A VDM-RT METHODOLOGY FOR THE HW/SW CO-DESIGN OF EMBEDDED SYSTEMS

BY

JOSÉ ANTONIO ESPARZA ISASA

20097706

MASTER'S THESIS

IN

TECHNICAL INFORMATION TECHNOLOGY

DISTRIBUTED REAL-TIME SYSTEMS

SUPERVISOR: PROF. PETER GORM LARSEN

Aarhus School of Engineering

December 15, 2011

José Antonio Esparza Isasa	Peter Gorm Larsen
20097706	Supervisor

FACULTY OF SCIENCE

AARHUS UNIVERSITY

Abstract

This thesis presents a new methodology for Hardware/Software Co-design of embedded systems. The proposed methodology is based on the modelling language VDM-RT. This new approach allows system engineers to perform design space exploration at an abstract level earlier in the development process. This methodology supports the analysis of real-time deadlines under different hardware/software architectures, without making use of virtual or actual prototyping. Based on the information gained during this analysis, system engineers are able to allocate the required system functionality in hardware and software blocks. This allocation or partitioning process can be done through a rigorous study of the design trade-offs by applying the VDM-RT methodology. As an additional advantage, the system engineers are able to specify and capture clearly and unambiguously the system structure and behaviour. Besides presenting this new methodology, this thesis will show its application in two case studies and present a pragmatic analysis of its performance.

Acknowledgements

I would like to thank my academic supervisor Peter Gorm Larsen for his attention and advice during this thesis and my Master's degree. I would like to thank my industrial supervisors from Bang & Olufsen Gert Schilkowski and Karsten Langhoff Sørensen for their contribution in the AVB case study. Finally, I would like to thank Joey Coleman for his input regarding the content and structure of this work. Additional thanks go to my family and friends for their patience and support.

Table of Contents

List of Figures

Chapter 1

Introduction

This chapter presents the work carried out in this thesis. We start with a description of the background of this work in section 1.1, the motivation that lead to its development in section 1.2, and the thesis goals in section 1.3. The methodology has been applied to two case studies, presented in section 1.4. A brief review of related work is presented in section 1.5. The typographical conventions used in this thesis are presented in the reading guide, section 1.6. Finally, the structure of the thesis is described in section 1.7.

1.1. Background

Today efficient embedded systems require hybrid solutions that are able to integrate hardware and software components. Complexity has grown exponentially over the last few decades, reaching sizes up to 1 million VHDL Lines Of Code (LOC) [Black&04], even in small sized Integrated Circuits (IC). On top of the hardware layer, one has to consider the software functionality implemented in the system. Software has challenges similar to the ones introduced by hardware but typically at a higher level of abstraction. Mastering both hardware and software is a must for all engineering teams working in the area of quality embedded systems.

An emerging challenge that results from the combination of hardware and software components in order to create systems, is the functionality allocation. This is also known as the *partitioning process*. The partitioning process is the stage in the design process in which it has to be decided whether a functionality has to be implemented in hardware or in software. This allocation/partitioning process is influenced by multiple factors like price, used silicon area, performance, meeting real-time deadlines and so on.

In order to master to incorporate such a variety of details in the design process, multiple authors agree on the possibility of using a heterogeneous modelling approach [Gajski&09, Waddington&06, Shaout&09]. The abstractions that are introduced in those models are as important as the modelling itself. Abstraction allows engineers to leave out selected details so as to attend to others, enabling a step-wise approach to system design. Some of the main results of using modelling techniques in the development process are a better level of problem understanding and a higher level of confidence in the developed solution.

Current methodologies in embedded systems design are derived from the System-Level-Design (SLD) approach. SLD pursues the paradigm of specify/explore/refine [Gajski&09]. It aims to start the development process by modelling at the highest level of abstraction and then perform refinement down to the final implementation. Hardware/Software Co-Design is a promising SLD approach and a top-down methodology [Wolf03]. This approach is a model-driven engineering methodology, that aims to derive a system architecture from behavioural models. Deriving the system architecture implies performing the partitioning process that results in the definition of the hardware and the software blocks in a well-founded manner.

It would be fair to conclude that the application of modelling techniques play a major role in the development of new Hardware/Software systems. Models and abstraction are the only avenues to master the complexity present in today systems. Finally, traditionally it was preferable to implement functionality in software in order to lower the costs. Today's newer technologies and lower hardware prices are making possible to start considering hardware or software solutions as equal candidates. Approaches like Hardware/Software Co-design are gaining importance since they are focused on developing the functionality to be delivered, postponing the partitioning and functionality allocation process to a later stage in the process.

1.2. Motivation

This thesis work has been motivated by the author's interest in the Embedded Systems field. This interest was fostered in several courses during the Master's degree. The courses *Model driven development using VDM++ and UML* and *Hardware/Software Co-design* deserves special mention. The Model driven development course presented the way concepts like abstraction and formal modelling can be applied in the development of systems. Through the application of abstraction a complex problem can be boiled down to its essence, being easier to be approached and solved. A precise and abstract representation of a system can be created by the use of formal models. This representation allows the analysis and lays a solid basis for the implementation stage. Hardware/-Software co-design makes very extensive use of models in order to represent and analyse different architectural solutions. The applied models in this field can be at different abstraction levels, and can range from general system models for initial system specification to concrete subcomponent representations.

There are already several modelling technologies for Hardware/Software co-design in the market. Every modelling technology presents an associated methodology that describes its application throughout the development process. Some of these technologies are too generic and, even though they can be fitted for the purpose of communication and overall analysis, a thorough study of the problem becomes difficult to perform (e.g. SysML, UML profiles). On the other hand, there are very specialized technologies that tackle concrete problems in a proficient way, but are weak on representing more generic scenarios (e.g. Matlab Simulink). It is the author's belief that there is a need for a modelling workflow that takes the most beneficial aspects of both approaches and provides a way to conduct hardware/software design space exploration.

1.3. Thesis goal, approach and scope

This section describes the Master's thesis goal, the approach used and the scope of this work.

1.3.1 Goals

The main goals of the thesis are:

1. **To propose a general purpose methodology based in the VDM-RT modelling language in order to support the design decisions in the Hardware/Software co-design field.**
2. **To propose improvements and extensions to the VDM-RT language and the modelling tool Overture, seeking to contribute on the creation of a modelling tool that can be applied in a more specialized field.**

An additional personal learning goal of this thesis, as important as the previous ones, is to improve the author's skills and knowledge in the VDM-RT modelling technology and in the Hardware/-Software Co-design field.

1.3.2 Approach

The methodology applied in this thesis is structured in two phases, each containing several points of action.

Phase 1: Study, analysis and reflections
Reading and research activities that will allow the author to: evaluate the need for new methodologies and draw the requirements for new methodologies in the case they are needed. The result of this phase will allow to build the rationale behind the work that will be carried out during the rest of the thesis. This phase is decomposed in the following steps:

- Research on the state of the art of modelling techniques applied in system-level design.
- Exploration of current processes and workflows used in hardware/software co-design. This research includes both academic readings and interviews with experts in the field.
- Research on the current challenges in the hardware/software co-design field.
- Additional study on the FPGA technology.

Phase 2: Elaboration
The elaboration phase covers some of the methodological needs detected in Phase 1 by the design of a VDM-RT methodology for Hardware/Software co-design. This phase is decomposed in the following steps:

- Development of a VDM-RT based methodology in order to provide partial support in hardware/software co-design decisions.
- Application of the proposed methodology in case studies in order to validate its performance.
- Pragmatic analysis of results and evaluation of the proposed VDM-RT based methodology.

1.3.3 Scope

This work will be focused on:

- Proposing a methodology that supports the process of making partitioning decisions during the development of hardware/software systems.
- Focusing on partitioning decisions that can be reached through the analysing of real-time deadline accomplishments, interaction between components, and approximate system behaviour modelling.

This thesis does not have as an ultimate goal the implementation of industrial solutions for the problems presented in the case studies. Actual incorporation of the proposed improvements to the tool Overture or the modelling language VDM in any of its flavours are out of the scope of this thesis work.

1.4. Case studies

The proposed methodology has been applied in two case studies. A core part of the thesis has been the evaluation of different partitionings with regard to real-time deadlines. The first case is conceptually simple with a single real-time deadline, aiming to show how the methodology should be applied. The second case is based on a real industrial application and with multiple real-time constraints and several possible partitionings. A brief description of these cases is provided below, for further information refer to chapters 5 and 6.

- **Development of a servo controller**: The servo controller case will present how different implementation strategies can be modelled by the use of VDM-RT. This case shows the application of the methodology in a single-process scenario with real-time deadlines. This is an introductory case study based on the workshop [Speedway11SDK].
- **Development of an AVB endpoint device**: **A**udio **V**ideo **B**ridging is a technology that aims to offer real-time multimedia content streaming over Ethernet network. This technology requires the implementation of protocols for time synchronization (802.1AS), traffic shaping (802.1Qat) and stream reservation (802.1Qav). The case is ideal for testing the performance of the methodology in a time-critical, multi-process setting. The modelling focus will be set on the time critical aspects of AVB. This case study has been provided by Bang & Olufsen.

1.5. Related work

Before starting considering a new approach for Hardware/Software co-design, a review of the current modelling techniques, methodologies and challenges has been performed. In this section, only an overview of the considered work is presented. Additional details in existing approaches will be presented in chapters 2 and 3.

Formal modelling processes for Hardware/Software engineering
[Huang&04] proposed the **S**oftware **H**ardware **E**ngineering methodology, based on the POOSL modelling language. This methodology is based on a formal object-oriented modelling language. It has been applied in a number of industrial projects and several tools have been developed even allowing model compilation. Unfortunately this area of research seems to be inactive. In his thesis, Verhoef introduces a real-time extension for the formal modelling language VDM [Verhoef05]. This extension provides the necessary constructs for the evaluation of different system architectures in a distributed real-time deployment. Additional research and development effort has been spent on this and a concrete implementation of this is in the modelling tool Overture. A complete development process entirely based on the VDM method is proposed in [Lausdahl&10b, Larsen&09]. This methodology makes use of the VDM-Specification Language, the VDM-PP Object Oriented modelling language and, finally the VDM-Real Time extension. This process makes use of formal model during the whole project life-cycle. The tool support

for this process is provided by Overture [Larsen&10b] and by VDMTools [VDMTools]. There is no previous work related to the application of the VDM methodology in the Hardware/Software Co-design field. The current research in VDM is working toward the integration with other modelling platforms in fields like mechanical co-simulation [DESTECS09] or System-of-Systems [COMPASS11].

Specialized processes for FPGA based systems
Xilinx presents a development process for FPGA-based system design in [XilinxToolFlow11, Speedway11EDK]. This workflow is highly tool-oriented and slightly covers the problems associated to hardware/software co-design of embedded systems. Simulink provides a sophisticated modelling framework that can be used in many modelling activities [Simulink]. One of these activities is the modelling of FPGA-based systems. Ou et alin [Ou&05] makes use of Matlab/Simulink in order to co-simulate different hardware partitions. The same authors propose an interesting contribution where Matlab models are applied for energy estimation consumption in FPGA soft-processors [Ou&04]. The existing literature is focused on using Matlab as an specialized tool in order to create models that will help on taking concrete partitioning decisions. Little effort has been spent on relating these modelling activities with a complete methodology to be applied for complete system development.

Combined SystemC - UML profiles based processes
Mischkalla et al. and Prevostini et al. propose the combination of SystemC with UML profiles in order to create a sustainable modelling-implementation process through the development process [Mischkalla&10, Prevostini&07]. Such an approach is interesting since two modelling technologies are combined. This heterogeneous application is taking the most beneficial aspects of both technologies and mitigates possible shortcomings. However, UML profiles can be limited when it comes to model execution and SystemC models might be biasing partitioning decisions towards software (SystemC is a C++ class library). [Mueller&10] describes the approach followed in the ongoing EU FP-7 project SATURN (SysML Based Modelling Architecture exploration, simulation and synthesis for complex embedded systems). In this approach an addition to Artisan Studio is used to create SysML models that can be synthesised. The generated code can be co-simulated in a virtual platform for system verification. Finally, generated code can be deployed in the actual platform.

The proposed methodology for Hardware/Software Co-design tries to approach the hardware partitioning problem, that is not explicitly addressed in the existing VDM-RT development process [Larsen&09, Lausdahl&10b]. The proposed methodology benefits from several ideas proposed in the previously cited papers.
One of the most relevant features of the VDM-RT based methodology for hardware/software co-design is that it is not specialized to address particular problems. This is beneficial from the system perspective, as it is better suited than the specialized processes for FPGA-based system. However, when it comes to solve particular implementation details [XilinxToolFlow11] and [Simulink] might be more adequate.
Finally, the combination of graphical notations applied in the combined SystemC - UML profile based processes are interesting and the synthesis possibilities outstanding. Note that this has been reached after years of development and considerable investments, therefore it will not be fair to compare the methodologies considering the state of the tools. Considering only the modelling methods used, it could be conclude that SysML might lead to an informal use of the modelling technology, implying imprecisions and modelling errors. SystemC is too close to the implementation language C++ and might alter the focus of the modelling and bias the system engineer to

more software-based implementations. On the other hand VDM-RT is far from final implementations of the system, since it is an abstract modelling language by construction. A more in-depth comparison of these approaches will be provided in chapter 3.

1.6. Reading guide

This thesis makes use of the following conventions:

References:
Articles, books, technical reports, figures and anything that is not the author original work is referenced. The references are placed between square brackets containing the surname of the primary author and the year of publication, e.g. [Brooks86]. In the case there are several authors for the same resource the character & will be placed between the surname of the primary author and the year of publication, e.g. [Black&04]. Finally, in the case the author has published several articles during the same year, a letter will be added following the year of publication, e.g. [Lausdahl&10b].

Emphasis:
Concepts, words or sentences with special relevance are emphasized in the text by the use of *italic*.

Guidelines:
General instructions and modelling advice are given in the form of guidelines, that present the following style:

Guideline 9: Each hardware candidate should be deployed on an individual CPU configured as a hardware block.

Keywords and language instructions:
C, VHDL and VDM keywords are shown in **`typewriter boldface`**. Other instructions are shown in `plain typewriter font`.

VDM model listings:
VDM models are placed in listings that present the following style:

```
private generateCycle: () ==> ()
generateCycle() == ();
```

VHDL code listings:
VHDL source code snippets are placed in listings that present the following style:

```
control_counter_process : process (clk, rst)
begin
end
```

C code listings:
C source code snippets are placed in listings that present the following style:

```
void rc_servo_set_servo_position(int baseaddr,
                                 int servo_number,
                                 int position);
```

1.7. Structure

This thesis is organized in seven chapters, of which this introduction is the first. Figure 1.1 presents the thesis structure in a graphical manner.
Chapters 2 and 3 summarize the theoretical foundation of this work. These chapters will provide the necessary concepts in System Level Design, Hardware/Software Co-design, Workflows and Modelling.

- Chapter 2 introduces the concept of abstraction and its application in the Hardware/Software engineering field. Special emphasis is made in the importance of the partitioning decisions during the analysis and design phases of the project.
- Chapter 3 gives and overview of the most relevant modelling languages used in system level design. Together with the languages, workflows that make use of them in order to model and study Hardware/Software systems are reviewed.

The following chapters present and apply in practical cases a new methodology for design space exploration in Hardware/Software co-design:

- Chapter 4 presents the VDM-RT methodology proposed by the author. Concrete guidelines and modelling advice will be given so it can be applied by a modeller with minimal knowledge of VDM-RT. A critical view on the methodology will be presented.
- Chapter 5 introduces the case study on servo control. The proposed methodology in this thesis will be applied in a case that is easy to understand by a reader with minimal knowledge on hardware design. The methodology will be applied from the analysis and design stage down to the final implementation of two different architectures. This case study will give the possibility of validating the hypothesis derived from the VDM models against real implementation details.
- Chapter 6 introduces the second case study of this thesis which is based on the development of an AVB endpoint device. The system under consideration in this case study is complex and involves several protocols. The VDM-RT methodology will help to detect critical aspects and to provide information in order to make design decisions. An actual implementation in this second case study has not been performed due to time limitations. This case study has been proposed by the commercial electronics manufacturer Bang & Olufsen.

This thesis is concluded in chapter 7, which summarizes and analyses the achieved results. It also establishes areas in which further work could be performed.

This thesis contains five appendixes. Appendix A provides a list of definitions and abbreviations of hardware/software terms used in this thesis. Appendixes B and C present the produced

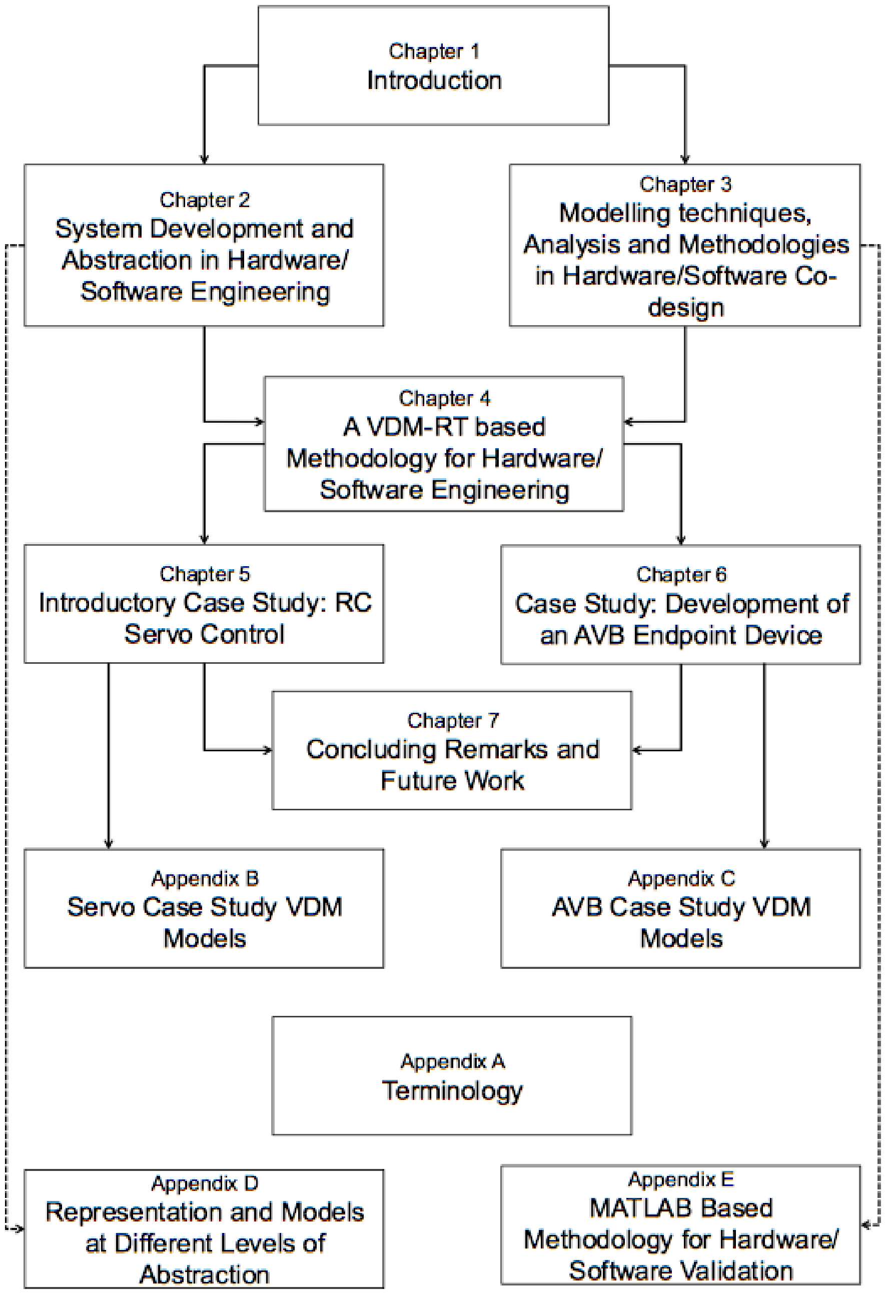

Chapter 1
Introduction
Chapter 2
System Development and Abstraction in Hardware/ Software Engineering
Chapter 3
Modelling techniques, Analysis and Methodologies in Hardware/Software Co-design
Chapter 4
A VDM-RT based Methodology for Hardware/ Software Engineering
Chapter 5
Introductory Case Study: RC Servo Control
Chapter 6
Case Study: Development of an AVB Endpoint Device
Chapter 7
Concluding Remarks and Future Work
Appendix B
Servo Case Study VDM Models
Appendix C
AVB Case Study VDM Models
Appendix A
Terminology
Appendix D
Representation and Models at Different Levels of Abstraction
Appendix E
MATLAB Based Methodology for Hardware/ Software Validation

Figure 1.1: Thesis structure

VDM models for the servo and the AVB case studies respectively. Additional details on representations and models used at different levels of abstraction are given in appendix D. Finally, more information about the MATLAB based validation methodology is provided in appendix E.

Chapter 2

System development and abstraction in hardware/software engineering

This chapter introduces system development and complexity, presenting the rationale for new methods and tools in section 2.1. Abstraction levels that can be used in a Hardware/Software Engineering context are described in section 2.2. The approach System-Level Design and the modelling languages used under it are described in sections 2.3 and 2.4. The concrete System Level Design approach hardware/software co-design is presented in section 2.5. Finally, the chapter is summarized in section 2.6.

2.1. Introduction

Before presenting methods, tools and system related abstractions, the term system itself requires to be defined. According to the Systems Engineering Handbook [INCOSEseh10] a system is defined as:

> "An integrated set of elements, subsystems, or assemblies that accomplish a defined objectives. These elements include products (hardware, software, firmware), processes, people, information, techniques, facilities, services and other support elements.

Besides the system engineering perspective present in the development, that is considering even facilities and services, the element under study in this thesis is what is defined as *product*. Both, [NASAseh10] and [INCOSEseh10] qualify *hardware*, *firmware* and *software* as crucial elements to produce system-level results.
Systems are developed by carrying out different processes which, according to [INCOSEseh10], can be classified as organizational, technical and managerial. The development processes proposed in this thesis work falls under the category of *technical processes*.
The purpose of the lines above is to contextualize the methodologies applied within the technical processes, that will be discussed in chapters 2, 3 and 4. The ultimate aim of these processes is to support and perform the development of a product, composed by *hardware*, *software* and *firmware*.

Hardware/Software systems are composed by a wide variety of elements and components of different nature, presenting a growing complexity. This complexity has increased exponentially over the last decades and traditional approaches, based on simple methodologies and the engineer's experience, are not always enough to tackle the project complexity. [Black&04]. This kind of situation leads to cause *analysis-paralysis*[1] in the system development process or *poor decision making* overlooking at critical details, something that may flyback later on in the project life-cycle. With the purpose of dealing with this complexity, [Patel&04] proposes to rise the level of abstraction in system design and to use different methods and tools in system development. A similar argument is hold by [Black&04]. Gajski in [Gajski&09] presents that it is needed a systematic and well-founded approach to design combined hardware/software solutions, that supports the specification phase and provides solid arguments to perform design decisions such as system hardware partitioning.

The current approach in System Design Methodologies is, according to [Gajski&09], and from the early 2000s, *Specify, Explore-and-Refine methodology*, which introduces the so called *System-Level (SL) perspective*. This approach is primarily using the concept of abstraction in order to define different levels. Abstraction levels are enabling an stepwise system development, which helps on facing a challenging project, in which the development team has limited experience or is specially complex.

2.2. Abstraction levels overview in a single system

Before going in detail on abstraction levels, the concept of abstraction itself deserve special consideration. There are different definitions explaining what abstraction is focusing on different aspects. Kramer in [Kramer07] is presenting abstraction as the process of *removing detail to simplify* and as a *generalization to identify* the common core. The first definition, which is the most connected with this thesis work is a quote from Kramer:

> "the act or process of leaving out of consideration certain properties of a complex object so as to attend to others."

In the system development context, abstraction is playing a key role allowing the engineers to tackle the complexity inherent in hardware/software systems. As Kramer explained and applied in this context, this tool will allow the designers to consider some of the properties that should be present in a system without considering others, that will be taken in account further in the development process. Therefore, abstraction is a enabling tool to describe the system at different levels and to perform different kind of analysis.

The way from a system described in terms of inputs and outputs (blackbox approach), to a system described with a fully detailed hardware implementation is organized in several layers of abstraction. All of them are relevant during the development process, providing details in different implementation areas. Abstraction is therefore, mainly, a tool to deal with the complexity and the amount of details present in a system.

In a combined hardware/software context, multiple authors ([Gajski&09, Black&04, Sudhakar05]) agree on six abstraction levels for single-device based systems. The levels that this thesis work is covering are:

[1]Common term to refer to over-analysing a situation, causing no productive results and no progress in decision-making.

Algorithmic Level: The algorithmic level is focused on the behaviour the system has to implement. At this level, no description of the hardware implementation is made. The effort is focused on describing the information the system is receiving, how this information is processed and which outputs should be provided. At this level it is possible to analyse the software logic. Relevant aspects could be concurrency, protocols, policies or data consistency.

Interface Level: The interface level incorporates details about the interface between the functionality implemented in hardware and the software components. This level is commonly referred in the literature as the "programmers view" At this level the hardware partition is clearly defined. Some of the analysed aspects are data availability, data format and protocols.

Transaction Level: The transaction level view adds timing details to the interface. It offers time related details that can be useful to estimate real-time deadline accomplishment. Appropriate real-time performance at this level is necessary condition but not sufficient. It is also know as the "timed programmers view", and models at this level are named Transaction Level Models (TLM). At the transaction level approximate time details are used to analyse timing details related to the hardware/software interface.

At a lower level of abstraction we can find the *Cycle Accurate Level*, the *Register Transfer Level* and the *Silicon Level*. These levels are not relevant for this thesis work and further information can be found in [Gajski&09].

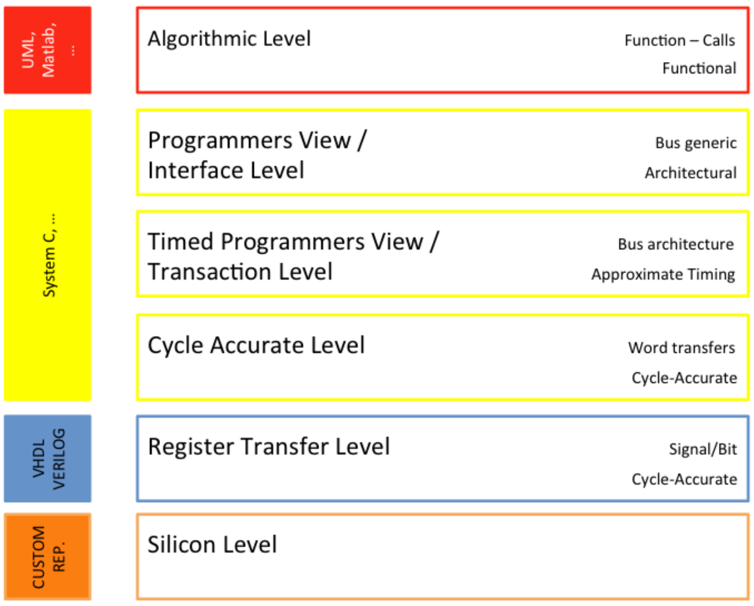


Figure 2.1: Abstraction levels in a hardware modelling context.

Figure 2.1 is detailing the abstraction levels discussed above. Note that the targeted levels by this work are at the top of the abstraction stack. The algorithmic level is often stratified in several sub-layers and varies in complexity. This last issue has to do more with the embedded software development field rather than with the partitioning process, therefore this thesis will not elaborate further in that area.
More information about the different representation and models that can be used at each abstraction level can be found in appendix D.

2.3. The System-Level perspective

The system-level design approach is aiming to start the design process at the highest abstraction level of the ones explained above. From this level, the system is represented by an executable specification that represents the behaviour (*Specify*). Additional models are incorporated to this holistic view in order to illustrate different facets of the system (*Explore*). These models can provide details on design decisions and serve as a tool to evaluate different configurations. Once exploration be the means of these models is concluded, additional implementation details are added (*Refine*). Each model refinement will be carried out after taking more design decisions, therefore incorporating more details about the final system. A relevant characteristic of SL design is to reuse components and Intellectual Property (IP) blocks whenever is possible, minimizing the development effort.

The system level design approach follow a sound engineering procedure applied during many years in multiple fields. SL starts with a initial specification, followed by possibilities exploration and finally model refinement down to actual implementation [Gajski&09].

2.3.1 Different approaches in system-level design

There are three main system-level design methodologies that make use of the approach Speficy-Explore-Refine introduced above [Shaout&09, Gajski&09]:

Hardware/Software Co-design: also known as top-down methodology or Model-Based Design. This methodology pursues to model the behaviour of a certain system in order to derive an architecture from it. Deriving this architecture implies cooperative design of hardware and software.

Platform-Based Design: this approach allocate the functionality implementing the behaviour to a previously defined architecture.

Component-Based Design: also known as bottom-up methodology. This methodology starts by creating the building blocks at the lowest level of abstraction. These blocks will be used in the creation of the immediate superior layer in the implementation. All blocks are organized at libraries present at the different abstraction levels.

2.3.2 Application of system-level design approaches

The application of System-Level design methodologies depends on the encountered situation. *Platform-Based Design* is the required approach in the case a certain platform has to be used. For instance, in the case an update over a certain system has to be performed and the architecture for this system has been already defined. The system might be featuring several processing units, so the issue considered in a platform based design approach is in which out of these processing units the functionality has to be allocated.
Hardware/Software co-design will support the development of an optimized platform, supporting this task with metrics on different factors. The use of this flavour of System-Level design approach is ideal for development of new products, in which a higher degree of flexibility is present. Design space exploration is fostered in this approach.
Finally, *Component-Based Design* requires an extensive creation of components library at each abstraction level. These components have to be created, ideally, before moving to the superior layer. It is an arduous task, if not impossible, to make a prediction of exactly which components

are to be used in a certain product, therefore there have to be components implementing many different functionalities which are not going to be used in a certain system. As an advantage, the bottom-up methodology facilitates the creation of re-usable building blocks, clearly allocated in an abstraction level. This approach could be adequate for a large hardware manufacturer, which will possibly reuse components in later developments.
From now on, the focus of this work will be the system-level design approach pursuing Hardware/Software co-design.

2.4. Modelling languages in system-level design

The essence of System Level (SL) design, as expressed by [Shaout&09] could be resumed as:

> "System Level Design maps a high-level abstract specification model of an entire system onto a target architecture."

Besides Shaout et al., many authors agree that speaking about System-Level Design without considering modelling would be impossible. Since the moment in which the concept of abstraction is present in this way of designing systems, it is clear that certain kind of system representation is going to be present. This directly implies that models are going to be used, which lead to consider two issues: a) The kind of languages needed to make such a representation and b) The level of and kind of details that should be present at this initial, abstract representation. This issue has been widely discussed in the literature.
At this point, more relevant than reviewing concrete languages, it is more interesting to consider which possibilities a modelling language should offer in order to be useful in a System-Level Design context. According to [Shaout&09, Gajski&00, Niemann98] a modelling technology used to specify a system should incorporate facilities to represent the following elements/properties:

Hierarchy: as explained above, increasing size of systems makes necessary to introduce structures in order to organize system behaviour and architecture. These structures typically have subordinated components solving part of the problem that they are responsible for. Making a proper representation of a system structure is a basic feature.

State transition: system operation can always be described as a state machine. Depending on external inputs (current, previous or both), the state can be in different operational modes or states of execution. Some embedded systems are purely implemented as state machines, its use is a common practice at both Software and Hardware level.

Concurrency: in many cases a system is executing several processes or threads at the same time. There is a leap in complexity from single-task based solutions to multi-task ones, both at the implementation and specially at the debug stage. The modelling language should be a tool to support this situation and foresee possible conflicts and concurrency problems.

Synchronization: The need for synchronization modelling support is derived from concurrency. Race conditions, deadlocks and other concurrency related problems can cause complete system malfunctioning. Like concurrency problems, these cases can be extremely difficult to detect. In some situations, problems are detected when the system has been already deployed. With the introduction of multi-core processors, synchronization must be guaranteed even at the hardware level. A modelling language supporting concurrency must support synchronization in order to be useful in the multi-task domain.

Interrupt handling: interrupts are common mechanisms in embedded-systems, often used to signal events. These kind of events may require the interruption of the current task and, therefore, transition to an alternate state to process them. Some embedded systems are purely interrupt driven. An modelling language used to model embedded systems must be able to represent this kind of situations.

Timing: many embedded systems present soft/hard deadlines in its operation. Not satisfying these deadlines could imply complete system malfunctioning. Modelling constraints of this kind, specially in concurrent systems is a desirable property.

Constructs to support formal verification: it should be possible to express certain system properties so logic and discrete mathematics can be used to prove that they are present in the system. While powerful, formal verification is sometimes far from the embedded engineer mindset. Strong tool support is required to make formal verification usable in an industrial setting.

Dynamic behaviour: in the case the system is making use of dynamic scheduling in order to coordinate software operation, the modelling language, ideally, should provide a way to represent and analyze it. Dynamic scheduling is often implemented by the use of a Real Time or ordinary[2] operating system.

In [Huang&04], the authors rise the level of abstraction and require different features that a modelling language should present:

Adequate expressive power: the modelling language has to be capable of expressing complex properties, behaviour and structure.

Platform-independent semantics: the semantics of the language shall not be bounded by a a certain technology. The use of the language should not be restricted to a concrete platform.

Operational semantics: this property refers to the *executability*. By executing a model, simulation and/or model-checking techniques can be applied in order to detect inconsistencies and prove correctness.

Modularity support: as explained above, the possibility of expressing the internals of a component and its interfaces is a must.

Automatic and correctness-preserving transformation: the system-level design model should be a *golden model*[3], that will be refined until the actual system implementation is achieved. During this process, a large number of refinements and transformations will be applied to the initial model. By using a tool that is automating these transformations and, furthermore, ensuring that certain properties are satisfied, the development of the system is easier.

It is difficult to find a modelling language that incorporates all the features suggested by Huang, Gajski and Niemann. A compromise solution must be reached when deciding on a concrete language. Further details on concrete languages and application areas will be introduced in chapter 4. This requirements are giving an idea about the kind of details that should be present at a System-Level Design model, or at least considered in an initial specification.

[2]Non-Real Time Operating System

[3]Common term to refer to a model that is used as a reference during the whole development process.

2.5. Hardware/Software co-design

Nowadays embedded system are often implemented in heterogeneous architectures, that combine dedicated hardware blocks, processors and software components deployed in processors[4].
The possibility of using different hardware and software blocks opens a wide range of possibilities when it comes to the architectural design stage. The challenge in this situation is to decide whether a certain behaviour should be implemented as a software or as a hardware component. The Hardware/Software co-design discipline provides the tools and the methods to generate architecture from behaviour by adding implementation details to the design.

Before digging in to some of the most outstanding aspects about this discipline, it is worthwhile considering definitions from the literature:

[Schaumont10] defines this discipline as:

> "Hardware/Software co-design is the design of cooperating hardware components and software components in *a single design effort*."

The same author remarks as well:

> "Hardware/Software co-design *focus on the partitioning and design* of an application in terms of fixed and flexible components."

[Wolf03] remarks:

> "... hardware/software co-design tries to increase the predictability of embedded system design by providing analysis methods that tell designers if a system *meets its performance, power and size goals* and synthesis methods that let researchers and designers rapidly *evaluate many potential design methodologies*."

Finally, [Shaout&09] characterize Hardware/Software co-design as follows:

> "... also referred to system synthesis, is a top-down approach. Starts with system behaviour, and *generates the architecture from the behaviour*. It is performed by gradually adding implementation details to the design."

These four quotes jointly give a good overview of what Hardware/Software co-design is: "a single design effort", what the main focus is: "the partitioning and design", which criterion are relevant in partitioning: "performance, power and size goals" and, finally, how the system is produced: "architecture derived from behaviour". The following sections elaborates on the details outlined by these three authors and presents FPGAs as platforms for using this methodology.

2.5.1 Trade-offs and impact of Hardware/Software co-design

Every system specification presents a certain collection of targets that shall be fulfilled in order to produce a successful implementation. Some targets, as they were mentioned by [Wolf03], are specified in the form of requirements that *must* be satisfied. Other targets, are design variables that can be optimized in order to produce a higher quality system. In order to consider a design target as fulfilled, and in order to optimize its variables, it is needed to apply a specific criteria. By

[4]In this case "processor" is a common term to denote to a "microcontroller", a "Digital Signal Processor" or a "microprocessor".

applying it, it is possible to compare solutions and determine which one is valid/better. Connected criterion with opposite gradients in its optimization function are known as trade-offs.
Considering a trade-off situation; a change on one of the involved variables might have an impact on the rest. For example, raising the frequency of the processor speed up the computation therefore, the performance. The higher the frequency the higher the energy consumption. In some cases the higher the temperature of the silicon.
The challenge here is to compromise in order to find the optimum balance in these trade-offs. The most relevant criterion, that might imply trade-offs from different perspectives in the design space, are explained below. These criterion is proposed by the author taking inspiration from [Chen&09] and [Schaumont10].

From a technical point of view, the most important variables that might present trade-offs are:

Performance: The level of abstraction in which a certain solution is implemented has a direct connection with its performance. By construction, the hardware layer is at a lower level of abstraction than the software layer. The existing overhead in a hardware functionality implementation is practically null. A hardware implemented functionality presents a higher degree of performance in *all* the cases. A hardware implementation might be more expensive to create in terms of time and material costs.

Flexibility: Functionality implemented as a software component is more flexible in terms of reusability and modification. A hardware block is typically suited for a single purpose and can hardly be reused for a different purpose.

Energy efficiency: energy constraints are present nowadays in many systems. Some systems require to be energy efficient so their operational life is longer e.g. a battery powered multimedia system. Others require to be energy efficient in order to be environmentally friendly e.g. a class-A home appliance. Finally, due to pure physical reasons, the more energy an electronic device is consuming the more heat is going to be dissipating. This might lead to performance decrease or even compromise system availability. A system that is operating at a higher frequency might perform better, but it will consume more energy.

Complexity: The heterogeneity present in a System-On-Chip (SoC), presents a challenge itself. Considering a hardware block in isolation, several thousands lines of VDHL code have to be used in order to implement a functionality that might require hundreds in the C programming language. This means that, a hardware implementation is harder to design, create and maintain. Furthermore, debugging facilities in software are more simple than the ones available in hardware. The latter kind often implies a combination of external equipment, in-system logic analyzers[5], ... On the other hand a hardware solution generally presents better performance than a software solution.

Co-design decisions have an impact as well from a managerial point of view. A certain architecture will present particular time and cost demands:

Engineering hours cost: Development effort is one of the most relevant issues when evaluating different design possibilities. The amount of man-hours that will be spent in the development depends on the available time-budget assigned to the project. A heavily based hardware solution will have a considerably higher demands on engineering development hours than a combined hardware/software approach.

[5]An example of in-system logic analyzer is Xilins Chipscope. This tool enables the developer to analyse the logical signals inside the FPGA. For additional details refer to `www.xilinx.com`.

Material cost: Material cost is one of the most relevant recurring design variables. Even minimum savings per unit lead to an important revenue in the long run. This might be in conflict with with the performance or the energy efficiency variables.

Development time availability: Current product developments are characterized by the short time-to-market. Reusability of hardware blocks whenever is possible, integration of IP cores or partitioning are some of the factors that will make certain architecture feasible within the available time frame for the development.

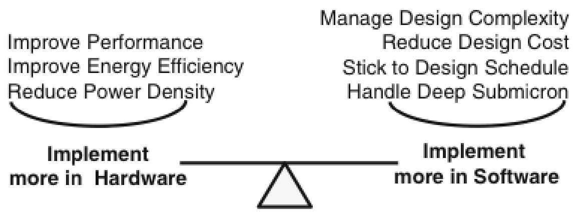


Figure 2.2: Trade-offs in Hardware/Software co-design. Source: [Schaumont10]

The technical criterion could be refined even further and consider more particular details. These details depend on the problem under study. In [Brogioli&06], the partitioning is performed considering mainly the real-time deadlines for a 3.5G mobile receiver. In this case the data and computational related aspects are especially relevant. The goal in this case is to derive an architecture that fulfils the real-time deadlines. The approach is to improve the performance of the system. The performance sub-criterion used in this partitioning problem were:

Data spatial location: Data can be stored in a number of different hardware elements in a certain architecture. Latency, access and setup times or bus usage make a difference when it comes to read or write data. Where to place the data, besides the used memory technology, will affect the time performance of a concrete system architecture.

Data level parallelism: Parallel data availability can be exploited from an implementation point of view. FPGAs and DSPs to some extent are strong on receiving a parallel stream of data reducing data input latency in the system.

Task level parallelism: An implementation in which multiple elements are solving a complex problem in a parallel way is considerably more efficient than a serial processing approach. When combined with a high degree of *Data level parallelism*, a parallel implementation might present an outstanding level of performance. However, not all computational problems are suitable to be solved by parallel computing. For example the image processing field present numerous algorithms suitable for parallel computing. On the other hand, the algorithms used in a network interface are not the ideal case for parallel processing.

Computational complexity: An algorithm running in a DSP or CPU might present bottlenecks that can be detrimental to the overall performance. The solution in many cases comes in the form of *hardware off-loading*[6], which implies moving the worst performing algorithms executing in the host processor to a hardware implementation in a FPGA. The criteria to determine which algorithm is performing best is not always the same and, in many cases, it is ad-hoc. In some cases the validation criteria can be time deadlines accomplishment, in other cases throughput, etc...

[6]A concrete hardware off-loading case will be presented in chapter 5

These aspects might not be critical or not even present in all applications. The conclusion that can be drawn is that, besides the general issues described above, particular partitioning sub-goals might arise in a certain project. This presents, a challenge for a co-design methodology. The applied techniques should be flexible enough in order to incorporate the *heterogeneous details* and *particular* considerations that each problem might present.

2.5.2 Overview of the Hardware/Software co-design process

There are several methodologies and techniques that can be applied in order to do Hardware/-Software co-design. All of them are pursuing the objectives described above. Even though they might differ in techniques, artefacts or performed analysis, all of them follow the overall structure presented in figure 2.3.

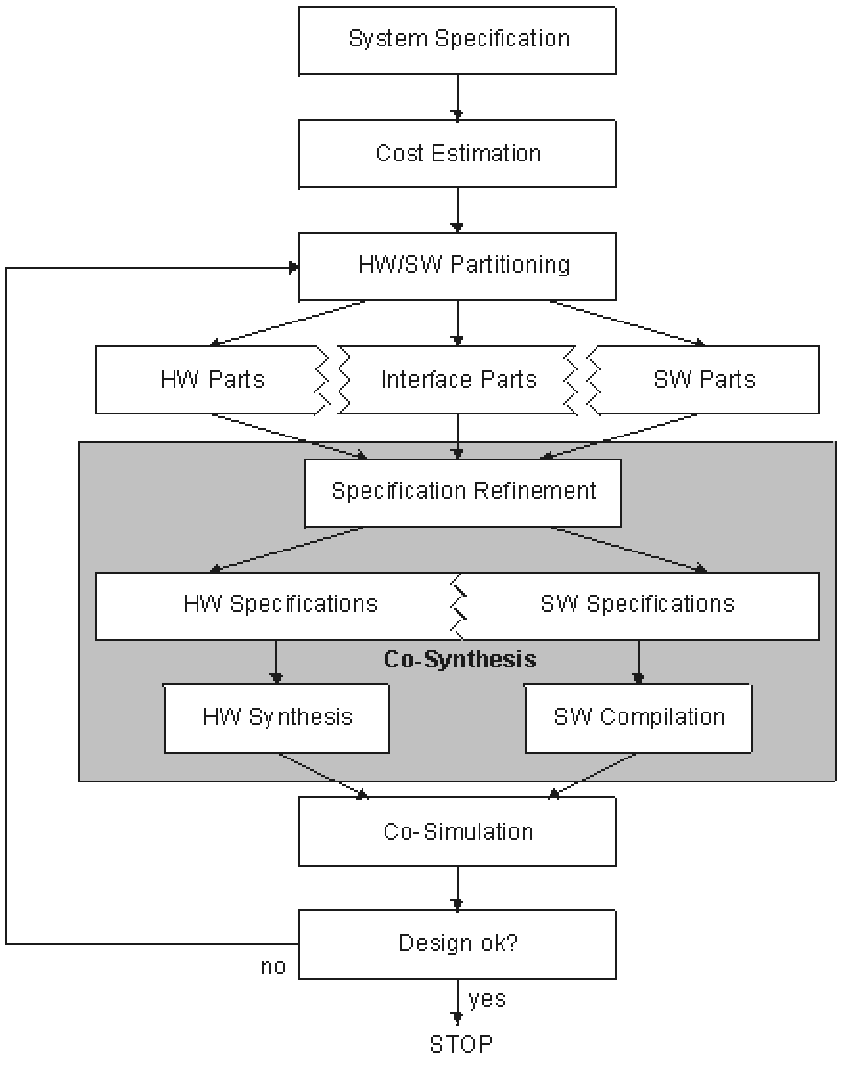


Figure 2.3: Hardware/Software co-design process overview. Source: `www.tu-dortmund.de`

The system design stage starts with the phase *system specification* in which functional and non-functional requirements are defined. In *cost estimation*, a cost factor for each design alternative is determined. This cost factor enables the system architects to evaluate the design alternatives against a set of criterion previously defined. These criterion might be similar to the ones introduced above. There are different models that can be used to perform cost estimation. Ad-hoc models, system models or previous experience are the most typical ways of carrying out this phase. This phase is where the design space exploration takes place. Once the optimal solution given the situation has been determined, the *Hw/Sw Partitioning* takes place. This phase will determine the system architecture, and will result on the definition of *Hardware parts, Interface parts and Software parts*. The next stage in the design is the co-synthesis, in which the Hardware and Software blocks are created in parallel. A key role in this stage is presented by the *refinement towards hardware specifications and software specifications*. The result of the co-synthesis is a model that can be co-simulated, so hardware and software blocks can be run together in a software platform. Results can be validated and verified against the initial requirements. In the case the validation is successful a bitstream for a concrete FPGA and a binary for a certain processor can be generated and deployed on target. Further testing activities on a real[7] hardware platform are performed and finally on the actual platform. The *Hardware/Software co-synthesis* and the *Real and Actual hardware testing* stages are out of the scope of this thesis.

2.5.3 The role of modelling in Hardware/Software co-design

As explained in sections 2.3 and 2.4, modelling techniques are fundamental tools in the System Level Design approach. Since Hardware/Software co-design is a model-driven engineering approach, the relevance of modelling is even greater. Many authors highlight the importance of models of computation and its application through the design process [Gajski&09, Black&04, Schaumont10, Shaout&09, Chen&09].

Two concepts are crucial in order to understand models in System Level design and, in particular, in Hardware/Software Co-design: a) Abstraction and b) Refinement. Both of them have been already explained in section 2.1. When considering models in this context, and according to Gajski et al. and Black et al. in [Gajski&09, Black&04], it is important to keep in mind that:

- The less abstract a model is, the closer to the final implementation it is;
- refinement is performed when lower level details are incorporated to a higher abstraction level model;
- the added details are relevant from the communication or from the computation (functionality) point of view and
- the precision of the notion of time can be improved by refinement.

The initial modelling activities start with the creation of a Specification Model (A), this model is named by [Black&04] as System Architectural Mode (SAM). A SAM model does not incorporate any notion of time. The main outcome is the *causal ordering between processes*. The causal ordering allows to describe what happens before and after considering certain event, in other words ordering of events and actions. The next refinement considering computation is the *Timed Functional Model* (B). At this step time annotations of certain computations and estimated delay results are incorporated. The first model incorporating an approximated notion of time in both communication and computation aspects is a Transaction Level Model (TLM) (C). From this point it is possible to refine the models in terms of communication or in terms of computation. A TLM model that is reaching a Cycle Accurate representation is called Bus-Cycle-Accurate

[7]The term real hardware platform refers to an FPGA development board. The term actual hardware platform refers to the final hardware device, the electronic platform that will be mounted inside the product to be developed.

Model (BCAM) (D). On the other hand, a TLM model refined in terms of computation is called Computation Cycle Accurate Model (CCAM) (E). When both BCAM and CCAM models are combined a Cycle-Accurate-Model (CAM) (F) is obtained.

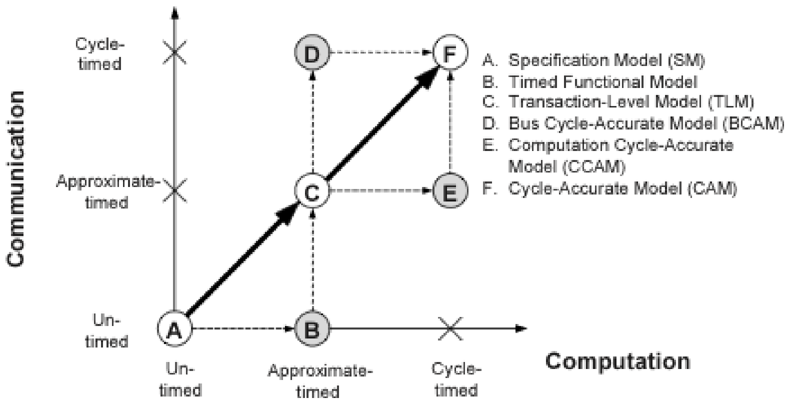

Communication
Cycle-timed
Approximate-timed
Un-timed
A
B
C
D
E
F
Un-timed
Approximate-timed
Cycle-timed
Computation
A. Specification Model (SM)
B. Timed Functional Model
C. Transaction-Level Model (TLM)
D. Bus Cycle-Accurate Model (BCAM)
E. Computation Cycle-Accurate Model (CCAM)
F. Cycle-Accurate Model (CAM)

Figure 2.4: Level of details with regard to communication and time. Source: [Gajski&09]

Figure 2.5 shows a similar model classification, provided in [Black&04]. Following this classification, BCAM models are called Bus Functional Models and CAM models are referred as Register Trasfer Level (RTL) models. Besides different terminology, the interesting issue is that Black shows intermediate refinements both in the communication and functionality aspects that were not provided by Gajski. These refinements fall under the category of Transaction Level Models. Note that, according to this criteria, it is not required to incorporate an approximate notion of time regarding functionality or communication to reach an RTL (CCAM or BCAM) level of representation in one of the areas. However, a complete RTL representation of the system is reached when both communication and computation aspects have been described in the model.

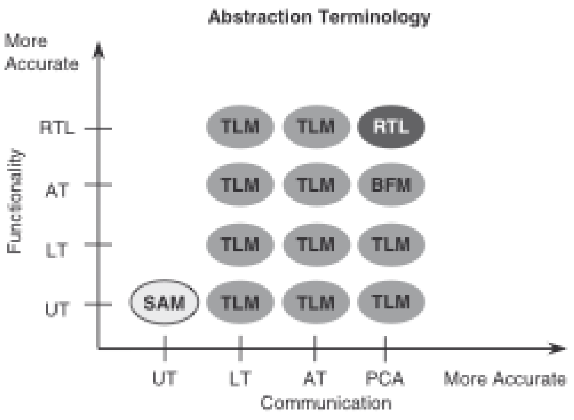


Figure 2.5: Level of details with regard to communication and time. Source: [Black&04]

2.5.4 Outcomes of Hardware/Software co-design

Applying Hw/Sw Co-design in system development provides advantages that are more complicated to obtain from other approaches. Among them, the following must be highlighted:

- Development of an architecture fitted for a particular behaviour, optimized according to concrete design criterion.
- Wider exploration of the Design Space, evaluation of multiple architectures before selecting. [Schaumont10] refers to this as "architectural awareness".
- Reuse of IP blocks in order to reduce implementation effort. Pursues to eliminate the *Not Invented Here* syndrome.
- Parallel and cooperative development of hardware and software.
- Consideration of faults, errors and possible problems earlier in the development process.
- Faster time to market, meeting the time requirements of todays business.
- Higher confidence in the achieved design, gained through a model based, well-structured process.

Current research and successful cases demonstrate that a methodology that is making use of Hardware/Software Co-design ends up creating better and higher quality embedded systems. However, one has to consider that implementing such a process in a development team, used to work by following an informal development process is not an easy task. Such a change should be introduced gradually and, to start with, being applied in a small project (risks mitigated). However, methodology adoption considerations are out of the scope of this work.

2.5.5 Enabling technologies

The main technology behind hardware/software co-design is reconfigurable hardware, which is supported primarily by FPGAs. FPGA manufacturers are creating custom toolchains especially adapted for the FPGA hardware they are producing. Both FPGA hardware and development environments are considered as enabling technologies for hardware/software co-design.

2.5.5.1 FPGAs and Hardware/Software Co-design

A FPGA is a semiconductor based device composed by Clustered Logical Blocks (CLB). The CLBs implement the logic present in the FPGA and are communicated through buses. The connections between the CLBs are established depending on the functionality that has to be implemented. Therefore, the wire segments that are conforming the buses are connected to what is called *Programmable Switches*, which state can be modified to fit the deployed implementation.
The most relevant feature of the technology described above, is that it enables the system developer to define and reconfigure a hardware structure to fit system design necessities. Some FPGAs allow hardware reconfiguration even in runtime. Such a flexible platform is idoneous for moving a certain functionality from the software the hardware side and vice versa. [Wolf03] refers to FPGAs and its connection with Hardware/Software co-design as follows:

> "The platform FPGA seems to be the chip for which co-synthesis was created: The chip's internal architecture is exactly what hardware/software partitioning targets."

Current researchers are using FPGAs as primary target platform for the application of Hardware/-Software co-design techniques. Material costs are decreasing considerably and its versatility, among other exotic features, is making them the perfect technological choice for many designs. Acceptance in industry is in constant increment.

2.5.5.2 Hardware/Software development environments

FPGA vendors are providing the end-users with development environments that facilitate the creation of hardware and software blocks. Taking as example the tools offered by Xilinx and Altera, the two main FPGA manufacturers in the market, the kind of tools that are offered are: VHDL Integrated Development Environments, system builders, software IDEs and some auxiliary tools[8]. The used tools in this work are:

System builder: gives the developer the possibility of creating a system with IP cores, processors and its own hardware components developed with the VHDL IDE. The possibility of creating heterogeneous FPGA solutions is supported by this tool.

Software IDE: allows the developer to create software for the hardware system defined in the System builder tool. The application logic developed in this IDE interfaces the hardware side through a Board Support Package (BSP). The BSP contains the hardware dependant software necessary to work with the created custom platform. It can be automatically generated by the tools.

Third parties are providing simulation and modelling tools. An insight on these technologies, and its connection with the hardware/software co-design process, will be given in chapter 3.

2.5.5.3 Other platforms

There are other platforms in the market supporting the partial reconfiguration of hardware blocks. It must be remarked that, even though they present some conceptual similarities to FPGAs, none of them reach the same level of flexibility. In decreasing order of complexity and affordability, some of the current available platforms are: Application-Specific Instruction-Set Processors, Programmable System-on-Chip (PSoC) and microcontrollers featuring Configurable Logic Cells (CLC). Additional information about PSoCs and microcontrollers with a CLC can be found in `www.cypress.com` and [PIC10F320] respectively.

2.6. Summary

Due to the complexity in nowadays system, abstractions and models making use of them have emerged as a necessity. The chapter has introduced the concept of abstraction and applied it in the context of system design. Different abstraction levels in hardware/software systems have been presented, showing which details can be *described* and *analyzed* at each of them. System-Level Design has been presented, remarking its connection with modelling languages and technologies. Finally, a certain way of doing System-Level Design, called Hardware-Software Co-design, has been described. The partitioning problem, aspects under study and role of modelling in this approach has been introduced.
In order to reach a better understanding on how modelling technologies are used in the development of new hardware/software systems, it is needed to conduct a more thorough review of the existing languages and methodologies in the market.

[8]Front-ends to configure additional hardware or tools to support the debugging process. For example, the application Chipscope, from Xilinx, makes it possible to insert a logic analyzer block in the FPGA and monitor the signals in different lines.

Chapter 3

Modelling techniques, analysis and methodologies in hardware/software co-design

This chapter provides an overview of the different modelling technologies and methodologies used for the hardware/software co-design of embedded systems. We start with the introduction in section 3.1, discussing the necessity of abstract modelling languages for representation and analysis. Section 3.2 reviews different modelling languages used for hardware/software design. Time representations used by the languages VDM-RT, SystemC and VHDL are explained in section 3.3. Section 3.4 analyses some methodologies using the modelling languages described in section 3.2. Finally the chapter is summarized in section 3.5.

3.1. Introduction

Multiple authors agree that there is a need for platform-independent languages, not directly associated with hardware or software implementations [Huang&04, Gajski&09]. Platform independent languages do not alter the partitioning process and avoid biasing. The consequence of avoiding biasing is to perform partitioning based on technical reasons and not based on the modelling technology used. Wolf in [Wolf03] refers to the modelling languages and their impact on partitioning decisions as follows:

> "Software languages like C bias the implementation in favour of software, while hardware languages bias the results towards hardware."

As an alternative, it is proposed to describe the system by applying an *heterogeneous* modelling approach. This approach proposes to:

- Use a hardware modelling language to represent behaviour that has to be *clearly* allocated in hardware and
- use a software modelling language to represent the rest of the system.

Note that, if it can be foreseen that certain part of the system clearly has to be implemented in hardware, there is no partitioning challenge in that case. Note as well that, if [Wolf03] approach is

followed, the rest of the system, considered as "not-clearly hardware", will be modelled by using a software description language. According to the same article "Software languages bias the implementation in favour of software". It will be fair to conclude that this approach will be biasing the whole implementation towards a software solution.
Even though Wolf's reasoning is too simplistic, the connection between the modelling language used and the final design decisions is well-founded. An alternative approach is to make use of more abstract modelling languages, that are not clearly connected with a particular implementation technology. The languages Parallel Object Oriented Specification Language (POOSL) and the Vienna Development Method (VDM) notations are examples of abstract modelling languages languages. Both languages are not connected with particular implementation technologies and can therefore be considered as platform-independent.
There are a number of authors proposing to apply heterogeneous modelling, using abstract modelling languages, different Models of Computation (MoC) and graphical modelling languages ([Gajski&09, Shaout&09, Patel&04]). The languages System Modelling Language (SysML) and Unified Modelling Language (UML) are examples of graphical modelling languages. The MoC used for System Level Design (SLD) are out of the scope of this thesis work so they are not considered further in this thesis (additional details are provided in [Gajski&09]).
This chapter provides a description of the modelling languages presented above and some of their associated methodologies for SLD. Additionally, the specialized languages MATLAB/Simulink and SystemC are presented. Even though they are not abstract languages, the fact that they are widely applied in the hardware/software system development field is making them worthwhile consideration.

3.2. Modelling languages

Two modelling approaches are considered in this work: a) informal modelling and b) formal modelling. These approaches could be defied as follows:

Formal modelling: that makes use of rigorous techniques, based on discrete mathematics in order to create models. These models are strong in the use of abstraction and precision. Formal modelling can be applied in the specification, design and verification of software and/or hardware systems. An example of a formal modelling language could be VDM.

Informal modelling: which does not make use of mathematical techniques in the process of model creation. Even though they lack formality and, to some extent, precision, they are strong in communication. An example of an informal modelling language could be the Unified Modelling Language (UML).

Current approaches in industry are making use of formal modelling mostly in the verification stage of the project, as a way of ensuring that the system hold certain properties [Gajski&09]. This section is making a distinction between formal and informal methods, but it is important to keep in mind that the differences between the techniques can be exploited in order to get more communicative models at variable levels of rigour [Bruel&98].

3.2.1 SysML

The System Modelling Language (SysML) is a graphical modelling language developed specifically to support Model-Based Systems Engineering (MBSE). SysML is a UML profile that is reusing some of the UML semantics and incorporating additional ones. This relation is depicted in figure 3.1.

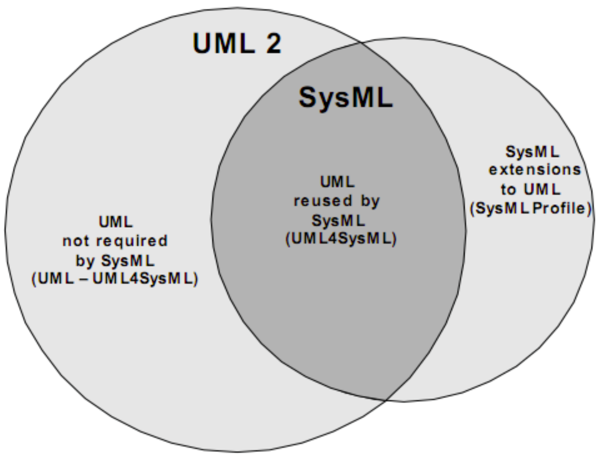

UML 2
SysML
UML not required by SysML (UML – UML4SysML)
UML reused by SysML (UML4SysML)
SysML extensions to UML (SysML Profile)

Figure 3.1: Venn diagram depicting SysML/UML relationship. Source: [SysML1.2]

According to [Rosenberg&10], the four pillars of SysML are supporting modelling the aspects of the system in terms of *requirements*, *structure*, *behaviour* and *parametrics*. The application of requirements diagrams is not directly connected with this thesis work, however the rest deserve additional consideration:

Structure: represented in SysML by two kind of diagrams: a) *block diagrams* and b) *internal block diagrams*. Block diagrams can represent hardware, software or firmware blocks. Internal block diagrams are used to illustrate the internal composition of each block. They are making use of parts, ports and connectors.

Behaviour: expressed in SysML by four kind of diagrams: a) *use cases*, b) *activity diagrams*, c) *sequence diagrams* and d) *state machines*. Note that these elements are inherited from the UML language, which only uses them in a software representation context. On the other hand, SysML make use of them in order to illustrate behaviour in heterogeneous systems.

Parametrics: enables the incorporation of concrete and detailed characteristics and constraints that the system might present. A *parametrics diagram* makes use of constraints blocks in a certain analysis context.

If the elements presented above are analysed, they could be classified in two groups:

(a) New graphical representations: Under this category the diagrams for representing requirements, structure and behaviour can be found.

(b) New representations that enables additional analysis: Parametrics makes possible to integrate analysis with graphical design models.

It can be conclude that: new graphical representations enable the application of a UML based approach in a generic context, make it possible to incorporate a wider range of details (hardware and software) and improve the communication outcome of the models; new representations that enables additional analysis, parametrics, is one of the most relevant SysML contributions, since it enables the evaluation of engineering trade-offs. As it was explained in section 2.5.1, trade-offs

are widely present in the Hardware/Software Co-design field.
Even though, SysML is not a methodology and is tool independent, tool support is relevant in the exploitation of parametrics. A second tool related issue is the *code generation* support. Some tools, like Enterprise Architect or Artisan Studio, enable the code generation in different languages. Here it is important to remark that not only software languages, but hardware description languages are considered as well. Under the appropriate tool support, a SysML model can be used to perform design space exploration, partitioning and co-synthesis of the hardware and software components. A complete case study, illustrating the application of SysML diagrams, usage of parametrics and code generation, can be found in [Rosenberg&10].

- SysML provides intuitive and expressive graphical notation to represent *structure* and *behaviour*.
- *Parametrics* enables the representation of characteristics and constraints by the use of numerical expressions, that can be analysed in constraints blocks diagrams.

3.2.2 SystemC

SystemC is an open source class library for the programming language C++ to model digital hardware. Chen et al. in [Chen&09] show that such a modelling approach is possible due to the integration of hardware constructs and a simulation kernel. [Black&04], elaborate further on the SystemC hardware modelling enabling features, which are summarized below.

Time model: which is managed by the simulation kernel. Time is implemented in kernel space, with a resolution of 64 bits. The time model provides a notion of time to the simulation, but it does not provide the possibility of instantiating clocks.

Hardware data types: that are close to the hardware representation of data. For example, integers and fixed-point representation of a user defined width and the four-state logical data type.

Module hierarchy: so it is possible to represent complex hardware/software structures. Modules can be connected to other modules in the upper, lower or at the same level using channels.

Communication: simple channels and complex communication buses can be modelled. The first type with the integrated communication constructs (for example, fifo queues, signals or ports) the latter, making use of the language in order to model them.

Concurrency model: implemented as a cooperative multi-tasking model. This means that processes are responsible to suspend themselves in order to give processor time to other process. In other words, the simulation kernel *never* initiate a context switch (a non pre-emptive model).

SystemC present a high dynamic range in terms of abstraction [Patel&04]. It is possible to apply it in order to create SAM models and refine it down to the Register Transfer Level (see [Black&04, Gajski&09, Chen&09]). Some tools support the generation of VHDL code from SystemC. This bridges the gap between the hardware and the software development world.
On the other hand, it is generally argued that the use of a programming language in a modelling

context, influences towards the incorporation of implementation details in wrong levels of abstraction. In some cases it is needed to incorporate implementation details, for example, while performing refinement. In other cases it is negative, for example while creating a model that is used as initial specification of the system. The key is, from the modeller perspective; a) to have the abstraction level in which the model under creation is supposed to be clear in mind and b) to incorporate implementation details only when they are necessary.
Finally, remark that [Wolf03] stated that applying a software modelling language to describe a system completely, might bias the design decisions towards software solutions.
The last disadvantage SystemC presents, even though it is not exhibited by the class library itself, is the overwhelming price of the simulation and modelling tools. Even though SystemC can be compiled with the open source compiler *gcc*, and executed with a free implementation of *bash* its capabilities cannot be fully exploited without the appropriate tool support. Due to its possibilities and integration with other technologies, special mention deserve the commercial tools *CoFluent Design*[1] and *Modelsim*[2]. Application of this tools in the co-design process are out of the scope of this thesis work, further details to be found in [Robert&10].

- SystemC is strong in modelling hierarchy and structure.
- The abstraction range covered by this modelling language is very high, ranging from overall system representation to Cycle Accurate Models.

3.2.3 MATLAB - Simulink

MATLAB is an environment that enables the user to perform complex mathematical calculations. It is optimized to offer better performance than ad-hoc models developed in programming languages like C, C++ or FORTRAN [Matlab]. *Simulink* is a MATLAB extension, used for dynamic systems modelling and simulations. Its functionality is organized in toolboxes depending on their application are. These areas range from physical modelling to system verification and validation [Simulink]. Code generation facilities are integrated as part of Simulink, enabling the tool to generate code in the programming languages C, C++ and VHDL among others.
Simulink makes it possible to model particular implementations at the TLM level. Furthermore, hardware blocks provided by Xilinx are already modelled in the tool. This makes it possible to create system models in the form of block diagrams and analyse the performance before having the actual hardware. Finally, the generated model in Simulink can be verified against the final implementation. The implementation can be simulated in a software environment or deployed in hardware. This link is performed by the Simulink component *EDA Simulator Link*.
The main advantage that is offered by the combination of MATLAB-Simulink, is the strong tool support in the modelling of complex systems in terms of: a) Physics and b) Mathematical algorithms. The outcome of using this modelling technology will reach its maximum only if the problem under study presents a challenge in one of those areas (or both).

[1]Cofluent official website: `http://www.cofluentdesign.com/`
[2]Modelsim official website: `http://model.com/`

- MATLAB - Simulink has a remarkable performance when it comes to model mathematical based processes. Special attention deserves its application in the signal processing field.
- There are a number of extensions that allow the application of this modelling technology in a number of fields.
- The language is not appropriate to model complex structures or concurrent problems.

3.2.4 POOSL

POOSL stands for **P**arallel **O**bject-**O**riented **S**pecification **L**anguage. This language was created at the Technical University of Eindhoven with the purpose of supporting Hardware/Software Engineering through System Level Design (SLD). [POOSL] describes its syntax as expressive and composed by a powerful set of primitives. These syntax primitives are organized around three layers:

Data layer: POOSL is an Object Oriented language, so data entities are represented by classes. Information is represented by attributes and associated operations by methods. Data classes are passive and their methods are sequential.

Process layer: provides the constructs to represent basic active classes. Active classes can be combined to represent composite processes. Process classes give the possibility of expressing *concurrent* and *active* behaviours.

Architecture layer: POOSL gives the possibility of defining architecture by the use of clusters. Clusters are composite structures that combine clusters, process classes and data classes. A cluster does not extend the behaviour of the individual elements composing it, but uses them to offer a more complex functionality. Clusters define their interfaces with external components by the use of messages. Creating clusters insides clusters gives the possibility of representing *hierarchy* in POOSL models.

There is an UML profile for the graphical representation of models created under POOSL. Table 3.1 shows the relation between the layers and the stereotypes introduced by the POOSL UML profile (see [POOSL]). In figure 3.2 the graphical representation of these profiles are shown.

Syntax layer	**Equivalent UML stereotype**
Data Layer	«data»
Process Layer	«process»
Architecture Layer	«cluster»

Table 3.1: Equivalence between POOSL constructs and UML stereotypes.

It can be conclude that: the general idea about using the POOSL language (formal representation), together with the POOSL UML profile (graphical, informal representation), is a clear application of the ideas proposed by [Bruel&98] (previously discussed at the beginning of section 3.2).

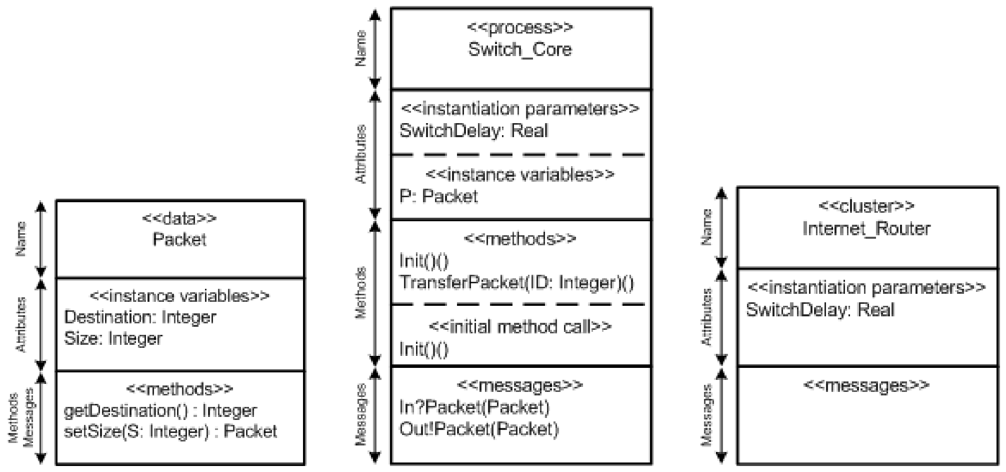


Figure 3.2: Representation of POOSL classes under POOSL UML profile. Source: [POOSL]

.

POOSL is supported by the tool *SHESim* SHESim is a graphical tool that allow the interactive composition of the system through blocks. Blocks can be detailed with attributes and methods and other elements depending on their type. An initial graphical model can be refined with the use of the actual language. SHESim can interpret the model so execution can be performed.
Code generation is partially covered by the tools. SHE integrates a formal framework for the *generation of Real-Time Control Software*. However, computation-intensive software and hardware synthesis are not currently available.
POOSL is the most relevant enabling tool of the SHE methodology, which will be described further in the text.

- POOSL presents platform independent semantics.
- A combined graphical-formal approach has been developed for its application in POOSL.
- POOSL allows the creation and execution of formal models in order to verify properties.
- The lack of current research makes it not recommended for new projects.

3.2.5 VDM-RT

VDM-RT is the **R**eal **T**ime extension for the modelling language/formal method technique VDM, the **V**iena **D**evelopment **M**ethod. The pillars of VDM rely on the the VDM-SL ISO standard [ISOVDM96short], [Plat&92]. In order to support new problems and satisfy industrial needs, the language has evolved and several dialects have been developed. The considered dialects in this work are:

VDM-SL: is the standardized base language use in the formal specification of systems.

VDM++: is the object oriented extension of the language. Concurrency support by the means of thread and synchronization predicates is provided.

VDM-RT: is the real time extension of VDM, which allows the modelling of distributed and real time systems.

Tool support exist on the market in the form of open-source and commercial solutions. Some of the available tools are:

VDM Tools: commercial tool supporting the base language VDM-SL and the dialect VDM++. It supports the code generation in the programming languages Java and C++. Reverse engineering can be applied over Java code. It is possible to generate UML diagrams from the VDM model and perform test coverage analysis [VDMTools].

Overture: open-source tool supporting the base language VDM-SL and the dialects VDM++ and VDM-RT [Larsen&10c].

One of the most relevant characteristics of the VDM-RT language is the possibility of exploring different deployment options. From the model perspective it allows to perform static system deployment. This deployment is made through the use of the CPU and BUS classes.

CPU: represents a single processing unit. VDM modelled functionality can be allocated on a CPU which will execute it. CPU operation can be configured to run under two different models First Come First Served (FCFS) or Fixed Priority (FP). Under the FCFS mode, the CPU will execute a certain number of operations per thread before it is swapped out. On the other hand, under the FP mode, the CPU will execute threads depending on its execution priority . Besides the operating policy, the frequency at which the CPU is running has to be set.

BUS: Buses serve as communication channels between CPUs. Each bus has to be configured in terms of transmission policy, transmission speed and connected CPUs. The available transmission policies at the time of writing are FCFS and Carrier Sense Multiple Access/Collision Detection (CSMACD). In an static deployment context, buses are configured before the model execution start, and remain in the specified topology until the end of the model execution.

Additional work has been carried out in the field of dynamic reconfiguration of VDM models [Nielsen10a]. The application of this technique would allow the modelling, simulation and analysis of changing network topologies in a VDM setting. Such an approach would be worthwhile considering in the modelling of reconfigurable computing systems, which is a hot research topic with potential applications in the FPGA technology. However, this field is out of the scope of this thesis.
Additional work has been carried out in the application of these dialects in concrete industrial cases. Special consideration deserves the application of VDM-RT in modelling and validation of distributed embedded real-time control systems [Verhoef05, Verhoef08].
For additional information on the VDM modelling language in general and on VDM-RT in particular, refer to [Fitzgerald&05] and [Larsen&10b] respectively.

- VDM allows the creation and execution of formal models in order to verify system properties.
- VDM presents platform independent semantics in all its flavours.

	SysML	SystemC	Simulink	POOSL	VDM-RT
Hierarchy and modularity	++	+++	++	+++	+++
RTOS support			+	++	++
State transition	+++	++	+	++	++
Concurrency and Synchronization	+	+++	+	+++	+++
Timing	+	+	++	++	+++
Constructs to support formal verification		+		+++	++
Team communication possibilities	+++	++	+ +	++	+

Table 3.2: Modelling language comparison

- Strong in representing hierarchy, structure and behaviour in concurrent and real time scenarios.
- Growing tool support and used in on-going research projects.

3.2.6 Language comparison

Table 3.2 presents a comparison between the presented modelling languages and the different desirable features described in section 2.4. The languages have been assessed with a scale from zero to three "+". Zero "+" means that the discussed feature is not supported, three "+" represents a good support of the considered feature.

In general "Hierarchy & modularity" are properly supported in most modelling languages. The support for "RTOS and timing" analysis is weaker in modelling languages based in graphical notations. Executable languages are stronger in representing Concurrent and timing aspects. Finally, graphical languages are stronger in supporting communication across team members. It could be conclude that a single modelling language cannot cover all the aspects that require representation and analysis in a Hardware/Software project. A modelling approach in which several modelling languages are used in order to study the aspects in which they are stronger is the most promising avenue.

3.3. Time and modelling languages

This section aims to explain how time is represented in different modelling languages and how time information is affecting system simulation and modelling. The notions of time introduced in here can be used in order to make a logical system working on discrete basis, react to a continuous phenomenon like time. Such a system is considered as *Real-Time System* [Burns&09].

VHDL and SystemC make use of a notion of time that can reach the femtosecond level [Perry02, Black&04]. Such a precise notion of time is needed because both languages are used to describe hardware/software systems at the implementation level.

No notion of time is incorporated in the modelling languages VDM-SL and VDM++. In the case the modeller wants to incorporate a representation of time in models created in these languages, it will have to be modelled explicitly. A different approach is presented in the real time version of

the language VDM-RT. The VDM-RT kernel incorporates the notion of time by using *time-ticks*. Each time-tick corresponds to the physical notion of time 1 nanosecond.
Additional explanation deserves the real-time mechanisms incorporated in VDM-RT. Time is considered by new types of threads, and the expressions duration and cycles.
Two different types of threads are available in VDM-RT:

procedural: threads, statements executed until completion. They are subject to scheduling and descheduling.

periodic: threads, offer the possibility of repeated execution of the instructions contained on it. The execution frequency together with additional parameters can be set up at declaration time.

Periodic threads can be precisely adjusted to model the real-time problem under study. These threads can be tuned by the use of the constructs **`period`**, **`jitter`**, **`delay`** and **`offset`**. This work makes use of the **`period`** construct, which express the time interval between successive executions of the periodic thread. The inverse of the period is equal to the threads execution frequency.
The VDM-RT language provide constructs to use the notion of time on the created models.

duration: specifies that a certain amount of time units are going to be used in order to execute the accompanied statement. This duration is *absolute* and *platform-independent*. This implies that no matter the speed of the processing unit, the instruction is going to take x time ticks to be executed.

cycles: specifies that a *relative* amount of time units are going to be used in order to execute the accompanied statement. The instruction cycles expresses the number of CPU clock cycles required to complete execution. This time information is *platform-dependant*, total time required for complete model execution will depend on CPU frequency.

VDM-RT simulation is time-driven. The VDM-RT interpreter integrated in Overture is implementing threading under a *pre-emptive* paradigm. The scheduler gives a certain amount of time so each thread can perform computations during this time slice. The duration of this time step is determined by the minimum duration presented by one of threads considered in the simulation. In figure 3.3 it can be seen how the time step duration is determined for a concrete case. In this scenario two CPUs running two different threads are modelled. Since thread 1 running in CPU 1 is taking 2 time units to be executed, that will be the size of the minimum time step. Besides the scheduling time slices considered above, different policies can be used. These policies are established with the constructs explain in the beginning of this section.

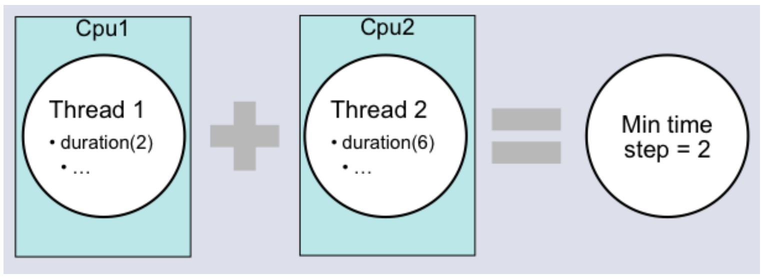


Figure 3.3: Threading and step size in VDM. Source: [Lausdahl&10b]

In figure 3.4, the resource scheduling process together with the thread policy section is shown. As it is shown there is a central component named *resource scheduler*, responsible for pointing all the system CPUs at each time step. Each time a CPU is pointed by the resource scheduler, it is pre-empted for execution. Once a CPU is selected a computation time equals to the time step is given to it. During that computation time, a certain policy previously defined for the thread will be applied. In the case the duration of the thread reach an equal value to the assigned time step, the scheduler will swap out the thread and assign computation time to the next CPU. It is possible to make such a simulation because there is no drift in the notion of time of each CPU. In other words, all the CPUs used in the model are synchronized.

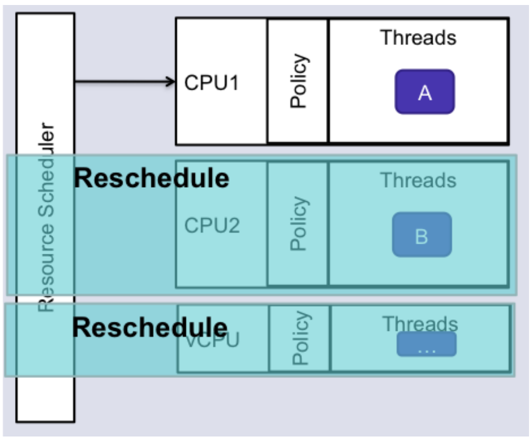


Figure 3.4: Resource scheduling. Source: [Lausdahl&10b]

SystemC and VHDL can represent and conduct analysis at the femtosecond level where as VDM-RT is operating at the nanosecond level. This implies that there is a precision gap of 6 orders of magnitude between the modelling technology VDM-RT and SystemC and VDHL languages.

3.4. Methodologies

The languages introduced in section 3.2, cannot produce results if they are applied in an unsystematic way or isolated. Therefore, they are used within a methodology or development process. There are two ways of using models in the development process: a) use ad-hoc models in order to illustrate and analyse particular aspects of the system and b) use models as a guide in the process of model-driven engineering approach. An example of the first case: the application of MATLAB models in order to develop the signal processing component of a system. The rest of the system is not modelled in MATLAB and it is developed using a different approach. The second approach, model-driven engineering, is applied in all the projects in which a Hardware/Software Co-design methodology is used. Since the latter kind is the core part of this thesis, this section will be primarily focused on this approach.

3.4.1 Xilinx methodology

The non-modelling based methodology introduced in this section is proposed by the tool and FPGA vendor Xilinx. It must be remarked that, this is a tool oriented methodology, and it is not forcing or pursuing the application of any SLD methodology like Hardware/Software Co-design. Therefore, they are mere guidelines on how their technologies and development environments have to be used. This methodology is located at the implementation level, while the rest presented in this section start at a higher level of abstraction, considering the system as a whole.
Figure 3.5 shows the tool flow recommended by Xilinx. This flow is used in the creation of the hardware/software system. As it is depicted, the design starts with hardware. Once the hardware platform is defined, a BSP containing the hardware drivers is geneterated, so software developers can create the software logic.
It is important to consider this methodology in the appropriate project stage. Considering a project in which Hardware/Software Co-design is applied, several project phases have to be accomplished before using Xilinx methodologies described in [XilinxToolFlow11, Speedway11EDK]. If they are applied before, the development team might face the risk of starting the implementation phase too early, without creating a detailed enough specification or without evaluating enough possible architectures.
The purpose of this section is to show that tool vendors might propose ways of implementing systems, which are highly biased by the tools they are selling. Pragmatism and analysis of vendors methodologies have to be done before incorporating them in the whole development process.

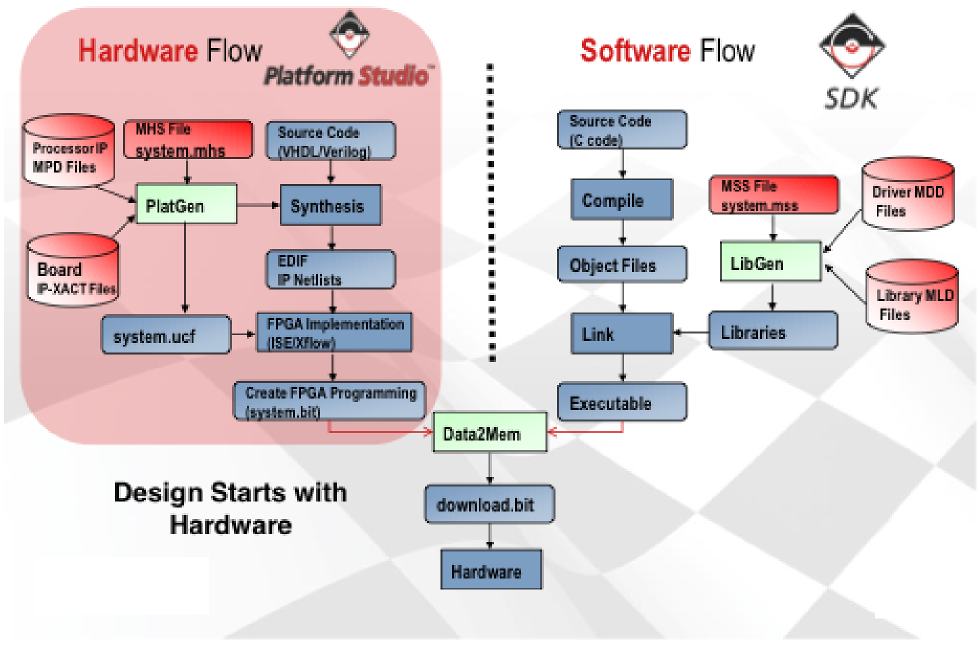


Figure 3.5: Xilinx Microblaze development tool-flow. Source: [Speedway11EDK]

3.4.2 A MATLAB - Simulink based methodology

The official MATLAB methodology is considering primarily the verification of hardware/software designs. A brief summary of the verification approaches supported by MATLAB - Simulink is provided in the appendix E. Since this thesis is focused in the partitioning process, these verification strategies are not relevant. There is research work using MATLAB/Simulink as a tool for design space exploration. [Ou&05] is proposing the application of MATLAB in order to evaluate different partitionings through the use of model based simulation. This approach is aiming to create high-level abstractions in order to represent processors, buses and dedicated hardware blocks. The equivalence between models and actual implementations is shown in figure 3.6. [Ou&05] is considering these models as high-level abstraction representations, while still they are defined as cycle-accurate. Some authors do not consider, cycle-accurate models to be at a high abstraction level. The explanation behind is that, according to Ou, the simulation will consider the number of actual clock cycles used to perform each computation. Instead of cycle-accurate, the model could be more precisely considered as *cycle aware*. Since not all the cycle related information is taken in consideration, the simulation performance regarding time is improved in orders of magnitude.

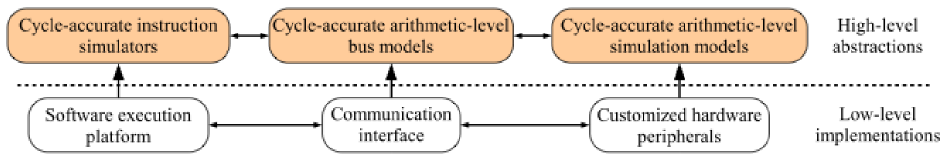

Cycle-accurate instruction simulators
Cycle-accurate arithmetic-level bus models
Cycle-accurate arithmetic-level simulation models
High-level abstractions
Software execution platform
Communication interface
Customized hardware peripherals
Low-level implementations

Figure 3.6: Proposed approach by [Ou&05]

Note that this modelling approach is not tackling the energy performance of the simulated solutions. According to [Ou&05], additional research has to be done in order to incorporate consumption considerations. One of the proposed ways to model and analyse this issue is by adding an instruction-level energy estimation. Energy estimations require a more detailed modelling of the final implementation [Ou&04] and are out of the scope of this thesis work.

3.4.3 A combined SysML - SystemC based methodology

This approach is proposed by the EU-FP7 project SATURN. SATURN (**S**ysML b**A**sed modeling, architec**TU**re explo**R**ation, simulation and sy**N**thesis for complex embedded systems). An overall description of the phases proposed by SATURN is presented in [Mueller&10]. The design process starts with modelling the requirements by using SysML *Requirements diagrams*. Structure and behaviour can be described by the use of the correspondent diagrams presented in section 3.2.1. The SATURN modelling frontend, based on *Artisan Studio*, provides tool support to generate SystemC and C code from this graphical description. Generated results can be co-simulated a SystemC simulator is supporting the hardware simulation and QEMU is supporting the software simulation in a virtual machine. Further analysis can be performed with MATLAB/Simulink. Once the co-simulation of different candidate architectures has been performed, and a final one has been chosen. Hardware can be synthesized and software loaded in the FPGA CPU. The infrastructure supporting this process is shown in figure 3.7.
One of the relevant proposals introduced in the SATURN project, shown in [Mueller&10], are the equivalences between the SystemC language and the graphical notation based on SysML. Similar considerations were presented in [Mischkalla&10, Prevostini&07]. In figure 3.8 the applied equivalences in this case are shown.
There is not much documentation about the SATURN project because it is still under development.

3.4.4 The SHE methodology

SHE stands for **S**oftware **H**ardware **E**ngineering methodology. This methodology is using the POOSL modelling language introduced in section 3.2.

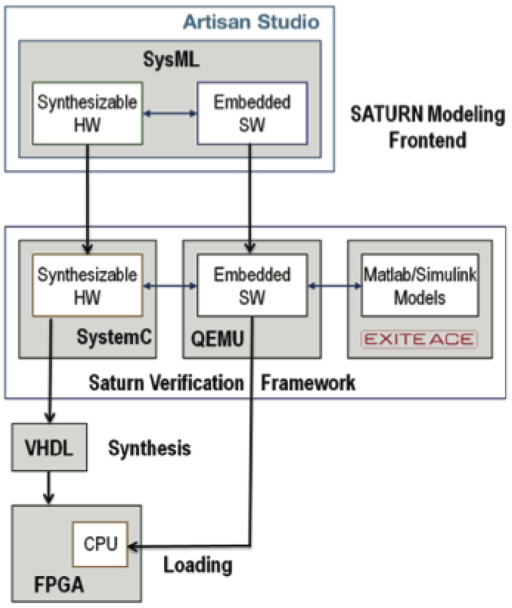


Figure 3.7: SATURN SysML based methodology. Source: [Mueller&10]

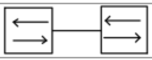
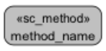
SystemC	Stereotype	Metaclass	Notation
Module	<<sc_module>>	Class	
Interface	<<sc_interface>>	Interface	
Port	<<sc_port>>	Port	
Primitive Port	<<sc_in>> <<sc_out>>	Port	
Signal	<<sc_signal>>	Property, Connector	
Process	<<sc_method>> / <<sc_thread>>	Action	
Main	<<sc_main>>	Operation	None
Clock	<<sc_clock>>	Class	

Figure 3.8: Proposed equivalences between SystemC and SysML. Source: [Mueller&10]

This methodology starts with the requirements stated in plain English. The requirements specification output is composed by several documents and artefacts. These elements are informal and its structure might vary. This depends on the requirements engineering techniques applied in each case. Requirements elicitation process and formulation techniques are not considered by the SHE methodology.
The core issues that might influence the design are extracted from these documents, and an *Essential Specification* is produced. This specification presents essential behaviour and essential structure. As a result of this phase a conceptual solution is obtained. Upon this basic specification *refinement* is performed in order to create an *Extended Specification*. The extended specification phase incorporates details enough to perform design decisions and evaluate engineering trade-

offs. Even though it is not shown in figure 3.9 and according to [POOSL], the proposed solution is validated against the initial requirements. In the case this validation, or some part of it, is not successful, it is possible to go back to the *Extended Specification* and refactor part of it. In the case more radical changes are required it is possible to go back to the initial *Essential Specification*. Once the partitioning is definitive and supported by the model analysis, the implementation takes place.
Implementation is partially supported by the tool SHEsim. Implementation of real time control software is possible, but more complex software or hardware VDHL code generation is not supported.
The SHE methodology and the modelling language POOSL has been applied in a number of industrial cases provided by leading companies in the technology sector (Siemens, Alcatel, IBM and Lucent among others). Despite industrial applications and a considerable list of publications, it is not clear from [POOSL] if this research line is active (last related publication in year 2007 and last industrial case in year 2005).

3.4.5 A VDM-RT based methodology

This methodology relies heavily on the application of VDM models in order to support all the project phases and activities. Due to its integration with the project lifecycle, its application is shown embedded in a V-model. The following approach is proposed by [Larsen&10a].

The application starts with the system specification and requirements capture, supported by the use of use-case diagrams (UML, SysML) and VDM-Specification Language. The result of these two models are reflected in a VDM++ sequential model, that helps on modelling a sequential system description. The principal outcomes of such a model is the hierarchical characterization of the system, introduction of the causal ordering of events and an abstract approximate notion of time.
A concurrent model is derived in order to show the concurrent aspects of the system. *Threads* and *synchronization* predicates are incorporated to the concurrent VDM++ model.
Once basic structure and concurrent behaviour has been specified, the modeller can create a *real-time* model of the system by using VDM-RT. The accomplishment of real-time deadlines can be studied at this point. As an additional step, *distributed real-time* models can be evaluated. The models explained above, have evolved from an initial specification to a distributed real-time model (finally refined to the implementation). This flow of information is shown in figure 3.10 by green arrows. In the horizontal plane, purple arrows are representing the acceptance activities that are performed at each level, and how the models developed previously can act as a reference for validation.

The application of the VDM technology is under continuous research. Some of the on-going EU FP-7 projects DESTECS and COMPASS, are extending the language and the tool support in order to cover a wider area. Both projects are relevant in the hardware/software co-design area. DESTECS allows the co-simulation of physical systems. COMPASS is focused on formalising the interaction between systems, that together conform Systems-of-Systems. Note that physical interaction with the environment and logical interaction with other systems, are factors that might have an impact in partitioning decisions. Further research on the combination of COMPASS and DESTECS and its application in hardware/software co-design still has to be done. However, that study is out of the scope of this work.

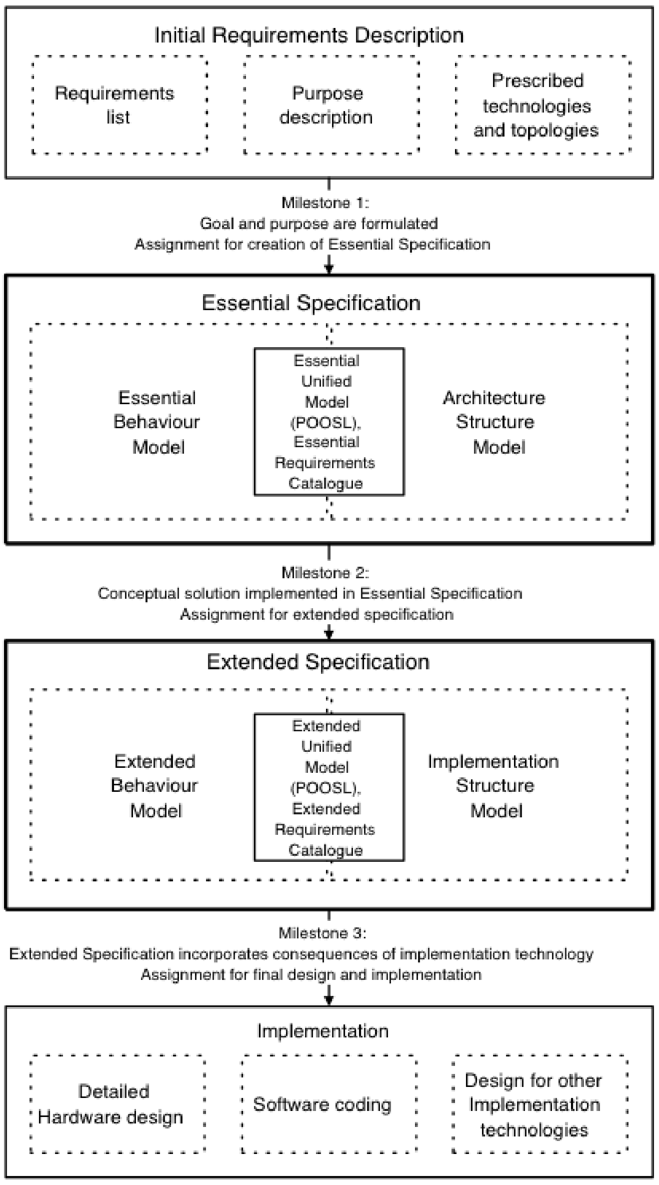

Initial Requirements Description
Requirements list
Purpose description
Prescribed technologies and topologies
Milestone 1:
Goal and purpose are formulated
Assignment for creation of Essential Specification
Essential Specification
Essential Behaviour Model
Essential Unified Model (POOSL), Essential Requirements Catalogue
Architecture Structure Model
Milestone 2:
Conceptual solution implemented in Essential Specification
Assignment for extended specification
Extended Specification
Extended Behaviour Model
Extended Unified Model (POOSL), Extended Requirements Catalogue
Implementation Structure Model
Milestone 3:
Extended Specification incorporates consequences of implementation technology
Assignment for final design and implementation
Implementation
Detailed Hardware design
Software coding
Design for other Implementation technologies

Figure 3.9: Overview of the SHE engineering design methodology. Source: [vanderPutten&07]

3.5. Summary

This chapter has presented several modelling languages with very different approaches to system modelling. Graphical languages like SysML and UML and informal languages like SystemC and Matlab/Simulink have been reviewed. Formal languages like POOSL and VDM have been presented. Different methodologies associated to the reviewed modelling languages have been described. This chapter has shown the need of new modelling languages and/or methodologies that allow the system engineer to put aside implementation considerations and focus on the problem under study at a high level of abstraction.

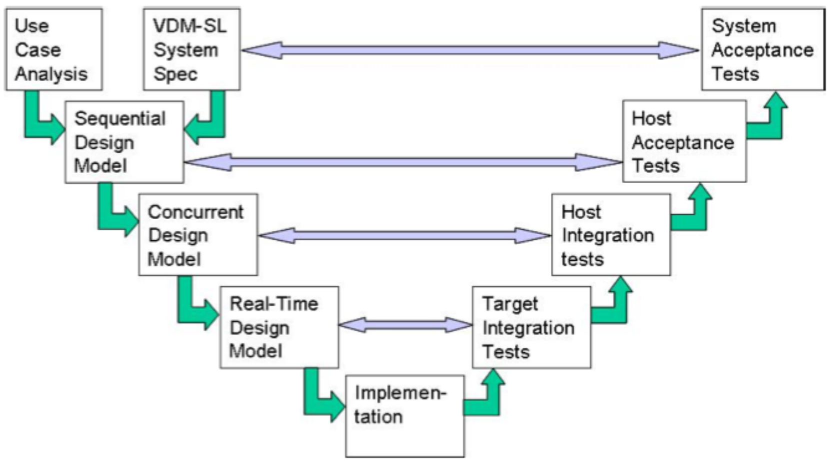


Figure 3.10: VDM based methodology. Source: [Larsen&10a]

Chapter 4

A VDM-RT based modelling methodology for Hardware/Software engineering

This chapter is presenting the proposed methodology for the hardware/software co-design of embedded systems, the core part of the thesis. Section 4.1 gives an overview of the rationale and aim of the methodology presented. Section 4.2 describes the proposed methodology. This description is built on top of the existing VDM-RT development process [Larsen&09] and makes special emphasis on how it can be applied in order to take Hardware/Software co-design partitioning decisions. The methodology is complemented by the early model evaluation against a prototype, which is described in section 4.3. Proposed improvements to the VDM technology are presented in section 4.4. Section 4.5 describes how new platforms can be incorporated to VDM-RT. Finally, the chapter is summarized in section 4.6.

4.1. Introduction

This chapter presents the VDM-RT modelling methodology proposed in this thesis work in order to provide support to the Hardware/Software Co-design of embedded systems. The goal is to provide a way to tackle the fundamental problems in hardware/software co-design described in chapter 2. The only avenue to accomplish this objective is to apply a systematic methodology, based on the ones presented in chapter 3. The intent of the proposed VDM-RT methodology is to provide the system engineer with a tool to describe and analyse Hardware/Software systems from a formal perspective, that will allow him to perform design space exploration earlier in the project life-cycle. A number of guidelines based on the approach explained in the chapter will be given. The guidelines give modelling advice according to the hardware/software co-design process proposed in this chapter.

4.2. Methodology description

The presented methodology is structured in three modelling stages and one exploration phase. The methodology starts with the *early* production of sequential VDM models, that will start being far away from the final one but will help on getting familiarized with the problem under study. Finally, a sequential VDM model with enough details to take partitioning decisions later on in the process will be created. This candidate model will be moved to a concurrent VDM model, with the purpose of focusing on the concurrent behaviour of the system. A real-time VDM model will be derived from the concurrent model, substituting the very limited notion of time for a more precise one. In the exploration phase, the real-time VDM model will be evaluated as a distributed real-time model. It is here where VDM-RT possibilities for supporting hardware/software co-design decisions have to be studied more in detail. Note that this is a "waterfall-like" process, in which errors introduced in the sequential system model can have a very negative impact in the rest of the modelling stages and exploration phases.

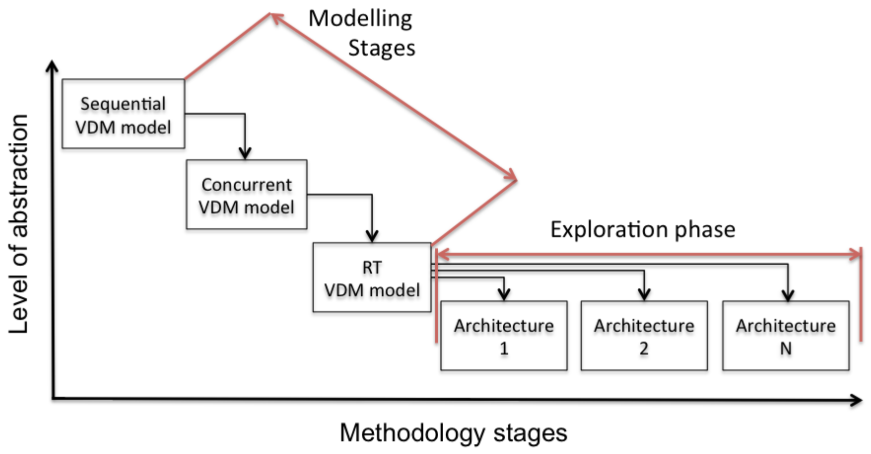


Figure 4.1: Methodology overview.

4.2.1 Creation of a sequential model of the system

The first step of the methodology consist on the creation of a sequential VDM model. As the name says, this model is representing the system operation in a sequential model, modelling the operation execution in a sequence of steps. The notion of time incorporated in this model is very limited, and can only be used to determine whether an event is happening before or after another, considering the system deployed in the same computation unit.
The purpose of such a model is to represent the most relevant entities involved in system operation. The relevant term refer to entities that fall directly under the following categories:

Model entities: represent elements that are a straightforward logical representation of an existing entity in the physical world that has to be processed by the system. For example, a clock class that represents a device to measure time in a model of a time synchronization system (more details about this on a concrete example in chapter 6).

Entities with core behaviours: that represent crucial processes in the system operation. An example of this kind of entity would be an entity responsible for the generation of a Pulse Width Modulated signal controlling a motor (more details about this on a concrete example chapter 5).

System boundaries: that represent the limits of the system. Boundary entities define how and where the system is interacting with the environment.

At this point of time, a "raw" sequential model has been created. Taking advantage of the Object Oriented (OO) techniques, the model can be expressed by a UML class diagram. OO techniques from the software engineering world can be applied, for example design patterns. Once the structure of the system has been created, operational details, used data and system properties can be incorporated.

Guideline 1: The sequential VDM model gives the possibility of focusing on the key hardware/software entities to be considered for partitioning. It is a pre-condition to reach a good understanding of the system before moving to more complex modelling.

Guideline 2: In the case the problem under study is new to the modeller, it is difficult to create a useful model in the first attempt. In this case, the creation of the model should be iterative, and this process should be used as a tool to help on system understanding.

4.2.2 Creation of a concurrent model of the system

The concurrent VDM model of the system is an intermediate step between the sequential VDM model and the real-time VDM model. In this model, the entities with core behaviours most likely will be active classes, with threads running the key operations identified in the sequential model. Additional effort has to be spent on ensuring a safe concurrent access to shared data. The notion of time will remain in discrete timesteps, as defined in the firsts sequential models. In order to make the threads advance progressively, the `TimeStamp` pattern presented in [Lausdahl&10b] can be used.

Guideline 3: It is important to make sure that *all* the synchronization issues are tackled during the concurrent modelling phase. If not, problems may arise during the RT modelling phase. Fixing synchronization problems in the RT modelling phase is altering the focus of that phase and it might be more complex.

4.2.3 Creation of a distributed real-time model

The distributed real-time VDM model introduces three major changes: the introduction of a new notion of time, the introduction of a RTOS layer and the possibility of deploying different model entities in different computation units. This third modelling stage allows to get the models closer to real systems performance.

Guideline 4: The first step in the process of moving to the real-time world is the creation of a non-distributed real-time model. This model is an intermediate step between the concurrent and the distributed real-time model. In this case all the objects will be deployed in the same CPU.

The notion of time introduced by the `TimeStamp` class should be removed and the Overture time reference should be used instead. In the case there were threads using `sleep` calls based on `TimeStamp`, they should be refactored to use the information provided by the **`time`** expression. In the case this sleep period was constant in time, the thread could be modelled as periodic.

4.2.4 Evaluating hardware partitions from VDM-RT models

This methodology proposes the application of VDM-RT CPUs in order to model a hardware partition. It is possible to create a *coarse-grained* model of the hardware partition if this process is carefully applied. The key principle behind it is that there should be a difference of orders of magnitude in the processing speed between CPUs emulating a general-purpose CPU and a CPU emulating an specialized hardware block (hardware partition).
The communication between the CPU emulating the hardware block and the CPU emulating the general-purpose CPU will be modelled as a BUS with a very high bandwidth, orders of magnitude above the bandwidth used in order to represent a pure software-based communication. The reason behind using a high bandwidth is to simulate how a hardware block is using a register mapped scheme for interaction with the control software. This process can be repeated with different alternative system definitions easily. The Overture log files are a powerful tool in order to study the real-time evolution of the system. More details will be provided in the case studies (chapters 6 and 7).

Guideline 5: A VDM-RT CPU that represents a hardware partition should be configured with a processing speed orders of magnitude higher than general-purpose CPUs.

Guideline 6: A VDM-RT BUS that communicates a hardware partition with a general-purpose CPU should be configured with a very high bandwidth. This bandwidth should be orders of magnitude higher than the ones used to communicate general-purpose CPUs running software.

Guideline 7: The instruction **`time`** will give the value of the notion of time at invocation time. It is a good idea to use it several times during the critical operations under study are being run. This will give an overall idea of the time spent by different operations. Custom logs can be generated, with the advantage of being able to get the relevant data faster than by comprehending the Overture logs.

4.2.4.1 Evaluation of a single hardware partition

The study of a system that is making use of a single specialized hardware partition is a straightforward application of the idea presented above. The steps could be summarized as follows:

1. Identify the component that is going to be executed in hardware.
2. Define a separate CPU configured with a processing speed orders of a magnitude above the speed used in the general-purpose CPUs.
3. Communicate the defined CPU through a bus with a very high bandwidth with the general-purpose CPU. The general-purpose CPU is running the rest of the entities as software components.
4. Deploy the functionality that should be running in hardware in the CPU defined in step 2.
5. Analyse the possible improvement in the system performance.

Guideline 8: In the case the communication to be modelled is not as fast as the one that could be achieved by a registered mapped configuration, the bus bandwidth can be lowered.

4.2.4.2 Evaluation of several hardware partitions

In some situations it is desirable to consider how the system would be performing in the case *several components* are deployed in hardware. Considering the process described above, the most intuitive approach to model such a scenario would be to define a second CPU and deploy the hardware candidates there. However, this process turns to be erroneous and can lead to false results. A more concrete example is presented in order to explain this challenge.

Consider a system that is modelled by three components: A, B and C. Components B and C depend directly from component A, that needs to be communicated with them. Components B and C do not present any kind of interaction between them. The system designer is interested on evaluating the performance of the system when A is running in a general-purpose CPU and B and C are off-loaded to specialized hardware blocks. In the case the process explained above is applied, the system would be deployed as illustrated in figure 4.2.

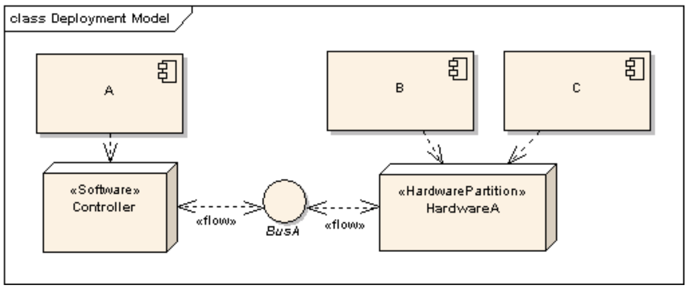


Figure 4.2: Wrong usage of the hardware partition.

If such a deployment is performed, the specialized hardware block would not be acting as such. In this scenario the CPU representing the specialized hardware block will be behaving as a hi-speed CPU with an OS running two threads. Naturally, such a model could possibly be performing better than a purely software based architecture. These improvements are still not reflecting how a hardware off-loading for components B and C would be performing.

The correct way of performing the hardware off-loading evaluation is to make a deployment like the one presented in figure 4.3. As it can be seen, two CPUs (HardwareA and HardwareB) representing specialized hardware partitions have been introduced. Both CPUs are communicated by hi-speed BUSes to the general-purpose CPU (Controller). Component A is deployed in Controller, components B and C are deployed in HardwareA and HardwareB respectively.

In the case the components B and C would be interacting, a hi-speed bus should be connecting the CPUs in which they are deployed, in this case HardwareA and HardwareB.

Guideline 9: Each hardware candidate should be deployed on an individual CPU configured as a hardware block.

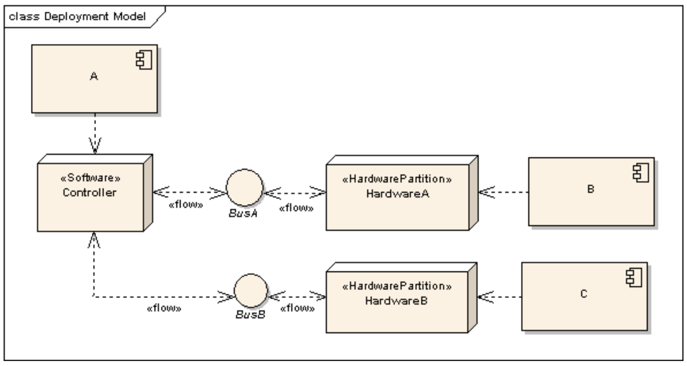


Figure 4.3: Correct usage of the hardware partition.

4.2.4.3 Modelling complex communication between partitions

The BUS class provided by Overture allows the modelling of a bus under the policy *First Come First Served* **`<FCFS>`**. In some cases it might be interested to make use of other policies. Two possible situations in which this would make sense could be:

1. A system in which not all the information sources have the same priority when it comes to access the transmission medium.
2. A system with different kind of policies depending on if the information is sent or received.

Since these cases are too particular, it is understandable that are not incorporated by default in the language. However, both scenarios can be easily modelled in VDM-RT controlling how the component is accessing the bus. The idea is to make use of a `Proxy` class that implements the desired policy. Figure 4.4 illustrates how this pattern could be applied in the second example presented above. In this case the BUS is communicating using CPU1 and CPU2. The BUS is accessed through a `Proxy` class. This class specifies that a policy has to be applied over the data. However this policy is abstract and has to be implemented in derived classes. `IncomingData` and `OutgoingData` are specific implementations of `Proxy`, that will implement the required policy depending on if the information is received or sent. The interaction with the `Proxy` class within the component is straightforward, in this case there is a controller class `Logic` making use of it.

Guideline 10: Use a proxy class in order to incorporate a more complex behaviour in the transmission/reception of data. As an additional advantage, Proxy classes separate access to the physical communication medium from the system logic, which leads to a lean model structure.

4.2.4.4 Modelling external measuring hardware

During the development of hardware systems it is very common to use external equipment like oscilloscopes or logic analysers. The purpose of such equipment is to monitor the value of a signal over time. There are no classes in VDM-RT implementing oscilloscopes or logic analysers, but certainly some log facilities can be built into the models to provide a similar function. The process to log data is based on a class that obtains the values of the target variable over the model execution time. Once model execution is finished, data is dumped to a file and ploted using a third

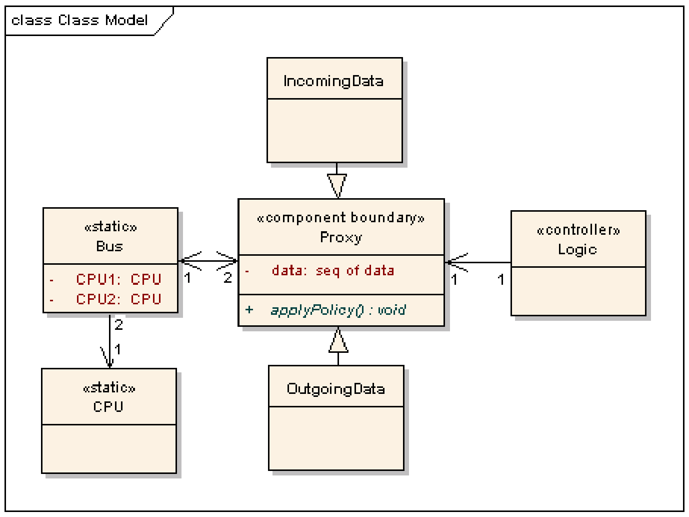


Figure 4.4: Application of a proxy class

party application. The critical part of the logging process is how the data is obtained. Logging implies introducing additional logic in the model. The execution of this logic might alter time related results.
The most simple way to avoid altering time results is to deploy the logging functionality in the virtual CPU and force the class containing the variable to be monitored to store its values periodically in the log class by using a `pushValue` operation. This operation should be defined as a an **`async`** log class method and its invocation should preceded by a **`duration`**`(0)`.

Guideline 11: Using graphics is a fast way of evaluating the I/O signals that are fed/generated by a certain architecture. A second advantage is that models generating a visual output are better at communicating modelling results to development team members. Finally, keeping together models and generated plots are a good way to document the designs.

Guideline 12: In general, log operations must be preceded by **`duration`** `(0)`, otherwise the time spent on logging will affect model execution results.

4.3. Early prototype generation and model validation

Taking one step further in the proposed process, a preliminary prototype implementing some of the proposed functionality could be produced. The value of generating a prototype with regard to the models created in the previous stages, is that it is possible to compare the model predictions with a preliminary implementation. In the case the models have been created correctly, resemblance between the created model and the preliminary implementations is expected, at least in the conceptual model. Some FPGA manufacturers provide tool support in order to profile the produced implementations. Combining the prototype performance, measured by dedicated test equipment (scopes, logic analysers) and assessed with a previously defined criteria with profiling information, can be a way to obtain feedback from a real implementation. In the case the events/behaviour/performance problems predicted in the models are present in the preliminary hardware implementation, confidence in the generated designs will be supported by well founded design decisions. In the case the obtained results are differing in aspects like time units, while still keeping the expected results, scale factors can be introduced in the models. Finally, if the obtained results are not related to the model predictions, there is a problem in the model, in the prototype, in the measurements or in a combination of all these elements. The solution to such issues depends on the particular case.

Guideline 13: Early feedback incorporation improves the models and bridges the gap between abstract representations and real implementations. In order to exploit the application of models in the development process, it is important to keep aligned models and produced/generated components.

4.4. Proposed improvements to the VDM technology

Since the Overture platform is in continuous development, and this work is carried out in the same institution in which this Master's thesis work is presented, it is interesting to discuss possible additions that could be done in order to improve the tool. Most of these additions/new features are directly connected with the hardware/software co-design field. Others would be beneficial in general, and would increase the value of the tool.

Models of new platforms: It would be desirable to have new specialized hardware blocks ready to use in the form of a VDM-RT library. These blocks could be modelling specialized processors like DSPs, microcontrollers or bus interfaces. A certain set of benchmarks could be run over different platforms and its results incorporated as concrete CPUs able to host components. The benchmarks could evaluate the time it takes to perform certain mathematical operations, communication through built-in buses, etc... Results could be mapped to concrete VDM-RT CPUs. Performance evaluation over these platforms could be more reliable and closer to reality.

Model profiler: Some of the coverage information generated by Overture could be extended so it incorporates the time it has taken to execute each operation. This information can be already extracted from the generated overture log files. However, it is something that has to be done manually for each operation. The possibility of generating such a model profile would simplify the analysis activities, by making the information easier to be read and found.

Improved bus support: The current bus configuration is limited to the First Come First Served policy. A general-purpose bus with a transmission/reception logic that could be user defined would be very interesting from the system design perspective. Such a field could contain the logic to filter packets, introduce noise, establish priorities or cause delays.

Automatic variable monitoring: In some cases it is interesting to monitor the evolution of a variable over time. Overture could incorporate the possibility of automatic plotting of certain variables selected by the user. This new feature is currently under development.

Code generation: Such a feature would bridge the generated models during the design phase with the implementation activities. In the case full code generation is not possible it would be interesting to have at least the stub of the application. Even though this would not be a considerable leap forward regarding implementation time, it establish a relationship between models and code. This relation can be backtracked and models can act as a reference for the developers.

Annotations for Real-Time deadline evaluation: A strong point of VDM-RT is the possibility of real-time deadline accomplishment evaluation. In the case a language could be used in order to define the real-time languages, Overture could automatically generate a report after model execution presenting which deadlines have been missed and when. This idea has been evaluated in [Ribeiro&11] and it is currently under development.

4.5. Modelling new platforms under VDM-RT

Some of the proposed improvements to the VDM technology like "Improved bus support" or "Models of new platforms" would require to study the aspects that characterize real BUSes and CPUs, which aspects are relevant for incorporation and which additions to the VDM language would be needed. These additions could be implemented as new constructs or as annotations. In this section some of the required additions to model new CPUs are considered.
Incorporating models of new platforms could be carried out by subclassing the VDM-RT **CPU** class. A microcontroller is an interesting platform in the hardware/software development area. In this case, a worthwhile considering feature would be the addition of interrupts. This would imply that the microcontroller CPU constructor would need to register an interrupt, a microcontroller class method would have to associate this interrupt with a triggering event and, finally, a microcontroller class method would have to associate the interrupt with an operation defined in the logic running in the microcontroller. Finally, the Microcontroller class constructor would have to skip the scheduling policy. This detailed example is shown in the VDM listing below.

```
instance variables
  int1   : Interrupt := new Interrupt(priority);
  event1 : Event     := new Event(Timer`onTimerOverflow);
  mcu    : Mic       := new Mic(speed, int1);

  public static timer      : Timer     := new Timer(10,event1);
  public static counter    : Counter   := new Counter(timer);
  public static ledControl : LEDdriver := new LEDdriver(counter);

operations
public System : () ==> System
System () ==
(
  mcu.event(int1, event1);
  mcu.setOp(int1,LEDdriver`turnOnLED);

  mcu.deploy(ledController1);
  mcu.deploy(counter);
  mcu.deploy(timer);
);
```

Considering as a second example a Digital Signal Processor[1], some of the operations deployed on it are executed with the same performance if compared to a general-purpose CPU. However, other mathematically complex operations run orders of magnitude faster under DSPs. Since a DSP is a specialized CPU, subclassing could be used in order to inherit the common features with CPUs. The use of annotations would allow to characterize operations and functions in VDM as DSP efficient without having to alter the description of their logic. This example is shown in the VDM listing below.

```
-- DSPefficient
private fft: () ==> ()
fft() =
  is not yet specified
post doneFlag = true and overflow = false;
```

4.6. Summary

This chapter has presented a VDM-RT based methodology for the Hardware/Software co-design of embedded systems. An overall view of the methodology has been provided and special emphasis has been made on the proposed additions to support the partitioning decisions. A set of guidelines have been presented.

As it has been argued above, there is no perfect methodology in order to design systems. It is the belief of the author that the proposed methodology compensate several shortcomings of existing ones, such as the high degree of specialization of Matlab based workflows, the complex setup of the SATURN methodology or the lack of planning and analysis in a Xilinx based approach. Comparisons aside, *pragmatism* should be present when selecting the modelling technology.

The following chapters 6 and 7 will present two case studies in which the VDM-RT hardware/-software co-design methodology is used. These case studies will give a practical insight on the methodology.

[1]Specialized processor with a high performance in heavy mathematical computations (e.g. computer vision algorithms, digital filtering algorithms)

Chapter 5

Introductory case study: RC servo control

This chapter applies the methodology presented in chapter 4 in a concrete case study. This case study ties together all the previous information presented so far. A model-driven approach will be applied in this problem, and it will be shown how implementation details are abstracted, as described in chapters 2 and 3. This chapter shows the importance of Hardware Software Co-Design. The chapter is structured in four sections. An introduction to basic servo control will be given in section 5.1. The application of the modelling methodology will be presented in section 5.2. An actual implementation will be described in section 5.3. Analysis of the different implementations and reflections on the correlation between the modelled architectures and the final implementations will be provided in section 5.4. Finally, a summary of the chapter is given in section 5.5.

5.1. Introduction to basic servo control

The problem presented in this chapter is the control of a servo motor. Servo motors are electromechanical devices able to orientate its axis between 0 and 180 degrees. Servos contain a direct current (DC) motor, several gears, a potentiometer and an electronic control board. The electronic control board acts as an internal servo motor controller and as an interface between the servo and the external controlling device. A detailed internal view of the servo is shown in figure 5.1.

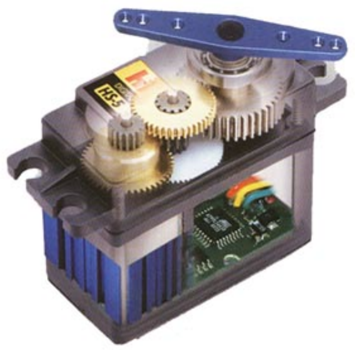


Figure 5.1: Servo internals. Source: `hitecrcd.com`

The servo electrical connections are: *Vcc* in order to power the servo, *GND* electrical ground required in all electrical devices and *Signal* the control signal used to command the servo. The

control signal is *Pulse Width Modulated.* A signal of these characteristics will change the duration of the pulse according to target position for the servo. Therefore, there is a direct connection between the pulse duration and the final servo position. Note that the frequency of the signal remains constant, the only parameter that changes is the *duty cycle*.
The required command signals vary from one servo to another, and it is something that the servo manufacturer might change. It is common to use pulses between 1ms to 2ms in order to control the servo. The period of the control signal normally varies from 18ms to 25ms (55 to 40 Hz). The period of the control signal depends as well on the servo manufacturer. In order to reach a correct servo operation it is a requirement that the servo is pulsed at the required control frequency, otherwise *jitter* will take place.

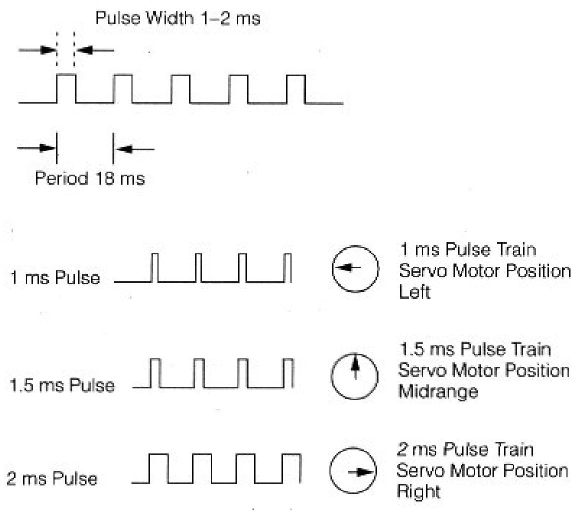

Pulse Width 1–2 ms
Period 18 ms
1 ms Pulse
1 ms Pulse Train Servo Motor Position Left
1.5 ms Pulse
1.5 ms Pulse Train Servo Motor Position Midrange
2 ms Pulse
2 ms Pulse Train Servo Motor Position Right

Figure 5.2: Typical servo control signals. Source: `seattlerobotics.org`

Figure 5.2 is an example showing the working principles introduced above. In this particular servo case the pulse duration is 18ms. This means that the signal control frequency is working at 55Hz. The pulse width will vary between 1ms to 2ms, representing 0 and 180 degrees respectively. Intermediate pulse widths can be used in order to position the servo at different angles of the described semicircle.
The *partitioning decision* in this case is that a servo controller can be implemented in software or in hardware. In the case the servo is controlled from a software component deployed in a microcontroller without an OS, the solution will consist on toggling a logical output between low and high values respecting the required timing constraints. These can be established by inserting properly wait routines in between the toggling functions. In the case the solution is implemented in hardware, the functionality will be exactly the same but implemented in VDHL, and running independently from the software side. Additional details on both implementations will be presented in the following sections. Even though functionality can be reached in several ways, timing constraints will play a major role on deciding upon a concrete implementation.

5.2. Modelling of a servo controller

Following the thesis aim of illustrating model-based engineering using the VDM method, the first step in this case study is to model the problem. The modelling flow will proceed according to the process explained in chapter 4.

5.2.1 Sequential model

The first step is to create a sequential model of the problem under study. This model will allow to represent unambiguously the entities involved in the system and the causal ordering of events (application of guideline 1).
This model is composed by the following entity classes:

Controller: represents the main control logic responsible for the system operation. In terms of hierarchy this would be the top-module of the system.

PWMgenerator: represents the logic used for the generation of the PWM signal described in the previous section.

Clock: periodic signal regulating the execution of the model. This signal is a logical notion of time: the simulated time.

GPIOinterface: represents a hardware block. It represents the boundary between the microcontroller unit and the environment. GPIO stands for General Purpose Input Output. Each GPIO channel has an associated pin in the microcontroller package. This pin can be at low level (0 Volts) or high level (5 Volts[1]). The GPIO hardware block is mapped to a control register that can is loaded with the control value, determined by the algorithm running in the microcontroller CPU.

Additional classes have been used to run the simulation. These classes are mere helpers and not as relevant as the ones introduced above. However they are relevant in order to generate the simulation results.

PositionLogger: keeps track on the target positions issued by the system control logic. This class will generate a Comma Separated Value (CSV) file with the targeted positions.

Sampler: offers the same functionality as an oscilloscope. In this model implementation this class is sampling the outputted GPIO values. It can be attached to different GPIO lines in order to simulate a multi-channel scope. However, this functionality has not been used in the servo case. Once simulation is completed the sampler class is able to dump all the readings in a CSV file.

Figure 5.3 shows a UML class diagram illustrating the relations between the entities presented above. The model is started and configured by the `Environment` class, which will be responsible as well for tearing down the execution (in this case only dumping the acquired signals to files). The Environment class will access as well the simulation clock represented by the `Clock` entity. Each time a clock tick is generated by the Clock, a sequence of `timeStep` operations will be triggered from the environment in all the instantiated classes. This can be seen in the code listing below. Each class that holds an active responsibility is implementing a `timeStep` operation.

```
while (clk.getSimTime() <= simTimeMax) do
(
 currentSimTime := clk.getSimTime();
 controller.timeStep(currentSimTime);
```

[1]Considering Transistor Transistor Logic (TTL)

```
  pwmUnit.timeStep();
  sampler1.sample();
  clk.tick();
 );
```

In the case of the `PWMgenerator` entity, the `timeStep` operation is triggering the execution of the `generateCycle` operation. This operation is responsible for creating the control signal that is provided to the servo. The code listing below shows how it has been modelled in VDM. An internal `tickCounter` variable is representing the number of ticks that will be used to generated a full PWM cycle (composed by the high and low state). The high state of the PWM signal starts at logical tick `0` and ends at logical tick `tickLimit`. As described in the model, these boundaries will be triggering the `setOn` and `setOff` operations causing, therefore, the proper logic level transition in the associated GPIO class instance.

```
private generateCycle: () ==> ()
generateCycle() ==
  cases tickCounter:
     (0) -> setOn(),
     (tickLimit) -> setOff(),
     others -> skip
  end;
```

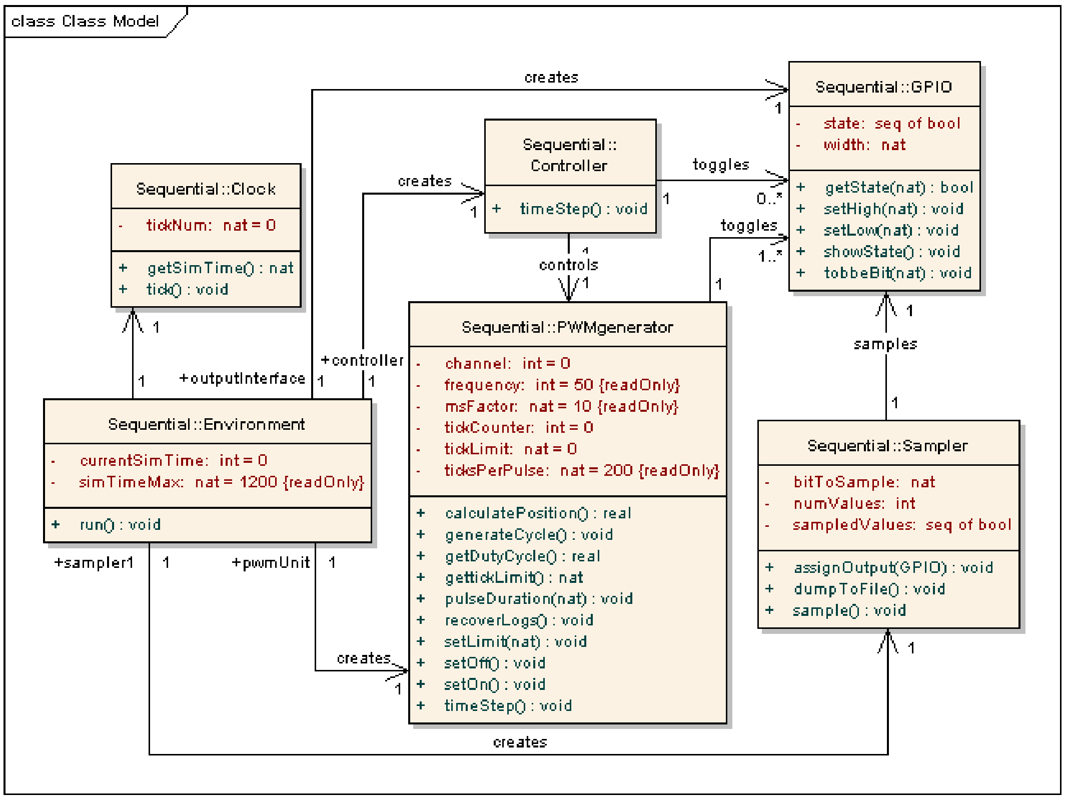


Figure 5.3: UML class diagram view of part of the sequential model.

The main outcome of the creation of a sequential model of the system is the unambiguous representation of the system hierarchy, and the separation of responsibilities among entities. An additional advantage derived from the application of the VDM methodology is the possibility of executing the models under different scenarios, something that is not that easy to do under UML[2].
The notion of time incorporated in the model is used in order to determine the ordering of events. However this notion of time is not sufficient in order to study time dependant properties of the system, something that will have to be studied in the real time models. Concurrent operation is not incorporated in the sequential model either.
The execution of the model produces as a result a CSV file that can be plotted (application of Guidelines 11 and 12). The resulting plot is a representation of the generated PWM signal by the modelled system. In this case the tool *gnuplot* and a set of bash scripts have been used in order to generate the signal plot. Figure 5.4 show the predicted signal output by the VDM sequential model.

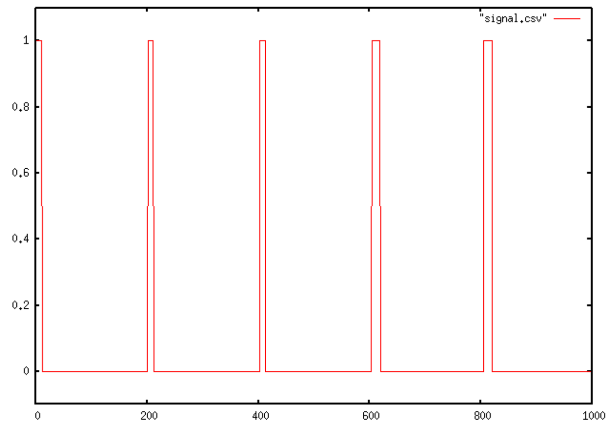
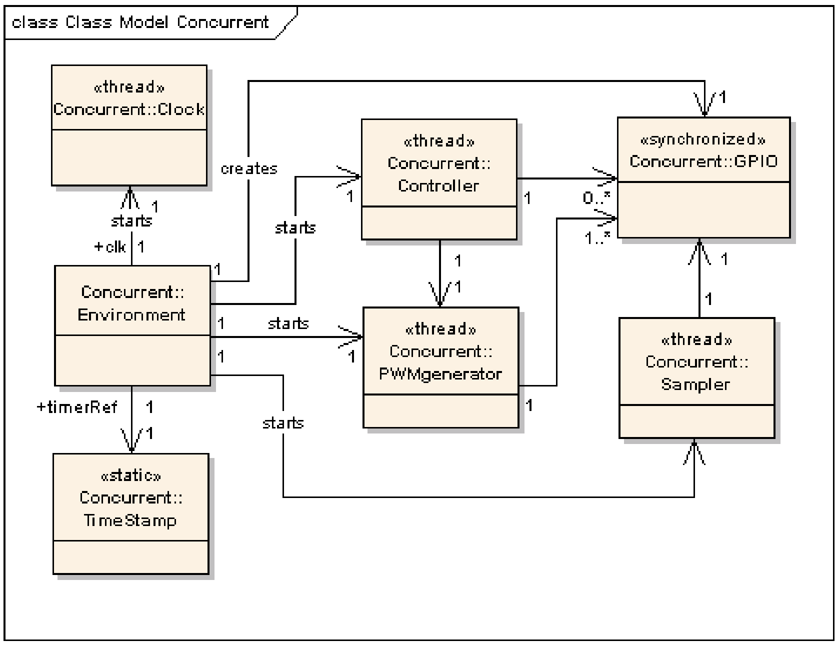


Figure 5.4: Generated servo control signal.

5.2.2 Concurrent model

The concurrent model gives the possibility of showing the concurrent aspects of the systems. The outcomes of such a model are to present behaviours that should be running in parallel and to study how these behaviours are interacting in a safe manner. In this model the timestamp pattern is used. This pattern allows a smooth conversion from the sequential to the concurrent world, by the introduction of a thread barrier. This barrier allows the regulation of the time progression, making sure that at least all the threads have been executed `timeStep` once before making the simulation clock advance one time unit More details on this pattern can be found in can be found in [Lausdahl&10b].
Figure 5.5 presents a simplified UML class diagram of the concurrent VDM model. Attributes and methods are not shown in this class diagram in order to increase readability. These details can be consulted on the previous class diagram 5.3. Stereotypes «`thread`» and «`synchronized`» are used in order to display the concurrent behaviour of the entities.

[2]Some Computer Aided Software Engineering (CASE) tools available in the market provides the necessary facilities to execute UML sequential models and generate different execution traces. An example of such a tool is IBM Rational Rhapsody. These tools are complex and expensive. An additional advantage is that, executable UML models have to be following the syntax specified by the UML standard. The result is a syntactically correct and executable model.



class Class Model Concurrent
«thread» Concurrent::Clock
«thread» Concurrent:: Controller
«synchronized» Concurrent::GPIO
Concurrent:: Environment
«thread» Concurrent:: PWMgenerator
«thread» Concurrent:: Sampler
«static» Concurrent:: TimeStamp
creates
starts
starts
starts
starts
+clk
+timerRef
0..*
1..*
1

Figure 5.5: Concurrent model simplified UML class diagram.

From the VDM perspective, the model infrastructure is constructed by the `Environment` class. The threads are started from this entity as well, which will block until the finishing condition has been reached.

```
  start(clk);
  start(pwmUnit);
  start(controller);
  start(sampler1);
```

Then `Environment` class makes sure all the threads have progressed at least one time unit by invoking the static operation contained in the TimeStamp class `WaitRelative`. The `TimeStamp` class maintains as well the notion of time used in the simulation.

```
while timerRef.GetTime() <= simTimeMax do
(
  timeStep();
  Environment`timerRef.WaitRelative(1);
);
```

The most outstanding change in the VDM models is that now all the active classes, which are running in threads, are incorporating a thread section in the class body. This thread section defines the concurrent behaviour the class is going to present. Thanks to organizing the model evolution in the form of discrete `timeSteps` contained in a single function, the evolution towards a concurrent version is relatively simple. As an example, the concurrent aspect of the `PWMgenerator` class is presented in the code listing below. As it is shown in the model, the `timeStep` operation will be running once and then the thread will block until it is notified by the `timeStamp` instance class (declared as `timerRef`). The thread will be executing continuously until the Environment class finishes the execution.

```
thread
  while true do
  (
    timeStep();
    Environment`timerRef.WaitRelative(1);
  );
```

The `timeStep` operation invoked from the while loop, is modelling exactly the same behaviour as the one presented in the sequential model subsection. Besides detailing the active behaviour of the system, synchronization primitives have been used in order to ensure a safe concurrent execution (application of guideline 3). The primitive **`mutex`** has been used in order to express the necessity of mutual exclusion between readings and writings during the information exchange. The most significant case is present at the GPIO class, which is updated by the `PWMgenerator` and sampled by the `Sampler` class. The used protection is shown in the next code listing. As it is modelled, the state presented by a certain gpio bit can be consulted if it is not being updated. Additionally, a certain bit cannot be being altered at the same point of time.

```
mutex(setLow,setHigh);
mutex(setHigh,getState);
mutex(setLow,getState);
```

At this point of the modelling stage, a model that presents proper structure, hierarchy and behaviour, besides a concurrent operation description has been reached. The next step in order to study how the system behaves in real time and under different deployments (different architectures).

5.2.3 Distributed real-time model

The distributed real time VDM model brings the possibility of exploring the real time behaviour of the previously developed models. Besides studying real time properties, different architectural candidates can be evaluated. In order to model these architectures, the `deployment` class has been incorporated to the model:

Deployment: defines the system infrastructure by the use of processing blocks (CPU) and communication interfaces (Buses). It contains the static class declarations of the modelled entities. This class is also responsible for deploying the modelled entities in the processing units.

Since the notion of time is provided by the overture kernel, the clock class is not needed. The changes in the classes can be seen in the class diagram depicted in figure 5.6. Methods and attributes are not displayed in order to increase readability of the diagram. For additional details refer to figure 5.3.
Initially, a non-distributed real time mode has been created. In this model all the components are deployed in the same CPU. Even though this does not provides information in order to perform partitioning decisions, it gives the possibility to make sure that the model is working correctly

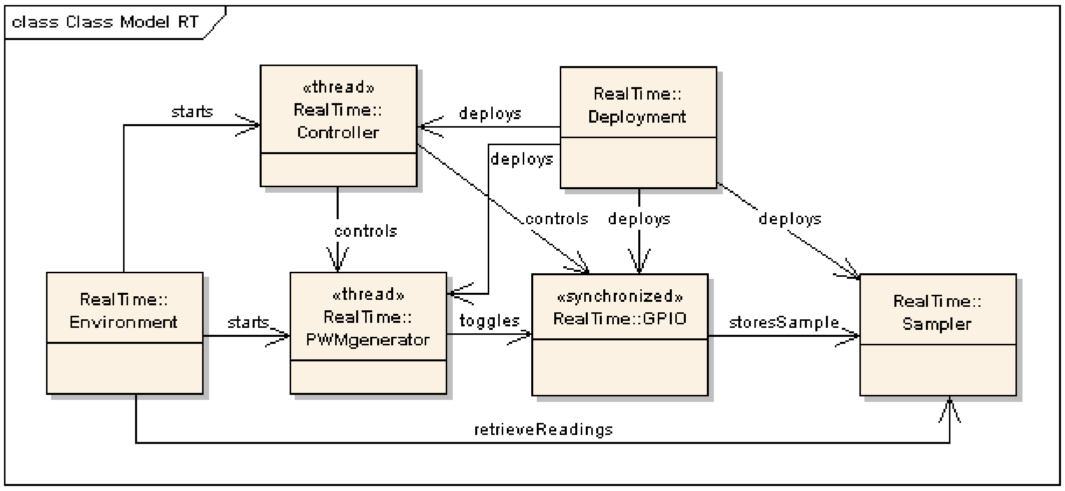


Figure 5.6: Distributed Real-Time model simplified UML class diagram.

under the Overture real-time kernel before moving to the architectural exploration phase (guideline 4).
During the exploration phase, three different architectures have been studied. Two of them are pure software solutions and a third one is making use of a dedicated hardware partition for the PWM signal generation.

5.2.3.1 Architecture 1: an OS-less software based solution

This architecture is purely based on software. The PWM signal generation logic is deployed together with a controller class in a CPU. The GPIO block has been deployed in a separated hardware block communicated at a very high speed (guideline 6) with the CPU software controller. The purpose of such a high bandwidth in the bus used is to model the fast access to a hardware component mounted in the same chip as the CPU. The separated hardware block representing the hardware partition is running at a frequency orders of magnitude above the general purpose CPU frequency (guideline 5).
An overview of the architecture as generated by Overture is shown in figure 5.7.

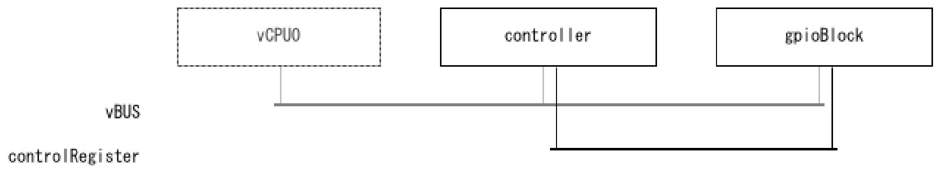


Figure 5.7: Architecture of the software based solutions.

The deployment of the entities has been carried out as described in the UML deployment diagram shown in figure 5.8.
In order to study the load response of the CPU running the PWM generation logic, the function `timeConsumer` has been introduced. This function is taking processing time progressively each time the controller thread is executed. Therefore, the longer the simulation has been running the more time the `timeConsumer` function will be taking.

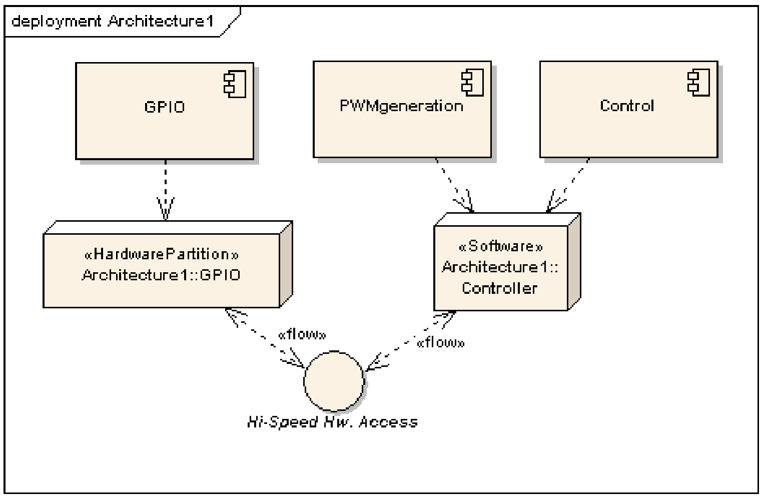


Figure 5.8: Deployment diagram of the software based solutions.

In order to maintain the proper time limits in the signal, duration statements have been used. In this version of the model the `generateCycle` operation in the PWMgenerator class has been modelled as shown in the listing below. In this concrete case the target PWM signal has a pulse duration of 2 milliseconds and a low state duration of 18 milliseconds. The toggleBit operations are not considered in the time analysis, since the time it takes to toggle the hardware output is orders of magnitude below the phenomena we want to study (generation of a signal in the millisecond range). The increment of the `runCount` has not been considered either, since this is a model dependant parameter that will not be present in the actual implementation.

```
duration (2E6) outputInt.toggleBit(1);
duration (18E6) outputInt.toggleBit(1);
duration (0) runCount := runCount +1;
```

The PWMgenerator thread is modelled as a procedural thread that runs until completion. The controller thread is modelled in the same way. The purpose behind this modelling decision is to avoid using periodic constructs, that will be used under an RTOS based implementation.

The generated signal by the OS-less based software solution can be seen in figure 5.9. In this diagram three control pulses can be seen. The duration of the pulses is correctly generated in all three cases. The separation between control pulses is incorrect. This means that the system is failing to generate the control pulses at the required frequency. Note that the time between the second and the third control pulses is considerably higher than time between the first and second pulse. The separation between the pulses keep augmenting while the model execution progresses, making the situation even worse. This means that a pure software based solution would be correct in the case the CPU is not loaded excessively. Load boundaries would depend on concrete platforms, however, the same situation would appear at certain point of load if this control scheme is used. Remark as well that this architecture is the fastest to implement and the cheapest in terms of used silicon area (assuming that a processing unit is already present in the system). Generation of the PWM signal from a OS-less based software solution, while plausible, should be carefully considered due to is load sensitivity.

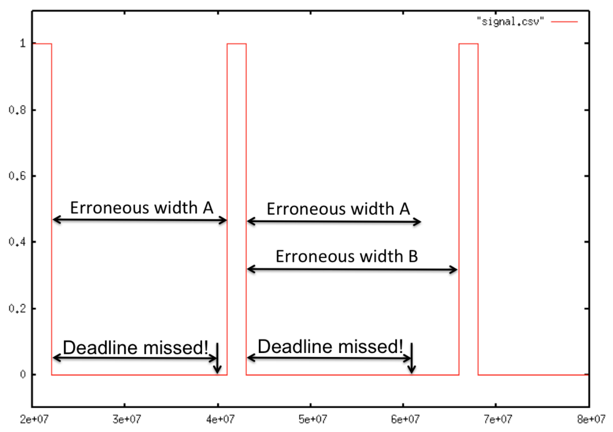


Figure 5.9: Generated control signal by the OS-less software only solution.

5.2.3.2 Architecture 2: a RTOS based software solution

The second architecture evaluated for this case study is making use of a Real Time Operating System (RTOS). The purpose is to exploit the scheduling possibilities that such a software solution offer in order to respect the hard real time deadlines. The used deployment and hardware architecture are exactly the same as the one introduced in section 5.2.3.1. However the scheduling policy in the CPU unit has been altered from *First Come First Served* to *Fixed Priority*. Additionally the time dependant functionality (generateSignal) has been given a higher priority than the controller operations, which are not time critical. Both aspects are illustrated in the following model listing:

```
controller : CPU := new CPU(<FP>, 1E9);
...
controller.setPriority(PWMunit`generateSignal,10);
controller.setPriority(Controller`timeConsumer,1);
```

As it can be imagined, this second software based approach is able to generated the PWM signal properly and respecting the deadlines (see chart on the right side of figure 5.10). It is interesting to study the impact of turning off the priorities and evaluate whether this approach is still valid. In the case no higher priority is given to the generate signal operation, the model is indicating that some deadlines will be eventually missed (see chart on the left side of figure 5.10). Finally, a more detailed view on the correctly generated control signal is depicted on the left side of figure 5.11.

This architecture is more complex that the OS-less software based solution. It is possible to use an open-source RTOS and achieve the expected results. Economically speaking this would not be necessarily more expensive. From the resource consumption point of view, it is clear that this solution requires additional hardware infrastructure (more memory and a dedicated timer to be used by the RTOS). In principle, no additional silicon are would be required.

5.2.3.3 Architecture 3: a combined hardware/software solution

The last evaluated architecture is considering a hardware partition for the generation of the PWM signal, which is show in figure 5.12.

This architecture has been defined in VDM as shown in the next model listing. Since the PWM generation block is deployed in a hardware block, the chosen frequency is orders of magnitude

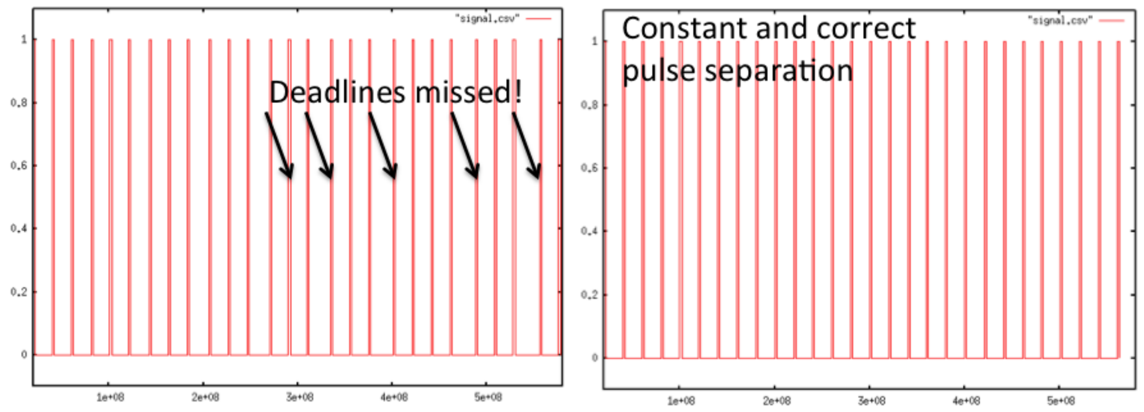


Figure 5.10: Generated signals by the RTOS based software only solution.

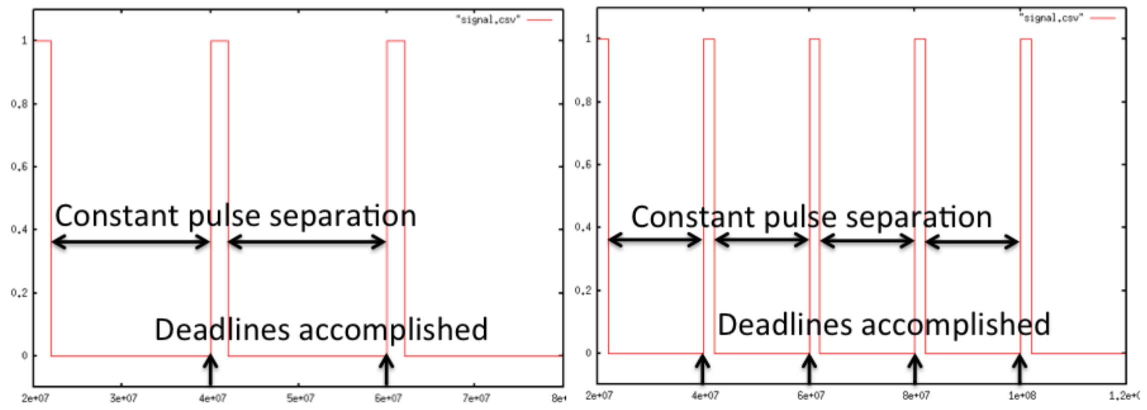


Figure 5.11: Generated signals by the RTOS based with priorities and the Hardware based solutions.

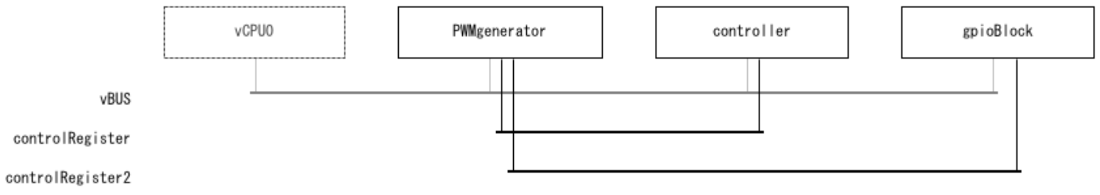


Figure 5.12: Architecture of the hardware based solution.

above the controller frequency (guideline 5). The hardware partition is running at a frequency of 1E9, while the controller is running at 1E5. The communication between the CPU and the hardware blocks is established through hi-speed buses, modelling the available high speed interfaces between hardware and software (guideline 6).

```
PWMgenerator      : CPU := new CPU(<FCFS>, 1E9);
controller        : CPU := new CPU(<FCFS>, 1E5);
gpioBlock         : CPU := new CPU(<FCFS>, 1E9);
controlRegister   : BUS := new BUS(<CSMACD>,
                                  72E13,
                                  {PWMgenerator,
                                   controller}
                                 );
controlRegister2 : BUS := new BUS(<CSMACD>,
                                  72E13,
                                  {PWMgenerator,
                                   gpioBlock}
                                 );
```

The deployment used in this case is depicted in the UML diagram show in figure 5.13. As it can be seen, each entity has been mapped to a separate processing unit.

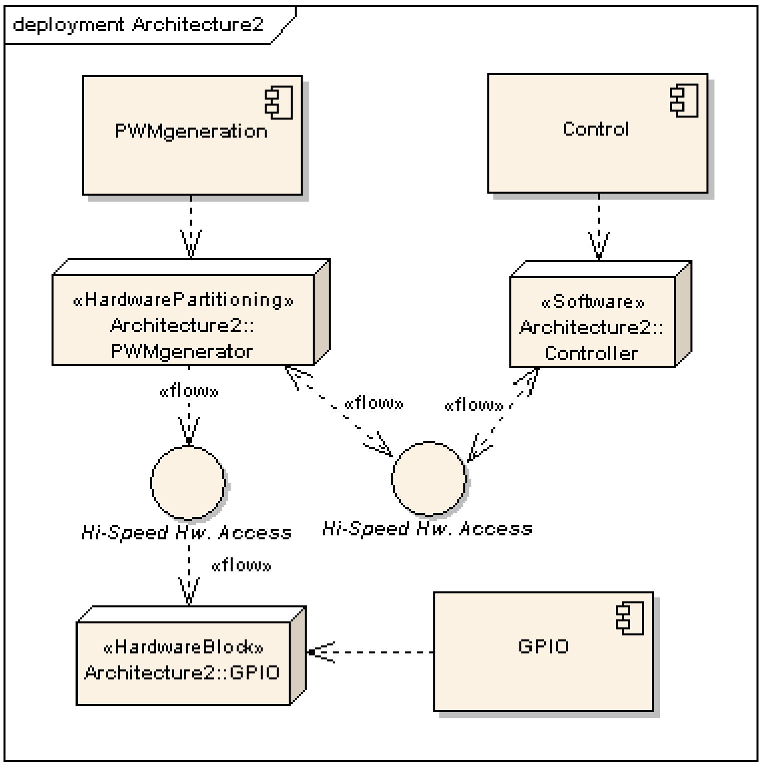

deployment Architecture2
PWMgeneration
Control
«HardwarePartitioning» Architecture2:: PWMgenerator
«Software» Architecture2:: Controller
«flow»
«flow»
«flow»
Hi-Speed Hw. Access
Hi-Speed Hw. Access
«flow»
«HardwareBlock» Architecture2::GPIO
GPIO

Figure 5.13: Deployment diagram of the hardware based solution.

The logic used is the same as the one explained in section 5.2.3.1. The generated signal is depicted in figure 5.11 on the right. As it can be seen, this architecture is able to generate the PWM signal correctly and within the established time frames. The use of a separate block for the PWM signal generation makes the generated signal independent from the CPU load. The signal is generated correctly no matter how much the CPU is loaded.

This solution is the most adequate regarding performance. On the other hand it requires more silicon area on the FPGA since an additional hardware block has to be deployed on it. In the case an IP block is ready to integration additional development effort would not be necessary. In the case no IP block is available, the required time for VHDL development would have to be considered in the project time budget.

5.3. Implementation of the servo controller

The studied implementations were originally presented in [Speedway11SDK]. The presented results in this section relates the extracted information from the implementation and analyses the results. In addition these results are related to the formal models presented above. The study of the information provided by the actual implementation allows the evaluation of the Hardware/-Software partitioning decisions, and bridges the gap between abstract representation and real implementation (application of guideline 13).

5.3.1 Preliminary details

The models presented in the previous section are making use of a `CPUloader` function that delays the execution at a constant rate of 1 millisecond per iteration. This function is not performing any specific calculation. In this implementation, the `CPUloader` software component has been substituted by an algorithm finding prime numbers. This computational complexity of the algorithm causes a variable execution time per execution. The fact that the `CPUloader` is different does not affect the study of the real-time deadlines fulfilment. The purpose of the models were to illustrate how a delay was affecting the servo control problem, not to model how that delay was created (abstracted detail).

5.3.2 Implementing the conflictive architecture

This architecture is based on a pure software solution. The control logic is running together with the `CPUloader` function on the same CPU and without any kind of operating system. The servo is controlled by the software implementation of the logic described in the sequential VDM model. The function responsible for the `generateCycle` functionality described in VDM is `adjust_servo_manually`. The most relevant parts of this function are the sleep routines `millisleep` and `usleep`, which obtain the appropriate point of time in which the GPIO has to be toggled in order to generate the PWM signal. The variable `high_time_us` is used in order to calculate the duration of the pulse in order to reach the target position, determined by the coded expression in the function. Note that this expression vary depending on the used servo.

```
void adjust_servo_manually(int baseaddr,
                           int servo_number,
                           int position)
{
  int high_time_us;
  high_time_us = 700 + ((2000/256) * position);
  millisleep(25);
  rc_servo_assert_manual_output(baseaddr, servo_number);
  usleep(high_time_us);
  rc_servo_deassert_manual_output(baseaddr, servo_number);
}
```

In the code listing presented below, the main control loop is shown. In this loop the performed operations are the servo control operation and the auxiliary cpu loader operation (`findPrime`).

```
while(latest_prime_found < highest_prime_to_find)
```

```
{
  adjust_servo_manually(XPAR_AXI_RC_SERVO_CONTROLLER_0_BASEADDR,
                        1,
                        0x00 + servo_value);

  latest_prime_found = findPrime(latest_prime_found,
                                 highest_prime_to_find);
}
```

5.3.3 Implementing the optimal architecture

The optimal architecture is offloading the PWM signal generation by the use of the IP block `axi_rc_servo_controller`. The prime number finder is still run in the system processor. The structure of the used IP block can be seen in figure 5.14. The figure shows that the device is clocked, represented by `>0` on the upper left corner, that it has one signal output, represented by `1-` in the upper right corner, and that it is attached to the AXI interface of the microblaze processor. Figure 5.15 shows the device deployed together with other slaves in the final hardware design. As example of other hardware blocks, two GPIO blocks 4-bit with connected to LEDs and switches and an Serial Peripheral Interface Flash memory controller are shown. These blocks are irrelevant to the problem under study, however they give an idea of the variety of devices that can be integrated in the system.

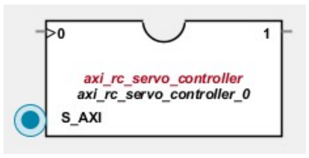


Figure 5.14: IP block implementing the servo control functionality.

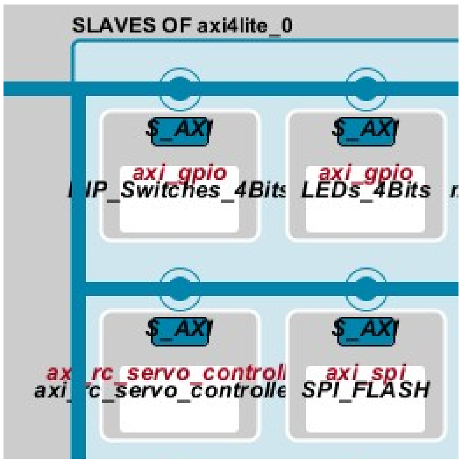


Figure 5.15: IP block deployed.

Even thought the use of an IP block abstract the system engineer from VHDL details, it is worth to inspect how the PWM generation logic is implemented in hardware. The VHDL implementation of the PWM block is shown in the next code listing. The PWM block is organized around a state machine, which is implemented by the use of a case statement. This state machine presents three possible states: "*initial* or *reset state*", "*low period*" and "*high period*". A counter is used in order to determine which state should be the next one. The counter is controlled by a clock dependant process. The values the counter may present vary from 0 to `servo_PWM_clock_periods`. This second variable represents the number of elapsed clock periods per PWM cycle. As it can be imagined the number of clock cycles has to be *much higher* than the desired target frequency for the PWM signal in order to get a good resolution. The value the counter is providing is compared with the number of required `low_pulse_width_clock_periods` and `high_pulse_clock_periods` in order to trigger a transition in the state machine or remain in the same state.

```
servo_control_out <= '0';
  reset_control_counter <= '0';
  case current_state is
  when reset => reset_control_counter <= '1';
    next_state <= low_period;

  when low_period =>
    if (control_counter >= low_pulse_width_clock_periods) then
      reset_control_counter <= '1';
      next_state <= high_period;
    else
      next_state <= low_period;
    end if;

  when high_period => servo_control_out <= '1';
   if (control_counter >= high_pulse_width_clock_periods) then
    reset_control_counter <= '1';
    next_state <= low_period;
  else
    next_state <= high_period;
  end if;
end case;
```

```
control_counter_process : process (clk, rst)
begin
  if clk'event and clk = '1' then
    if reset_control_counter = '1' then
      control_counter <= 0;
    elsif (control_counter = servo_PWM_clock_periods) then
      control_counter <= 0;
    else
      control_counter <= control_counter + 1;
    end if;
  end if;
end process;
```

The most interesting conclusion that can be drawn from this analysis, is that *the logic behind the PWM generation in VHDL is the same as the one modelled in VDM*. Besides the control logic, the VHDL implementation contains the required logic to interact with the processor. The IP servo control block is integrated as a slave hardware device, mapped to a control register in the microblaze processor. From the software side, the servo is controlled by invoking the function `rc_set_servo_position`, which will take as a parameter the register address, the servo number and the target position. The IP hardware block will be continuously polling the associated register and repositioning the servo according to the hold value. This process is run in parallel and independently of the state of the software functionality running in the microblaze CPU.

```
void rc_servo_set_servo_position(int baseaddr, int servo_number,
int position)
{
  servo_number--;
  if (servo_number >=0)
    rc_servo_set_servo_register(baseaddr,
                                8 + (servo_number*4),
                                position);
}
```

5.3.4 RTOS based solution

A third possible implementation, which has been explored from the VDM-RT context, is the integration of a Real Time Operating System. This third design alternative has been studied but not implemented in the FPGA. However, similar results to the ones exhibited by the VDM-RT model can be expected. The considered RTOS for this implementation have been *Xilikernel* and *FreeRTOS*. Xilikernel is not a viable option for new designs or research activities. Xilinx representatives have stated in informal conversations that Xilikernel is not a priority for Xilinx when it comes to development effort and maintenance. As an alternative they highly recommend the use of *FreeRTOS* [3].

As it was explored in the RTOS based VDM model, the generation of the control signal could be incorporated in a periodic thread that is executed at a period of 20ms. A thread (called task in FreeRTOS) can be used as shown in the next code listing. One of the most relevant parameters of the thread creator is the function entry point. In this case, the entry function will be associated to the generate signal function, as it was shown in the models. The execution of this task will be started by a software timer, which will be generating an interrupt every 20ms. The task should be executing the `generateSignal` functionality within a *critical section*, so a preemtive context switch will not occur. Interrupts should be deactivated *before* entering the critical section.

```
portBASE_TYPE xTaskCreate(
  pdTASK_CODE entryPoint,
  const portCHAR * name,
  unsigned portSHORT usStackDepth,
  void * Parameters,
  unsigned portBASE_TYPE uxPriority,
```

[3]FreeRTOS is open-source and is available for download at `http://www.freertos.org/`.

```
  xTaskHandle * callback // If needed
);
```

The next code listing shows the thread signature that should be used in a FreeRTOS based implementation.

```
xTaskCreate(
  generateSignal,
  "RCservoControl",
  SMALL_SIZE,
  (void*)targetTime,
  HIGH_PRIORITY,
  NULL
);
```

While possible, this software implementation can be highly inefficient, since the system can be busy generating the PWM signal up to 10% of the time. Besides performance considerations, a more critical situation could be that other real-time events are not being processed because the PWM task is being served.

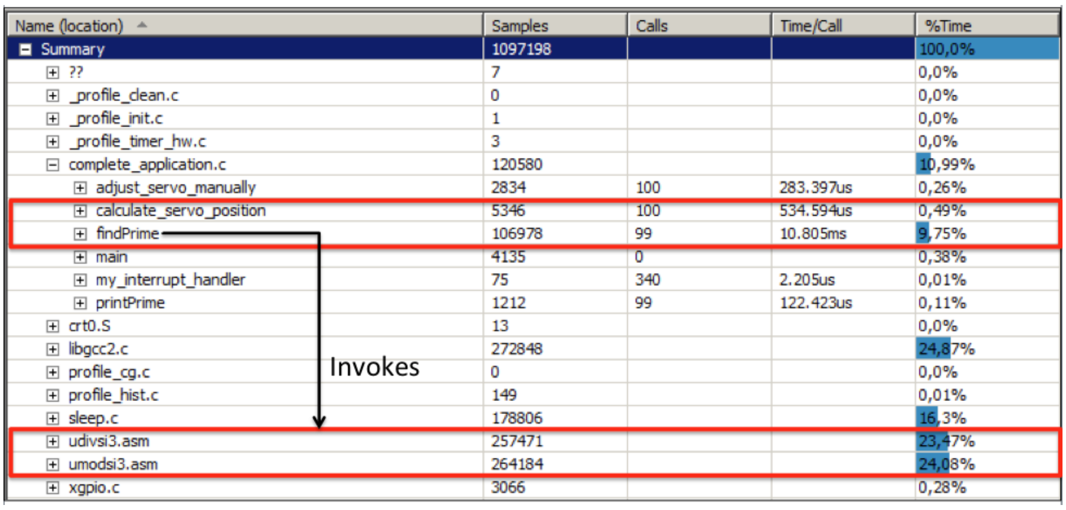
5.4. Evaluating the implemented architectures

In this section two implementations will be evaluated. These architectures correspond to the OS-less software solution and to the combined hardware/software solution. The results provided by this evaluation were initially predicted by the models presented above, which were used to make the partitioning decisions of the system. In a real development setup, only the most suited architectural candidate will be selected for implementation. In order to validate the thesis that VDM-RT models can provide the necessary information to make partitioning decisions, the two opposite architectures have been evaluated. As it will be presented below, the performance of the different solutions was properly anticipated by the abstract VDM-RT models.

5.4.1 Evaluation of the conflictive architecture

Xilinx Software Development Kit provides a tool to profile the execution of a certain application. The advantage of the Xilinx profiler is that the evaluation is performed when the application is running on the actual hardware, providing real measurements instead of predictions. The profiler has been used in order to evaluate the performance of the "software only" implementation. In order to get more meaningful results, the serial traffic has been eliminated by commenting out the print statements in the application. The reason behind this is that a considerable amount of time is used in order to send information through the console. This process may mask the potential bottlenecks that may appear in the real application. After eliminating the print statements and some preliminary profiler setups, a test run can be executed. This test run offered the results shown in figure 5.16.

The time critical functions `adjust_servo_manually` and `calculate_servo_position` are taking 0.26% and 0.49% of execution time respectively. On the other hand the CPU loading function `findPrime` is requiring 9.75% of execution time, and invoking arithmetic functions that



Name (location)	Samples	Calls	Time/Call	%Time
Summary	1097198			100,0%
??	7			0,0%
_profile_clean.c	0			0,0%
_profile_init.c	1			0,0%
_profile_timer_hw.c	3			0,0%
complete_application.c	120580			10,99%
adjust_servo_manually	2834	100	283.397us	0,26%
calculate_servo_position	5346	100	534.594us	0,49%
findPrime	106978	99	10.805ms	9,75%
main	4135	0		0,38%
my_interrupt_handler	75	340	2.205us	0,01%
printPrime	1212	99	122.423us	0,11%
crt0.S	13			0,0%
libgcc2.c	272848			24,87%
profile_cg.c	0			0,0%
profile_hist.c	149			0,01%
sleep.c	178806			16,3%
udivsi3.asm	257471			23,47%
umodsi3.asm	264184			24,08%
xgpio.c	3066			0,28%

Figure 5.16: Software based solution profiling.

are requiring almost 50% of processing time if considered together (`hudivsi3` and `umodsi3`). This shows that the software bottleneck is introduced by the CPU loading function `findPrime`. This function is making the control function `adjust_servo_manually` missing its real time deadlines. The profiler shows how the problem predicted by the VDM-RT models is present in the actual system implementation.
Besides detecting the conflictive situation by the use of the profiler, it is possible to monitor the generated PWM signal with a scope, and study how it changes over time.
During the first seconds of execution, it can be seen how the PWM signal is generated properly, respecting the time limits for the pulse and at the required frequency. Figure 5.17 shows both constraints on the left and right scope dumps respectively.

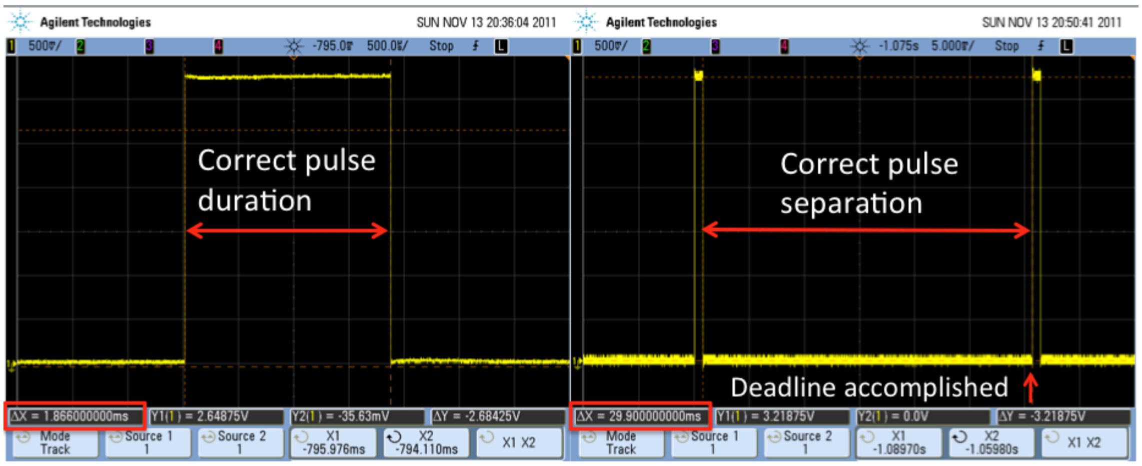


Figure 5.17: High pulse duration and correct signal period.

After the system has been running for a period of time of approximately 20 seconds, it is possible to appreciate the problems that arise if the software only architecture is used. Figure 5.18 on the left shows how the spacing between two consecutive control pulses is taking up to 1.3 seconds, orders of magnitude greater than the initial 25 ms deadline. Moreover, since the CPU loader is consuming processing time at a variable rate, the delay introduced in the control pulse generation

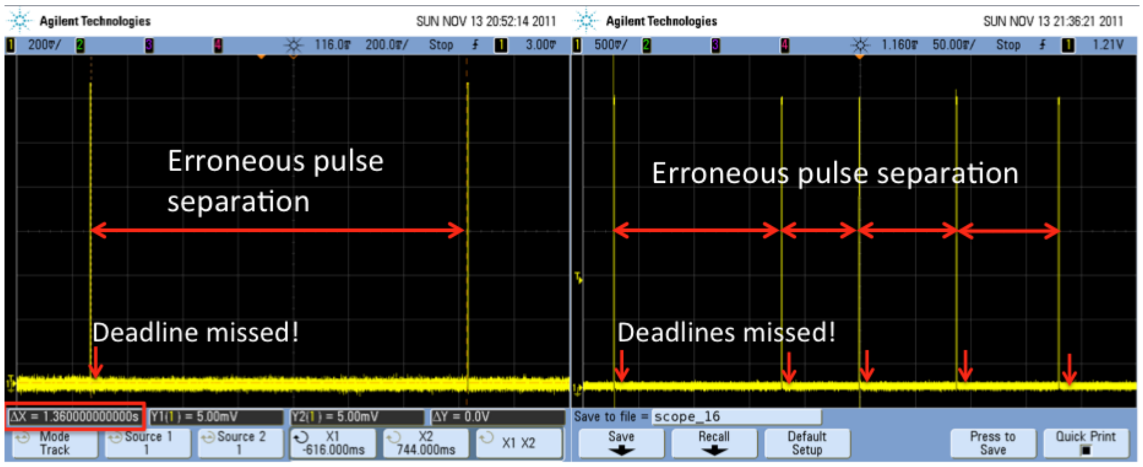


Figure 5.18: Generated control signals after 20 seconds of operation.

is varying as well. This makes the control error variable during system operation. Uneven spacing between the control pulses can be seen in the chart on the right in figure 5.18.

5.4.2 Evaluation of the optimal architecture

The optimal architecture, which makes use of a hardware block for the PWM signal generation, has not been profiled. This second solution has been directly deployed into the FPGA and its generated signal monitored with the scope. An overview of the generated signal can be seen in figure 5.20. As it is depicted, the spacing between the control pulses seems to be correct and evenly distributed. The measurement was repeated at different moments of time over several minutes with no changes.
In order to take a closer look at the signal, the timebase was decreased. This allowed to analyse the duration of the control pulse and the exact separation between consecutive pulses. The duration of the control pulse, shown in figure 5.19 on the left is correct, presenting a value of 1.76ms. The separation between two consecutive control pulses, shown in figure 5.19 is correct as well, presenting a value of 29.4 ms.

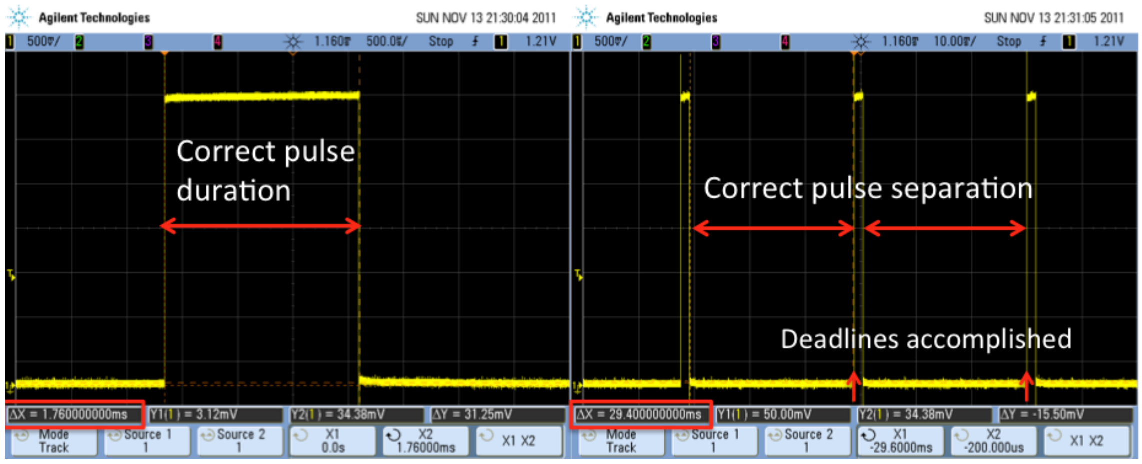


Figure 5.19: Detailed view of the generated PWM signal by the IP PWM block.

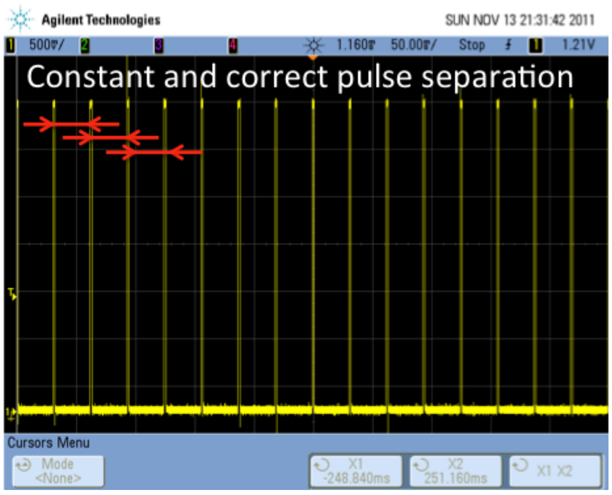


Figure 5.20: Overview of the generated PWM signal by the IP PWM block.

5.4.3 Physical results

In order to see the results of both architectures physically, a servo was connected to the GPIO pin in the FPGA. When the software only architecture was in use, it was possible to see the jitter in the servo axis after a short period of operation. The jitter was increasing considerably until a full oscillation between 0 and 180 degrees was reached. Remark that a continuous operation under this control signal can damage the servo itself. If the servo is performing a critical control, the consequences of missing the real time deadlines can be serious depending on the concrete application. The hardware based architecture was physically studied as well. As it was predicted by the VDM RT models and shown by the scope analysis, the target position is maintained constant making impossible to move the servo axis. In this case, a complete implementation was possible to be carried out due to the existing literature, material and the simplicity of the case study. However, there are different control cases that are challenging, where different logical control strategies and partitionings might have an impact on the physical performance of the solutions. These more complex cases can be harder to implement, and a model-based setting is the only way to perform the design space exploration required in order to perform a sound partitioning. Evaluation of partitionings and performance of different control logic can be performed by VDM (discrete event modelling). On the other hand, the study of a continuous time process is hard to perform under a modelling environment like VDM. In this kind of scenario, a tool like DESTECS can be a good solution in order to contextualize partitioning decisions and analyse their impact on the physical environment. Additional information on the DESTECS EU FP7 project can be found on [Broenink&10].

5.5. Summary

This chapter has presented how a VDM-RT based modelling methodology can be applied in the hardware software co-design of embedded systems. The methodology has helped to evaluate several architectures and decide on a concrete implementation, making use of IP-cores. Finally, the system implementation has been carried out. Feedback from that concrete implementation has been obtained through application profiling. With demonstration purposes, a conflictive architec-

ture has been implemented as well. Correlation between created models and actual results has been shown.
The VDM models have helped on creating an unambiguous representation of the system, that can be executed under different scenarios. By applying modelling it has been possible to represent the hardware/software co-design partitioning issues and analyse the performance regarding real-time deadline accomplishment. Even though the servo case is simple from a technical point of view, creating models that provide the same kind of information as the one obtained from VDM models using, for example, SysML, could have been more complex.

Chapter 6

Case study: Development of an AVB endpoint device

This chapter is focused on a complex industrial case study, the hardware/software co-design of an Audio Video Bridging (AVB) endpoint device. The methodology proposed in chapter 4 will be applied. This chapter is structured in seven sections. An introduction is presented in section 6.1. An overall description of the AVB protocol will be given in section 6.2. The relevant aspects for the hardware partitioning of the system will be presented in section 6.3. The created models for the hardware/software co-design of the system will be presented in section 6.4. Evaluation of different architectures will be described in section 6.5. The outcome of the model would be briefly evaluated in section 6.6. Finally, section 6.7 summarizes the chapter.

6.1. Introduction

This chapter presents a second case study proposed by the home electronics manufacturer Bang & Olufsen. The focus of this case is the specification and analysis of an Audio Video Bridging compliant endpoint device. The preliminary analysis will create the basis for the hardware software co-design of the system, and it will assist in making the initial partitioning decisions of the system.

6.2. Description of the AVB protocol

The AVB protocol is used in the transmission of multimedia content over AVB compliant networks. It provides the necessary mechanisms to ensure the **real-time deadlines** associated with this kind of content, as well as the **bandwidth reservation** and the facilities to avoid **traffic bursting**. The AVB protocol is a layer 2 transport protocol according to the Open Systems Interconnection (OSI) classification. This protocol is specified in the standard [IEEE-std1722-2011]. This standard specifies that in order to create an AVB compliant device the following protocols have to be supported:

802.1AS: time synchronization protocol. This protocol is responsible for distributing a common notion of time among all the nodes that belong to the AVB network.

802.1Qav: stream reservation protocol. This protocol is responsible for allocating 75% of the available network bandwidth for transmission of AVB traffic.

802.1Qat: traffic shaping protocol. This protocol ensures that the traffic is sent over the network smoothly and at a constant pace.

IEEE1722 can be used on top of multiple media layers. These physical mediums include optical fibre, wireless networks and Ethernet. The main development effort at the time of writing is focused on the 802.3 physical layer (Ethernet). Note that AVB has nothing to do with TCP/IP. TCP/IP traffic can be used in parallel to IEEE1722, for example, in order to send control commands. TCP/IP is not necessary in order to stream multimedia content with AVB. A contextualized view of the involved protocols and layers is depicted in figure 6.1. An overview of AVB and its relation to these protocols is given in [Garner&07].

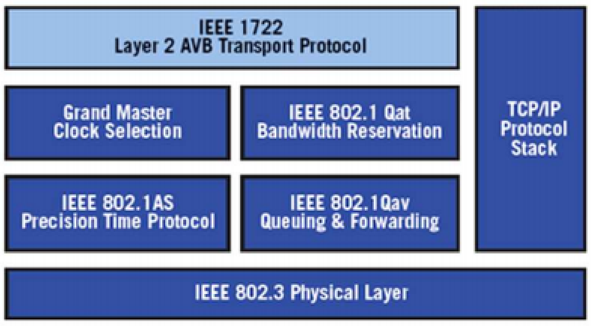


Figure 6.1: Protocol stack showing the involved protocols in AVB. Source: [IEEE-std1722-2011]

Due to its timing performance, the AVB protocol application is not limited to the transmission of multimedia content. Current developments include industrial automation and robotics. Even though these areas are currently being researched, the immediate business case for AVB is the home entertainment market.

6.2.1 The traffic shaping protocol 802.1Qav

The traffic shaping protocol is known as well as a "queuing and forwarding protocol". All the network equipment in an AVB network must be compliant with the traffic shaping protocol. This protocol is specified in [IEEE-Std802.1Qav-2009]. The purpose of this protocol is to provide a uniform traffic at a constant pace. By applying traffic shaping bursting traffic is avoided. A second consequence of its application is that the receiver buffers are not overflowed. The traffic shaper creates several time slots and assigned them to AVB and other traffic (for example, TCP/IP traffic) depending on the listener capabilities and the time deadlines. This scheme resembles the Time Division Multiplexing (TDM) technique used in many communication systems. Figure 6.2 shows the basic structure of a traffic shaping unit. The shaper is composed by two multiplexers, a Priority-Based multiplexer and a Timing aware scheduler. The latter is the only multiplexer with access to the physical medium. The Priority-based scheduler manages non-critical data, such as TCP/IP traffic. This traffic is delivered as best-effort to the Timing aware scheduler. The queues for critical traffic, for example AVB traffic, are labelled as *Class 4/5 queues*, and are directly connected to the final multiplexer. Finally, the timing aware scheduler classifies the traffic and is responsible for putting the data on to the physical medium when it is scheduled. Further details on the protocol operation are out of the scope of this thesis work. For additional information refer to [Garner&07] and the standard [IEEE-Std802.1Qav-2009].

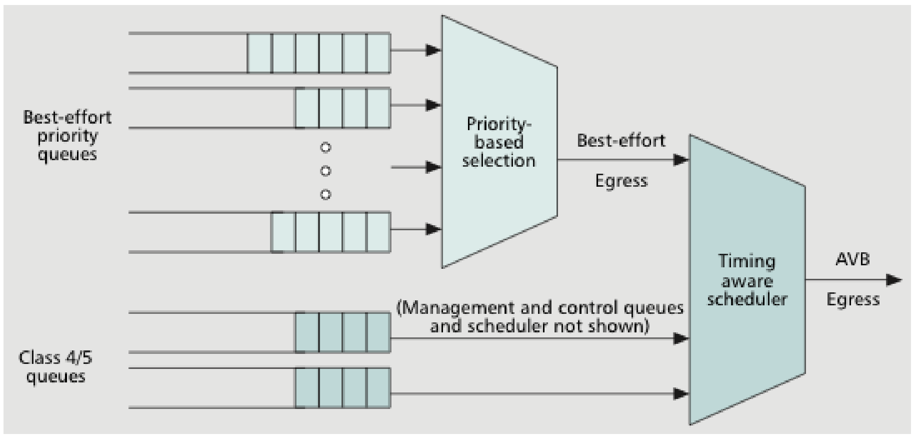


Figure 6.2: AVB and TCP/IP traffic managed by packet filtering. [Garner&07]

6.2.2 The stream reservation protocol 802.1Qat

AVB is effective in time-sensitive content transmission in part due to the network resources that are allocated for its operations. This process is done by applying the stream reservation protocol, specified in [IEEE-Std802.1Qat-2010]. The application of the stream reservation protocol makes it possible to minimize jitter and guarantee a *deterministic* low latency. All the network equipment in an AVB network must be compliant with the stream reservation protocol in order to enable operation. The stream reservation protocol uses the Virtual Local Area Network (VLAN) tagging mechanism in order to filter the network packets before they are transferred to the physical medium. This implies that the packet filtering process is carried out in a very low layer of the communications stack, and can be done by using bitwise operators. The process of configuring the network trees is more complex and involves a dialogue between the talker (information source) and the listener (information sink). Additional details on protocol setup, maintenance and operation are out of the scope of this work. For further details refer to [Garner&07] and the standard [IEEE-Std802.1Qat-2010].

6.2.3 The time synchronization protocol 802.1As

This protocol is specified in [IEEE-Std802.1As-2011]. According to this standard, a device that is able to receive or generate time corrections is called **Time Aware System** (TAS). Each TAS is considered as a system by itself. Therefore, the term system should not be associated with the set of devices conforming the network. A network composed by several TAS could be considered as a network of systems.

6.2.3.1 TAS Structure

Each TAS is composed by several modules. These modules are depicted in figure 6.3. A brief description of each module is presented in the following lines:

Media Dependant Layer: is responsible for the access to the physical medium. Functions that vary depending on the used physical connection are implemented at this layer.

Port Sync: is the information entry point for the TAS. Each Port Sync is attached to a Media Dependant Layer access point.

Site Sync: retrieves the information from every Port Sync structure present on the system. It coordinates the time information dissemination depending on the port functionality. It acts as an interface between the registered ports and the Clock Slave and the Clock Master.

Clock Master: is present only in the case a TAS is Grand Master capable. In the case the TAS is the Grand Master, the Clock Master will be generating the time information that will be sent to the rest of the TAS present in the network. In the case the TAS is not acting as Grand Master, the Clock Master will be registering the difference between the time information received and the time information generated by itself. In the case the current Grand Master leaves the network, and this TAS assumes that role, then it is aware of the existing difference in time, avoiding potential discontinuities.

Clock Slave: is responsible for receiving the time information from the Site Sync entity and processing it. The time information will be made available to the time consuming application through the interfaces implemented by the Clock Slave. The time information will be notified to the Clock Master in the case the TAS is not acting as the network Grand Master, in this case the above explained offset will be computed by the Clock Master.

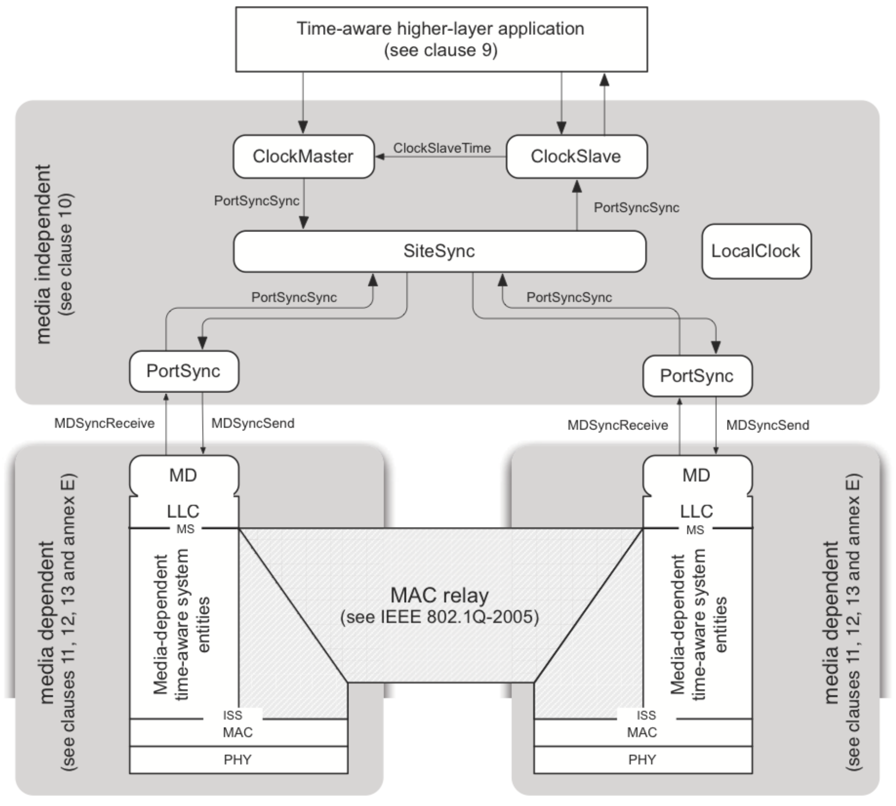


Figure 6.3: Overall structure of a Time Aware System. [IEEE-Std802.1As-2011]

Ports may present different roles: *Master*, *Slave*, *Passive* or *Disabled.* The Master Ports in a TAS are sending the information to other TAS, which are listening in the Slave Ports. Therefore, there is a one-to-one association between each Master and Slave port. Passive and Disabled ports are used to avoid loops in the network topology. These kind of ports are not playing an active role in the network operation. Figure 6.4 shows a Grand Master Time-Aware Bridge connected to a

Time-Aware Bridge. As it is shown all the ports of the Grand Master are acting as Master ports. The second port is connected to the slave port of the Time-Aware Bridge. This second device is forwarding time information over the two lower master ports. Left and right ports are configured as Passive ports, and are not sending nor receiving time information.

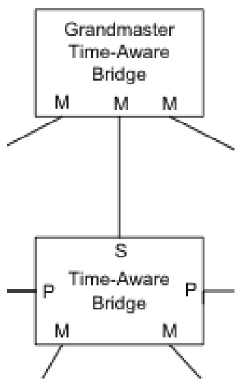


Figure 6.4: A Time-Aware Bridge connected to a Grand Master. [IEEE-Std802.1As-2011]

6.2.3.2 TAS Behaviour

The time synchronization is a two step process in which two separated messages have to be sent. These messages are the *SyncEvent* and the *FollowUpMessage*.

SyncEvent: signals the point of time in which the time information is sent to a slave.

FollowUpMessage: contains the necessary information to compute the point of time in which the SyncEvent was originated at the Master. The follow up has to be generated after the Sync Event was sent, so the precise timestamp of the sending event can be obtained. Further details will be provided in the example below.

The FollowUpMessage is composed by the following fields:

Precise Origin Timestamp: is the original timestamp generated at the Grand Master when a SyncEvent was sent to the Media Dependant Layer.

Cumulative Rate Ratio: is the ratio between the frequency of the master TAS and the slave TAS. Each TAS will generate a different cumulative rate ratio. Further details will be presented in the coming example.

Correction Field: time correction computed at every node after forwarding the SyncEvent. Further details will be presented in the coming example.

Remark that the Precise Origin Timestamp remains constant, and it is never altered by the TASs that are sending a modified Follow Up Message. Only the information contained in the Cumulative Rate Ratio and Correction Field is altered. An example is given below in order to give a better explanation of the synchronization process.
The example network is composed by four TASs. The first system is acting as Grand Master. The remaining three systems are connected one after another (daisy chained). Device two and three are *bridges*, since they can communicate with more than one TAS and forward information. The

devices are deliberately operating at different frequencies to make the example more illustrative. A deployment diagram of this network is shown in figure 6.5. Frequency is specified the generic unit pulses per time unit (ppt). Time is specified in time units (tu).

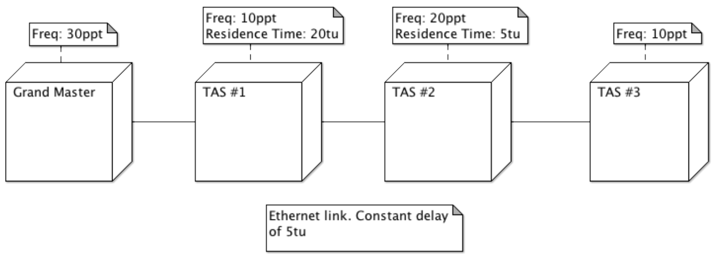


Figure 6.5: Example network scenario.

The synchronization process starts with the slave ports measuring the propagation delay and retrieving the frequency of the devices with master ports. Once the frequencies for each master device associated to each slave port have been determined, the *Neighbour Rate Ratio* for each TAS is computed. The Neighbour Rate Ratio is defined in equation 6.1.

$$NeigbourRateRatio_{TAS\#n} = \frac{Freq_n}{Freq_{n-1}} \tag{6.1}$$

The product of two or more Neighbour Rate Ratio will produce the Cumulative Rate Ratio. This figure allows to compare time measurements carried out in a certain node in relation to the Grand Master. The Cumulative Rate Ratio is computed by applying the equation 6.2.

$$CRR_{TAS\#n} = \frac{Freq_{TAS\#1}}{Freq_{GM}} \cdot \prod_{2}^{n} \frac{Freq_{TAS\#n}}{Freq_{TAS\#n-1}} \tag{6.2}$$

Which is equivalent to equation 6.3.

$$CRR_{TAS\#n} = \prod_{1}^{n} NeighbourRateRatio_{TAS\#n} \tag{6.3}$$

Considering TAS # 2, its Cumulative Rate Ratio would be computed as shown in equation 6.4.

$$CRR_{TAS\#2} = \frac{Freq_{TAS\#1}}{Freq_{GM}} \cdot \frac{Freq_{TAS\#2}}{Freq_{TAS\#1}} = \frac{Freq_{TAS\#2}}{Freq_{GM}} \tag{6.4}$$

Any measurement performed at the TAS #2 can be multiplied by the Cumulative Rate Ratio computed for that TAS, having as a result a time measurement related to the time base used in the Grand Master.
Table 6.1 shows the result of applying systematically these expressions to the Time Aware Systems deployed in the example network.
After a delay of 10 time units, introduced by the physical link layer, the Sync Event sent by the GM is received and timestamped by the TAS#1 producing the *IngressTimeStamp*. The Sync Event is

	GM	TAS#1	TAS#2	TAS# 3
Freq	30	10	20	10
Neighbour Rate Ratio	1	10/30	20/10	10/20
Cumulative Rate Ratio	1 1	1 * 10/30 = 1/3 = 0.33	1 * 10/30 * 20/10 = 2/3 = 0.66	1* 10/30 * 20/10 * 10/20 = 1/3 = 0.33

Table 6.1: Neighbour Rate Ratio and Cumulative Rate Ratio computation

GM generated Follow Up Message	Values
Precise Origin Timestamp	A
Cumulative Rate Ratio	1
Correction Field	0

Table 6.2: Grand Master Follow Up Message

followed by the Follow Up Message. TAS#1 forwards the Sync Event and timestamps the sending process, generating the *EgressTimeStamp*. By calculating the difference of these two timestamps the *Residence Time* for TAS#1 is determined. The Residence Time indicates the time the system has used on receiving, processing and forwarding the event. Residence time is calculated as follows:

$$ResidenceTime_{TAS\#n} = EgressTimeStamp_{TAS\#n} - IngressTimeStamp_{TAS\#n} \tag{6.5}$$

The ResidenceTime has to be taken into consideration so the hops can be compensated when calculating the original GM time. At this point, it is possible to create the Follow Up message contents that will be sent to TAS#2. As explained above, the Precise Origin Time Stamp will not be altered. the Cumulative Rate Ratio has been previously determined. The Link Delay has been measured by the slave port. Finally, the residence time has been obtained applying the last expression. The expression 6.6 links all this variables together, and enables the calculation of the new Correction Field.

$$\begin{aligned} CorrectionField_{TAS\#n} = ResidenceTime_{TAS\#n} \cdot CRR_{TAS\#n} + LinkDelay + \\ + CorrectionField_{TAS\#n-1} \end{aligned} \tag{6.6}$$

Applied in the case for TAS#1:

$$CorrectionField_{TAS\#1} = 20 \cdot \frac{10}{30} + 10 + 0 = 16.66 \tag{6.7}$$

Finally, the follow up message is sent to the TAS#2. The second TAS will repeat the same procedure as explained above. A special case is presented by the TAS#3. Since this TAS is the limit of the system, an endpoint device, it does not forward the time information. Therefore it does not create a Follow Up Message. TAS#3 consumes the received follow up information and associates this data to the generated timestamp when the sync event was received locally. With the contents of the Follow Up Message, any TAS is able to determine the GM time that corresponds to the moment of time in which the Sync Event is received locally and timestamped. In order to make this example more clear, the Follow Up Message related calculations for all the TAS in the network are shown in tables 6.2, 6.3, 6.4.

TAS#1 generated Follow Up Message	Values
Precise Origin Timestamp	A
Cumulative Rate Ratio	10/30
Correction Field	20 * 10/30 + 10 + 0 = 16.66

Table 6.3: TAS#1 generated Follow Up Message

TAS#2 generated Follow Up Message	Values
Precise Origin Timestamp	A
Cumulative Rate Ratio	10/30 * 20/10 = 20/30
Correction Field	5 * 20/30 + 10 + 16.66 = 29.99

Table 6.4: TAS#2 generated Follow Up Message

Details on the timing performance of the 802.1As can be found in [Garner&09]. For a more thorough description of the protocol refer to [Garner&11]. Do not read [IEEE-Std802.1As-2011] before having an overall understanding of [Garner&11]. The standard document is appropriate in order to find specific technical details on the protocol. On the other hand, the same document fails to give an overall simple explanation.

6.3. Partitioning decisions and modelling the AVB compliant device

In order to model an AVB compliant endpoint device, the following aspects have to be taken into considerations:

6.3.1 Protocol modelling

An AVB compliant device has to provide an implementation of the protocols presented in the previous section: the *traffic shaping protocol*, the *stream reservation protocol* and the *time synchronization protocol*. Both traffic shaping and stream reservation protocols are heavily based on operations over data queues. If the case under study is the SoS and network perspective, it makes more sense to build models by using *queue theory*. Such a modelling approach could provide a mathematically proven architecture with the optimized number of queues, frames per second and nodes in the network. In this sense, it is clear that VDM-RT is not the most promising modelling technology. Taking in consideration a single node in the network, the implementation of the traffic shaping and stream reservation protocols is reduced to a set of packet filtering operations over the hardware buffers. These are mostly bitwise comparisons over the buffers, that will determine when the Ethernet packets will be sent. Since both protocols are amendments to the widespread 802.1Q tagging protocol, they are available and integrated in COTS Ethernet interfaces. From the partitioning perspective it is clear that there are highly efficient hardware implementations in the market ready for system integration.

6.3.2 Real-time characteristics

Considering the real time perspective the time synchronization protocol is more challenging than traffic shaping and stream reservation protocols. The real time analysis possibilities offered by Overture will allow a coarse grained study of the system time performance. The most relevant issues to analyse for a slave endpoint device, considering time synchronization, are:

Introduced error at Sync time stamping: The sync event is time stamped when it is received by the endpoint AVB compliant device. The time difference between the reception of the Sync event and the point of time in which the time stamping takes place is the introduced time error in the device. It is desirable to introduce the least error possible. The error budget for a set of commercial WiFi speakers is set to +/- 10 microseconds [Stanton08].

Sync processing time: The performance of the device in processing sync events determines the maximum rate at which it can be receiving them. This limit should not be overridden, otherwise bandwidth will be wasted[1].

Note that not incorporating traffic shaping and stream reservation protocols in the time synchronization models will not have any impact on the obtained results. Time synchronization related traffic will always be treated as a high priority traffic, with access to the physical channel as soon as it is requested by its implementation. Therefore, it is fair to model the buses as dedicated point to point connections between the connected devices.

6.3.3 Partitioning factors to be considered by the models

The VDM-RT models will help on evaluating the **precision** offered by different architectures. This factor will be measured by the error introduced in the different designs. The lower the introduced time error the more precision the architecture is offering. The aim is to provide executable models that will act as enabling tool to study the **time feasibility of several architectures**. The different architectures will be composed by a variety of hardware and software components. It is worth to keep in mind that, as exposed in chapter 2, a certain partitioning decision will be accompanied by a set of trade-offs in terms of: **used silicon area**, **development time**, **flexibility of the design**, **scalability** and **price**.
It is the intention that the proposed VDM-RT model will help on finding the best compromise for a solution depending on the target application.

6.4. Modelling the AVB time synchronization protocol

6.4.1 Sequential model

The initial sequential model of the system gives the possibility of identifying the main entities involved in system operation. The main outcome of the sequential model is to represent the structure and the time independent behaviour of the system. This model is incorporating a very limited notion of time, represented as *timesteps*. Even thought the notion of time introduced in this model is not providing the mechanisms for timing analysis, it is sufficient in order to show the event ordering. Abstracting time details is beneficial in an initial modelling approach, since it allows the modeller to focus only on designing the system structure and distributing the

[1]In a poorly designed system, even overflows may occur.

responsibilities among the components. An additional outcome is that properties that should be exhibited by the system can be incorporating at this stage. Finally, the application of modelling gives the possibility of reaching a good understanding of the protocol. In the case of complex protocols, like the one considered in this section, the right model will be created after several attempts (guideline 2).
The sequential model of the system has served to detect the following entities in the time synchronization subsystem:

App: represents the time consuming application.

ClockSlave: represents the clock dependant on the time corrections received by the TAS. It provides the time information to the `App` entities.

SiteSync: represents the intermediate layer between the received information in the ports and the `ClockSlave` entity.

Port: represents an abstract entity that can be acting as a Master or as a Slave port depending on the device configuration.

Slave: represents a particular kind of port, used to received incoming time corrections.

Clock: represents the local system clock. It is used for timestamping and as local time reference for the TAS using it.

PhyMessage: represents the buffer receiving the sync event and the follow up message. This class acts as a proxy between the endpoint and the external devices (application of guideline 10).

Figure 6.6 shows the UML class diagram of the sequential model. The environment class has been omitted in order to make the diagram more readable. The information interface is represented by the `PhyMessage` class, which is associated to the abstract class Port. A concrete instance of the Port class should be used in order to manage the information exchange. The `Clock` class is associated to the Ports so a timestamp can be applied to the incoming messages. The class Slave is the concrete implementation used in this model. The Ports are managed by the `SiteSync` entity. The received time corrections are provided to the `ClockSlave` attached to the `SiteSync` entity. This clock will be responsible for forwarding the corrections to the applications requiring time information.
In order to reach a better understanding of the protocol, a generic deployment that involves bridges and endpoint devices has been modelled. In this deployment concrete implementations of the Port class, in the form of Master and Slave ports have been used. Figure 6.7 shows the object diagram of the created model. Even though the focus of this case study is the development of AVB compliant endpoints (TAS1 and TAS4 in figure 6.7), this deployment models as well the use of AVB compliant bridges (TAS2 and TAS3 in figure 6.7). Such a model does not bring additional information about how an endpoint device should be partitioned. However, it is possible to contextualize the operation of the endpoint device in a more concrete and generic scenario. Only after thoroughly understanding how the target system is interacting with the environment, makes sense to move forward towards the more focused modelling.

6.4.2 Concurrent model

The purpose behind creating a concurrent model of the system is the same as the one exposed in section 5.2.2. It is worth to mention that the notion of time in this model remains unaltered with respect to the sequential model. In order to represent the concurrent progression of time the

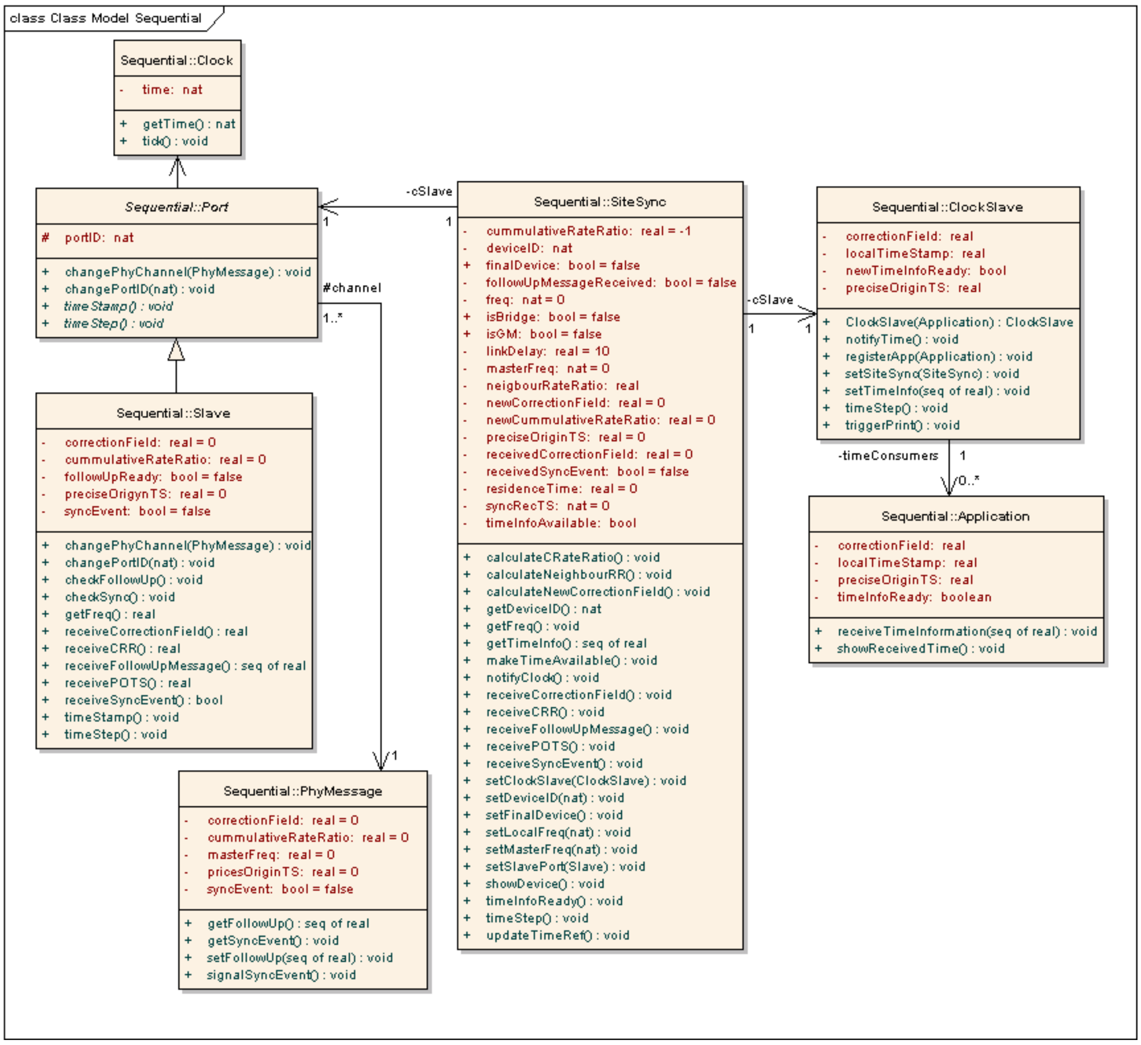

class Class Model Sequential
Sequential::Clock
- time: nat
+ getTime() : nat
+ tick() : void
Sequential::Port
# portID: nat
+ changePhyChannel(PhyMessage) : void
+ changePortID(nat) : void
+ timeStamp() : void
+ timeStep() : void
-cSlave
1
1
#channel
1..*
Sequential::SiteSync
- cummulativeRateRatio: real = -1
- deviceID: nat
+ finalDevice: bool = false
- followUpMessageReceived: bool = false
- freq: nat = 0
+ isBridge: bool = false
+ isGM: bool = false
- linkDelay: real = 10
- masterFreq: nat = 0
- neigbourRateRatio: real
- newCorrectionField: real = 0
- newCummulativeRateRatio: real = 0
- preciseOriginTS: real = 0
- receivedCorrectionField: real = 0
- receivedSyncEvent: bool = false
- residenceTime: real = 0
- syncRecTS: nat = 0
- timeInfoAvailable: bool
+ calculateCRateRatio() : void
+ calculateNeighbourRR() : void
+ calculateNewCorrectionField() : void
+ getDeviceID() : nat
+ getFreq() : void
+ getTimeInfo() : seq of real
+ makeTimeAvailable() : void
+ notifyClock() : void
+ receiveCorrectionField() : void
+ receiveCRR() : void
+ receiveFollowUpMessage() : void
+ receivePOTS() : void
+ receiveSyncEvent() : void
+ setClockSlave(ClockSlave) : void
+ setDeviceID(nat) : void
+ setFinalDevice() : void
+ setLocalFreq(nat) : void
+ setMasterFreq(nat) : void
+ setSlavePort(Slave) : void
+ showDevice() : void
+ timeInfoReady() : void
+ timeStep() : void
+ updateTimeRef() : void
-cSlave
1
1
Sequential::ClockSlave
- correctionField: real
- localTimeStamp: real
- newTimeInfoReady: bool
- preciseOriginTS: real
+ ClockSlave(Application) : ClockSlave
+ notifyTime() : void
+ registerApp(Application) : void
+ setSiteSync(SiteSync) : void
+ setTimeInfo(seq of real) : void
+ timeStep() : void
+ triggerPrint() : void
-timeConsumers
1
0..*
Sequential::Application
- correctionField: real
- localTimeStamp: real
- preciseOriginTS: real
- timeInfoReady: boolean
+ receiveTimeInformation(seq of real) : void
+ showReceivedTime() : void
Sequential::Slave
- correctionField: real = 0
- cummulativeRateRatio: real = 0
- followUpReady: bool = false
- preciseOrigynTS: real = 0
- syncEvent: bool = false
+ changePhyChannel(PhyMessage) : void
+ changePortID(nat) : void
+ checkFollowUp() : void
+ checkSync() : void
+ getFreq() : real
+ receiveCorrectionField() : real
+ receiveCRR() : real
+ receiveFollowUpMessage() : seq of real
+ receivePOTS() : real
+ receiveSyncEvent() : bool
+ timeStamp() : void
+ timeStep() : void
1
Sequential::PhyMessage
- correctionField: real = 0
- cummulativeRateRatio: real = 0
- masterFreq: real = 0
- pricesOriginTS: real = 0
- syncEvent: bool = false
+ getFollowUp() : seq of real
+ getSyncEvent() : void
+ setFollowUp(seq of real) : void
+ signalSyncEvent() : void

Figure 6.6: Sequential model UML class diagram.

timeStamp pattern has been applied. This pattern is implemented in the new class `TimeStamp` and used from the rest of the threads.

Figure 6.8 shows the class diagram of the concurrent version of the model. Attributes and methods have been omitted in order to increase readability. For additional details refer to figure 6.6. The stereotypes *«thread»* and *«synchronized»* are used to show active and concurrently accessed classes respectively.

6.4.3 Distributed real-time models

The distributed real time model gives he possibility of: A) Exploring different architectures and B) Analysing the real time deadlines accomplishment. Some changes are required before being able to carry out both studies. The first change has been the usage of the notion of time provided by Overture. The second change has been the addition of the Deployment class:

Deployment: defines the system infrastructure by the use of processing blocks (CPU) and communication interfaces (Buses). It contains the static class declarations of the modelled enti-

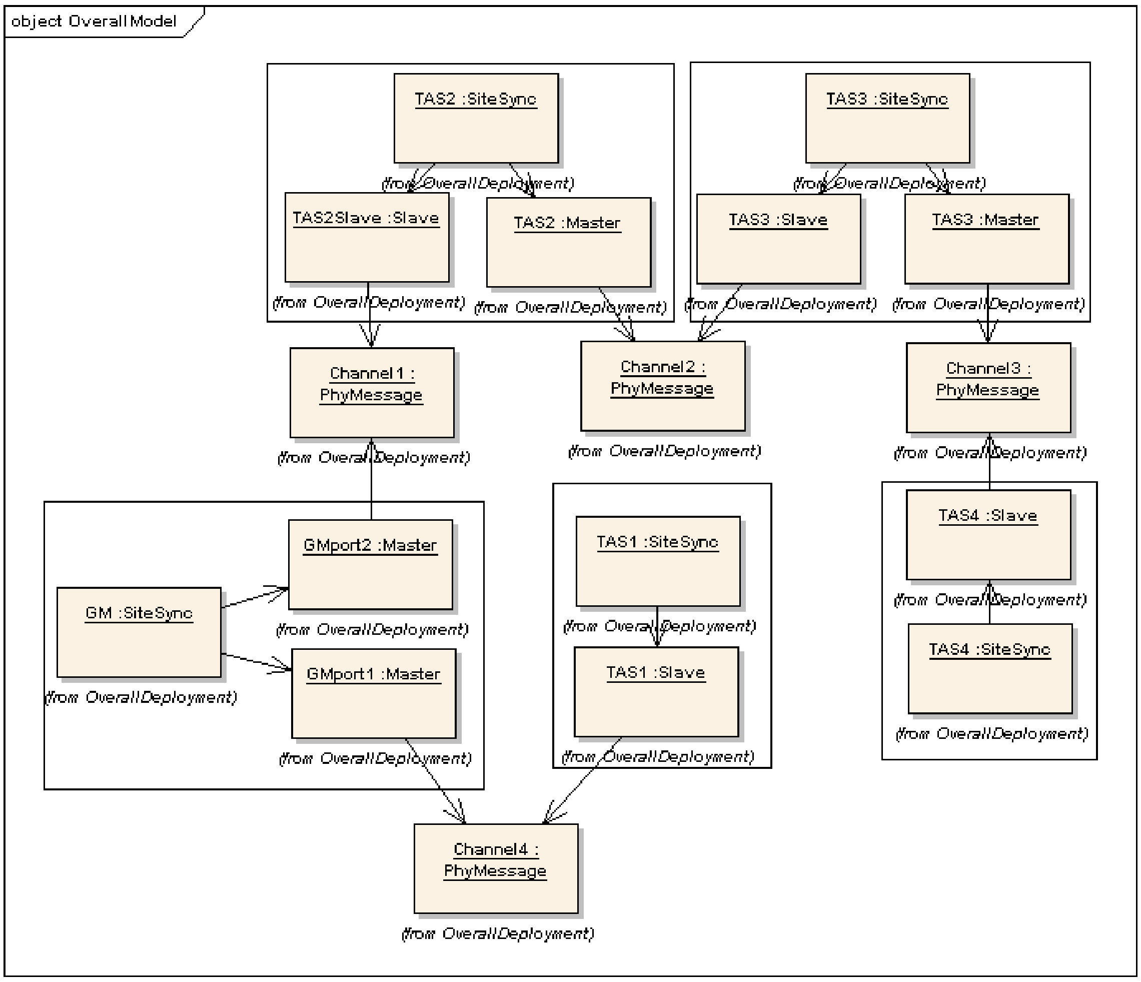


Figure 6.7: Context model UML object diagram.

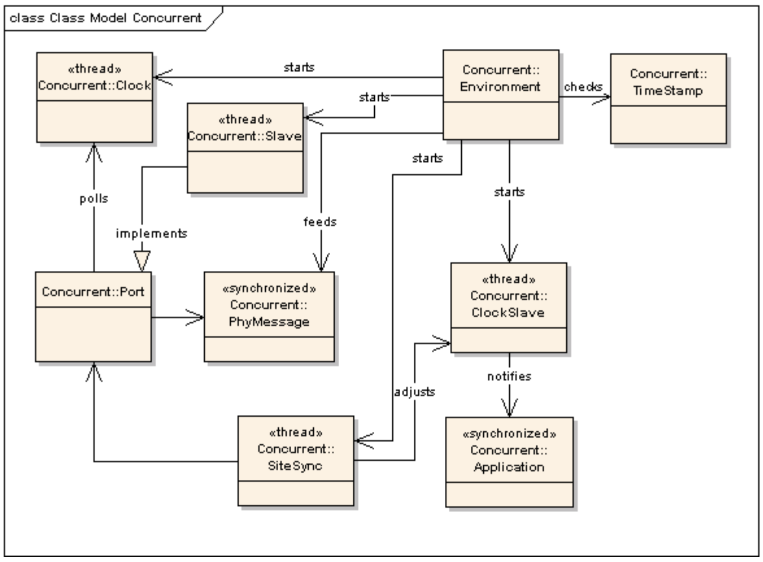

class Class Model Concurrent
«thread» Concurrent::Clock
Concurrent:: Environment
Concurrent:: TimeStamp
«thread» Concurrent::Slave
Concurrent::Port
«synchronized» Concurrent:: PhyMessage
«thread» Concurrent:: ClockSlave
«thread» Concurrent:: SiteSync
«synchronized» Concurrent:: Application
starts
checks
polls
implements
feeds
adjusts
notifies

Figure 6.8: Concurrent model UML class diagram.

ties. This class is as well responsible for deploying the modelled entities in the processing units.

Figure 6.9 shows a class diagram modelling the main entities of the model and the existing relations among them.

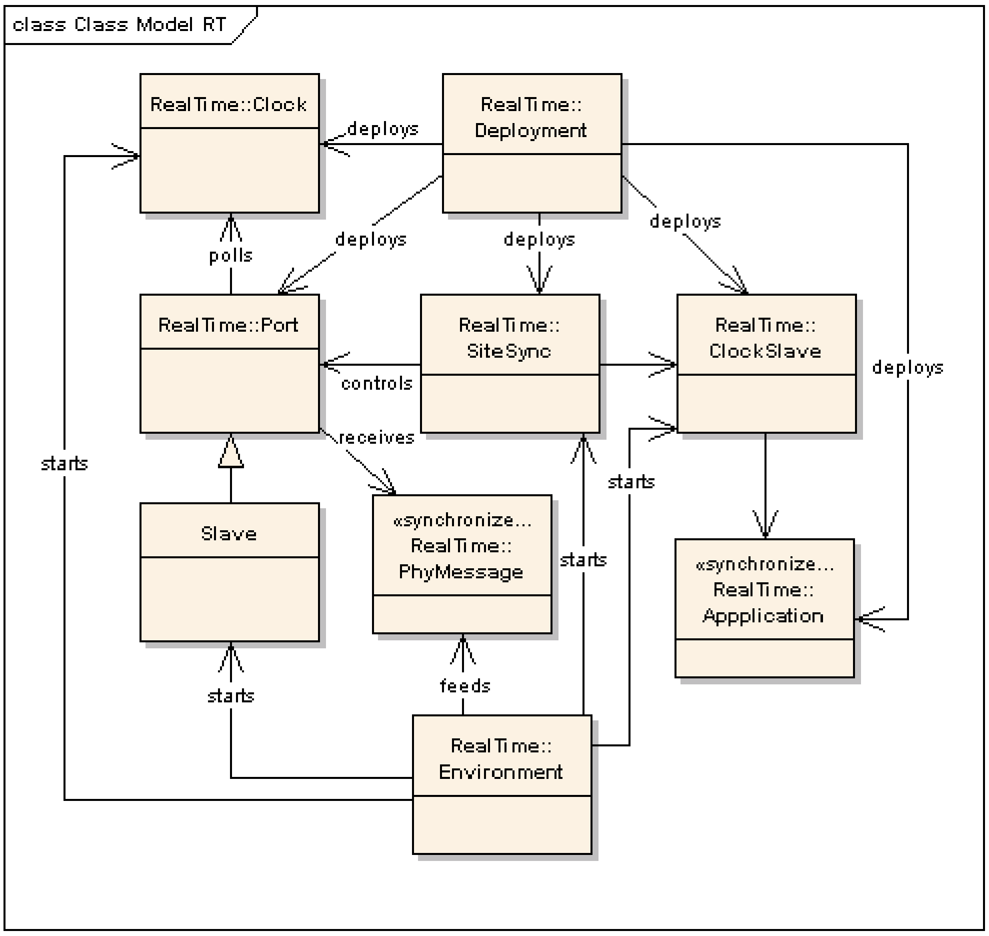


Figure 6.9: Real-Time model UML class diagram.

6.5. Evaluating different hardware/software architectures in a simulated deployment

Four different architectures have been evaluated during the exploration phase. In order to study the relevant timing aspects, log sequences have been created (application of guideline 12). In order to get the relevant values of time during model execution, the `time` instruction has been used (guideline 7).

6.5.1 Architecture 1: highly software based architecture

The first considered architecture is highly software based. Two CPUs have been used communicated through a hi-speed bus, modelling the available hi speed for hardware access. The access to the channel is running on the CPU representing the hardware partition. This implies that the PhyMessage channel representing the hardware buffer for incoming time corrections will be deployed in hardware. A UML deployment diagram representing this architecture is shown in figure 6.10.

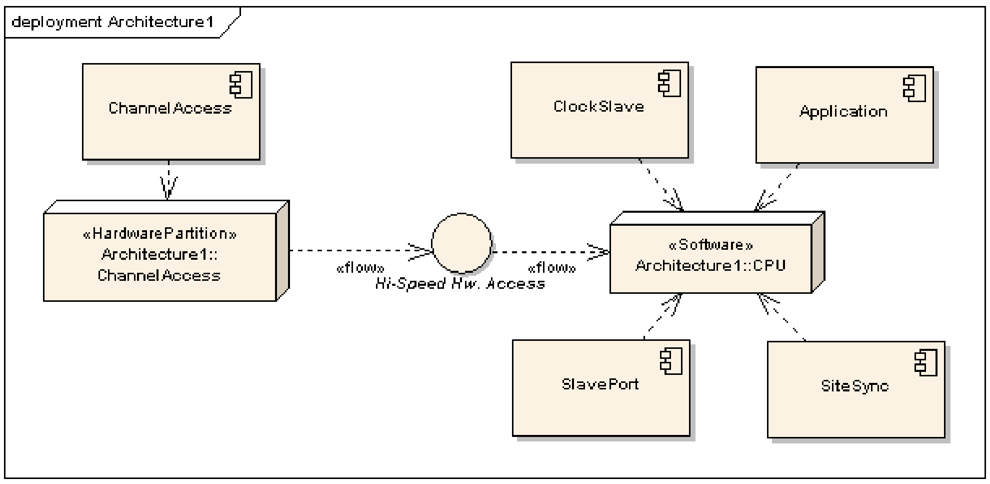


Figure 6.10: Deployment diagram for the highly software based architecture.

Model execution reveals that there is a considerable delay on the time stamping process under this architecture. This model has been analysed with CPUs of different frequencies. In the first case, a CPU running at a frequency of 1E7 Hz has been used. In this execution, the sync event was received at the hardware buffers 14ns after the simulation started. The same Sync event was timestamped after 4604 nanoseconds had elapsed. This constitutes a total delay of 4590 ns. In the second scenario a CPU running at a frequency of 10E6 Hz has been selected. The Sync event was still received at 14 ns, since the conditions of the environment did not change. On the other hand, the Sync event was time stamped after 46004 ns were elapsed, producing an error of 45990 ns. It can be conclude that such an implementation would be introducing an considerable delay of orders of magnitude in the synchronization process. Furthermore, this error would be depending on the speed of the used CPU and/or on its load at the moment of time in which the sync event was received.
Additionally, the maximum rate at which the endpoint device can receive time corrections is considerable reduced by this delay. If two sync events are received with a separation of 45 microseconds considering a cpu of 1Mhz, they will be timestamped at the same time therefore, the time information will not be valid[2].

6.5.2 Architecture 2: mostly software and offloaded SlavePort

The second architecture is making use of three different CPUs. The first CPU is modelling a hi-performance hardware partition. This hardware partition is communicated through a hi-speed bus to a second CPU. This intermediate is communicated to the final one through a low-speed bus. CPUs one and two are modelling regular general purpose processors. The channel access and the slave port have been deployed in the hardware partition. The synchronization logic SiteSync and ClockSlave is running in he intermediate CPU. Final time correction are made available to the third CPU through the Low-speed software interface. A UML deployment diagram representing this architecture is shown in figure 6.11.

[2]Unless this situation is properly identified and the second packet filtered out.

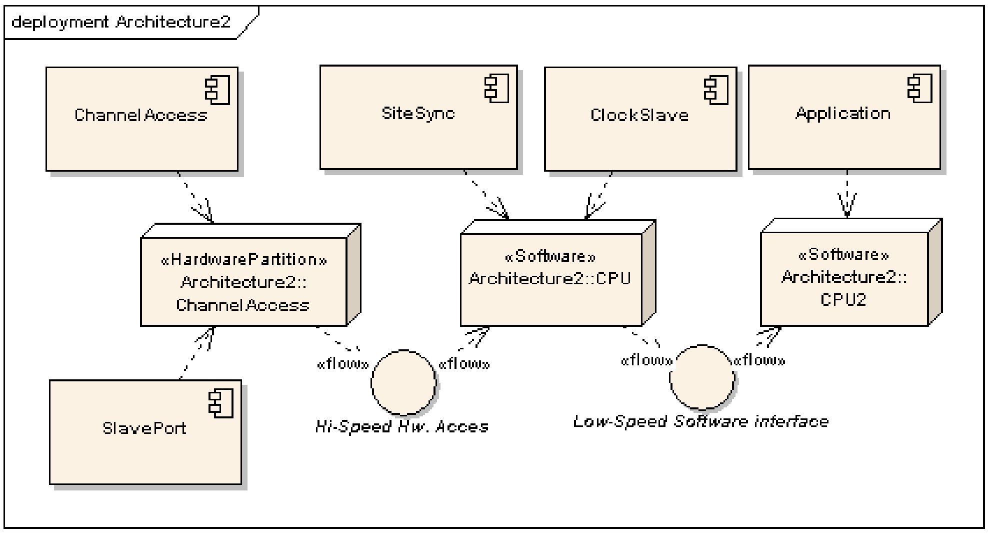

deployment Architecture2
ChannelAccess
SiteSync
ClockSlave
Application
«HardwarePartition» Architecture2:: ChannelAccess
«Software» Architecture2::CPU
«Software» Architecture2:: CPU2
«flow»
«flow»
«flow»
«flow»
SlavePort
Hi-Speed Hw. Acces
Low-Speed Software interface

Figure 6.11: Deployment diagram for the mostly software with offloaded SlavePort architecture.

This architecture turns out to be much more accurate if compared with the first one. Model execution reveals that the introduced error in the time stamping is minimum. The model has been executed under two different configurations. The first configuration is the one shown in figure 6.11. In this case the Sync event is received after 28ns. The hardware is timestamping the sync event at 34ns. The total introduced error is 6ns. This result is very positive if compared with the one obtained by architecture 1. In order to explore a more cost-effective option, a second variant of architecture 2 is proposed. In this variant, the second CPU is not going to be used; the time consuming application is going to be deployed together with SiteSync and ClockSlave in the same CPU, running all of them as software components. The results are exactly the same as the ones obtained in architecture 2 configuration 1. This definitely supports the following thesis: A) The most critical and time sensitive process is the Sync event time-stamping, and B) The delay is considerably mitigated if the time-stamping process is carried out in the processing unit that is buffering the network traffic. The sync interval supported by architecture 2 is orders of magnitude above the one supported by architecture 1, and could be applied in very hi-speed transmission mediums such as optical fibres.

6.5.3 Architecture 3: offloaded SiteSync and SlavePort

The third architecture is keeping the access to the channel together with the slave port access in the hardware partition. The SiteSync logic has been allocated as well in the same hardware partition. A hi-speed bus is used to communicate the hardware partition with the second CPU. The second CPU is a general purpose processor running ClockSlave and Application as software components. A UML deployment diagram representing this architecture is shown in figure 6.12.

The purpose of this model is to explore how the endpoint device structure could be expanded to give support to more complex functionality, like the one that could be offered by a network bridge. In this case it has been decided to deploy the site sync entity in hardware. The prime focus of the model is to evaluate if there is a penalty in the time corrections reception while keeping the possibility of a more scalable design. The mode has shown that, the sync event is received 48 ns after the simulation has started, and this sync event is not timestamped until 74 ns have passed. This leads to a delay of 26 ns. The conclusions that can be drawn in this case are a little bit fuzzy. While it makes sense to consider hardware partitions as very high speed VDM-RT CPUs, it does

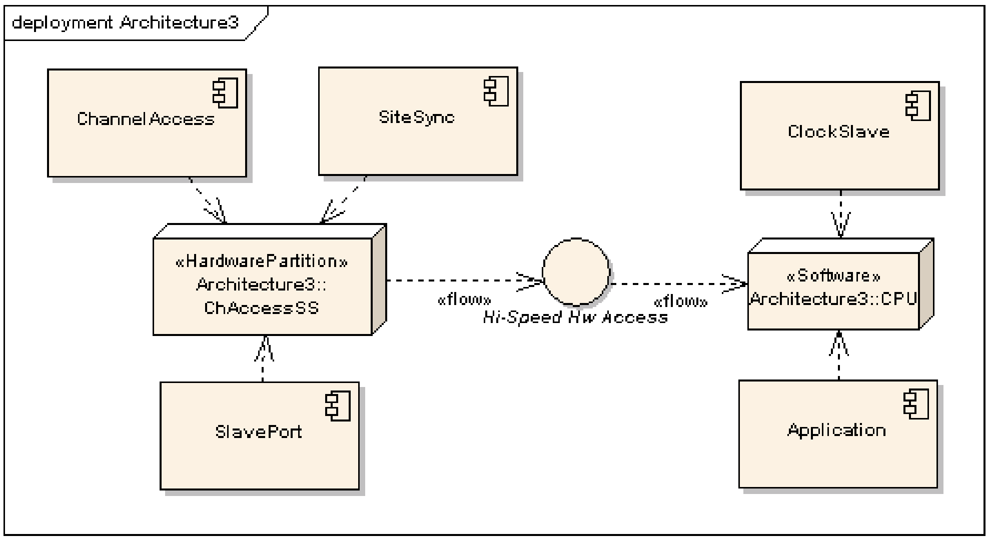


Figure 6.12: Deployment diagram for offloaded SiteSync and SlavePort architecture.

not make sense to use a single high speed CPU to deploy two potential hardware components on it. The reason behind is that active components are modelled as active classes, running as threads. These threads are going to be sharing computation time of the VDM-RT CPU. In the actual implementation, separate hardware partitions that can run in parallel will be implemented as purely separated hardware units, that will not be sharing "computation time".

6.5.4 Architecture 4: multiple hardware partitions

The fourth architecture is considering an improved hardware architecture. The goal is to improve the results obtained from architecture 3. In this case the first hardware partition is running the `SlavePort` and the `ChannelAccess`. This partition is communicated with a hi-speed bus with the second hardware partition, where `SiteSync` has been deployed. A second hi-speed bus has been used to communicate with the `ClockSlave` and the time consuming application. A UML deployment diagram representing this architecture is shown in figure 6.13.

This fourth architecture aims to model correctly the scenario proposed in architecture 3, a more scalable implementation with good time performance. In order to illustrate separate hardware units, two different hardware partitions have been used (application of guideline 9). Architecture 2 proved that the time stamping should be done at the hardware level, in the same block in which the buffers are located, so this design decision is going to be respected. Following the intent of architecture 3, site sync is going to be moved to hardware. Site sync is going to be able to access the hardware ports through the hi-speed hardware interface explained above, therefore the communication latency is cut down below 3 ns. Model execution showed that, the sync event is received at 28 ns and timestamped at 34 ns. This is keeping the 6 ns delay presented by the architecture 2. Site sync is able to retrieve the time corrections and execute the communication logic. In the case this architecture for the endpoint device would have to be used as a base to implement a compliant bridge, able to forwared the time correction making use of master ports, this architecture would be correct. In the case the focus is to create a compliant endpoint device, this architecture would require additional development effort and very little positive impact (if any).

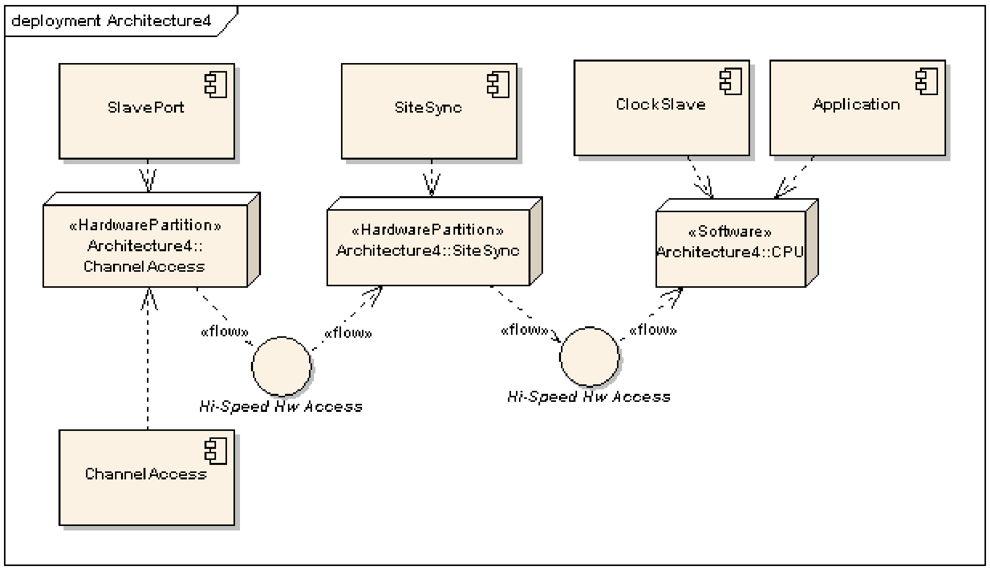


Figure 6.13: Deployment diagram for the multiple hardware partition architecture.

6.6. Model outcome assessment

The created models have been created much faster than the time it would have taking to produce partial prototypes of all four architectures before selecting the most appropriate for final implementation. The models have helped the system engineer to evaluate rapidly different architectures and, furthermore, to conduct analysis that will help him on deciding which factors will be improved among different architectures (such as scalability or possibility of further improvements). Besides the help on taking technical decisions, the application of modelling has bridged the mathematical description of the time synchronization protocol with a model that can be relatively easily understood by other professionals involved in system design.
Considering the time it takes to create a model like the one presented in this case study with regard the advantages it brings to the development process, the relation between invested resources/model outcome can be qualified, objectively, as very high.

6.7. Summary

This chapter has presented the application of the VDM-RT methodology for Hardware/Software co-design of embedded systems in a second case study, the development of an AVB endpoint device. This second case study is more complex than the introductory case study about servo control. The problem under study is more abstract as well, since the AVB case is at a system level, where the critical partitioning decisions have to be determined after initial study of the system. The proposed methodology have had a very positive impact on the problem understanding, helping the modeller on detecting the potential bottlenecks and critical aspects related to a problem unknown in advance. This has proved that the proposed approach not only performs well under "toy examples", but in real industrial cases as well. In the case this information would be presented to a systems engineer responsible for the AVB development project, he would be able to select which architectural candidate is the best one for the project needs. Furthermore, this selection would be based on solid information obtained by the application of a sound engineering approach.

Chapter 7

Concluding remarks and future work

This chapter is composed of five sections. A brief review of the achieved results will be given in section 7.1. A more detailed assessment of the proposed methodology is presented in section 7.2. Future work and a review of further research lines in the application of the VDM-RT technology in Hardware/Software Co-design of embedded systems is presented in section 7.3. Reflections on the personal learning outcomes are presented in section 7.4 Final remarks are given in section 7.5.

7.1. Achieved results

The initial goals of this thesis, as presented in chapter 1 were:

1. **To propose a general-purpose methodology based in the VDM-RT modelling language in order to support the design decisions in Hardware/Software co-design field.**

2. **To propose improvements and extensions to the VDM-RT language and the modelling tool Overture, seeking to contribute on the creation of a modelling tool that can be applied in a more specialized field.**

Before being able to propose a new methodology for Hardware/Software co-design, it was important to study and review the state of the art in this field. Chapters 2 and 3 present a review of modelling languages and current approaches relevant for the development of Hardware/Software systems. Special emphasis was made on the System Level Design approach to Hardware/Software co-design. The modelling languages were evaluated with regard to the criterion presented by [Shaout&09], [Gajski&09], [Niemann98] and [Huang&04]. This technology and methodology review demonstrated that there is a need for a language and a methodology that enables the system engineer to perform: precise descriptions at a high abstraction level and allow rapid evaluation of different alternative architectural candidates in order to find the most appropriate hardware/software architecture.

The approach followed to accomplish the first goal is composed by two subgoals: the proposal of the methodology itself and the validation of the proposed methodology through its application in two case studies. In the three subsections below we assess our work in those two subgoals and the second goal of the thesis.

7.1.1 Development of a VDM-RT based methodology for Hardware/Software partitioning

Chapter 4 has proposed a new methodology that can support the hardware/software co-design process. The proposed methodology provides a reference for making partitioning decisions. These decisions are based on the accomplishment of real-time deadlines. Even though making partitioning decisions based on time constraints might be enough in some cases, there are other situations in which additional criterion shall be consider. As described in chapter 2, energy efficiency, delay at the hardware level or material costs are factors that should be taken into account in hardware/-software co-design. The proposed methodology fails on considering these factors directly. This can be compensated by using other modelling technologies to extract information in those areas and consider it together with the architectures selected from the VDM-RT models. One of the strong points of the proposed methodology is the possibility of evaluating different hardware/software partitions with minimum changes on the models. On the other hand, the results provided by this exploration phase have to be used carefully and a certain tolerance factor should be used (additional details will be provided in section 7.2). Finally, the proposed methodology fails when approaching some localized partitioning problems. An example of a problem hard to tackle by this methodology is the signal processing field. This limitation is introduced by the language VDM-RT since it lacks of the necessary mathematical support in order to model signal processing algorithms. As it has been described above, the proposed approach in this case is to apply an heterogeneous modelling approach.
Additional assessment of this subgoal is presented in section 7.2.

7.1.2 Application in two case studies

The proposed methodology has been applied in two case studies with the purpose of validating it (presented in chapters 5 and 6). The case studies have shown that the application of the VDM-RT methodology for Hardware/Software co-design is providing relevant and sound information that can be used in the partitioning process. Case one, the development of a servo controller showed that this methodology can work on small cases. Predictions and results matched and it was shown that the methodology performed well under this scenario. Case two, the development of an AVB endpoint device shown that partitioning information can be extracted in complex problems as well. However, we cannot conclude that the partitioning decisions taken in that case were correct since we have not been able to produce the implementation for the four architectural candidates. Note that this would have implied producing and testing hundreds of thousands of VHDL code. The practical application of the methodology has shown that the usage of modelling in complex projects presents a high Return On Investment (ROI). The creation of models takes a considerable amount of time, but this initial investment can potentially be recovered later on in the development process. Additional assessment of this subgoal is presented in section 7.2

It can be concluded that we have accomplished goal one and reached the initial objectives set by its two subgoals. Once we proposed the methodology and after we applied it in the two case studies, we detected the need for concrete improvements for the VDM technology.

7.1.3 Proposal of improvements to the VDM technology

Chapter 4 has proposed a number of improvements to the VDM-RT language and the modelling tool Overture. Some of the proposed changes would be beneficial for making partitioning decisions with the proposed VDM-RT methodology.

The addition to *models of new platforms* would allow to conduct a more precise functionality performance evaluation. The concrete precision improvements would depend on the platform under study and they are complex to predict without studying particular cases. The *profiling of a model* can be already done by manually extracting the relevant information from the Overture log files. In some situations it is complicated to identify and to locate this information. The analysis that can be conducted with model profile information can lead to make partitioning decisions, hence the relevance of it. Modellers would have easier and clearer access to the information they need to make partitioning decisions if an structured and formatted execution report would be generated. Incorporating *annotations for real-time deadline evaluation* can have a very positive impact in the partitioning process. Some systems present complex time requirements that might depend on multiple factors. Trying to analyse all these factors by using log files every time the model is executed can be time consuming. Using annotations in order to describe time-critical aspects helps on formalizing the required time behaviour. Thanks to this formalization it is possible to automatically evaluate the real-time performance of the considered architectural candidates. An additional advantage is that this an step towards automatic design space exploration. An *improved bus support* would help on incorporating some relevant communication details. The improved bus support would rise the expressiveness of the models and it would simplify the communications modelling by eliminating the proxy pattern. Additional assessment on communication modelling is provided in section 7.2.

Automatic variable monitoring would make it easier to analyse the evolution of a certain variable over time. Like annotations for real-time deadline evaluation, this feature would be a step forward towards automatic design space exploration. Automatic variable monitoring would make the modelling and analysis process less complex but it would not improve the quality of the partitioning decisions.

The addition of *code generation* is complex to implement in an efficient manner. Considering that this methodology is targeting the development of embedded systems optimization with regard to speed and code size is a requirement. While mapping VDM++ models to C++ code (or another Object Oriented language) is something that could be explored and partially implemented in a Master's thesis, doing the same with VHDL would be more difficult. The reason resides in the existing abstraction gap between VDM and VHDL. Even though code generation would allow a smooth progression from the modelling to the implementation stage, it would have very little positive impact (if any) on making partitioning decisions from VDM-RT models. This makes code generation optional in this context.

The proposed improvements are worthwhile considering from the user perspective. Incorporating models of new platforms and improving the bus support would allow the user to extract more precise and reliable information, gaining confidence in the methodology proposed. The incorporation of annotations for real-time deadline evaluation would give the user the possibility of carrying out more complex analysis in a faster way, increasing his performance considerably. Finally, model profiling and automatic variable monitoring would make the methodology more usable and it would eliminate the need for external scripts and tools.

It can be concluded that we have accomplished goal two, not only by just presenting a features wishlist but by describing a set of improvements, the rationale for its consideration and how this methodology can benefit from them. Additional assessment of this subgoal is presented in section 7.2

7.2. Assessment of the VDM-RT methodology for Hardware/Software co-design

This thesis has made extensive use of abstract models in order to represent the combination of hardware and software in the development of systems. More precisely, the models have been used to extract information to support the hardware/software partitioning process. During the process of creating these abstract models, some details have not been considered, since they have been left out intentionally by the modeller. Other details have been left out by the modelling technology itself. Being aware of what has been lost during this formalization is required in order to move forward in the development process, bridging the gap between modelling and implementation. A weakness of this methodology is that how to bridge this gap might not be obvious in all cases. Some of these problematic situations are introduced by the notion of time used in VDM-RT, the VDM-RT kernel and the modelling of communication.

7.2.1 The notion of time

The VDM-RT language assumes that every CPU deployed in a model is perfectly synchronized with regard to the rest. Considering a single FPGA system, such an assumption can be accepted. In the case the problem under study is a distributed real-time system, it might be necessary to consider additional synchronization mechanisms in the implementation stage. Some implementation technologies provide abstraction layers taking care of the time synchronization so, depending on the used technology, engineers might not need to produce the time synchronization implementation. Note that these synchronization problems cannot be captured by the VDM-RT models and therefore this is a case in which the new methodology does not support the systems engineer. This is a limitation introduced by the modelling technology since VDM-RT is targeting the "big picture" of the system rather than time synchronization in a distributed environment. Further details and mechanisms to compensate this problem are described in [Babaoglu&93].

7.2.2 Time precision

A word of caution has to be given about the precision of the time predictions offered by the VDM-RT models. As it has been described above, the VDM-RT system models are coarse-grained. The data these models are providing for analysis is good enough to take co-design decisions by comparison. If this comparison is performed between figures provided by the model, no additional precautions are needed. In the case this comparison is performed between figures provided by the abstract model and figures provided from the real world, a tolerance of at least one order of magnitude should be introduced. This might be imprecise in some situations, but these models are created at a high level and are not targeting very fine-tuned implementation details. As described in section 3.3, the System C modelling library incorporates a notion of time that can reach the femtosecond level, orders of magnitude below the nanosecond level reached in VDM-RT. Furthermore, this notion of time is directly connected with the notion of time used in VDHL. Making use of SystemC can be a good alternative to reach a higher degree of precision in the models. The price to pay for this precision: less abstract and complex representations if compared to VDM-RT models.

7.2.3 The RTOS abstraction

The VDM-RT kernel is modelling a Real-Time Operating System layer. This layer provides the mechanisms to manage different types of threads and synchronization predicates like mutexes and history counters. These mechanisms are implemented in many different ways by RTOS in the market. VDM-RT uses a different paradigm for synchronization and protection of critical sections. For example, the functionality offered by history counters in VDM-RT might not be explicitly available in a certain RTOS, but surely a different mechanism will be provided in order to accomplish the same task. This relation holds in the other way. Some of the features provided by RTOS implementations are not present in VDM-RT by default. For instance, critical sections are present in many RTOS implementations but not in VDM-RT. Mechanisms like counting semaphores have to explicitly modelled in VDM-RT if they are going to be used in the models, while they are available by default in RTOS implementations. The system engineer has to be aware about the possible implementation options offered by different technologies while mapping a model to a real implementation. Besides these implementation details, time performance of different RTOS implementations needs to be taken into account. Under different RTOS it might take a different amount of time to perform a context switch, to spawn a thread or to acquire a lock. VDM-RT is introducing its own timing characteristics in those aspects. Again, these differences have to be carefully evaluated when moving to the final implementation.

7.2.4 Modelling communication

Communication between systems and subsystems is one of the problematic issues in system design. The suggested methodology is focused on the functionalities, more precisely on the real-time performance of the system functionalities. In order to evaluate the functionality the communication can be partially modelled. The proposed approach could be used to determine the time-frame and approximate boundaries for the communication facilities required by the modelled functionality. If the problem under study is focused on the communication performance, this methodology will most likely be not appropriate to address it. This shortcoming is introduced by the representation of the communication link by BUSes, which might not be precise enough. Some high-level aspects need to be incorporated in the BUS representation, like the bus load or possible traffic losses. Other aspects are very difficult to capture in VDM and therefore it does not make sense to consider them. Communication depends heavily on physical phenomena, like noise, temperature, distance or wireless propagation. These factors are entirely abstracted away in VDM-RT, even though they are relevant in communication modelling, it is difficult to analyse them from the proposed methodology. As seen in [Nielsen10b] VDM-RT as it is at the moment is not ideal to model dynamic architectures with wireless devices. As a conclusion regarding the proposed methodology and the communication problem, the modeller must be aware of the root purpose of the analysis conducted on the model, the consequences and the final impact in the system functionality. By doing so, many communication aspects can be considered with regard to the created models even though they have not been fully detailed.

7.2.5 Code generation

Code generation has been described as one of the desirable tool features from which this methodology could benefit. Considering that the code is generated from models that are not depicting the whole system in detail, it is not possible to obtain a final implementation automatically. The problematic details discussed above regarding time, concrete RTOS behaviour and communication would have to be very carefully studied, making sure that the assumptions taken are respected at the implementation level. It is important to keep in mind that the final product is going to be implemented by using technologies different from the paradigms used in the models. The differences between the implementation and the modelling technologies, have to be considered in order to overcome the gap between the abstract model and the real world.

7.2.6 Application in two case studies

The new suggested methodology has only been applied in two case studies, and compared to final implementations in only one. The results look promising but, to determine if such a methodology would work in general, it should be applied in more cases (preferably of different nature) and have other people to test it. This assessment cannot be carried out in one MSc. thesis, so therefore we can conclude that the results of the suggested methodology look promising but a wider analysis in terms of different projects, professionals and longer time is needed to determine the limits of its capabilities.

The servo controller case was conceptually and technically simple, and it made it possible to show in practice how the methodology could be applied. Thanks to this simplicity it was possible to go down to the implementation level and compare the predictions with actual implementation results. The AVB case was much more challenging than the servo case from a technical point of view. The protocols used in AVB were complex and reaching a good understanding was not an easy task. The application of models helped us on going through these problems, since they forced us to be systematic and to evaluate which details where worth considering in the partitioning problem. The models created made it possible to select among several different architectural candidates without having to perform an actual implementation. Due to time limitations an implementation of the AVB endpoint was not carried out. Like in the servo case, an implementation would have made it possible to relate predictions with actual results. However, an AVB implementation can take up to several VHDL thousands of LOC and is not feasible for a Master's thesis of 6 man-month. It is difficult to provide an accurate estimate of the time it would take to produce functional implementations for the four architectural candidates we have considered in this work. The development time can be shortened if we implement a limited architecture aiming to validate the models. These limited implementations, even though not fully functional AVB devices, could provide a preliminary structure for further development. The real-time features studied in the models could be implemented and deployed in these preliminary hardware/software stubs. If we follow this strategy, we could create the four implementations and support the model conclusions. This kind of implementation considered alone is expected to have a workload comparable to this MSc. thesis project.

Considering the size of the models, the servo case has been modelled with only 500 LOC in VDM-RT. On the other hand, the final implementations have required 260 LOC in C for the software only solution and 1195 LOC in VHDL and 382 LOC in C for the hardware/software solution. This shows that only 500 lines of VDM have been enough to evaluate architectural implementations with a total size of 1738 LOC of varying complexity. In the AVB case study the size of the VDM-RT model is 1083 LOC. This model is presenting how the time synchronization protocol is handled by the device in four different architectures, some of them using different configurations. The AVB protocol implementation provided by Xilinx[1] is composed by 55000 VHDL LOC. Note that the architectural pillars of this solution have been tackled from a VDM-RT model of 1083 LOC. Considering the complexity of the protocol and the number of evaluated architectures it would be fair to conclude that it is an expressive model. It would be difficult to extract the same kind of information from a SystemC, SysML or Matlab/Simulink model of similar size.
Finally, it is worth to remark that the servo case models were created at a much lower level of abstraction than the AVB protocol models. The servo case was focused on a very concrete opera-

[1]Xilinx produces an IP core implementing the AVB protocol. Additional information can be found in [XilinxAVB11]

tional detail. This explains why there is such a small difference in size between VDM-RT models and final implementation.

7.2.7 Comparison with other methodologies

Chapter 3 presented different methodologies for the development of Hardware/Software systems. It is worth to assess how the proposed methodology performs in comparison to them. The methodology proposed by Xilinx is at an operation level and can hardly cope with complex partitioning decisions. The methodology based on Matlab/Simulik is strong in modelling problems involving heavy mathematical computations, for example signal processing problems. However, it fails when considering problems that would benefit from being described at a higher level of abstraction. The Saturn methodology, a combined SysML-SystemC approach, exploits co-simulation possibilities bridging the gap between modelling and implementation. However, it might be too close to the implementation level. The SHE methodology is close to the methodology proposed in this thesis. The SHE methodology is using the modelling language POOSL instead of VDM-RT. The expressive power of POOSL is comparable to VDM-RT and it presents many other similarities in terms of constructs. Additional effort has been spent in the development of predictable control software able to cope with real-time deadlines, a feature that is missing in VDM-RT. There has been no new information in the POOSL/SHE area since year 2007 and the state of the project is unknown. Even though the SHE/POOSL methodology is interesting, taking into account its current development state, it can only be considered as a good source of inspiration for further developments in VDM-RT.

Compared to the Matlab/Simulink approach, the methodology proposed in this thesis is more abstract and far from concrete implementation technologies. However, it cannot offer the same performance on problems involving complex mathematics. The Saturn approach is better in virtual prototyping, but limited to concrete platforms. In comparison, the proposed methodology is platform independent. The VDM-RT based methodology is strong in high level design space exploration but further effort is required to relate the design alternatives considered at this level with the technologies uses in the implementation phase.

7.3. Future Work

This thesis work has given the possibility of detecting interesting areas and concrete action points that could be explored in further developments. Below suggestions for these are provided.

7.3.1 Implementation of the AVB endpoint device proposed architectures

In chapter 6 four different architectural candidates were modelled. These models provided information relevant for the partitioning process and helped on selecting the optimal candidate for implementation. Due to time constraints and complexity of the problem, no early prototype generation was performed. Possible further work could be the implementation of the four architectures discussed. Once the implementation would be complete, it would be interesting to validate the information extracted from the VDM-RT models against the prototypes created. This would reinforce the validity of the proposed methodology, by demonstrating its applicability in a complex case from the design phase down to the implementation level.

7.3.2 Guidelines for the transformation of VDM-RT to SystemC

SystemC is one of the most used languages in the modelling and design of hardware/software systems. There are tools available in the market that allow the generation of C/C++ code for software components and VHDL code for hardware components. SystemC could act as a bridge between abstract VDM models and the final implementation.
Besides partially solving the of lack of VDHL code generation from the VDM modelling environments, SystemC will bring a stepwise evolution towards the final implementation. This would result in an intermediate modelling stage in SystemC, that will allow to perform modelling at a lower abstraction level. Such a low level model would possibly enable the incorporation of more low level details in the case they are needed for more concrete architectural evaluation. On the other hand this level would be less abstract and would require more modelling time. Aspects that were not detailed at the VDM level would have to be incorporated at the SystemC level. This would require additional modelling time and the complexity of the representation of the system will increase.
A systematic approach would be required in order to move the VDM models to SystemC models. The proposal of such an approach, and the study of the equivalences between the represented entities in VDM and the SystemC constructs should be studied in detail in further work.

7.3.3 Exploiting further the possibilities offered by Overture and VDM-Tools

Both Overture and VDM-Tools present features that make creating higher quality models easier. These possibilities were not explored during the development of the case studies. Some of these features are combinatorial testing, proof obligations and model coverage. The application of combinatorial testing allows to evaluate the created models against all possible combinations of input values, performing a thorough testing of the model. By ensuring that the model is performing properly under all kind of inputs from the environment we are reducing modelling errors. The application of proof obligations make sure that the error sources introduced by the way the model has been created are fixed. This improves the overall robustness of the model. Finally, model coverage indicates to which extent the model has been evaluated. A coverage of 100% is obtained in case all the parts of the model have been executed at least once. It is desirable to have the highest percentage of coverage possible, implying that most of the model has been covered under different runs. By verifying and testing the model by the means exposed above, confidence in the used method for taking the partitioning decisions is increased.

7.3.4 Applying a heterogeneous modelling approach based on the VDM-RT language, 20-sim and SysML

As discussed in this thesis, only a combination of several modelling technologies can produce a complete description of the system. In the same way, a complete analysis of the system can only be achieved by using multiple modelling technologies. Following this idea, the proposed extension to this work is the incorporation of additional information in terms of interaction with external systems and physical world. New modelling languages should be used in order to described these specialized areas, the languages SysML and 20-sim. A link between these modelling languages and the VDM methodology would be required in order to fully exploit the possibilities of this heterogeneous approach.
The projects [COMPASS11] and [DESTECS09] are providing tools linking SysML and 20-sim with the modelling language VDM. Additionally, these projects incorporate specific facilities to conduct analysis in the logical and physical world. Such an analysis would be very difficult

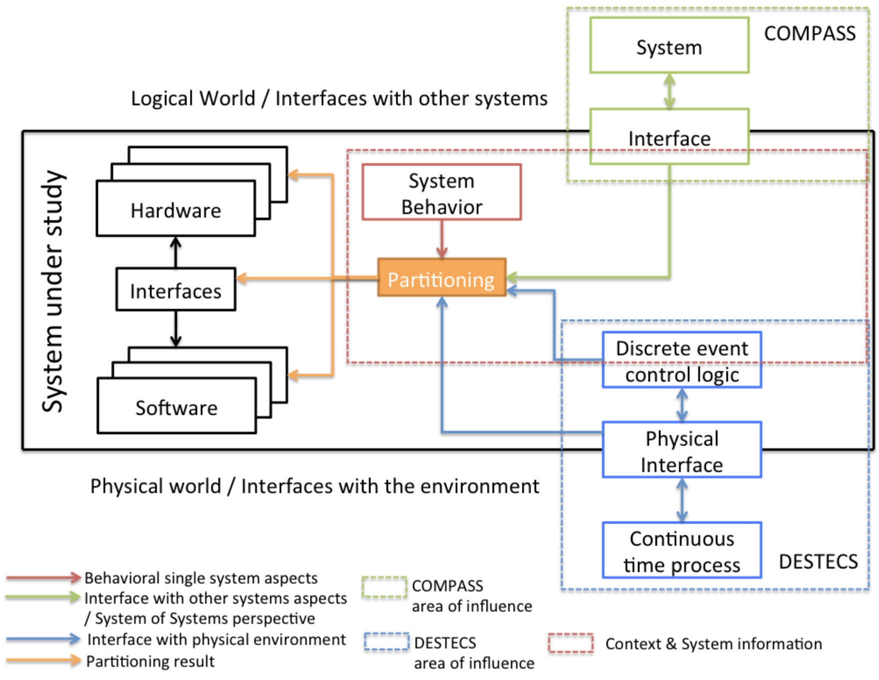

COMPASS
System
Interface
Logical World / Interfaces with other systems
System under study
Hardware
Interfaces
Software
System Behavior
Partitioning
Discrete event control logic
Physical Interface
Continuous time process
DESTECS
Physical world / Interfaces with the environment
Behavioral single system aspects
Interface with other systems aspects / System of Systems perspective
Interface with physical environment
Partitioning result
COMPASS area of influence
DESTECS area of influence
Context & System information

Figure 7.1: Heterogeneous modelling approach.

to be achieved by a VDM-only approach. Integrating the models created under COMPASS and DESTECS in the partitioning process would relate the implications of certain system architectures to the environment in which the system would be operating.
Figure 7.1 shows the proposed relation between the COMPASS and DESTECS projects and the partitioning process.

7.4. Personal Learning Outcomes

As described above, one of the case studies was based on the Audio Video Bridging protocol. Before modelling an AVB compliant device it was a necessary to understand the protocols behind it and the hardware and software internals of an AVB device. This was specially complicated and it took me weeks of reading, meetings, and several e-mails with hardware manufacturers. AVB has been specified in 2011 therefore the documentation available aside from the standards is reduced to a few academic papers. One of the mistakes I made while trying to understand the protocols behind AVB was to start with the 450 pages standard, trying to get every single detail on it. After two weeks following this track I decided to read the few available papers about AVB instead and revisit the standard right after that. This turned out to be a good approach. The papers gave me the overall picture I needed to relate the details presented in the standard. Surprisingly, this kind of "overall picture" was not part of the standard that was focused on operational details. The papers allowed me to start creating an overall VDM model of the system. I continued expanding this initial model with the relevant details extracted from the standard documentation. As a result of this modelling stage, I personally think that the executable VDM model I have created expresses better how AVB works than 400 pages of text. The conclusion that I can extract from this experience is that the next time I will face the problem of understanding a complex system or protocol I will read the scientific papers before delving into the standard documentation.

FPGAs are the most relevant platforms in the Hardware/Software Co-design field. My experience with this technology was very limited when I started this MSc. thesis. Conducting this work gave me the opportunity to learn more about them, mostly by reading papers, doing small experiments with FPGAs (not included in this thesis work since they are not related to it) and following a 4-day course organized by the Xilinx FPGA manufacturer. Even though I am not an expert on FPGAs I can say that I have a better understanding of its internals, possibilities and limitations. This gives me the basic knowledge to judge if they are a good technological alternative in the case I have to design an embedded system in the future.

The competences acquired in the courses "Model-Driven Development" and "Hardware/Software co-design" have been complemented with more in-depth new learning outcomes. In the model-driven development course we were introduced to modelling and the VDM language, but we did not analyze in detail the information obtained from models. One of the main pillars of this thesis has been to study and extract information from models, therefore, I have improved my skills and understanding of how modelling can be used in practical cases. In the Hardware/Software co-design course we were introduced to informal models, graphical notations and methodologies for Hardware/Software systems development. In this thesis I have reviewed these modelling technologies and methodologies, compared them and identified their strong and weak points. This has given me the opportunity to exercise critical thinking and analytical skills in a technical context.

Additionally, the process of creating this thesis has been an starting point in the process of becoming better in writing and communicating technical ideas. After finishing this thesis I see that one of the most complex parts of writing is being selective, analysing the contribution of part of the text to the global argument presented by the thesis. This implies to decide whether certain information has to be written as part of the main body, incorporated in appendixes or only cited.

It is the first time during my academic life in which I have had to create an argument of this size and support it properly. If I compare this MSc. thesis with my previous work I can clearly see a difference between them. My Bachelor's thesis was very product-oriented, therefore its report was a descriptive document focused on technical details. During my Master's degree some of the courses I have followed have required the creation of projects with an additional level of judgement, comparison and reflection. I now feel for the first time that I understand what doing research involves. Finally, this thesis has required a more mature approach, better planning and organization and to relate my work with current research work.

7.5. Final Remarks

This thesis has successfully approached the initial objective of developing a VDM-RT based methodology that can help on taking partitioning decisions in the Hardware/Software Co-design of embedded systems. This initial objective was a challenge, given the abstraction gap between a VDM representation and a final implementation involving the development of hardware. The development of the proposed methodology has been justified and compared against the existing approaches, showing that there is a place for a VDM-RT based approach in the development of Hardware/Software systems.
The application of the proposed methodology in two case studies has shown that the presented thesis can provide solid arguments for a particular candidate architecture. The proposed additions

to the tools and the language can improve the performance of the proposed VDM-RT methodology considerably for hardware/software co-design.
Some of the additions could be developed under new student projects or MSc. thesis projects. The application of formal methods and modelling in the Hardware/Software Co-design field is a topic that still needs further research and development. This thesis has opened a new way in which this research could continue.

References

[Amos&02] S. W. Amos and R. S. Amos. *N*ewnes Dictionary of Electronics. Newnes, fourth edition, 2002. 389 pages. [cited at p. 113, 114, 115]

[Babaoglu&93] Özalp Babaoglu and Keith Marzullo. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*. Technical Report UBLCS-93-1, University of Bologna, Piazza di Porta S. Donato, 5, 40127 Bologna (Italy), January 1993. [cited at p. 94]

[Black&04] David C. Black and Jack Donovan. *S*ystemC: From The Gourd Up. Springer, first edition, 2004. 244 pages. [cited at p. ix, 1, 6, 10, 19, 20, 26, 31]

[Broenink&10] Broenink, J. F. and Larsen, P. G. and Verhoef, M. and Kleijn, C. and Jovanovic, D. and Pierce, K. and Wouters F. Design Support and Tooling for Dependable Embedded Control Software. In *P*roceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems, pages 77–82, ACM, April 2010. [cited at p. 70]

[Brogioli&06] Michael Brogioli and Predrag Tadosavljevic and Joseph R. Carvallaro. Hardware/Software Co-design Methodology and DSP/FPGA Partitioning: A Case Study for Meeting Real-Time Processing Deadlines in 3.5G Movile Receivers. 2006. 5 pages. [cited at p. 17]

[Brooks86] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *I*nformation Processing, 7, 1986. 1069-1075 pages. [cited at p. 6]

[Bruel&98] Jean-Michel Bruel. Integrating Formal and Informal Specification Techniques. Why? How? In *P*roceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, pages 50–, IEEE Computer Society, Washington, DC, USA, 1998. [cited at p. 24, 28]

[Burns&09] Alan Burns and Andy Wellings. *R*eal-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX. Addison-Wesley, fourth edition, 2009. 602 pages. [cited at p. 31]

[Chen&09] Sao-Jie Chen and Guang-Huei Lin and Pao-Ann Hsiung and Yu-Hen Hu. *H*ardware Software Co-Design of a Multimedia SOC Platform. Springer, 2009. 151 pages. [cited at p. 16, 19, 26]

[Cofer&06] R.C. Cofer and Benjamin F. Harding. *R*apid System Prototyping with FPGAs: Accelerating the design process. *E*mbedded Technology, Elsevier Inc., Linacre House, Jordan Hill, Oxford OX2 8DP, UK, 2006. 301 pages. [cited at p. 114, 115, 116]

[COMPASS11] Comprehensive Modelling for Advanced Systems of Systems. 2011. http://www.compass-research.eu/. [cited at p. 5, 98]

[DESTECS09] DESTECS (Design Support and Tooling for Embedded Control Software). European Research Project, June 2009. http://destecs.org. [cited at p. 5, 98]

[Edwards&97] Stephen Edwards and Luciano Lavagno and Edward A. Lee and Alverto Sangiovanni-Vicentelli. Design of Embedded Systems: Formal Models, Validation and Synthesis. *P*roceedings of the IEEE, 85(3):366–390, March 1997. 22 pages. . [cited at p. 255]

[Fitzgerald&05] John Fitzgerald and Peter Gorm Larsen and Paul Mukherjee and Nico Plat and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [cited at p. 30]

[Gajski&00] D. D. Gajski and J. Zhu and A. Gerstlauer and S. Zhao. *S*pecC, Specification Language and Design Methodology. Springer, 2000. 336 pages. [cited at p. 13]

[Gajski&09] Daniel D. Gajski and Samar Abdi and Andreas Gerstlauer and Gunar Schirner. *Embedded System Design, Modelling Synthesis and Verification*. Springer, 2009. 347 pages. [cited at p. ix, 1, 10, 11, 12, 19, 20, 23, 24, 26, 91]

[Garner&07] Garner, G.M. and Feifei Feng and den Hollander, K. and Hongkyu Jeong and Byungsuk Kim and Byoung-Joon Lee and Tae-Chul Jung and Jinoo Joung. IEEE 802.1 AVB and Its Application in Carrier-Grade Ethernet [Standards Topics]. *C*ommunications Magazine, IEEE, 45(12):126 –134, December 2007. 8 pages. [cited at p. x, 74, 75]

[Garner&09] Garner, G.M. and Gelter, A. and Teener, M.J. New simulation and test results for IEEE 802.1AS timing performance. In , editor, *P*recision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on, pages 1–7, IEEE, IEEE, Los Alamitos, CA, USA, October 2009. 7 pages. [cited at p. 80]

[Garner&11] Garner, G.M. and Hyunsurk Ryu. Synchronization of audio/video bridging networks using IEEE 802.1AS. *C*ommunications Magazine, IEEE, 49(2):140–147, February 2011. 7 pages. . [cited at p. 80]

[Huang&04] Jinfeng Huang and Jeroen Voeten and Andre Ventevogel and Leo van Bokhoven. *Platform-independent Design for Embedded Real-Time Systems*, chapter 1, pages 35–50. Kluwer Academic Publishers, Norwell, MA, USA, 2004. [cited at p. 4, 14, 23, 91]

[IEEE-std1722-2011] The Institute of Electrical and Electronics Engingeers, Inc. *I*EEE Standard for Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks. Technical Report, IEEE, NY, USA, May 2011. 45 pages. [cited at p. x, 73, 74]

[IEEE-Std802.1As-2011] The Institute of Electrical and Electronics Engingeers, Inc. *I*EEE Standards for Timing and Syncrhonization for Time-Sensitive Applications in Bridged Local Area Networks – Local and Metropolitan Area Networks. Technical Report, IEEE, NY, USA, 2011. 274 pages. [cited at p. x, 75, 76, 77, 80]

[IEEE-Std802.1Qat-2010] The Institute of Electrical and Electronics Engingeers, Inc. *I*EEE Standards for Virtual Bridged Area Networks. Amendment 14: Stream Reservation Protocol (SRP) – Local and Metropolitan Area Networks. Technical Report, IEEE, NY, USA, 2010. 101 pages. [cited at p. 75]

[IEEE-Std802.1Qav-2009] The Institute of Electrical and Electronics Engingeers, Inc. *I*EEE Standards for Virtual Bridged Area Networks. Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams – Local and Metropolitan Area Networks. Technical Report, IEEE, NY, USA, 2009. 71 pages. [cited at p. 74]

[INCOSEseh10] International Council on Systems Engineering. *S*ystems Engineering Handbook. A guide for system life cycle processes and activities. Technical Report, INCOSE, 7670 Opportunity Rf, Suite 220 San Diego, CA, January 2010. [cited at p. 9]

[ISOVDM96short]	P. G. Larsen and B. S. Hansen and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. December 1996. International Standard ISO/IEC 13817-1. [cited at p. 29]
[Kramer07]	Jeff Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007. [cited at p. 10]
[Larsen&09]	Peter Gorm Larsen and John Fitzgerald and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *I*ntl. Journal of Software and Informatics, 3(2-3), October 2009. [cited at p. 4, 5, 41]
[Larsen&10a]	Peter Gorm Larsen, Sune Wolff, Nick Battle, John Fitzgerald and Ken Pierce. *D*evelopment Process of Distributed Embedded Systems using VDM. Technical Report TR-2010-02, The Overture Open Source Initiative, April 2010. [cited at p. ix, 37, 39]
[Larsen&10b]	Peter Gorm Larsen and Nick Battle and Miguel Ferreira and John Fitzgerald and Kenneth Lausdahl and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *A*CM Software Engineering Notes, 35(1):, January 2010. 6 pages. [cited at p. 5, 30]
[Larsen&10c]	Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle. *The VDM-10 Language Manual*. Technical Report TR-2010-06, The Overture Open Source Initiative, April 2010. [cited at p. 30]
[Lausdahl&10b]	Kenneth Lausdahl and Peter Gorm Larsen. *V*DM-RT Scheduling in Overture. April 2009. 16 pages. Model driven development using VDM++ and UML II. [cited at p. ix, 4, 5, 6, 32, 33, 43, 55]
[Matlab]	MathWorks. http://www.mathworks.com. October 2011. Matlab official website. [cited at p. 27]
[Mischkalla&10]	Mischkalla, Fabian and He, Da and Mueller, Wolfgang. Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems. In *P*roceedings of the Conference on Design, Automation and Test in Europe, pages 1201–1206, European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2010. 6 pages. [cited at p. 5, 35]
[Mueller&10]	Mueller, W. and Da He and Mischkalla, F. and Wegele, A. and Whiston, P. and Penandil, P. and Villar, E. and Mitas, N. and Kritharidis, D. and Azcarate, F. and Carballeda, M. The SATURN Approach to SysML-Based HW/SW Codesign. In *V*LSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on, pages 506 –511, IEEE, Los Alamitos, CA, USA, July 2010. 5 pages.

[cited at p. ix, 5, 35, 36]

[NASAseh10] National Aeronautics and Space Administration. *S*ystems Engineering Handbook. Technical Report, NASA, NASA Headquarters. Whasington, D.C. 20546, December 2007. [cited at p. 9]

[Nielsen10a] Claus Ballegaard Nielsen. *Dynamic Reconfiguration of Distributed Systems in VDM-RT*. Master's thesis, Aarhus University, December 2010. [cited at p. 30]

[Nielsen10b] Claus Ballegaard Nielsen. Towards Dynamic Reconfiguration of Distributed Systems in VDM-RT. In *S*emantic Issues in VDM: a BCS-FACS and Overture Workshop, September 2010. [cited at p. 95]

[Niemann98] Ralf Niemann. *H*ardware/software co-design for data flow dominated embedded systems. Springer, 1998. 244 pages. [cited at p. 13, 91]

[Ou&04] Jingzhao Ou and Viktor K. Prasanna. Rapid energy estimation of computations on FPGA based soft processors. In *I*EEE Intern. SoC Conf, IEEE Computer Society, Washington, DC, USA, 2004. 4 pages. [cited at p. 5, 35]

[Ou&05] Jingzhao Ou and Viktor K. Prasanna. MATLAB/Simulink Based Hardware/Software Co-Simulation for Designing Using FPGA Configured Soft Processors. In *P*roceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04, pages 148.2–, IEEE Computer Society, Washington, DC, USA, 2005. [cited at p. ix, 5, 35]

[Patel&04] Hire D. Patel and Sandeep K. Dhukla. *S*ystemC Kernel Extensions for Heterogeneous System Modelling. A Framework for Multi-MoC Modeling and Simulation. Kluwer Academic Publishers, P.O. Box 17, 3300 AA Dordrecht, The Netherlands, 2004. 172 pages. [cited at p. 10, 24, 26]

[Perry02] Douglas L. Perry. *V*HDL: Programming by Example. McGraw Hill, fourth edition, 2002. 476 pages. [cited at p. 31]

[PIC10F320] Microhip Technology Inc. *P*IC10(L)F320/322 Data Sheet. 6/8-Pin, High-Performance, Flash Microcontrollers. Preliminary DS41585A, Microhip Technology Inc., 2355 West Chandler Blvd.Chandler, AZ 85224-6199, 2011. 210 pages. [cited at p. 22]

[Plat&92] Nico Plat and Hans Toetenel. *A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax*. Technical Report 92-07, Delft University, March 1992. 75 pages. [cited at p. 29]

[POOSL] SHE/POOSL Website. http://www.es.ele.tue.nl/she/index.php?select=32. October 2011. . [cited at p. ix, 28, 29, 37]

[Prevostini&07] Mauro Prevostini and Elena Zamsa. SysML Profile for SoC Design and SystemC Transformation. May 2007. 7 pages. [cited at p. 5, 35]

[Ribeiro&11] Augusto Ribeiro and Kenneth Lausdahl and Peter Gorm Larsen. Run-Time Validation of Timing Constraints for VDM-RT Models. In *9*th Overture Workshop, June 2011, Limerick, Ireland, 2011. [cited at p. 49]

[Robert&10] Thomas Robert and Vincent Perrier. *C*oFluent Methodology for UML. UML SysML MARTE Flow for CoFluent Studio. White Paper, CoFluent Design, 2010. 19 pages. [cited at p. 27]

[Rosenberg&10] Doug Rosenberg and Sam Mancarella. *Embedded Systems Development using SysML*. Report, Sparx Systems Ltd and ICONIX, 2010. pages. . [cited at p. 25, 26]

[Schaumont10] Patrick R. Schaumont. *A* Practical Introduction to Hardware/Software Codesign. Springer, 2010. 396 pages. [cited at p. ix, 15, 16, 17, 19, 21]

[Shaout&09] Adnan Shaout and Ali H. El-Mouse and Khalid Mattar. Specification and Modelling of HW/SW CO-Design for Heterogenious Embedded Systems. July 2009. [cited at p. 1, 12, 13, 15, 19, 24, 91]

[Simulink] MathWorks. http://www.mathworks.com/. October 2011. Simulink official website. [cited at p. x, 5, 27, 259, 260]

[Speedway11EDK] Introduction to MicroBlaze Hardware Development. Featuring ISE Embedded Design Suite 13. Speedway course material, May 2011. 112 pages. [cited at p. ix, 5, 34]

[Speedway11SDK] *D*eveloping MicroBlaze Software with Xilinx SDK. Avnet Inc., August 2011. 113 pages. Speedway course material. [cited at p. 4, 63]

[Stanton08] Kevin B. Stanton. *8*02.1AS Tutorial. Presentation, IEEE, November 2008. [cited at p. 81]

[Sudhakar05] Sudahakar Yalamanchili. *V*HDL A starters guide. Pearson Prentice Hall, Upper Salle Reiver NJ 07458, second edition edition, 2005. 241 pages. [cited at p. 10]

[SysML1.2] *Systems Modeling Language (SysML) Specification*. Technical Report Version 1.2, SysML Modelling team, June 2010. 236 pages. http://www.sysml.org/docs/specs/OMGSysML-v1.2-10-06-02.pdf. [cited at p. ix, 25]

[vanderPutten&07] Petrus Henricus Antonius van der Putten and Jeroen Peter Marie Voeten. *S*pecification of Reactive Hardware/Software Systems. The method Software/Hardware Engineering. PhD thesis, Technical University of Eindhoven, May 1997. 479 pages. [cited at p. ix, 38]

[VDMTools] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007. pages. . [cited at p. 5, 30]

[Verhoef05] Marcel Verhoef. On the Use of VDM++ for Specifying Real-Time Systems. *P*roc. First Overture workshop, November 2005. [cited at p. 4, 30]

[Verhoef08] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. [cited at p. 30]ISBN 978-90-9023705-3

[Waddington&06] Daniel Waddington and Patrick Lardieri. Model-Centric Software Development. *I*EEE Computer, 39(2):28–29, February 2006. 2 pages. . [cited at p. 1, 255]

[Wolf03] Wolf, Wayne. A Decade of Hardware/Software Codesign. 36:38–43, April 2003. [cited at p. 2, 15, 21, 23, 27, 255]

[XilinxAVB11] Xilinx. *LogiCore IP Ethernet AVB Endpoint v3.1*. DS677, Xilinx, March 2011. pages. [cited at p. 96]

[XilinxToolFlow11] Xilinx. Xilinx Tool Flow. Speedway course material., 2011. 22 pages. [cited at p. 5, 34]

Appendices

Appendix A

Terminology

$^{\mathcal{T}}$ASIC

Application Specific Integrated Circuit. Generic integrated circuit that has been tailored for a specific application. A single ASIC might replace several conventional logic integrated circuits, allowing a reduction in both circuit board size and power consumption. [Amos&02].

$^{\mathcal{T}}$ASIP

Application Specific Instruction Set Processor. Processors that present a partially reconfigurable Instruction Set Architecture. This allows tuning and optimization of the the processor for certain applications.

$^{\mathcal{T}}$AVB

Audio Video Bridging refers to a set of standards developed by the IEEE 802.1 AVB Task Group. It provides the specifications that will allow time-synchronized lo latency streaming services through IEEE 802 networks.

$^{\mathcal{T}}$BCAM

Bus Cycle Accurate Model. Model illustrates the communication aspects of the system at the clock cycle level.

$^{\mathcal{T}}$BFM

Bus Functional Model. Model that illustrates the communication aspects of the system incorporating no or an approximate notion of time.

$^{\mathcal{T}}$CAM

Cycle Accurate Model. Model that illustrates the communication and functional aspects of the system at the clock cycle level.

$^{\mathcal{T}}$CCAM

Computational Cycle Accurate Model. Model that illustrates the computational aspects of the system at the clock cycle level.

$^{\mathcal{T}}$CLC

Configurable Logic Cell. Reduced programmable silicon area that can implemented functionality at the hardware level defined by the end user. CLC are currently integrated in some commercial microcontrollers enabling the developers the implementation of very simple logical functions on them.

^{T}CMOS
Complementary Metal-Oxide Semiconductor. A for of construction of logic monolithic integrated circuits using complementary insulated-gate field-effect transistors in pairs. [Amos&02]

^{T}COTS
Commercial Off-The-Shelf. Common term to refer to the commercial availability of a certain product or component.

^{T}CSV
Comma Separated Value. Format in which several values are separated by a combination of commas and line breaks.

^{T}DUT
Design Under Test. Solution that is being subject of testing activities.

^{T}EDA tool
Electronic Design Automation Tool. Software tool used to support the electronic design activities, automating repetitive process involved in the design and speeding up the development process.

^{T}Golden Model
Model that is used as a reference during the whole development process

^{T}GPIO
General Purpose Input Output. Interface between a processor and external hardware. No specific usage or application is specified, and it can be generally accessed from custom defined logic in order to generate control signals.

^{T}IP core
Intellectual Property core. Hardware block implemented by a third party and COTS available. Thoroughly tested and ready for integration in new systems.

^{T}ISA
Instruction Set Architecture. Layer between the processor micro-architecture and the software logic. The ISA defines the operations that are available for software execution, as well as other hardware based facilities like available register, memory addressing or interrupt handlers.

^{T}Firmcore
IP block implementing functionality via HDL. It has been optimized for a specific target technology. The implementation optimization may be physical layout aware, allowing a highly efficient implementation resulting in improved performance, power or area characteristics. [Cofer&06]. An example of a firmcore block could be a NIOS or Microblaze processors, optimized for Altera and Xilinx platforms respectively.

^{T}FPGA
Field Programmable Gate Array is an integrated circuit that can be re-configured by the end user. Its hardware can be implemented by using hardware description languages, like VHDL or Verilog.

^{T}Hardcore
IP block implementing functionality at the hardware level. The functionality has been created in fixed-logic at the gate and signal route level rather than within the programmable

FPGA logic fabric. The functionality is fixed at the silicon level during the manufacture fo the device. The functionality cannot be removed or modified by the design team. [Cofer&06]. A example of a hardcore block could be an Atmel 8-bit microcontroller.

^{T}**Hardware/Software Co-design**
System Level Design methodology that pursues the derivation of architecture from behaviour and the parallel and cooperative development of hardware and software.

^{T}**MBSE**
Model-Based System Engineering. System engineering approach in which models are guiding the development process, supporting engineering trade-offs evaluation and design decissions.

^{T}**Partitioning**
process/stage of the project in which it is being decided whether a certain functionality should be implemented as a hardware or as a software block.

^{T}**POOSL**
Parallel Object-Oriented Specification Language. Formal language that supports the modelling of concurrent solutions and Object-Oriented techniques. It is used the Software Hardware Engineering methodology.

^{T}**PSoC**
Programmable System-on-Chip. User configurable hardware platform, that allows the incorporation of a number of analog and digital hardware blocks provided by the PSoC manufacturer. Hardware blocks are used by the integrated processor in the PSoC, which can execute a user defined software logic.

^{T}**PWM**
Pulse Width Modulation. Also known as Pulse Duration Modulation. Modulation technique in which the duration of the pulses in a pulse carrier is made to vary in accordance with the instantaneous value of the modulating signal. [Amos&02]

^{T}**RTC**
Real Time Clock. Electronic or electro-mechanical device able to measure physical time.

^{T}**RTL**
Register Transfer Level. Lowest abstraction level used in hardware design. It describes the system by the use of the hardware modelling language VDHL. It enables the developer the performance of very detailed modelling and analysis activities. Due to its low level of abstraction it is one of the most complex representations that can be used in system design.

^{T}**RTOS**
Real Time Operating System. Operating system that offers the possibility of coordinating processes and operations according to physical time limitations.

^{T}**SAM**
System Architectural Model. High level abstraction model of a system. It is normally created at the beginning of the project development activities, and serve as a base model from which more refined TLM models are created.

^{T}SDK

Software Development Kit. Collection of software components that enables the development of additional software.

^{T}SL

Specification Language. Formal or graphical language that allow the representation of system requirements.

^{T}SLD

System Level Design. Engineering approach used in the creation of complex systems. SLD pursues starting the system development at the highest abstraction level possible, as a way to tackle complexity. SLD current approaches are: Component Based Design, Platform Based Design and Hardware/Software Co-Design.

^{T}SM

Specification Model. Model created using a Specification Language, with the purpose of presenting unambiguously the requirements that apply on a certain system.

^{T}Softcore

IP block implementing functionality via HDL with no or minimal optimization for a specific target technology. [Cofer&06]. An example of a softcore could be an ARM core, that can be deployed in several FPGA platforms.

^{T}SysML

System Modelling Language. Graphical modelling language developed by the Object Management Group. It is based on UML and it provides additional constructs for the representation of common problems in the systems engineering domain. SysML is not restricted to hardware or software applications, and it can represents a variety of system components.

^{T}TLM

Transaction Level Model. System model that incorporates no or approximate notion of time. It can detail functional or communication aspects or both. It is an intermediate representation between the SAM models and the CAM models

^{T}UML

Unified Modelling Language. Graphical modelling language developed by the Object Management Group. Widely applied in the software industry, provides constructs to represent object-oriented solutions. Representations can be describing architecture, behaviour and interaction between different software units.

^{T}V model

or V-life cycle. Model of a general development process, that shows the existing relationship between analysis, design and implementation and validation and verification phases.

^{T}VDM

Vienna Development Method. Formal method enabling the modelling of systems. It can be used along all the phases described by the V model. The VDM method is currently supporting three languages: VDM-SL, VDM++ and VDM-RT. These languages cover the specification, object oriented design and real-time modelling respectively.

^{T}VHDL

Very-high-speed Hardware Description Language. Low level language used for the creation of hardware components. The implementation of these hardware components can be carried out in different platforms (FPGAs, ASICs, ...).

Appendix B

Servo case study VDM models

This appendix contains the VDM models for the servo case study. Section B.1 presents the sequential VDM models. Section B.2 presents the concurrent VDM models. Finally, sections B.12, B.13, B.14 present the produced VDM-RT models.

B.1. Sequential VDM model

B.1.1 Clock

```

class Clock

instance variables

 private tickNum : nat := 0;

operations

public tick : () ==> ()
tick() ==
(
 tickNum := tickNum +1;
);

public getSimTime: () ==> nat
getSimTime() ==
 return tickNum;


end Clock
```

B.1.2 Controller

```

class Controller

instance variables

  private pwmGenerator : PWMgenerator;
  private outputInterface : gpioInterface;

operations

public Controller: gpioInterface * PWMgenerator  ==> Controller
Controller (gpio,p) ==
(
  outputInterface := gpio;
  -- Create a position logger
  pwmGenerator := p;
);

public getOutputInterface: () ==> gpioInterface
getOutputInterface() ==
 return outputInterface;

-- Operations running in the controller
public timeStep: nat ==> ()
timeStep(time) ==
(
 skip;
);

end Controller
```

B.1.3 Environment

```

class Environment

instance variables

  clk : Clock;
  simTimeMax : nat := 1200;
  currentSimTime : nat := 0;

  outputInterface : gpioInterface;
  controller : Controller;

  sampler1 : Sampler;
  posLog : PositionLogger;

  pwmUnit : PWMgenerator;

operations

public run : () ==> ()
run() ==
(
  -- ### Simulation setup
  -- Create a clock
  clk := new Clock();

  -- Define a microcontroller

  posLog := new PositionLogger();

  -- Attach the gpio generating the PWM signal to the environment
  outputInterface := new gpioInterface(2);
  -- Create a PWMgenerator and attach it to the output
  pwmUnit := new PWMgenerator(outputInterface,1,posLog);

  controller := new Controller(outputInterface,pwmUnit);
  -- Set the initial pulse duration to 10 ms
  pwmUnit.pulseDuration(10);

  -- Create a virtual scope and attach it to the generated
  -- PWM output.
  -- Using channel 1 from the GPIO block.
  sampler1 := new Sampler(outputInterface,1);

  -- ### Run simulation
  while (clk.getSimTime() <= simTimeMax) do
  (
    currentSimTime := clk.getSimTime();
    IO`print("\nSimulation time step: ");
    IO`print(clk.getSimTime());

    controller.timeStep(currentSimTime);
    pwmUnit.timeStep();


    sampler1.sample();
    clk.tick();
  );

```

```
   -- ### Simulation tear down
   sampler1.dumpToFile();
   pwmUnit.recoverLogs();
   sampler1.dumpClock();

   IO`print("## Model Over ##");
);

end Environment
```

B.1.4 PWMgenerator

```

class PWMgenerator

values

  -- Number of clock ticks per cycle
  private ticksPerPulse :nat = 200;

  -- 10 ticks = 1 ms
  private msFactor : nat = 10;

  -- Signal frequency in Hz
  private frequency : nat = 50;

instance variables

  -- Ussed GPIO block and channel number
  private output : gpioInterface;
  private channel : nat := 0;

  -- Internal counter used to define the
  -- transition from high to low inside
  -- each cycle.

  private tickCounter : nat := 0;
  inv tickCounter >= 0 and tickCounter <= ticksPerPulse;
  -- tickCounter --> is set ot 0 at the beginning of A
  --             --> incremented each clock tick
  --             --> when tickCounter reaches its limit
  --                 the line is asserted for the rest of the
  --                 cycle.
  --  The end of A and beggining of B deppends on the desired
  --  PWM signal
  --
  --  A      B      A      B
  -- <--><-------><--><------->
  --  __          __
  -- |  |        |  |
  -- |  |________|  |________

  private tickLimit : nat := 0;
  inv tickLimit > 0 and tickLimit < ticksPerPulse;

  -- Higher duty cycle values might damage the servo
  inv getDutyCycle() < 10;

  public posLog : PositionLogger;

operations

-- Constructor
-- Associates a GPIO block and a channel.
public PWMgenerator : gpioInterface *
                      nat *
                      PositionLogger ==>
                      PWMgenerator
PWMgenerator(out,ch,posL) ==
(
  output := out;
  channel := ch;
  tickLimit := ticksPerPulse/2;
  posLog := posL;
)
post output.getWidth() >= channel and
   not channel = 0 and
   getDutyCycle() = 0.5;

public calculatePosition: () ==> real
calculatePosition() ==
  return (180 * ((tickLimit/msFactor) -1));


-- Takes in account the duty cycle and generates the signal
private generateCycle: () ==> ()
generateCycle() ==
  cases tickCounter:
     -- Start of the new cycle
     (0) -> setOn(),
     -- Time limit for the high level has been reached
     (tickLimit) -> setOff(),
     -- Keep the line as it is
     others -> skip
  end;
```

```


public timeStep: () ==> ()
timeStep() ==
(
  -- Generate the part of the continuous signal
  -- that corresponds to this time step.
  generateCycle();
  IO`print("\n## Calculated position in degrees: ");
  IO`print(calculatePosition());
  posLog.pushValue(calculatePosition());

  -- Update the pulse counter in the case the resolution
  -- limit is reached
  if tickCounter >= ticksPerPulse then
    tickCounter := 0
  else
    tickCounter := tickCounter +1;
);

-- Sets pulse duration in milliseconds
public pulseDuration : nat ==> ()
pulseDuration(d) ==
 tickLimit := d;

-- Assert high the associated GPIO channel
private setOn: () ==> ()
setOn() ==
  output.setHigh(channel);

-- Assert low the associated GPIO channel
private setOff: () ==> ()
setOff() ==
  output.setLow(channel);

public gettickLimit: () ==> nat
gettickLimit() ==
 return tickLimit;

public setLimit : nat ==> ()
setLimit(l) ==
  tickLimit := l;

public recoverLogs: () ==> ()
recoverLogs() ==
(
  posLog.dumpToFile();
);

```

```
functions

-- Calculates the current duty cycle of the signal
-- defined by timeHigh/timeCycle
public getDutyCycle : () -> real
getDutyCycle() ==
 gettickLimit()/ticksPerPulse
pre ticksPerPulse > 0;

end PWMgenerator
```

B.1.5 PositionLogger

```

class PositionLogger

types

  private OutputTP = int * int;

instance variables

  private numValues : nat := 0;
  private sampledValues : seq of real := [];

  private writeResult : bool := false;
  private io : IO := new IO();

operations

public pushValue: real ==> ()
pushValue(angle) ==
(
  sampledValues:= sampledValues ^ [angle];
);

public dumpToFile:() ==> ()
dumpToFile () ==
(
  IO`print(sampledValues);

  for all r in set {1,..., len(sampledValues)}
  do
  (
    writeResult := io.fwriteval[real]("position.csv",
                                      sampledValues(r),
                                              <append>);
    writeResult := io.fecho("position.csv", ",1", <append>);
    writeResult := io.fecho("position.csv","\n", <append>);
  );

);

end PositionLogger
```

B.1.6 Sampler

```

class Sampler

types

 OutputTP = int * int;

instance variables

  gpioUnit : gpioInterface;
  sampledValues : seq of bool := [];
  bitToSample : nat;
  numValues : nat := 0;
  inv numValues = len sampledValues;

  toFile : seq of nat := [];

  writeResult : bool := false;
  io : IO := new IO();


operations

public sample : () ==> ()
sample () ==
  atomic
  (
    sampledValues := sampledValues ^
                        [gpioUnit.getState(bitToSample)];
    numValues := numValues +1;
  );

public Sampler : gpioInterface * nat ==> Sampler
Sampler(gpioBlock,output) ==
(
  gpioUnit := gpioBlock;
  bitToSample := output;
);

public dumpToFile : () ==> ()
dumpToFile () ==
(
  IO`print(sampledValues);

  for all r in set {1,...,numValues}
  do
  (
    IO`print(r);
    writeResult := io.fwriteval[int]("signal.csv", r, <append>);
    if sampledValues(r) then
    (
      --IO`print(",1");
      writeResult := io.fecho("signal.csv", ",1", <append>)
    )
    else
    (
      --IO`print(",0");
      writeResult := io.fecho("signal.csv",",0", <append>)
     );

      writeResult := io.fecho("signal.csv","\n", <append>);
      --IO`print("\n");
    );
);

public dumpClock : () ==> ()
dumpClock () ==
(

  for all r in set {1,...,numValues}
  do
  (
    writeResult := io.fwriteval[int]("clock.csv", r, <append>);
    writeResult := io.fecho("clock.csv", ",0", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[int]("clock.csv", r, <append>);
    writeResult := io.fecho("clock.csv", ",1", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[real]("clock.csv",
                                      r+0.5,
                                      <append>);
    writeResult := io.fecho("clock.csv", ",1", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[real]("clock.csv",
                                      r+0.5,
                                      <append>);
    writeResult := io.fecho("clock.csv", ",0", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);
   )
);

public assignOutput : gpioInterface ==> ()
assignOutput (gpioU) ==
(
  gpioUnit := gpioU;
);

end Sampler
```

B.1.7 Timer

```
class Timer

instance variables

  freq : nat;
  flag : bool := true;

operations

public Timer: (nat) ==> Timer
Timer(fq) ==
(
  freq := fq;
);

private wait: () ==> ()
wait() ==
(

);

private toggleFlag: () ==> ()
toggleFlag() ==
  flag := not flag;

public getFlag: () ==> bool
```

```
getFlag() ==
  return flag;

public activeBehaviour:() ==> ()
activeBehaviour() ==
(

);

thread

while true do
(
  wait();
  toggleFlag();
  IO`print("\nFlag state: ");
  IO`print(flag);
);

sync
  mutex(toggleFlag,getFlag);

end Timer
```

B.1.8 gpioInterface

```

class gpioInterface

instance variables

  width : nat := 0;
  state : seq of bool := [];
  inv len state = width;

operations

public gpioInterface: nat ==> gpioInterface
  gpioInterface(w) ==
(
  width := w;

  for all r in set {1,...,width}
    do state := state ^ [false];

 IO`print(state);

)
post width = len state and
  forall i in set {1,...,len state} & state(i) = false;


public setHigh: nat ==> ()
setHigh(i) ==
  state(i) := true
pre i <= len state;

public setLow: nat ==> ()
setLow(i) ==
  state(i) := false
pre i <= len state;

public toggleBit: nat ==> ()
toggleBit(i) ==
  if state(i)
  then state(i) := not state(i)
  else state(i):= true
pre i<=len state;

public getState: nat ==> bool
getState(i) ==
   return state(i)
pre i <= len state;

public showState: () ==> ()
showState() ==
  for all r in set {1,...,width}
  do showSingle(r);


public getWidth: () ==> nat
getWidth() ==
  return width;

public showSingle: nat ==> ()
showSingle(i) ==
(
  IO`print("\n bit ");
  IO`print(i);
  IO`print(": ");
  IO`print(getState(i));
);

sync
  mutex(setLow,setHigh);
  mutex(setHigh,getState);
  mutex(setLow,getState);

end gpioInterface
```

B.2. Concurrent VDM model

B.3. Clock

```

class Clock

instance variables

 private tickNum : nat := 0;

operations

public tick : () ==> ()
tick() ==
(
 tickNum := tickNum +1;
);

public getSimTime: () ==> nat
getSimTime() ==
 return tickNum;

thread
while true do
(
  tick();
  Environment`timerRef.WaitRelative(1);
);

end Clock
```

B.4. Controller

```

```

```
class Controller

instance variables

  private pwmGenerator : PWMgenerator;
  private outputInterface : gpioInterface;

operations

public Controller: gpioInterface * PWMgenerator  ==> Controller
Controller (gpio,p) ==
(
  outputInterface := gpio;
  -- Create a position logger
  pwmGenerator := p;
);

public getOutputInterface: () ==> gpioInterface
getOutputInterface() ==
 return outputInterface;

-- Operations running in the controller
public timeStep: () ==> ()
timeStep() ==
(
 skip;
);

thread

while true do
(  timeStep();
  Environment`timerRef.WaitRelative(1);
);

end Controller
```

B.5. Environment

```

class Environment

instance variables

```

```
  clk : Clock;
  simTimeMax : nat := 2400;
  currentSimTime : nat := 0;

  outputInterface : gpioInterface;
  controller : Controller;

  public  sampler1 : Sampler;
  posLog : PositionLogger;

  pwmUnit : PWMgenerator;

  public static timerRef : TimeStamp := new TimeStamp(4);

operations

public run : () ==> ()
run() ==
(
  -- ### Simulation setup
  -- Create a clock
  clk := new Clock();

  -- Define a microcontroller

  posLog := new PositionLogger();

  -- Attach the gpio generating the PWM signal to the environment
  outputInterface := new gpioInterface(2);
  -- Create a PWMgenerator and attach it to the output
  pwmUnit := new PWMgenerator(outputInterface,1,posLog);

  controller := new Controller(outputInterface,pwmUnit);
  -- Set the initial pulse duration to 10 ms
  pwmUnit.pulseDuration(10);

  -- Create a virtual scope and attach it to the generated
  -- PWM output.
  -- Using channel 1 from the GPIO block.
  sampler1 := new Sampler(outputInterface,1);

  start(clk);
  start(pwmUnit);
  start(controller);
  start(sampler1);

  -- ### Run simulation
  while (timerRef.GetTime() <= simTimeMax) do
  (
    timerRef.WaitRelative(1);
    wait();
  );

   -- ### Simulation tear down
   sampler1.dumpToFile();
   pwmUnit.recoverLogs();
   sampler1.dumpClock();

   IO`print("## Model Over ##");
);


public wait: () ==> ()
wait() ==
  skip;

end Environment
```

B.6. PWMgenerator

```

class PWMgenerator

values

  -- Number of clock ticks per cycle
  private ticksPerPulse :nat = 200;

  -- 10 ticks = 1 ms
  private msFactor : nat = 10;

  -- Signal frequency in Hz
  private frequency : nat = 50;

instance variables

  -- Ussed GPIO block and channel number
  private output : gpioInterface;
  private channel : nat := 0;

  -- Internal counter used to define the
  -- transition from high to low inside
  -- each cycle.

  private tickCounter : nat := 0;
  inv tickCounter >= 0 and tickCounter <= ticksPerPulse;
  -- tickCounter --> is set ot 0 at the beginning of A
  --             --> incremented each clock tick
  --             --> when tickCounter reaches its limit
  --                 the line is asserted for the rest of the
  --                 cycle.
  --  The end of A and beggining of B deppends on the desired
  --  PWM signal
  --
  --  A      B     A      B
  -- <--><-------><--><------->
  --  ___          ___
  -- |   |        |   |
  -- |   |________|   |________

  private tickLimit : nat := 0;
  inv tickLimit > 0 and tickLimit < ticksPerPulse;

  -- Higher duty cycle values might damage the servo
  inv getDutyCycle() < 10;

  public posLog : PositionLogger;

operations

-- Constructor
-- Associates a GPIO block and a channel.
public PWMgenerator : gpioInterface *
                      nat *
                      PositionLogger ==>
                      PWMgenerator
PWMgenerator(out,ch,posL) ==
(
  output := out;
  channel := ch;
  tickLimit := ticksPerPulse/2;
  posLog := posL;
)
post output.getWidth() >= channel and
   not channel = 0 and
   getDutyCycle() = 0.5;

public calculatePosition: () ==> real
calculatePosition() ==
  return (180 * ((tickLimit/msFactor) -1));


```

```
-- Takes in account the duty cycle and generates the signal
private generateCycle: () ==> ()
generateCycle() ==
  cases tickCounter:
     (0) -> setOn(),
     (tickLimit) -> setOff(),
     others -> skip
  end;


public timeStep: () ==> ()
timeStep() ==
(
  -- Generate the part of the continuous signal
  -- that corresponds to this time step.
  generateCycle();
  IO`print("\n## Calculated position in degrees: ");
  IO`print(calculatePosition());
  posLog.pushValue(calculatePosition());

  -- Update the pulse counter in the case the resolution
  -- limit is reached
  if tickCounter >= ticksPerPulse then
    tickCounter := 0
  else
    tickCounter := tickCounter +1;

  Environment`timerRef.WaitRelative(1);
);

-- Sets pulse duration in milliseconds
public pulseDuration : nat ==> ()
pulseDuration(d) ==
 tickLimit := d;

-- Assert high the associated GPIO channel
private setOn: () ==> ()
setOn() ==
  output.setHigh(channel);

-- Assert low the associated GPIO channel
private setOff: () ==> ()
setOff() ==
  output.setLow(channel);

public gettickLimit: () ==> nat
gettickLimit() ==
 return tickLimit;

```

```
public setLimit : nat ==> ()
setLimit(l) ==
  tickLimit := l;

public recoverLogs: () ==> ()
recoverLogs() ==
(
  posLog.dumpToFile();
);

functions

-- Calculates the current duty cycle of the signal
-- defined by timeHigh/timeCycle
public getDutyCycle : () -> real
getDutyCycle() ==
 gettickLimit()/ticksPerPulse
pre ticksPerPulse > 0;

thread

while true do
(
  timeStep();
  Environment`timerRef.WaitRelative(1);
);


end PWMgenerator
```

B.7. PositionLogger

```

class PositionLogger

types

  private OutputTP = int * int;

instance variables

  private numValues : nat := 0;
  private sampledValues : seq of real := [];

  private writeResult : bool := false;
  private io : IO := new IO();

operations

public pushValue: real ==> ()
pushValue(angle) ==
(
  sampledValues:= sampledValues ^ [angle];
);

public dumpToFile:() ==> ()
dumpToFile () ==
(
  IO`print(sampledValues);

  for all r in set {1,..., len(sampledValues)}
  do
  (
    writeResult := io.fwriteval[real]("position.csv",
                                      sampledValues(r),
                                      <append>);
    writeResult := io.fecho("position.csv", ",1", <append>);
    writeResult := io.fecho("position.csv","\n", <append>);
  );

);

end PositionLogger
```

B.8. Sampler

```

class Sampler

types

 OutputTP = int * int;

instance variables

  gpioUnit : gpioInterface;
  sampledValues : seq of bool := [];
  bitToSample : nat;
  numValues : nat := 0;
--  inv numValues = len sampledValues;

  toFile : seq of nat := [];

  writeResult : bool := false;
  io : IO := new IO();


operations

public sample : () ==> ()
sample () ==
(
  atomic
  (
    sampledValues := sampledValues ^
                     [gpioUnit.getState(bitToSample)];
    numValues := numValues +1;
  );
);

public Sampler : gpioInterface * nat ==> Sampler
Sampler(gpioBlock,output) ==
(
  gpioUnit := gpioBlock;
  bitToSample := output;
);

public dumpToFile : () ==> ()
dumpToFile () ==
(
  IO`print(sampledValues);

  for all r in set {1,...,numValues}
  do
  (
    IO`print(r);
    writeResult := io.fwriteval[int]("signal.csv", r, <append>);
    if sampledValues(r) then
    (
      --IO`print(",1");
      writeResult := io.fecho("signal.csv", ",1", <append>)
    )
    else
    (
      --IO`print(",0");
      writeResult := io.fecho("signal.csv",",0", <append>)
     );
```

```

      writeResult := io.fecho("signal.csv","\n", <append>);
      --IO`print("\n");
    );
);

public dumpClock : () ==> ()
dumpClock () ==
(

  for all r in set {1,...,numValues}
  do
  (
    writeResult := io.fwriteval[int]("clock.csv", r, <append>);
    writeResult := io.fecho("clock.csv", ",0", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[int]("clock.csv", r, <append>);
    writeResult := io.fecho("clock.csv", ",1", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[real]("clock.csv",
                                      r+0.5,
                                      <append>);
    writeResult := io.fecho("clock.csv", ",1", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);

    writeResult := io.fwriteval[real]("clock.csv",
                                      r+0.5,
                                      <append>);
    writeResult := io.fecho("clock.csv", ",0", <append>);
    writeResult := io.fecho("clock.csv","\n", <append>);
   )
);

public assignOutput : gpioInterface ==> ()
assignOutput (gpioU) ==
(
  gpioUnit := gpioU;
);

thread
  while true do
(
  sample();
  Environment`timerRef.WaitRelative(1);
);

end Sampler
```

B.9. TimeStamp

```
class TimeStamp

values

public stepLength : nat = 1;

instance variables

currentTime  : nat   := 0;
wakeUpMap    : map nat to [nat] := {|->};
barrierCount : nat1;

operations

public TimeStamp : nat1 ==> TimeStamp
TimeStamp(count) ==
 barrierCount := count;

public WaitRelative : nat ==> ()
WaitRelative(val) ==
  WaitAbsolute(currentTime + val);

public WaitAbsolute : nat ==> ()
WaitAbsolute(val) == (
  AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);

BarrierReached : () ==> ()
BarrierReached() ==
(
 while  (card dom wakeUpMap = barrierCount) do
   (
    currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set dom wakeUpMap
              & wakeUpMap(th) <> nil and
              wakeUpMap(th) <= currentTime }
  in
```

```
   for all t in set threadSet
   do
    wakeUpMap := {t} <-: wakeUpMap;
 );
)
post forall x in set rng wakeUpMap & x = nil or x >= currentTime;

AddToWakeUpMap : nat * [nat] ==> ()
AddToWakeUpMap(tId, val) ==
   wakeUpMap := wakeUpMap ++ { tId |-> val };

public NotifyThread : nat ==> ()
NotifyThread(tId) ==
 wakeUpMap := {tId} <-: wakeUpMap;

public GetTime : () ==> nat
GetTime() ==
  return currentTime;

Awake: () ==> ()
Awake() == skip;

public ThreadDone : () ==> ()
ThreadDone() ==
 AddToWakeUpMap(threadid, nil);

sync
  per Awake => threadid not in set dom wakeUpMap;

  mutex(AddToWakeUpMap);
  mutex(NotifyThread);
  mutex(BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread);
  mutex(AddToWakeUpMap, BarrierReached);
  mutex(NotifyThread, BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

B.10. Timer

```
class Timer
```

```

instance variables

  freq : nat;
  flag : bool := true;

operations

public Timer: (nat) ==> Timer
Timer(fq) ==
(
  freq := fq;
);

private wait: () ==> ()
wait() ==
(

);

private toggleFlag: () ==> ()
toggleFlag() ==
  flag := not flag;

public getFlag: () ==> bool
getFlag() ==
  return flag;

public activeBehaviour:() ==> ()
activeBehaviour() ==
(

);

thread

while true do
(
  wait();
  toggleFlag();
  IO`print("\nFlag state: ");
  IO`print(flag);
);

sync
  mutex(toggleFlag,getFlag);

end Timer
```

B.11. gpioInterface

```

class gpioInterface

instance variables

  width : nat := 0;
  state : seq of bool := [];
  inv len state = width;

operations

public gpioInterface: nat ==> gpioInterface
  gpioInterface(w) ==
(
  width := w;

  for all r in set {1,...,width}
    do state := state ^ [false];

 IO`print(state);

)
post width = len state and
  forall i in set {1,...,len state} & state(i) = false;


public setHigh: nat ==> ()
setHigh(i) ==
  state(i) := true
pre i <= len state;

public setLow: nat ==> ()
setLow(i) ==
  state(i) := false
pre i <= len state;

public toggleBit: nat ==> ()
toggleBit(i) ==
  if state(i)
  then state(i) := not state(i)
  else state(i):= true
pre i<=len state;

public getState: nat ==> bool
getState(i) ==
   return state(i)
```

```
pre i <= len state;

public showState: () ==> ()
showState() ==
  for all r in set {1,...,width}
  do showSingle(r);


public getWidth: () ==> nat
getWidth() ==
  return width;

public showSingle: nat ==> ()
showSingle(i) ==
(
  IO`print("\n bit ");
  IO`print(i);
  IO`print(": ");
  IO`print(getState(i));
);

sync
  mutex(setLow,setHigh);
  mutex(setHigh,getState);
  mutex(setLow,getState);

end gpioInterface
```

B.12. Real-time VDM model: Software based solution

B.12.1 CPUloader

```
class  Controller

-- Definition of the CPU loader running in a controller

values
  -- Execution limit for the CPU loader
  executionLimit : nat = 90;

instance variables

  -- Switch time mark. Initialize to a higher value
  -- in the case a discrete control is used from the
  -- loader.
  switchTimeMark :nat := 0;

  progressCounter : nat := 0;
  pwmref: PWMunit;
  cSent :bool := false;

operations

-- Specify the switch time mark in the case several
-- control comands are issued to the controller
public setSwitchTimeMark : nat ==> ()
setSwitchTimeMark(tm) ==
  switchTimeMark := tm;

-- Time consumer operation loading the CPU. The
-- function is not performing any computation, just
-- taking CPU time.
public timeConsumer : () ==> ()
timeConsumer () ==
(
  -- Simulated operations performed during
  -- the first 9 iterations.
  cases progressCounter:
    (1) -> duration(1E6) timeEater(),
    (2) -> duration(5E6) timeEater(),
    (3) -> duration(10E6) timeEater(),
    (4) -> duration(15E6) timeEater(),
    (5) -> duration(20E6) timeEater(),
    (6) -> duration(25E6) timeEater(),
    (7) -> duration(30E6) timeEater(),
    (8) -> duration(40E6) timeEater(),
    (9) -> duration(50E6) timeEater(),
    others -> ()
   end;

  -- Increment the progress counter
  progressCounter := progressCounter + 1;

  -- Send a control signal to the hardware block
  -- in the case a switchTimeMark has been specified
  if time > switchTimeMark and not cSent then
  (
    controlSignal();
    IO`print("[#] PWM generator notified at time: ");
    IO`print(time); IO`print(" [#]\n");
    cSent := true;
  );
);
```

```

-- Issues a change in the continuous time hardware
-- signal generation.
public controlSignal : () ==> ()
controlSignal () ==
(
  pwmref.setPeriod(2);
);

-- Establish the reference to the PWM unit
public setPWM : PWMunit ==> ()
setPWM(p) ==
  pwmref := p;

-- Do nothing
public timeEater : () ==> ()
timeEater() ==
  skip;

thread
  -- Procedural thread running unit completion
  while progressCounter < executionLimit do timeConsumer();

sync
  -- Ensures that only one timeConsumer instance is running
  -- at a certain point of time.
  mutex(timeConsumer);

end Controller
```

B.12.2 Deployment

```
system Deployment

instance variables

  -- Definition of a controller unit, acting as CPU
  controller : CPU := new CPU(<FCFS>, 1E9);

  -- Definition of a hardware GPIO block
  gpioBlock : CPU := new CPU(<FCFS>, 100E6);

  -- Associate the controller to the GPIO block
  controlRegister : BUS := new BUS(<CSMACD>,
                                   72E13,
                                   {controller,gpioBlock});

  -- Static definition of the deployable objects
  public static loader : Controller := new Controller();
  public static pwm : PWMunit := new PWMunit();
  public static gpio : gpioInterface := new gpioInterface(2);

operations

-- Setup and actual deployment
public Deployment : () ==> Deployment
Deployment () ==
(
  -- Set a reference to the pwm generator from the loader
  loader.setPWM(pwm);
  -- Set a reference to the GPIO block from the PWMgenerator
  pwm.setGPIO(gpio);

  -- Deploy the active objects in different CPUs
  controller.deploy(loader);
  controller.deploy(pwm);
  gpioBlock.deploy(gpio);

);

end Deployment
```

B.12.3 Environment

```
class Environment

values

  -- Time limit mark in the case two control signals
  -- are going to be generated.
  private timeLimitMark : nat = 0;

instance variables

  -- Create a sampler
  public static sampler1 : Sampler := new Sampler();

operations

-- Run operation, simulation entry point
public run : () ==> ()
run() ==
(
  -- Assocaite the limit time mark for the loader
  Deployment`loader.setSwitchTimeMark(timeLimitMark);

  -- Associate the sampler to the PWM generator
  Deployment`pwm.outLog := sampler1;

  -- Start the threads
  start(Deployment`pwm);
  start(Deployment`loader);

  wait(); -- Block until threads are done

 -- Dump the sampled signals to a file
  sampler1.dumpSignal();

  -- Simulation tear down
  printAndBye();

);

-- Simulation tear down
public printAndBye: () ==> ()
printAndBye() ==
(
  IO`print("[#] Generated signal: ");
  IO`print(Deployment`pwm.outLog.log);
  IO`print(" [#]");
  IO`print("\n[1] Model over");
  IO`print("\n[2] Logs generated");
);

-- Wait until it is down
public wait : () ==> ()
wait() ==
  skip;

sync
 -- Sync predicate for the wait operation
 per wait => Deployment`pwm.getFlag();

end Environment
```

B.12.4 PWMgenerator

```
class PWMunit
```

```

instance variables

  -- References to the sampler and the
  -- controller
  public outLog : Sampler;
  public controller : Controller;

  -- Allows the generation of two different
  -- predefined signals
  private periodSelector : nat := 1;

  -- Sets a reference to the GPIO interface
  public outputInt : gpioInterface;

  -- Done flag updated when operation completion
  -- has been reached
  public done : bool := false;

  -- Counter keeping track of the
  -- simulation progress
  public runCount : nat := 0;

  -- Number of control periods that have to
  -- be generated
  public exLimit : nat := 30;

operations

-- Associate a GPIO block to the PWM unit
-- controller
public setGPIO : gpioInterface ==> ()
setGPIO(g) ==
  outputInt := g;

-- Associate a controller to the PWMgenerator
public PWMunit : Controller ==> PWMunit
PWMunit(c) ==
  controller := c;

-- Generate a single contorol cycle. Two different
-- control puses can be selected depending on the
-- target position.
private generateSignal : () ==> ()
generateSignal() ==
(
  while runCount < exLimit do
  (
    if (getPeriod() = 2) then
    (-- Set output to high
      duration (0) outputInt.toggleBit(1);
      duration (2E6) ();-- Wait 2 milliseconds
      -- Set output to low
      duration (0) outputInt.toggleBit(1);
      duration (18E6) (); -- Wait 18 milliseconds
      duration (0) runCount := runCount +1;
    )
    -- In the second control case 1 millisecond
    -- is spent in the control pulse. The rest
    -- of the operation is analogous to the
    -- previous case.
    else if getPeriod() = 1 then
    (
      duration (0) outputInt.toggleBit(1);
      duration (1E6) ();
      duration (0) outputInt.toggleBit(1);
      duration (19E6) ();
      duration (0) runCount := runCount +1;
    )
  );
  duration (0) done := true -- Mark the thread as completed
);

-- Get completion flag
public getFlag : () ==> bool
getFlag() ==
  return done;

-- Get the selected period by the discrete processing
-- unit
private getPeriod : () ==> nat
getPeriod() ==
  return periodSelector;

-- Sets the control period signal
public setPeriod: nat ==> ()
setPeriod(p) ==
  periodSelector := p;

thread
   -- Procedural thread, running until completion
   generateSignal();

sync
  -- The signal generation should not be executed
  -- while the completion flag is requested
  mutex(generateSignal,getFlag);
  -- There should not be two instances of generateSignal
  -- running in parallel
  mutex(generateSignal);

  -- Target period must not be set while the period is
  -- retrieved
  mutex(setPeriod,getPeriod);

end PWMunit
```

B.12.5 Sampler

```
class Sampler

instance variables

  -- Sequence of read values
  public log : seq of nat := [];

  -- Auxiliary variable in order to store the result of
  -- the write to file operation.
  writeResult : bool := false;

  -- Instace of the IO class in order to write
  -- to the file
  io :IO := new IO();

  -- Skip the first three spurious values
  public  r : nat := 3;

operations

-- Toggle the read value in the sampling input.
async public toggle : () ==> ()
toggle() ==
(
  duration(0) log := log ^ [time];
);

public dumpSignal : () ==> ()
dumpSignal () ==
(
  -- Create signal.csv in the case it does not exist
  -- Clean up previous contents in the case the file exists
  writeResult := io.fecho("signal.csv", "", <start>);

  -- Write the logged time values with the proper format
  while r < (len Deployment`pwm.outLog.log - 1)
  do
  (
    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        <append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    r := r+2;
   )
);

sync
  -- Two toggle operations cannot be run at the same time
   mutex(toggle);

end Sampler
```

B.12.6 gpioInterface

```
class gpioInterface

```

```
instance variables

  -- Bus width
  width : nat := 0;
  -- State of each bit (true = high state, false = low state)
  state : seq of bool := [];

  -- For each bit there is a state entry in the sequence
  inv len state = width;

operations

-- Class constructor. All states initialized to false by
-- default
public gpioInterface: nat ==> gpioInterface
  gpioInterface(w) ==
(
  width := w;

  for all r in set {1,...,width}
    do state := state ^ [false];

 IO`print(state);
)
post width = len state and
  forall i in set {1,...,len state} & state(i) = false;

-- Set a concrete bit to high level
public setHigh: nat ==> ()
setHigh(i) ==
  state(i) := true
pre i <= len state; -- Bit position should be in bit sequence

-- Set a concrete bit to low level
public setLow: nat ==> ()
setLow(i) ==
  state(i) := false
pre i <= len state; -- Bit position should be in bit sequence

-- Toggles the state of a certain bit.
-- If the bit position is at high level goes to false
-- If the bit position is at low level goes to true
async public toggleBit: nat ==> ()
toggleBit(i) ==
(
  if state(i)
  then state(i) := not state(i)
  else state(i):= true;
  Environment`sampler1.toggle();
)
pre i<=len state; -- Bit position should be in bit sequence

-- Get the state of a certain bit
public getState: nat ==> bool
getState(i) ==
   return state(i)
pre i <= len state;

-- Shows the state of all the bits in the register
public showState: () ==> ()
showState() ==
  for all r in set {1,...,width}
  do showSingle(r);

-- Get the width of the gpio register
public getWidth: () ==> nat
getWidth() ==
  return width;

-- Prints out the state of a certain bit
public showSingle: nat ==> ()
showSingle(i) ==
(
  IO`print("\n bit ");
  IO`print(i);
  IO`print(": ");
  IO`print(getState(i));
)
pre i<= len state; -- Bit position should be in bit sequence

sync
  -- Only one operation can modify the state of a bit
  -- at a certain point of time.
  mutex(setLow,setHigh);

  -- Bit state cannot be retrieved if it is being set high
  -- and vice versa.
  mutex(setHigh,getState);

  -- Bit state cannot be retrieved if it is being set low
  -- and vice versa.
  mutex(setLow,getState);

end gpioInterface
```

B.13. Real-time VDM model: RTOS based solution

B.13.1 CPUloader

```
class  Controller

-- Definition of the CPU loader running in a controller

values
  -- Execution limit for the CPU loader
  executionLimit : nat = 90;

instance variables

  -- Switch time mark. Initialize to a higher value
  -- in the case a discrete control is used from the
  -- loader.
  switchTimeMark :nat := 0;

  progressCounter : nat := 0;
  pwmref: PWMunit;
  cSent :bool := false;

operations

-- Specify the switch time mark in the case several
-- control comands are issued to the controller
public setSwitchTimeMark : nat ==> ()
setSwitchTimeMark(tm) ==
  switchTimeMark := tm;

-- Time consumer operation loading the CPU. The
-- function is not performing any computation, just
-- taking CPU time.
public timeConsumer : () ==> ()
timeConsumer () ==
(
  -- Simulated operations performed during
  -- the first 9 iterations.
  cases progressCounter:
    (1) -> duration(1E6) timeEater(),
    (2) -> duration(2E6) timeEater(),
    (3) -> duration(3E6) timeEater(),
    (4) -> duration(4E6) timeEater(),
    (5) -> duration(5E6) timeEater(),
    (6) -> duration(6E6) timeEater(),
    (7) -> duration(7E6) timeEater(),
    (8) -> duration(8E6) timeEater(),
    (9) -> duration(9E6) timeEater(),
    others -> ()
   end;

  -- Increment the progress counter
  progressCounter := progressCounter + 1;
);

-- Issues a change in the continuous time hardware
-- signal generation.
public controlSignal : () ==> ()
controlSignal () ==
(
  pwmref.setPeriod(2);
);

-- Establish the reference to the PWM unit
public setPWM : PWMunit ==> ()
setPWM(p) ==
  pwmref := p;

-- Do nothing
public timeEater : () ==> ()
timeEater() ==
  skip;

thread
  -- Periodic thread executed every 55 milliseconds
  periodic (55E6,0,0,0) (timeConsumer);

sync
  mutex(timeConsumer);

end Controller
```

B.13.2 Deployment

```
system Deployment

instance variables

  -- Definition of a controller unit, acting as CPU
  controller : CPU := new CPU(<FP>, 1E9);

  -- Definition of a hardware GPIO block
  gpioBlock : CPU := new CPU(<FCFS>, 100E6);

  -- Associate the controller to the GPIO block
  controlRegister : BUS := new BUS(<CSMACD>,
                                   72E13,
                                   {controller,gpioBlock});

  -- Static definition of the deployable objects
  public static loader : Controller := new Controller();
  public static pwm : PWMunit := new PWMunit();
  public static gpio : gpioInterface := new gpioInterface(2);



operations

-- Setup and actual deployment
public Deployment : () ==> Deployment
Deployment () ==
(
  -- Set a reference to the pwm generator from the loader
  loader.setPWM(pwm);
  -- Set a reference to the GPIO block from the PWMgenerator
  pwm.setGPIO(gpio);



  -- Deploy the active objects in different CPUs
  controller.deploy(loader);
  controller.deploy(pwm);


  --controller.setPriority(loader.timeConsumer,5);

  gpioBlock.deploy(gpio);
  controller.setPriority(PWMunit`generateSignal,10);
  controller.setPriority(Controller`timeConsumer,1);

);

end Deployment
```

B.13.3 Environment

```
class Environment

values
```

```

  -- Time limit mark in the case two control signals
  -- are going to be generated.
  private timeLimitMark : nat = 0;

instance variables

  -- Create a sampler
  public static sampler1 : Sampler := new Sampler();

operations

-- Run operation, simulation entry point
public run : () ==> ()
run() ==
(

  -- Assocaite the limit time mark for the loader
  Deployment`loader.setSwitchTimeMark(timeLimitMark);

  -- Associate the sampler to the PWM generator
  Deployment`pwm.outLog := sampler1;

  -- Start the threads
  start(Deployment`loader);
  start(Deployment`pwm);


 -- wait(); -- Block until threads are done
 if not Deployment`pwm.getFlag() then wait();

  -- Dump the sampled signals to a file
  sampler1.dumpSignal();

  -- Simulation tear down
  printAndBye();

);

-- Simulation tear down
public printAndBye: () ==> ()
printAndBye() ==
(
  IO`print("[#] Generated signal: ");
  IO`print(Deployment`pwm.outLog.log);
  IO`print(" [#]");
  IO`print("\n[1] Model over");
  IO`print("\n[2] Logs generated");
);
```

```

-- Wait until it is down
public wait : () ==> ()
wait() ==
  skip;

sync
 -- Sync predicate for the wait operation
 per wait => Deployment`pwm.getFlag();

end Environment
```

B.13.4 PWMgenerator

```
class PWMunit

instance variables

  -- References to the sampler and the
  -- controller
  public outLog : Sampler;
  public controller : Controller;

  -- Allows the generation of two different
  -- predefined signals
  private periodSelector : nat := 1;

  -- Sets a reference to the GPIO interface
  public outputInt : gpioInterface;

  -- Done flag updated when operation completion
  -- has been reached
   public done : bool := false;

  -- Counter keeping track of the
  -- simulation progress
  public runCount : nat := 0;

  -- Number of control periods that have to
  -- be generated
  static public exLimit : nat := 30;

operations

-- Associate a GPIO block to the PWM unit
-- controller
```

```
public setGPIO : gpioInterface ==> ()
setGPIO(g) ==
  outputInt := g;

-- Associate a controller to the PWMgenerator
public PWMunit : Controller ==> PWMunit
PWMunit(c) ==
  controller := c;

public generateSignal : () ==> ()
generateSignal () ==
(
  if runCount < exLimit then
  (
     duration(2E6) outputInt.toggleBit(1);
     duration(0) outputInt.toggleBit(1);
     duration(18E6) ();
     runCount := runCount + 1;
  )
  else
  (
     duration (0)  done := true;
  );
);

-- Get completion flag
public getFlag : () ==> bool
getFlag() ==
  return done;

-- Get the selected period by the discrete processing
-- unit
private getPeriod : () ==> nat
getPeriod() ==
  return periodSelector;

-- Sets the control period signal
public setPeriod: nat ==> ()
setPeriod(p) ==
  periodSelector := p;

thread
   -- Periodic invocation to the generate signal generation
   -- Operation invoked once every 20 milliseconds
   periodic (20E6,0,0,0) (generateSignal)


sync

```

```
  mutex(generateSignal,getFlag);
  mutex(generateSignal);
  mutex(setPeriod,getPeriod);

end PWMunit
```

B.13.5 Sampler

```
class Sampler

instance variables

  -- Sequence of read values
  public log : seq of nat := [];

  -- Auxiliary variable in order to store the result of
  -- the write to file operation.
  writeResult : bool := false;

  -- Instace of the IO class in order to write
  -- to the file
  io :IO := new IO();

  -- Skip the first three spurious values
  public  r : nat := 3;

operations

-- Toggle the read value in the sampling input.
async public toggle : () ==> ()
toggle() ==
(
  duration(0) log := log ^ [time];
);

public dumpSignal : () ==> ()
dumpSignal () ==
(
  -- Create signal.csv in the case it does not exist
  -- Clean up previous contents in the case the file exists
  writeResult := io.fecho("signal.csv", "", <start>);

  -- Write the logged time values with the proper format
  while r < (len Deployment`pwm.outLog.log - 1)
  do
   (
    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        <append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    r := r+2;
  )
);

sync
  -- Two toggle operations cannot be run at the same time
   mutex(toggle);

end Sampler
```

B.13.6 gpioInterface

```
class gpioInterface

instance variables

  -- Bus width
  width : nat := 0;
  -- State of each bit (true = high state, false = low state)
  state : seq of bool := [];

  -- For each bit there is a state entry in the sequence
  inv len state = width;

operations

-- Class constructor. All states initialized to false by
-- default
public gpioInterface: nat ==> gpioInterface
  gpioInterface(w) ==
(
  width := w;

  for all r in set {1,...,width}
    do state := state ^ [false];
 IO`print(state);
)
post width = len state and
  forall i in set {1,...,len state} & state(i) = false;

-- Set a concrete bit to high level
public setHigh: nat ==> ()
setHigh(i) ==
  state(i) := true
pre i <= len state; -- Bit position should be in bit sequence

-- Set a concrete bit to low level
public setLow: nat ==> ()
setLow(i) ==
  state(i) := false
pre i <= len state; -- Bit position should be in bit sequence

-- Toggles the state of a certain bit.
-- If the bit position is at high level goes to false
-- If the bit position is at low level goes to true
async public toggleBit: nat ==> ()
toggleBit(i) ==
(
  if state(i)
  then state(i) := not state(i)
  else state(i):= true;
  Environment`sampler1.toggle();
)
pre i<=len state; -- Bit position should be in bit sequence

-- Get the state of a certain bit
public getState: nat ==> bool
```

```
getState(i) ==
   return state(i)
pre i <= len state;

-- Shows the state of all the bits in the register
public showState: () ==> ()
showState() ==
  for all r in set {1,...,width}
  do showSingle(r);

-- Get the width of the gpio register
public getWidth: () ==> nat
getWidth() ==
  return width;

-- Prints out the state of a certain bit
public showSingle: nat ==> ()
showSingle(i) ==
(
  IO`print("\n bit ");
  IO`print(i);
  IO`print(": ");
  IO`print(getState(i));
)
pre i<= len state; -- Bit position should be in bit sequence

sync
  -- Only one operation can modify the state of a bit
  -- at a certain point of time.
  mutex(setLow,setHigh);

  -- Bit state cannot be retrieved if it is being set high
  -- and vice versa.
  mutex(setHigh,getState);

  -- Bit state cannot be retrieved if it is being set low
  -- and vice versa.
  mutex(setLow,getState);


end gpioInterface
```

B.14. Real-time VDM model: Hardware based solution

B.15. CPUloader

```

class  Controller

-- Definition of the CPU loader running in a controller

values
  -- Execution limit for the CPU loader
  executionLimit : nat = 90;

instance variables

  -- Switch time mark. Initialize to a higher value
  -- in the case a discrete control is used from the
  -- loader.
  switchTimeMark :nat := 0;

  progressCounter : nat := 0;
  pwmref: PWMunit;
  cSent :bool := false;

operations

-- Specify the switch time mark in the case several
-- control comands are issued to the controller
public setSwitchTimeMark : nat ==> ()
setSwitchTimeMark(tm) ==
  switchTimeMark := tm;

-- Time consumer operation loading the CPU. The
-- function is not performing any computation, just
-- taking CPU time.
public timeConsumer : () ==> ()
timeConsumer () ==
(
  -- Simulated operations performed during
  -- the first 9 iterations.
  cases progressCounter:
    (1) -> duration(1E6) timeEater(),
    (2) -> duration(5E6) timeEater(),
    (3) -> duration(10E6) timeEater(),
    (4) -> duration(15E6) timeEater(),
    (5) -> duration(20E6) timeEater(),
    (6) -> duration(25E6) timeEater(),
     (7) -> duration(30E6) timeEater(),
     (8) -> duration(40E6) timeEater(),
     (9) -> duration(50E6) timeEater(),
    others -> ()
   end;

  -- Increment the progress counter
  progressCounter := progressCounter + 1;

  -- Send a control signal to the hardware block
  -- in the case a switchTimeMark has been specified
  if time > switchTimeMark and not cSent then
  (
    controlSignal();
    IO`print("[#] PWM generator notified at time: ");
    IO`print(time); IO`print(" [#]\n");
    cSent := true;
  );
);

-- Issues a change in the continuous time hardware
-- signal generation.
public controlSignal : () ==> ()
controlSignal () ==
(
  pwmref.setPeriod(2);
);

-- Establish the reference to the PWM unit
public setPWM : PWMunit ==> ()
setPWM(p) ==
  pwmref := p;

-- Do nothing
public timeEater : () ==> ()
timeEater() ==
  skip;

thread
  -- Procedural thread running unit completion
  while progressCounter < executionLimit do timeConsumer();

sync
  -- Ensures that only one timeConsumer instance is running
  -- at a certain point of time.
  mutex(timeConsumer);

end Controller
```

B.16. Deployment

```
system Deployment

instance variables

  -- Definition of a hardware block for the generation of the PWM
  -- signal
  PWMgenerator : CPU := new CPU(<FCFS>, 1E9);

  -- Definition of a controller unit, acting as CPU
  controller : CPU := new CPU(<FCFS>, 1E5);

  -- Definition of a hardware GPIO block
  gpioBlock : CPU := new CPU(<FCFS>, 100E6);

  -- Associate the PWMgenerator to the controller
  -- via control register
  controlRegister : BUS :=
   new BUS(<CSMACD>,
         72E13,
         PWMgenerator,controller});

  -- Associate the GPIOblock to the PWMgenerator
  -- via control register
  controlRegister2 : BUS :=
    new BUS(<CSMACD>,
           72E13,
           {PWMgenerator, gpioBlock});

  -- Static definition of the deployable objects
  public static loader : Controller := new Controller();
  public static pwm : PWMunit := new PWMunit();
  public static gpio : gpioInterface := new gpioInterface(2);

operations

-- Setup and actual deployment
public Deployment : () ==> Deployment
Deployment () ==
(
  -- Set a reference to the pwm generator from the loader
  loader.setPWM(pwm);
  -- Set a reference to the GPIO block from the PWMgenerator
  pwm.setGPIO(gpio);

  -- Deploy the active objects in different CPUs
  PWMgenerator.deploy(pwm);
  controller.deploy(loader);
  gpioBlock.deploy(gpio);
);

end Deployment
```

B.17. Environment

```
class Environment

values

  -- Time limit mark in the case two control signals
  -- are going to be generated.
  private timeLimitMark : nat = 0;

instance variables

  -- Create a sampler
  public static sampler1 : Sampler := new Sampler();

operations

-- Run operation, simulation entry point
public run : () ==> ()
run() ==
(
  -- Assocaite the limit time mark for the loader
  Deployment`loader.setSwitchTimeMark(timeLimitMark);

  -- Associate the sampler to the PWM generator
  Deployment`pwm.outLog := sampler1;

  -- Start the threads
  start(Deployment`pwm);
  start(Deployment`loader);

  wait(); -- Block until threads are done

  -- Dump the sampled signals to a file
  sampler1.dumpSignal();

  -- Simulation tear down
  printAndBye();

);

-- Simulation tear down
public printAndBye: () ==> ()
printAndBye() ==
(
  IO`print("[#] Generated signal: ");
  IO`print(Deployment`pwm.outLog.log);
  IO`print(" [#]");
  IO`print("\n[1] Model over");
  IO`print("\n[2] Logs generated");
);

-- Wait until it is down
public wait : () ==> ()
wait() ==
  skip;

sync
 -- Sync predicate for the wait operation
 per wait => Deployment`pwm.getFlag();

end Environment
```

B.18. PWMgenerator

```
class PWMunit

instance variables

  -- References to the sampler and the
  -- controller
  public outLog : Sampler;
  public controller : Controller;

  -- Allows the generation of two different
  -- predefined signals
  private periodSelector : nat := 1;

  -- Sets a reference to the GPIO interface
  public outputInt : gpioInterface;

  -- Done flag updated when operation completion
  -- has been reached
  public done : bool := false;

  -- Counter keeping track of the
  -- simulation progress
  public runCount : nat := 0;

  -- Number of control periods that have to
  -- be generated
  public exLimit : nat := 30;

operations

-- Associate a GPIO block to the PWM unit
-- controller
public setGPIO : gpioInterface ==> ()
setGPIO(g) ==
  outputInt := g;

-- Associate a controller to the PWMgenerator
public PWMunit : Controller ==> PWMunit
PWMunit(c) ==
  controller := c;

-- Generate a single contorol cycle. Two different
-- control puses can be selected depending on the
-- target position.
private generateSignal : () ==> ()
generateSignal() ==
(
  while runCount < exLimit do
  (
    if (getPeriod() = 2) then
    ( -- Set output to high
      duration (0) outputInt.toggleBit(1);
      duration (2E6) (); -- Wait 2 milliseconds
      -- Set output to low
      duration (0) outputInt.toggleBit(1);
      duration (18E6) (); -- Wait 18 milliseconds
      duration (0) runCount := runCount +1;
    )
    -- In the second control case 1 millisecond
    -- is spent in the control pulse. The rest
    -- of the operation is analogous to the
    -- previous case.
    else if getPeriod() = 1 then
    (
      duration (0) outputInt.toggleBit(1);
      duration (1E6) ();
      duration (0) outputInt.toggleBit(1);
      duration (19E6) ();
      duration (0) runCount := runCount +1;
    )
  );
  duration (0) done := true -- Mark the thread as completed
);

-- Get completion flag
public getFlag : () ==> bool
getFlag() ==
  return done;

-- Get the selected period by the discrete processing
-- unit
private getPeriod : () ==> nat
getPeriod() ==
  return periodSelector;

-- Sets the control period signal
public setPeriod: nat ==> ()
setPeriod(p) ==
  periodSelector := p;

thread
   -- Procedural thread, running until completion
   generateSignal();

sync
  -- The signal generation should not be executed
  -- while the completion flag is requested
  mutex(generateSignal,getFlag);

  -- There should not be two instances of generateSignal
  -- running in parallel
  mutex(generateSignal);

  -- Target period must not be set while the period is
  -- retrieved
  mutex(setPeriod,getPeriod);

end PWMunit
```

B.19. Sampler

```
class Sampler

instance variables

  -- Sequence of read values
  public log : seq of nat := [];

  -- Auxiliary variable in order to store the result of
  -- the write to file operation.
  writeResult : bool := false;

  -- Instace of the IO class in order to write
  -- to the file
  io :IO := new IO();

  -- Skip the first three spurious values
  public  r : nat := 3;

operations

-- Toggle the read value in the sampling input.
async public toggle : () ==> ()
toggle() ==
(
  duration(0) log := log ^ [time];
);

public dumpSignal : () ==> ()
dumpSignal () ==
(
  -- Create signal.csv in the case it does not exist
  -- Clean up previous contents in the case the file exists
  writeResult := io.fecho("signal.csv", "", <start>);

  -- Write the logged time values with the proper format
  while r < (len Deployment`pwm.outLog.log - 1)
  do
  (
    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
     io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r),
                        <append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
     io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        append>);
    writeResult := io.fecho("signal.csv", ",1", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    writeResult :=
      io.fwriteval[int]("signal.csv",
                        Deployment`pwm.outLog.log(r+1),
                        <append>);
    writeResult := io.fecho("signal.csv", ",0", <append>);
    writeResult := io.fecho("signal.csv","\n", <append>);

    r := r+2;
   )
);

sync
  -- Two toggle operations cannot be run at the same time
  mutex(toggle);

end Sampler
```

B.20. gpioInterface

```
class gpioInterface

instance variables

  -- Bus width
  width : nat := 0;
  -- State of each bit (true = high state, false = low state)
  state : seq of bool := [];

  -- For each bit there is a state entry in the sequence
  inv len state = width;

operations

```

```
-- Class constructor. All states initialized to false by
-- default
public gpioInterface: nat ==> gpioInterface
  gpioInterface(w) ==
(
  width := w;

  for all r in set {1,...,width}
    do state := state ^ [false];

 IO`print(state);

)
post width = len state and
  forall i in set {1,...,len state} & state(i) = false;

-- Set a concrete bit to high level
public setHigh: nat ==> ()
setHigh(i) ==
  state(i) := true
pre i <= len state; -- Bit position should be in bit sequence

-- Set a concrete bit to low level
public setLow: nat ==> ()
setLow(i) ==
  state(i) := false
pre i <= len state; -- Bit position should be in bit sequence

-- Toggles the state of a certain bit.
-- If the bit position is at high level goes to false
-- If the bit position is at low level goes to true
async public toggleBit: nat ==> ()
toggleBit(i) ==
(
  if state(i)
  then state(i) := not state(i)
  else state(i):= true;
  Environment`sampler1.toggle();
)
pre i<=len state; -- Bit position should be in bit sequence

-- Get the state of a certain bit
public getState: nat ==> bool
getState(i) ==
   return state(i)
pre i <= len state;

-- Shows the state of all the bits in the register
public showState: () ==> ()
```

```
showState() ==
  for all r in set {1,...,width}
  do showSingle(r);

-- Get the width of the gpio register
public getWidth: () ==> nat
getWidth() ==
  return width;

-- Prints out the state of a certain bit
public showSingle: nat ==> ()
showSingle(i) ==
(
  IO`print("\n bit ");
  IO`print(i);
  IO`print(": ");
  IO`print(getState(i));
)
pre i<= len state; -- Bit position should be in bit sequence

sync
  -- Only one operation can modify the state of a bit
  -- at a certain point of time.
  mutex(setLow,setHigh);

  -- Bit state cannot be retrieved if it is being set high
  -- and vice versa.
  mutex(setHigh,getState);

  -- Bit state cannot be retrieved if it is being set low
  -- and vice versa.
  mutex(setLow,getState);

end gpioInterface
```

Appendix C

AVB case study VDM models

This appendix contains the VDM models for the AVB case study. Section C.1 presents the initial overall sequential VDM models. Section C.2 presents the sequential VDM models. Section C.3 presents the concurrent VDM models. Finally, section C.4 presents the real-time VDM models.

C.1. Overall sequential model

C.1.1 App

```
class App

instance variables

  private preciseOriginTS : real;
  private correctionField : real;

  private localTimeStamp : real;


operations

public receiveTimeInformation : seq of real ==> ()
receiveTimeInformation (time) ==
(
  preciseOriginTS := time(1);
  correctionField := time(2);
  localTimeStamp := time(3);
);

public showReceivedTime : () ==> ()
showReceivedTime() ==
(
  IO`print("\nAt time: ");
  IO`print(localTimeStamp);
  IO`print("\nThe Grand Master time was: ");
  IO`print(preciseOriginTS);
  IO`print("\nIt has been corrected with: ");
  IO`print(correctionField);
  IO`print("\nFinal time for previos timestamp is: ");
  IO`print(preciseOriginTS+correctionField);
);

end App
```

C.1.2 Clock

```
class Clock

instance variables

  time : nat := 0;

operations

public tick: () ==> ()
tick() ==
(
  time := time +1;
);

public getTime: () ==> nat
getTime() ==
  return time;

end Clock
```

C.1.3 ClockSlave

```
class ClockSlave

instance variables

  private siteSync : Device;

  private preciseOriginTS : real;
  private correctionField : real;
  private localTimeStamp : real;

  private timeConsumers : seq of App := [];

operations

public registerApp : App ==> nat
registerApp(ap) ==
(
  timeConsumers := timeConsumers ^ [ap];
  return len timeConsumers;
);

public setTimeInfo: seq of real ==> ()
setTimeInfo(time) ==
(
  preciseOriginTS := time(1);
  correctionField := time(2);
  localTimeStamp := time(3);
  notifyTime();
  IO`print("\nSet time information");
  IO`print(preciseOriginTS);
  IO`print("\n");
  IO`print(correctionField);
  IO`print("\n");
  IO`print(localTimeStamp);
);

public notifyTime : () ==> ()
notifyTime() ==
  let t : seq of real =
     [preciseOriginTS,correctionField,localTimeStamp] in
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).receiveTimeInformation(t);

private setSiteSync : Device ==> ()
setSiteSync(dv) ==
  siteSync := dv;

public getTime:() ==> ()
getTime() ==
();

public ClockSlave: () ==> ClockSlave
ClockSlave() ==
(
  let i : nat = registerApp(new App()) in ();
);
```

```

public triggerPrint: () ==> ()
triggerPrint() ==
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).showReceivedTime();

end ClockSlave
```

C.1.4 Device

```
class Device

instance variables

  -- Device ID number
  private deviceID : nat := 0;

  public isBridge : bool := true;

  -- Frequency of the current device.
  private freq : nat := 0;

  -- Master ports associated to the device
  public masterPort : Master := new Master();

  -- Master ports attached to the device
  public masterPorts : seq of Master := [];
  inv isBridge <=> len masterPorts >= 1;

  -- Slave port associated to the device
  public slavePort : Slave := new Slave();

  -- Device with no external slaves connected
  -- (finalDevice = true)
  public finalDevice : bool := false;
  --inv finalDevice => len masterPorts = 0;

  -- Current device is Grand Master -> iGM = true
  -- if device isGM it should not have masters attached.
  public isGM : bool := false;
  inv isGM => not finalDevice;
  inv finalDevice => not isGM;

  -- Frequency of the master device
  private masterFreq : nat := 0;

  private receivedSyncEvent : bool := false;
  private syncRecTS : nat := 0;

  -- Neighbour rate ratio measured by the slave port
  private neighbourRateRatio : real := 1;

  -- Precise origin timestamp, defined by GM and
  -- never changed.
  private preciseOriginTS : real := 0;

  -- received correction field, to be recomputed
  -- before use.
  private receivedCorrectionField : real := 0;

  -- CumulativeRateRatio received from the previous
  -- node.
  private cummulativeRateRatio : real := 1;
  -- Delay from master to slave port
  private linkDelay : real := 10;
  -- Time that has been spent in the current node
  private residenceTime : real := 0;

  -- Correction field calculated at the current node,
  -- to be used in the current node.
  private newCorrectionField : real := 0;
  -- CummulativeRateRatio calculated at the current node,
  -- to be used in the current node.
  private newCummulativeRateRatio : real := 0;

  -- Clock attached to the device
  private clk : Clock;

  private timeInfoAvailable : bool;

  public cSlave : ClockSlave := new ClockSlave();

operations

public notifyClock : () ==> ()
notifyClock() ==
 cSlave.setTimeInfo([preciseOriginTS,
                     receivedCorrectionField,
                     syncRecTS]);

public calculateResidenceTime : () ==> ()
calculateResidenceTime() ==
  if not finalDevice and not isGM then
    residenceTime := masterPorts(1).getEgressTime()
                   - slavePort.getIngressTime()
```

```
pre masterPorts(1).getEgressTime() > slavePort.getIngressTime()
post residenceTime > 0;

public makeTimeAvailable : () ==> ()
makeTimeAvailable() ==
  timeInfoAvailable := true;

public getTimeInfo : () ==> seq of real
getTimeInfo() ==
  return [preciseOriginTS,
          receivedCorrectionField,
          cummulativeRateRatio];

public timeInfoRead : () ==> ()
timeInfoRead() ==
  timeInfoAvailable := false;

-- Registers a new master port in the siteSync entity
public registerMasterPort : Master ==> ()
registerMasterPort(m) ==
  masterPorts := masterPorts ^ [m];

-- Sets a single master port for the current device
public setMasterPort: Master ==> ()
setMasterPort(m) ==
  masterPort := m;

-- Sets the slave port that corresponds to the current device
public setSlavePort: Slave ==> ()
setSlavePort(s) ==
  slavePort := s;

-- Sets device ID
public setDeviceID: nat ==> ()
setDeviceID(id) ==
  deviceID := id
pre deviceID = 0
post deviceID <> 0;

-- Gets device ID
public getDeviceID: () ==> nat
getDeviceID() ==
  return deviceID
pre deviceID <> 0;

-- Sets the residence time. Time that will take the system
-- to process an incoming sync event.
public setResidenceTime : nat ==> ()
setResidenceTime(t) ==
  residenceTime := t
post residenceTime <> 0;

-- Generates the POTS
public generatePOTS : nat ==> ()
generatePOTS(ts) ==
  preciseOriginTS := ts
pre isGM
post preciseOriginTS <> 0;

-- Forwards the POTS
public sendPOTS : () ==> ()
sendPOTS() ==
  if not finalDevice then
   masterPort.sendPOTS(preciseOriginTS);

-- Receives the POTS
public receivePOTS: () ==> ()
receivePOTS() ==
  preciseOriginTS := slavePort.receivePOTS();

-- Sets the frequency of the master device.
public setMasterFreq : nat ==> ()
setMasterFreq (f) ==
  masterFreq := f
pre f <> 0;

-- Device constructor
public Device : nat ==> Device
Device (f) ==
(
  freq := f;
  registerMasterPort(masterPort);
);

-- Announces the device frequency
public announceFreq : () ==> ()
announceFreq() ==
  if (not finalDevice) then
   for all i in set {1,...,len masterPorts} do
      masterPorts(i).announceFreq(freq);

-- Get the frequency of the master device attached to
-- the slave port.
public getFreq : () ==> ()
getFreq() ==
  if (not isGM) then setMasterFreq(slavePort.getFreq());

-- Sends the follow up message to all the slave unit
-- attached to it.
-- Transmission process is performed in one operation.
public sendFollowUpMessage: () ==> ()
sendFollowUpMessage() ==
  if not finalDevice then
    for all i in set {1,...,len masterPorts} do
      masterPorts(i).sendFollowUpMessage(newCorrectionField,
                                         preciseOriginTS,
                                         newCummulativeRateRatio);

-- Sends the cummulative rate ratio to the slave
-- unit attached to it.
public sendCRR: () ==> ()
sendCRR() ==
  if (not finalDevice) then
    masterPort.sendCorrectionField(newCummulativeRateRatio);

-- Receives the follow up message sent by the master.
-- Reception is performed in one operation.
public receiveFollowUpMessage: () ==> ()
receiveFollowUpMessage() ==
  let s : seq of real = slavePort.receiveFollowUpMessage() in
  (
    receivedCorrectionField := s(1);
    preciseOriginTS := s(2);
    cummulativeRateRatio := s(3);
  );

-- Receives the cummulative rate ratio at the slave
-- port. Cummulative rate ratio is sent by the master
-- device attached to the slave port.
public receiveCRR: () ==> ()
receiveCRR() ==
  cummulativeRateRatio:= slavePort.receiveCRR();

-- Sets the current device as the grand master
-- for the network.
public setGM: () ==> ()
setGM() ==
  isGM := true;

-- Calculates the neighbour rate ratio. This calculation
-- is performed at every slave device.
public calculateNeighbourRR : () ==> ()
calculateNeighbourRR () ==
  if (not isGM) then
   neighbourRateRatio := freq / masterFreq
  else
    neighbourRateRatio := 1
```

```
pre not isGM => (freq <> 0 and masterFreq <> 0)
post neighbourRateRatio <> 0;

-- Calculates the new cummulative rate ratio to be sent
-- to the attached slave unit.
public calculateCRateRatio : () ==> ()
calculateCRateRatio () ==
  newCummulativeRateRatio := cummulativeRateRatio *
                             neighbourRateRatio
pre neighbourRateRatio <> 0 and cummulativeRateRatio <> 0;

-- Calculates the new correction field to be sent to the
-- attached slave unit.
public calculateNewCorrectionField : () ==> ()
calculateNewCorrectionField () ==
  if isGM then ()
  else (
    newCorrectionField := linkDelay + residenceTime *
    newCummulativeRateRatio + receivedCorrectionField;);

-- Sends a correction field to the slave unit.
public sendNewCorrectionField : () ==> ()
sendNewCorrectionField () ==
  if (not finalDevice) then
    masterPort.sendCorrectionField(newCorrectionField);

-- Receives a correction field sent by master unit.
public receiveCorrectionField : () ==> ()
receiveCorrectionField() ==
  receivedCorrectionField := slavePort.receiveCorrectionField();

-- Sets the current device as a final device.
-- A final device will not have slave devices and
-- will not be acting as a master device.
public setFinalDevice: () ==> ()
setFinalDevice() ==
  finalDevice := true
pre not isGM
post finalDevice;

-- Receives a synchronization sync event and
-- time stamps it.
public receiveSyncEvent: () ==> ()
receiveSyncEvent () ==
(
  receivedSyncEvent := slavePort.receiveSyncEvent();
  updateTimeRef();
);

```

```
-- Updates the time reference when the time stamp operation
-- is invoked.
public updateTimeRef: () ==> ()
updateTimeRef() ==
(
  IO`print("\n Sync event --> timestamp generated");
  -- syncRecTS := clk.getTime();
);

-- Sends a synchronization event to a slave unit.
public sendSyncEvent: () ==> ()
sendSyncEvent () ==
  if (not finalDevice) then
     masterPort.sendSyncEvent();


-- Shows the device state
public showDevice : () ==> ()
showDevice () ==
(
 IO`print("\nInfo for device: ");
 IO`print(deviceID);
 IO`print("\nIs GM:");
 IO`print(isGM);
 IO`print("\nIs final device: ");
 IO`print(finalDevice);
 IO`print("\nIs sync event received: ");
 IO`print(receivedSyncEvent);
 IO`print("\nSync event received at: ");
 IO`print(syncRecTS);
 IO`print("\nFrequency: ");
 IO`print(freq);
 IO`print("\nMaster frequency: ");
 IO`print(masterFreq);
 IO`print("\nLinkDelay: ");
 IO`print(linkDelay);
 IO`print("\nResidence Time: ");
 IO`print(residenceTime);
 IO`print("\nNeighbour Rate Ratio: ");
 IO`print(neighbourRateRatio);
 IO`print("\nRecCummulative Rate Ratio: ");
 IO`print(cummulativeRateRatio);
 IO`print("\nNew Cumulative Rate Ratio: ");
 IO`print(newCummulativeRateRatio);
 IO`print("\nReceived CorrectionField: ");
 IO`print(receivedCorrectionField);
 IO`print("\nNew calculated CorrectionField: ");
 IO`print(newCorrectionField);
 IO`print("\nPrecise Origin TimeStamp: ");
 IO`print(preciseOriginTS);

 --cSlave.triggerPrint();
);

end Device
```

C.1.5 Main

```
class Main

instance variables

  clock : Clock := new Clock();

  maxSimTime : nat := 0;

  -- Devices use the simulation
  device1 : Device := new Device(30);
  device2 : Device := new Device(10);
  device3 : Device := new Device(20);
  device4 : Device := new Device(10);

  device12 : Device := new Device(20);

  -- Channels used to cummunicate the devices
  ch1_2 : PhyMessage := new PhyMessage();
  ch2_3 : PhyMessage := new PhyMessage();
  ch3_4 : PhyMessage := new PhyMessage();

  ch1_12 : PhyMessage := new PhyMessage();

  testMP : Master := new Master(ch1_12);

  -- Sequence of devices involved in the scenario
  deployment : seq of Device := [device1,device2,device3,device4];

  -- Sequence of channels between the devices
  channels : seq of PhyMessage := [ch1_2,ch2_3,ch3_4];

operations

public run : () ==> ()
run () ==
(

  -- Sets up the network for the simulation
  setupNetwork();

  -- Device 1 operations

  device1.generatePOTS(10);
  device1.sendSyncEvent();

  device1.calculateCRateRatio();
  device1.calculateNewCorrectionField();
  device1.sendFollowUpMessage();

  device12.receiveSyncEvent();

  device12.receiveFollowUpMessage();
  device12.calculateCRateRatio();
  device12.calculateNewCorrectionField();


  -- Device 2 operations

  device2.receiveSyncEvent();
  device2.sendSyncEvent();

  device2.receiveFollowUpMessage();

  device2.calculateResidenceTime();
  device2.calculateCRateRatio();
  device2.calculateNewCorrectionField();

  device2.sendFollowUpMessage();

  -- Device 3 operations

  device3.receiveSyncEvent();
  device3.sendSyncEvent();

  device3.receiveFollowUpMessage();

  device3.notifyClock();
  device3.cSlave.triggerPrint();

  device3.calculateResidenceTime();
  device3.calculateCRateRatio();
  device3.calculateNewCorrectionField();

  device3.sendFollowUpMessage();

  -- Device 4 operations
```

```

  device4.receiveSyncEvent();
  device4.sendSyncEvent();

  device4.receiveFollowUpMessage();
  device4.calculateCRateRatio();
  device4.calculateNewCorrectionField();

  device4.sendFollowUpMessage();

  -- End of model

  showDevices();
  device12.showDevice();

);

public setupNetwork: () ==> ()
setupNetwork() ==
(
  for all i in set {1,...,len deployment}
  do
    deployment(i).setDeviceID(i);

  deployment(1).setGM();
  deployment(len deployment).setFinalDevice();
  device12.setFinalDevice();

  for all i in set {1,...,len deployment - 1}
  do
  (
    deployment(i).masterPort.changePhyChannel(channels(i));
    deployment(i+1).slavePort.changePhyChannel(
       deployment(i).masterPorts(1).getPhyChannel()
    );
  );

  device1.registerMasterPort(testMP);
  device12.slavePort.changePhyChannel(
    device1.masterPorts(2).getPhyChannel()
   );
  device12.setResidenceTime(10);

  -- Execute frequency announcement and
  -- neigbour rate ratio calculations
  -- for the daisy chained devices.
  announceFreqs();
  calculateNeighbourRR();

  -- Execute frequency announcement and
  -- neigbour rate ratio calcualations
  -- for the branched devices

  device12.announceFreq();
  device12.getFreq();
  device12.calculateNeighbourRR();

);

-- Set of actions for the daisy chained devices.
public announceFreqs : () ==> ()
announceFreqs() ==
  for all i in set {1,...,len deployment}
  do
  (
    deployment(i).announceFreq();
    deployment(i).getFreq();
  );


public calculateNeighbourRR : () ==> ()
calculateNeighbourRR() ==
  for all i in set {1,...,len deployment}
    do deployment(i).calculateNeighbourRR();

public showDevices : () ==> ()
showDevices () ==
  for all i in set {1,...,len deployment}
    do deployment(i).showDevice();


end Main
```

C.1.6 Master

```
class Master is subclass of Port

instance variables

  egressTime : real := 12;

operations

public timeStamp : () ==> ()
timeStamp() ==
  egressTime := 10;

public getEgressTime : () ==> real
getEgressTime () ==
  return egressTime;

-- Class constructor.
-- Receives the physical layer associated to the device
public Master : PhyMessage ==> Master
Master(phy) ==
  channel := phy;

-- Changes the physical channel associated to the device
public changePhyChannel : PhyMessage ==> ()
changePhyChannel(ch) ==
  channel := ch;

-- Sends a precise origin time-stamp
public sendPOTS : real ==> ()
sendPOTS(pots) ==
  channel.setPOTS(pots);

-- Sends a synchronization event
public sendSyncEvent : () ==> ()
sendSyncEvent() ==
  channel.signalSyncEvent();

-- Sends the cummulative rate ratio
public sendCRR : real ==> ()
sendCRR(crr) ==
  channel.setCRR(crr);

-- Sends the correction field
public sendCorrectionField : real ==> ()
sendCorrectionField(crf) ==
  channel.setCorrectionField(crf);

-- Sends the followup up message in one call
-- Follow up messsage is composed by Precise Origing Time Stamp,
-- cummulative rate ratio and correction
-- field in one invocation
public sendFollowUpMessage: real * real * real ==> ()
sendFollowUpMessage(crf,pots, crr) ==
  channel.setFollowUp(crf,pots,crr);

-- Announce the frequency of the master device controlling
-- the port
public announceFreq: nat ==> ()
announceFreq(f) ==
  channel.setMasterFreq(f);

end Master
```

C.1.7 PhyMessage

```
class PhyMessage

instance variables

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  -- Frequency sent by a master
  private masterFreq : real := 0;

operations

-- Gets the contents of the follow up message.
public getFollowUp: () ==> seq of real
getFollowUp() ==
  return [correctionField,
          preciseOriginTS,
          cummulativeRateRatio];

-- Sets the contents of the follow up message.
public setFollowUp: real * real * real ==> ()
setFollowUp(correction, pots, crr) ==
(
  setCRR(crr);
  setCorrectionField(correction);
  setPOTS(pots);
);

-- Signals a Synchronization event
public signalSyncEvent: () ==> ()
signalSyncEvent() ==
(
  IO`print("\nMaster port --> Sync event sent");
  syncEvent := true;
);

-- Check if there is a Synchronization event
public getSyncEvent: () ==> bool
getSyncEvent() ==
(
  IO`print("\nSlave port --> Sync event received");
  return syncEvent;
);

-- ##################################################
-- Getters and setters for the different fields in the
-- Physical Message.

public setMasterFreq : real ==> ()
setMasterFreq(f) ==
  masterFreq := f;

public getMasterFreq: () ==> real
getMasterFreq() ==
  return masterFreq;

public setCRR: real ==> ()
setCRR(crr) ==
  cummulativeRateRatio := crr;

public setCorrectionField: real ==> ()
setCorrectionField(crf) ==
  correctionField := crf;

public setPOTS: real ==> ()
setPOTS(pots) ==
  preciseOriginTS := pots;

public getCField: () ==> real
getCField() ==
  return correctionField;

public getPOTS: () ==> real
getPOTS() ==
 return preciseOriginTS;

public getCRR: () ==> real
getCRR() ==
  return cummulativeRateRatio;

sync

  mutex(getFollowUp,setFollowUp);
  mutex(signalSyncEvent,getSyncEvent);

end PhyMessage
```

C.1.8 Port

```
class Port

instance variables

  -- Physical channel associated to the port
  protected channel : PhyMessage;

  -- Associated ID to the port
  protected portID : nat;

  protected clk : Clock;

operations

public timeStamp : () ==> ()
timeStamp() ==
  is not yet specified;

-- Change the associated channel to the slave
public changePhyChannel : PhyMessage ==> ()
changePhyChannel(ch) ==
  channel := ch;

public getPhyChannel : () ==> PhyMessage
getPhyChannel() ==
  return channel;

-- Changes the associated ID to the port
public changePortID: nat ==> ()
changePortID(id) ==
  portID := id;

end Port
```

C.1.9 Slave

```
class Slave is subclass of Port
```

```

instance variables

  ingressTime : real := 2;

operations

public timeStamp : () ==> ()
timeStamp() ==
  ingressTime := 2;

public getIngressTime : () ==> real
getIngressTime () ==
  return ingressTime;

-- Class constructor
public Slave: PhyMessage ==> Slave
Slave(phy) ==
  channel := phy;

-- Receives the cummulative rate ratio
public receiveCRR: () ==> real
receiveCRR() ==
  channel.getCRR();

-- Receives the correction field
public receiveCorrectionField: () ==> real
receiveCorrectionField() ==
  channel.getCField();

-- Receives the precise origin time stamp
public receivePOTS : () ==> real
receivePOTS() ==
  channel.getPOTS();

-- Receives the follow up message
-- Receives Precise Origing Time Stamp, cummulative rate ratio
-- and correction field in one invocation
public receiveFollowUpMessage: () ==> seq of real
receiveFollowUpMessage() ==
  channel.getFollowUp();

-- Checks if there is a synchronization event
-- available
public receiveSyncEvent: () ==> bool
receiveSyncEvent() ==
 channel.getSyncEvent();

-- Gets the frequency associated to the master
```

```
public getFreq: () ==> real
getFreq() ==
  return channel.getMasterFreq();

end Slave
```

C.2. Sequential model

C.2.1 App

```
class App

instance variables

  private preciseOriginTS : real;
  private correctionField : real;

  private localTimeStamp : real;

  private timeInfoReady : bool := false;


operations

public receiveTimeInformation : seq of real ==> ()
receiveTimeInformation (time) ==
(
  preciseOriginTS := time(1);
  correctionField := time(2);
  localTimeStamp := time(3);

  timeInfoReady := true;
);

public showReceivedTime : () ==> ()
showReceivedTime() ==
(
  if timeInfoReady then
  (
  IO`print("\nAt time: ");
  IO`print(localTimeStamp);
  IO`print("\nThe Grand Master time was: ");
  IO`print(preciseOriginTS);
  IO`print("\nIt has been corrected with: ");
  IO`print(correctionField);
  IO`print("\nFinal time for previos timestamp is: ");
  IO`print(preciseOriginTS+correctionField);
  )
  else
    IO`print("\n Message from application:
             Time information not Ready");
);

end App
```

C.2.2 Clock

```
class Clock

instance variables

  time : nat := 0;

operations

public tick: () ==> ()
tick() ==
(
  time := time +1;
);

public getTime: () ==> nat
getTime() ==
  return time;

end Clock
```

C.2.3 ClockSlave

```
class ClockSlave

instance variables

  private siteSync : Device;

  private preciseOriginTS : real;
  private correctionField : real;
  private localTimeStamp : real;

  private timeConsumers : seq of App := [];

  private newTimeInfoReady : bool := false;

operations

public registerApp : App ==> ()
registerApp(ap) ==
  timeConsumers := timeConsumers ^ [ap];

public setTimeInfo: seq of real ==> ()
setTimeInfo(time) ==
(
  if len time > 2 then
  (
    preciseOriginTS := time(1);
    correctionField := time(2);
    localTimeStamp := time(3);
    newTimeInfoReady := true;
  )
);

public timeStep: () ==> ()
timeStep() ==
  if newTimeInfoReady then
  (
    notifyTime();
    newTimeInfoReady := false
  )
  else
    skip;

public notifyTime : () ==> ()
notifyTime() ==
  let t : seq of real =
  [preciseOriginTS,correctionField,localTimeStamp] in
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).receiveTimeInformation(t);

public setSiteSync : Device ==> ()
setSiteSync(dv) ==
  siteSync := dv;

public getTime:() ==> ()
getTime() ==
();

```

```
public ClockSlave: App ==> ClockSlave
ClockSlave(ap) ==
(
  registerApp(ap);
);

public triggerPrint: () ==> ()
triggerPrint() ==
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).showReceivedTime();

end ClockSlave
```

C.2.4 Device

```
class Device

instance variables

  -- Device ID number
  private deviceID : nat := 0;

  -- Set device as a bridge
  public isBridge : bool := true;

  -- Frequency of the current device.
  private freq : nat := 0;

  -- Slave port associated to the device
  public slavePort : Slave;

  -- Device with no external slaves connected
 -- (finalDevice = true)
  public finalDevice : bool := false;
  --inv finalDevice => len masterPorts = 0;

  -- Current device is Grand Master -> iGM = true
  -- if device isGM it should not have masters attached.
  public isGM : bool := false;
  inv isGM => not finalDevice;
  inv finalDevice => not isGM;

  -- Frequency of the master device
  private masterFreq : nat := 0;

  private receivedSyncEvent : bool := false;
  private syncRecTS : nat := 0;

  -- Neighbour rate ratio measured by the slave port
  private neighbourRateRatio : real := 1;

  -- Precise origin timestamp, defined by GM and
  -- never changed.
  private preciseOriginTS : real := 0;

  -- received correction field, to be recomputed
  -- before use.
  private receivedCorrectionField : real := 0;

  -- CumulativeRateRatio received from the previous
  -- node.
  private cummulativeRateRatio : real := 1;
  -- Delay from master to slave port
  private linkDelay : real := 10;
  -- Time that has been spent in the current node
  private residenceTime : real := 0;

  -- Correction field calculated at the current node,
  -- to be used in the current node.
  private newCorrectionField : real := 0;
  -- CummulativeRateRatio calculated at the current node,
  -- to be used in the current node.
  private newCummulativeRateRatio : real := 0;

  -- Clock attached to the device
  private clk : Clock;

  private timeInfoAvailable : bool;

  public cSlave : ClockSlave := new ClockSlave();

  private followUpMessageReceived : bool := false;

operations

public setClockSlave: ClockSlave  ==> ()
setClockSlave(c) ==
  cSlave := c;

-- Actions carried out at the device per time-step
public timeStep: () ==> ()
timeStep() ==
(
  if receivedSyncEvent = false then
  (   receiveSyncEvent(); IO`print("#syncEvent#");)
  else if receivedSyncEvent and not followUpMessageReceived then
  (  receiveFollowUpMessage(); IO`print("Follow up received");)
  else if receivedSyncEvent and followUpMessageReceived then
  (  notifyClock(); IO`print("CLocks notified"););
);


public notifyClock : () ==> ()
notifyClock() ==
 cSlave.setTimeInfo(
    [preciseOriginTS,receivedCorrectionField,syncRecTS]);

public makeTimeAvailable : () ==> ()
makeTimeAvailable() ==
  timeInfoAvailable := true;

public getTimeInfo : () ==> seq of real
getTimeInfo() ==
  return [
    preciseOriginTS,
    receivedCorrectionField,
    cummulativeRateRatio];

public timeInfoRead : () ==> ()
timeInfoRead() ==
  timeInfoAvailable := false;

-- Sets the slave port that corresponds to the current device
public setSlavePort: Slave ==> ()
setSlavePort(s) ==
  slavePort := s;

-- Sets device ID
public setDeviceID: nat ==> ()
setDeviceID(id) ==
  deviceID := id
pre deviceID = 0
post deviceID <> 0;

-- Gets device ID
public getDeviceID: () ==> nat
getDeviceID() ==
  return deviceID
pre deviceID <> 0;

-- Receives the POTS
public receivePOTS: () ==> ()
receivePOTS() ==
  preciseOriginTS := slavePort.receivePOTS();
```

```

public setLocalFreq : nat ==> ()
setLocalFreq(f) ==
  freq := f;

-- Sets the frequency of the master device.
public setMasterFreq : nat ==> ()
setMasterFreq (f) ==
  masterFreq := f
pre f <> 0;

-- Get the frequency of the master device attached to
-- the slave port.
public getFreq : () ==> ()
getFreq() ==
  if (not isGM) then setMasterFreq(slavePort.getFreq());


-- Receives the follow up message sent by the master.
-- Reception is performed in one operation.
public receiveFollowUpMessage: () ==> ()
receiveFollowUpMessage() ==
(  let s : seq of real = slavePort.receiveFollowUpMessage() in
  (
    receivedCorrectionField := s(1);
    preciseOriginTS := s(2);
    cummulativeRateRatio := s(3);
  );
  followUpMessageReceived := true;
);

-- Receives the cummulative rate ratio at the slave
-- port. Cummulative rate ratio is sent by the master
-- device attached to the slave port.
public receiveCRR: () ==> ()
receiveCRR() ==
  cummulativeRateRatio:= slavePort.receiveCRR();

-- Calculates the neighbour rate ratio. This calculation
-- is performed at every slave device.
public calculateNeighbourRR : () ==> ()
calculateNeighbourRR () ==
  if (not isGM) then
   neighbourRateRatio := freq / masterFreq
  else
    neighbourRateRatio := 1
pre not isGM => (freq <> 0 and masterFreq <> 0)
post neighbourRateRatio <> 0;

```

```
-- Calculates the new cummulative rate ratio to be sent
-- to the attached slave unit.
public calculateCRateRatio : () ==> ()
calculateCRateRatio () ==
  newCummulativeRateRatio :=
    cummulativeRateRatio * neighbourRateRatio
pre neighbourRateRatio <> 0 and cummulativeRateRatio <> 0;

-- Calculates the new correction field to be sent to the
-- attached slave unit.
public calculateNewCorrectionField : () ==> ()
calculateNewCorrectionField () ==
  if isGM then ()
  else (
    newCorrectionField := linkDelay + residenceTime *
    newCummulativeRateRatio + receivedCorrectionField;);

-- Receives a correction field sent by master unit.
public receiveCorrectionField : () ==> ()
receiveCorrectionField() ==
  receivedCorrectionField := slavePort.receiveCorrectionField();

-- Sets the current device as a final device.
-- A final device will not have slave devices and
-- will not be acting as a master device.
public setFinalDevice: () ==> ()
setFinalDevice() ==
  finalDevice := true
pre not isGM
post finalDevice;

-- Receives a synchronization sync event and
-- time stamps it.
public receiveSyncEvent: () ==> ()
receiveSyncEvent () ==
(
  receivedSyncEvent := slavePort.receiveSyncEvent();
  receivedSyncEvent := true;
  updateTimeRef();
);

-- Updates the time reference when the time stamp operation
-- is invoked.
public updateTimeRef: () ==> ()
updateTimeRef() ==
(
  --IO`print("\n Sync event --> timestamp generated");
  syncRecTS := slavePort.getIngressTime();
);
```

```

-- Shows the device state
public showDevice : () ==> ()
showDevice () ==
(
 IO`print("\nInfo for device: ");
 IO`print(deviceID);
 IO`print("\nIs GM:");
 IO`print(isGM);
 IO`print("\nIs final device: ");
 IO`print(finalDevice);
 IO`print("\nIs sync event received: ");
 IO`print(receivedSyncEvent);
 IO`print("\nSync event received at: ");
 IO`print(syncRecTS);
 IO`print("\nFrequency: ");
 IO`print(freq);
 IO`print("\nMaster frequency: ");
 IO`print(masterFreq);
 IO`print("\nLinkDelay: ");
 IO`print(linkDelay);

 IO`print("\nNeighbour Rate Ratio: ");
 IO`print(neighbourRateRatio);
 IO`print("\nRecCummulative Rate Ratio: ");
 IO`print(cummulativeRateRatio);

 IO`print("\nReceived CorrectionField: ");
 IO`print(receivedCorrectionField);
 IO`print("\nPrecise Origin TimeStamp: ");
 IO`print(preciseOriginTS);
);

end Device
```

C.2.5 Main

```
class Main

types

  public timeStimulus = seq of real;

instance variables

  -- Modelled entities in the system
  siteSync : Device;
  app : App;
  clkSlave : ClockSlave;
  slave : Slave;
  channel : PhyMessage;

  -- Auxiliar simulation entities
  simClk : Clock := new Clock();

  simSteps : nat := 3;

  timeStimuli : seq of timeStimulus := [];

operations

public loadStimuli: () ==> ()
loadStimuli() ==
(
 --loadStimulus([20,30,50]);
  let i : seq of timeStimulus = [[40,20,10],[30,80,90]] in
    for all e in set {1,...,len i} do
      loadStimulus(i(e));
);

public loadStimulus: timeStimulus ==> ()
loadStimulus(s) ==
  timeStimuli := timeStimuli ^ [s];

-- Model simulation entry point
public run: () ==> ()
run() ==
(
  IO`print("\nEndpoint device receiving time corrections");

  loadStimuli();
  IO`print(timeStimuli);
  setup();

  -- Simulation starts

  feedPhyMessage();

  for all i in set {1,...,simSteps} do
  (
    simClk.tick();
    slave.timeStep();
    siteSync.timeStep();
    clkSlave.timeStep();
  );

  -- Shows the simulation results
  siteSync.showDevice();
  app.showReceivedTime();

  IO`print("\nModel over\n");
  IO`print("####### Unused time stimuli: ");
  IO`print(timeStimuli);

);

public feedPhyMessage: () ==> ()
feedPhyMessage() ==
(
  channel.signalSyncEvent();
  channel.setFollowUp(hd timeStimuli);
  timeStimuli := tl timeStimuli;
);

-- Setup the simulation:
--   Creates the involved entities
--   Establish the relations between them
public setup: () ==> ()
setup() ==
(
  -- Creation of the entities present in a Time Aware System
  -- endpoint device
  siteSync := new Device();
  slave := new Slave();
  clkSlave := new ClockSlave();
  app := new App();
  channel := new PhyMessage();

  slave.clk := simClk;

  -- Establish the relationships between the entities

  -- 1. Register the Application in the clock slave entity
  clkSlave.registerApp(app);
  clkSlave.setSiteSync(siteSync);

  -- 2. Associate the clock slave to the siteSync entity
  siteSync.setClockSlave(clkSlave);

  -- 3. Configure the physical layer and associate it to the
  --   slave port
  slave.changePhyChannel(channel);

  -- 4. Associate the slave port to the siteSync entity
  siteSync.setSlavePort(slave);

  channel.setMasterFreq(10);
  siteSync.setLocalFreq(20);
  siteSync.getFreq();
  siteSync.calculateNeighbourRR();

  siteSync.setFinalDevice();
);

end Main
```

C.2.6 PhyMessage

```
class PhyMessage

instance variables

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  -- Frequency sent by a master
  private masterFreq : real := 0;

operations

-- Gets the contents of the follow up message.
public getFollowUp: () ==> seq of real
getFollowUp() ==
  return [correctionField,
          preciseOriginTS,
          cummulativeRateRatio];

-- Sets the contents of the follow up message.
public setFollowUp: seq of real ==> ()
setFollowUp(s) ==
(
  setCRR(s(1));
  setCorrectionField(s(2));
  setPOTS(s(3));
);

-- Signals a Synchronization event
public signalSyncEvent: () ==> ()
signalSyncEvent() ==
(
  --IO‘print("\nMaster port --> Sync event sent");
  syncEvent := true;
);

-- Check if there is a Synchronization event
public getSyncEvent: () ==> bool
getSyncEvent() ==
(
  --IO‘print("\nSlave port --> Sync event received");
  return syncEvent;
);

-- ################################################
-- Getters and setters for the different fields in the
-- Physical Message.

public setMasterFreq : real ==> ()
setMasterFreq(f) ==
  masterFreq := f;

public getMasterFreq: () ==> real
getMasterFreq() ==
  return masterFreq;

public setCRR: real ==> ()
setCRR(crr) ==
  cummulativeRateRatio := crr;

public setCorrectionField: real ==> ()
setCorrectionField(crf) ==
  correctionField := crf;

public setPOTS: real ==> ()
setPOTS(pots) ==
  preciseOriginTS := pots;

public getCField: () ==> real
getCField() ==
  return correctionField;

public getPOTS: () ==> real
getPOTS() ==
 return preciseOriginTS;
```

```

public getCRR: () ==> real
getCRR() ==
  return cummulativeRateRatio;

sync

  mutex(getFollowUp,setFollowUp);
  mutex(signalSyncEvent,getSyncEvent);

end PhyMessage
```

C.2.7 Port

```
class Port

instance variables

  -- Physical channel associated to the port
  protected channel : PhyMessage;

  -- Associated ID to the port
  protected portID : nat;

  protected clk : Clock;

operations

public timeStamp : () ==> ()
timeStamp() ==
  is not yet specified;

public timeStep : () ==> ()
timeStep() ==
  is not yet specified;

-- Change the associated channel to the slave
public changePhyChannel : PhyMessage ==> ()
changePhyChannel(ch) ==
  channel := ch;

public getPhyChannel : () ==> PhyMessage
getPhyChannel() ==
  return channel;

-- Changes the associated ID to the port
```

```
public changePortID: nat ==> ()
changePortID(id) ==
  portID := id;

end Port
```

C.2.8 Slave

```
class Slave is subclass of Port

instance variables

  ingressTime : real := 0;

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  private followUpReady : bool := false;

  public clk : Clock;


operations

public checkSync : () ==> ()
checkSync() ==
(
  syncEvent := channel.getSyncEvent();
  if syncEvent = true then
  timeStamp(); -- Time stamp
);

public checkFollowUp: () ==> ()
checkFollowUp() ==
(
  let i : seq of real = channel.getFollowUp() in
  (
    if len i = 0 then
      followUpReady := false
    else
    (
      correctionField := i(1);
      preciseOriginTS := i(2);
      cummulativeRateRatio := i(3);
    )
  );
);

public timeStamp : () ==> ()
timeStamp() ==
  ingressTime := clk.getTime();

public getIngressTime : () ==> real
getIngressTime () ==
  return ingressTime;

-- Class constructor
public Slave: PhyMessage ==> Slave
Slave(phy) ==
  channel := phy;

-- Receives the cummulative rate ratio
public receiveCRR: () ==> real
receiveCRR() ==
  channel.getCRR();

-- Receives the correction field
public receiveCorrectionField: () ==> real
receiveCorrectionField() ==
  channel.getCField();

-- Receives the precise origin time stamp
public receivePOTS : () ==> real
receivePOTS() ==
  channel.getPOTS();

-- Receives the follow up message
-- Receives Precise Origing Time Stamp, cummulative rate ratio
-- and correction field in one invocation
public receiveFollowUpMessage: () ==> seq of real
receiveFollowUpMessage() ==
  return [correctionField,preciseOriginTS,cummulativeRateRatio];

-- Checks if there is a synchronization event
-- available
public receiveSyncEvent: () ==> bool
receiveSyncEvent() ==
  return syncEvent;

```

```
-- Gets the frequency associated to the master
public getFreq: () ==> real
getFreq() ==
  return channel.getMasterFreq();

public timeStep: () ==> ()
timeStep() ==
(
  if not syncEvent then
    checkSync()
  else
    if not followUpReady then
      checkFollowUp();
);

end Slave
```

C.3. Concurrent model

C.3.1 App

```
class App

instance variables

  private preciseOriginTS : real;
  private correctionField : real;

  private localTimeStamp : real;

  private timeInfoReady : bool := false;

  private timeInfoProvided : bool := false;


operations

public receiveTimeInformation : seq of real ==> ()
receiveTimeInformation (time) ==
(
  preciseOriginTS := time(1);
  correctionField := time(2);
  localTimeStamp := time(3);

  timeInfoReady := true;

  markInfoProvided();

);

public showReceivedTime : () ==> ()
showReceivedTime() ==
(
  if timeInfoReady then
  (
  IO`print("\n#########################################");
  IO`print("\nAt time: ");
  IO`print(localTimeStamp);
  IO`print("\nThe Grand Master time was: ");
  IO`print(preciseOriginTS);
  IO`print("\nIt has been corrected with: ");
  IO`print(correctionField);
  IO`print("\nFinal time for previos timestamp is: ");
  IO`print(preciseOriginTS+correctionField);
  IO`print("\n#########################################");

  markInfoProvided();
  )
  else
    IO`print("\n Message from application:
        Time information not Ready");
);

private markInfoProvided : () ==> ()
markInfoProvided() ==
  timeInfoProvided := true;

public infoProvided : () ==> bool
infoProvided() ==
  return timeInfoProvided;



sync
  mutex(markInfoProvided,infoProvided);
  mutex(receiveTimeInformation);
  mutex(showReceivedTime, infoProvided);
  mutex(showReceivedTime,receiveTimeInformation);

end App
```

C.3.2 Clock

```
class Clock

instance variables

  time : nat := 0;

operations

public tick: () ==> ()
tick() ==
(
  time := time +1;
);

public getTime: () ==> nat
getTime() ==
  return time;

thread
  while true do
  (
    tick();
    Environment`timerRef.WaitRelative(1);
  );

sync
  mutex(tick,getTime);
  mutex(tick);

end Clock
```

C.3.3 ClockSlave

```
class ClockSlave

instance variables

  private siteSync : Device;

  private preciseOriginTS : real;
  private correctionField : real;
  private localTimeStamp : real;

  private timeConsumers : seq of App := [];

  private newTimeInfoReady : bool := false;

operations

public registerApp : App ==> ()
registerApp(ap) ==
  timeConsumers := timeConsumers ^ [ap];

public setTimeInfo: seq of real ==> ()
setTimeInfo(time) ==
(
  if len time > 2 then
  (
    preciseOriginTS := time(1);
    correctionField := time(2);
    localTimeStamp := time(3);
    newTimeInfoReady := true;
  )
);

public timeStep: () ==> ()
timeStep() ==
  if newTimeInfoReady then
  (
    notifyTime();
    newTimeInfoReady := false;
  )
  else
    skip;

public notifyTime : () ==> ()
notifyTime() ==
  let t : seq of real =
    [preciseOriginTS,correctionField,localTimeStamp] in
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).receiveTimeInformation(t);

public setSiteSync : Device ==> ()
setSiteSync(dv) ==
  siteSync := dv;

public getTime:() ==> ()
getTime() ==
();

public ClockSlave: App ==> ClockSlave
ClockSlave(ap) ==
(
  registerApp(ap);
);

public triggerPrint: () ==> ()
triggerPrint() ==
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).showReceivedTime();

thread
  while true do
  (
    timeStep();
    Environment`timerRef.WaitRelative(1);
  );

sync
  mutex(timeStep,setTimeInfo);
  mutex(notifyTime,setTimeInfo);

end ClockSlave
```

C.3.4 Device

```
class Device

instance variables

  -- Device ID number
  private deviceID : nat := 0;

  -- Set device as a bridge
  public isBridge : bool := true;

  -- Frequency of the current device.
  private freq : nat := 0;

  -- Slave port associated to the device
  public slavePort : Slave;

  -- Device with no external slaves connected
  -- (finalDevice = true)
  public finalDevice : bool := false;
  --inv finalDevice => len masterPorts = 0;

  -- Current device is Grand Master -> iGM = true
  -- if device isGM it should not have masters attached.
  public isGM : bool := false;
  inv isGM => not finalDevice;
  inv finalDevice => not isGM;

  -- Frequency of the master device
  private masterFreq : nat := 0;

  private receivedSyncEvent : bool := false;
  private syncRecTS : nat := 0;

  -- Neighbour rate ratio measured by the slave port
  private neighbourRateRatio : real := 1;

  -- Precise origin timestamp, defined by GM and
  -- never changed.
  private preciseOriginTS : real := 0;

  -- received correction field, to be recomputed
  -- before use.
  private receivedCorrectionField : real := 0;

  -- CumulativeRateRatio received from the previous
  -- node.
  private cummulativeRateRatio : real := 1;
  -- Delay from master to slave port
  private linkDelay : real := 10;
  -- Time that has been spent in the current node
  private residenceTime : real := 0;

  -- Correction field calculated at the current node,
  -- to be used in the current node.
  private newCorrectionField : real := 0;
  -- CummulativeRateRatio calculated at the current node,
  -- to be used in the current node.
  private newCummulativeRateRatio : real := 0;

  -- Clock attached to the device
  private clk : Clock;

  private timeInfoAvailable : bool;

  public cSlave : ClockSlave := new ClockSlave();

  private followUpMessageReceived : bool := false;

operations

public setClockSlave: ClockSlave  ==> ()
```

```
setClockSlave(c) ==
  cSlave := c;

-- Actions carried out at the device per time-step
public timeStep: () ==> ()
timeStep() ==
(
  if receivedSyncEvent = false then
  (   receiveSyncEvent(); IO`print("#syncEvent#");)
  else if receivedSyncEvent and not followUpMessageReceived then
  (  receiveFollowUpMessage(); IO`print("Follow up received");)
  else if receivedSyncEvent and followUpMessageReceived then
  (  notifyClock(); IO`print("CLocks notified"););
);


public notifyClock : () ==> ()
notifyClock() ==
 cSlave.setTimeInfo(
    [preciseOriginTS,receivedCorrectionField,syncRecTS]
  );

public makeTimeAvailable : () ==> ()
makeTimeAvailable() ==
  timeInfoAvailable := true;

public getTimeInfo : () ==> seq of real
getTimeInfo() ==
  return [preciseOriginTS,
          receivedCorrectionField,
          cummulativeRateRatio];

public timeInfoRead : () ==> ()
timeInfoRead() ==
  timeInfoAvailable := false;

-- Sets the slave port that corresponds to the current device
public setSlavePort: Slave ==> ()
setSlavePort(s) ==
  slavePort := s;

-- Sets device ID
public setDeviceID: nat ==> ()
setDeviceID(id) ==
  deviceID := id
pre deviceID = 0
post deviceID <> 0;

-- Gets device ID
```

```
public getDeviceID: () ==> nat
getDeviceID() ==
  return deviceID
pre deviceID <> 0;

-- Receives the POTS
public receivePOTS: () ==> ()
receivePOTS() ==
  preciseOriginTS := slavePort.receivePOTS();

public setLocalFreq : nat ==> ()
setLocalFreq(f) ==
  freq := f;

-- Sets the frequency of the master device.
public setMasterFreq : nat ==> ()
setMasterFreq (f) ==
  masterFreq := f
pre f <> 0;

-- Get the frequency of the master device attached to
-- the slave port.
public getFreq : () ==> ()
getFreq() ==
  if (not isGM) then setMasterFreq(slavePort.getFreq());


-- Receives the follow up message sent by the master.
-- Reception is performed in one operation.
public receiveFollowUpMessage: () ==> ()
receiveFollowUpMessage() ==
(  let s : seq of real = slavePort.receiveFollowUpMessage() in
  (
    receivedCorrectionField := s(1);
    preciseOriginTS := s(2);
    cummulativeRateRatio := s(3);
  );
  followUpMessageReceived := true;
);

-- Receives the cummulative rate ratio at the slave
-- port. Cummulative rate ratio is sent by the master
-- device attached to the slave port.
public receiveCRR: () ==> ()
receiveCRR() ==
  cummulativeRateRatio:= slavePort.receiveCRR();

-- Calculates the neighbour rate ratio. This calculation
-- is performed at every slave device.
```

```
public calculateNeighbourRR : () ==> ()
calculateNeighbourRR () ==
  if (not isGM) then
   neighbourRateRatio := freq / masterFreq
  else
    neighbourRateRatio := 1
pre not isGM => (freq <> 0 and masterFreq <> 0)
post neighbourRateRatio <> 0;

-- Calculates the new cummulative rate ratio to be sent
-- to the attached slave unit.
public calculateCRateRatio : () ==> ()
calculateCRateRatio () ==
  newCummulativeRateRatio :=
    cummulativeRateRatio * neighbourRateRatio
pre neighbourRateRatio <> 0 and cummulativeRateRatio <> 0;

-- Calculates the new correction field to be sent to the
-- attached slave unit.
public calculateNewCorrectionField : () ==> ()
calculateNewCorrectionField () ==
  if isGM then ()
  else (
    newCorrectionField := linkDelay + residenceTime *
    newCummulativeRateRatio + receivedCorrectionField;);

-- Receives a correction field sent by master unit.
public receiveCorrectionField : () ==> ()
receiveCorrectionField() ==
  receivedCorrectionField := slavePort.receiveCorrectionField();

-- Sets the current device as a final device.
-- A final device will not have slave devices and
-- will not be acting as a master device.
public setFinalDevice: () ==> ()
setFinalDevice() ==
  finalDevice := true
pre not isGM
post finalDevice;

-- Receives a synchronization sync event and
-- time stamps it.
public receiveSyncEvent: () ==> ()
receiveSyncEvent () ==
(
  receivedSyncEvent := slavePort.receiveSyncEvent();
  receivedSyncEvent := true;
  updateTimeRef();
);
```

```

-- Updates the time reference when the time stamp operation
-- is invoked.
public updateTimeRef: () ==> ()
updateTimeRef() ==
(
  --IO`print("\n Sync event --> timestamp generated");
  syncRecTS := slavePort.getIngressTime();
);

-- Shows the device state
public showDevice : () ==> ()
showDevice () ==
(
 IO`print("\nInfo for device: ");
 IO`print(deviceID);
 IO`print("\nIs GM:");
 IO`print(isGM);
 IO`print("\nIs final device: ");
 IO`print(finalDevice);
 IO`print("\nIs sync event received: ");
 IO`print(receivedSyncEvent);
 IO`print("\nSync event received at: ");
 IO`print(syncRecTS);
 IO`print("\nFrequency: ");
 IO`print(freq);
 IO`print("\nMaster frequency: ");
 IO`print(masterFreq);
 IO`print("\nLinkDelay: ");
 IO`print(linkDelay);

 IO`print("\nNeighbour Rate Ratio: ");
 IO`print(neighbourRateRatio);
 IO`print("\nRecCummulative Rate Ratio: ");
 IO`print(cummulativeRateRatio);

 IO`print("\nReceived CorrectionField: ");
 IO`print(receivedCorrectionField);
 IO`print("\nPrecise Origin TimeStamp: ");
 IO`print(preciseOriginTS);

);

thread
  while true do
  (
    timeStep();
    Environment`timerRef.WaitRelative(1);
  );
```

```

end Device
```

C.3.5 Environment

```
class Environment

types

  public timeStimulus = seq of real;

instance variables

  -- Modelled entities in the system
  siteSync : Device;
  app : App;
  clkSlave : ClockSlave;
  slave : Slave;
  channel : PhyMessage;

  -- Auxiliar simulation entities
  simClk : Clock := new Clock();

  simSteps : nat := 3;

  timeStimuli : seq of timeStimulus := [];

  public static timerRef : TimeStamp := new TimeStamp(5);

operations

public loadStimuli: () ==> ()
loadStimuli() ==
(
 --loadStimulus([20,30,50]);
  let i : seq of timeStimulus = [[40,20,10],[30,80,90]] in
    for all e in set {1,...,len i} do
      loadStimulus(i(e));
);

public loadStimulus: timeStimulus ==> ()
loadStimulus(s) ==
  timeStimuli := timeStimuli ^ [s];

-- Model simulation entry point
public run: () ==> ()
```

```
run() ==
(
  IO`print("\nEndpoint device receiving time corrections");

  loadStimuli();
  IO`print(timeStimuli);
  setup();

  -- Simulation starts

  start(simClk);
  start(slave);
  start(siteSync);
  start(clkSlave);

  feedPhyMessage();

/*
  for all i in set {1,...,simSteps} do
  (
    simClk.tick();
    slave.timeStep();
    siteSync.timeStep();
    clkSlave.timeStep();
  );*/

  while app.infoProvided() = false do
  (
    wait();
    timerRef.WaitRelative(1);
  );

  -- Shows the simulation results
  siteSync.showDevice();
  app.showReceivedTime();

  IO`print("\nModel over\n");
  IO`print("####### Unused time stimuli: ");
  IO`print(timeStimuli);

);

public feedPhyMessage: () ==> ()
feedPhyMessage() ==
(
  channel.signalSyncEvent();
  channel.setFollowUp(hd timeStimuli);
  timeStimuli := tl timeStimuli;
);
```

```

-- Setup the simulation:
--   Creates the involved entities
--   Establish the relations between them
public setup: () ==> ()
setup() ==
(
  -- Creation of the entities present in a Time Aware System
  -- endpoint device
  siteSync := new Device();
  slave := new Slave();
  clkSlave := new ClockSlave();
  app := new App();
  channel := new PhyMessage();

  slave.clk := simClk;

  -- Establish the relationships between the entities

  -- 1. Register the Application in the clock slave entity
  clkSlave.registerApp(app);
  clkSlave.setSiteSync(siteSync);

  -- 2. Associate the clock slave to the siteSync entity
  siteSync.setClockSlave(clkSlave);

  -- 3. Configure the physical layer and associate it to the
  --   slave port
  slave.changePhyChannel(channel);

  -- 4. Associate the slave port to the siteSync entity
  siteSync.setSlavePort(slave);

  channel.setMasterFreq(10);
  siteSync.setLocalFreq(20);
  siteSync.getFreq();
  siteSync.calculateNeighbourRR();

  siteSync.setFinalDevice();
);

wait: () ==> ()
wait() ==
  skip;

end Environment
```

C.3.6 PhyMessage

```
class PhyMessage

instance variables

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  -- Frequency sent by a master
  private masterFreq : real := 0;

operations

-- Gets the contents of the follow up message.
public getFollowUp: () ==> seq of real
getFollowUp() ==
  return [correctionField,
          preciseOriginTS,
          cummulativeRateRatio];

-- Sets the contents of the follow up message.
public setFollowUp: seq of real ==> ()
setFollowUp(s) ==
(
  setCRR(s(1));
  setCorrectionField(s(2));
  setPOTS(s(3));
);

-- Signals a Synchronization event
public signalSyncEvent: () ==> ()
signalSyncEvent() ==
(
  --IO`print("\nMaster port --> Sync event sent");
  syncEvent := true;
);

-- Check if there is a Synchronization event
public getSyncEvent: () ==> bool
getSyncEvent() ==
(
  --IO`print("\nSlave port --> Sync event received");
  return syncEvent;
);

-- ####################################################
-- Getters and setters for the different fields in the
-- Physical Message.

public setMasterFreq : real ==> ()
setMasterFreq(f) ==
  masterFreq := f;

public getMasterFreq: () ==> real
getMasterFreq() ==
  return masterFreq;

public setCRR: real ==> ()
setCRR(crr) ==
  cummulativeRateRatio := crr;

public setCorrectionField: real ==> ()
setCorrectionField(crf) ==
  correctionField := crf;

public setPOTS: real ==> ()
setPOTS(pots) ==
  preciseOriginTS := pots;

public getCField: () ==> real
getCField() ==
  return correctionField;

public getPOTS: () ==> real
getPOTS() ==
 return preciseOriginTS;

public getCRR: () ==> real
getCRR() ==
  return cummulativeRateRatio;

sync

  mutex(getFollowUp,setFollowUp);
  mutex(signalSyncEvent,getSyncEvent);

end PhyMessage
```

C.3.7 Port

```
class Port

instance variables

  -- Physical channel associated to the port
  protected channel : PhyMessage;

  -- Associated ID to the port
  protected portID : nat;

  protected clk : Clock;

operations

public timeStamp : () ==> ()
timeStamp() ==
  is not yet specified;

public timeStep : () ==> ()
timeStep() ==
  is not yet specified;

-- Change the associated channel to the slave
public changePhyChannel : PhyMessage ==> ()
changePhyChannel(ch) ==
  channel := ch;

public getPhyChannel : () ==> PhyMessage
getPhyChannel() ==
  return channel;

-- Changes the associated ID to the port
public changePortID: nat ==> ()
changePortID(id) ==
  portID := id;

end Port
```

C.3.8 Slave

```
class Slave is subclass of Port

instance variables
```

```

  ingressTime : real := 0;

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  private followUpReady : bool := false;

  public clk : Clock;


operations

public checkSync : () ==> ()
checkSync() ==
(
  syncEvent := channel.getSyncEvent();
  if syncEvent = true then
  timeStamp(); -- Time stamp
);

public checkFollowUp: () ==> ()
checkFollowUp() ==
(
  let i : seq of real = channel.getFollowUp() in
  (
    if len i = 0 then
      followUpReady := false
    else
    (
      correctionField := i(1);
      preciseOriginTS := i(2);
      cummulativeRateRatio := i(3);
    )
  );
);

public timeStamp : () ==> ()
timeStamp() ==
  ingressTime := clk.getTime();

public getIngressTime : () ==> real
getIngressTime () ==
  return ingressTime;

-- Class constructor
public Slave: PhyMessage ==> Slave
Slave(phy) ==
  channel := phy;

-- Receives the cummulative rate ratio
public receiveCRR: () ==> real
receiveCRR() ==
  channel.getCRR();

-- Receives the correction field
public receiveCorrectionField: () ==> real
receiveCorrectionField() ==
  channel.getCField();

-- Receives the precise origin time stamp
public receivePOTS : () ==> real
receivePOTS() ==
  channel.getPOTS();

-- Receives the follow up message
-- Receives Precise Origing Time Stamp, cummulative rate ratio
-- and correction field in one invocation
public receiveFollowUpMessage: () ==> seq of real
receiveFollowUpMessage() ==
  return [correctionField,preciseOriginTS,cummulativeRateRatio];

-- Checks if there is a synchronization event
-- available
public receiveSyncEvent: () ==> bool
receiveSyncEvent() ==
  return syncEvent;

-- Gets the frequency associated to the master
public getFreq: () ==> real
getFreq() ==
  return channel.getMasterFreq();

public timeStep: () ==> ()
timeStep() ==
(
  if not syncEvent then
    checkSync()
  else
    if not followUpReady then
      checkFollowUp();
);
```

```

thread
  while true do
  (
    timeStep();
    Environment`timerRef.WaitRelative(1);
  );

end Slave
```

C.3.9 TimeStamp

```
class TimeStamp

values

public stepLength : nat = 1;

instance variables

currentTime  : nat    := 0;
wakeUpMap    : map nat to [nat] := {|->};
barrierCount : nat1;

operations

public TimeStamp : nat1 ==> TimeStamp
TimeStamp(count) ==
 barrierCount := count;

public WaitRelative : nat ==> ()
WaitRelative(val) ==
  WaitAbsolute(currentTime + val);

public WaitAbsolute : nat ==> ()
WaitAbsolute(val) == (
  AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);

BarrierReached : () ==> ()
BarrierReached() ==
(
 while  (card dom wakeUpMap = barrierCount) do
   (
    currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set dom wakeUpMap
             & wakeUpMap(th) <> nil and
             wakeUpMap(th) <= currentTime }
  in
   for all t in set threadSet
   do
    wakeUpMap := {t} <-: wakeUpMap;
 );
)
post forall x in set rng wakeUpMap & x = nil or
     x >= currentTime;

AddToWakeUpMap : nat * [nat] ==> ()
AddToWakeUpMap(tId, val) ==
   wakeUpMap := wakeUpMap ++ { tId |-> val };

public NotifyThread : nat ==> ()
NotifyThread(tId) ==
 wakeUpMap := {tId} <-: wakeUpMap;

public GetTime : () ==> nat
GetTime() ==
  return currentTime;

Awake: () ==> ()
Awake() == skip;

public ThreadDone : () ==> ()
ThreadDone() ==
 AddToWakeUpMap(threadid, nil);

sync
  per Awake => threadid not in set dom wakeUpMap;

  mutex(AddToWakeUpMap);
  mutex(NotifyThread);
  mutex(BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread);
  mutex(AddToWakeUpMap, BarrierReached);
  mutex(NotifyThread, BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

C.4. Real-time model

C.4.1 App

```
class App

instance variables

  private preciseOriginTS : real;
  private correctionField : real;

  private localTimeStamp : real;

  private timeInfoReady : bool := false;

  private timeInfoProvided : bool := false;


operations

public receiveTimeInformation : seq of real ==> ()
receiveTimeInformation (timeSeq) ==
(
  preciseOriginTS := timeSeq(1);
  correctionField := timeSeq(2);
  localTimeStamp := timeSeq(3);

  timeInfoReady := true;

  markInfoProvided();

);

public showReceivedTime : () ==> ()
showReceivedTime() ==
(
  if timeInfoReady then
  (
  IO`print("\nAt time: ");
  IO`print(localTimeStamp);
  IO`print("\nThe Grand Master time was: ");
  IO`print(preciseOriginTS);
  IO`print("\nIt has been corrected with: ");
  IO`print(correctionField);
  IO`print("\nFinal time for previos timestamp is: ");
  IO`print(preciseOriginTS+correctionField);

  markInfoProvided();
  )
  else
    IO`print("\n Message from application:
      Time information not Ready");
);

private markInfoProvided : () ==> ()
markInfoProvided() ==
  timeInfoProvided := true;

public infoProvided : () ==> bool
infoProvided() ==
  return timeInfoProvided;



sync
  mutex(markInfoProvided,infoProvided);
  mutex(receiveTimeInformation);
  mutex(showReceivedTime, infoProvided);
  mutex(showReceivedTime,receiveTimeInformation);

end App
```

C.4.2 Clock

```
class Clock

instance variables

  currentTime : nat := 0;

operations

public tick: () ==> ()
tick() ==
(
  duration (0) currentTime := currentTime +1;
);

public getTime: () ==> nat
getTime() ==
  return time;

thread
 -- while true do
 -- (
 --   tick();
 -- );

 -- periodic (2000E6,0,0,0) (tick)
 -- periodic (9E9,0,0,0) (tick) (Working)
   periodic(1E1,0,0,0) (tick)

 -- periodic (1E3,0,0,0) (tick)

sync
  mutex(tick,getTime);
  mutex(tick);

end Clock
```

C.4.3 ClockSlave

```
class ClockSlave

instance variables

  private siteSync : Device;

  private preciseOriginTS : real;
  private correctionField : real;
  private localTimeStamp : real;

  private timeConsumers : seq of App := [];

  private newTimeInfoReady : bool := false;

operations

public registerApp : App ==> ()
registerApp(ap) ==
  timeConsumers := timeConsumers ^ [ap];

public setTimeInfo: seq of real ==> ()
setTimeInfo(timeSeq) ==
(
  if len timeSeq > 2 then
  (
    preciseOriginTS := timeSeq(1);
    correctionField := timeSeq(2);
    localTimeStamp := timeSeq(3);
    newTimeInfoReady := true;
  )

);

public timeStep: () ==> ()
timeStep() ==
  if newTimeInfoReady then
  (
    notifyTime();
    newTimeInfoReady := false;
  )
  else
    skip;

public notifyTime : () ==> ()
notifyTime() ==
  let t : seq of real =
    [preciseOriginTS,
     correctionField,
     localTimeStamp] in
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).receiveTimeInformation(t);

public setSiteSync : Device ==> ()
setSiteSync(dv) ==
  siteSync := dv;

public getTime:() ==> ()
getTime() ==
();

public ClockSlave: App ==> ClockSlave
ClockSlave(ap) ==
(
  registerApp(ap);
);

public triggerPrint: () ==> ()
triggerPrint() ==
  for all i in set {1,...,len timeConsumers} do
    timeConsumers(i).showReceivedTime();

thread
  while true do
  (
    timeStep();
  );

```

```
sync
  mutex(timeStep,setTimeInfo);
  mutex(notifyTime,setTimeInfo);

end ClockSlave
```

C.4.4 Deployer

```
system Deployer

instance variables

-- Architecture definition

/*
Computing units:
 Priority: <FP> - Fixed priority
           <PP> - Priority?

Speed is giving in MIPS - Millions of instructions per second
*/

  cpu1 : CPU := new CPU(<FP>, 1E7);
  cpu2 : CPU := new CPU(<FP>, 1E9);

  cpu3 : CPU := new CPU(<FP>, 1E7);

  cpu4 : CPU := new CPU(<FP>, 1E9);

/*
Communication bus:
  Modes: <CSMACD> - ?
*/
  bus : BUS := new BUS(<CSMACD>, 72E9,{ cpu1,cpu2});
  bus2 : BUS := new BUS(<CSMACD>, 20E5, {cpu1,cpu3});


  -- Used only to evaluate the fourth configuration
  --(independent SiteSync block)
  --bus : BUS := new BUS(<CSMACD>, 72E9,{ cpu1,cpu4});

  siteSyncCom : BUS := new BUS(<CSMACD>, 72E9, {cpu4,cpu2});


  public static siteSync : Device := new Device();
  public static slave : Slave := new Slave();
  public static clkSlave : ClockSlave := new ClockSlave();
  public static app : App := new App();
  public static channel : PhyMessage := new PhyMessage();
  public static simClk : Clock := new Clock();
  public static timerRef : TimeStamp := new TimeStamp(1);

operations

public Deployer : () ==> Deployer
Deployer () ==
(-- TODO Deploy deployable object to cpu's

  slave.clk := simClk;

  -- Establish the relationships between the entities

  -- 1. Register the Application in the clock slave entity
  clkSlave.registerApp(app);
  clkSlave.setSiteSync(siteSync);

  -- 2. Associate the clock slave to the siteSync entity
  siteSync.setClockSlave(clkSlave);

  -- 3. Configure the physical layer and associate it to the
  --   slave port
  slave.changePhyChannel(channel);

  -- 4. Associate the slave port to the siteSync entity
  siteSync.setSlavePort(slave);

  channel.setMasterFreq(10);
  siteSync.setLocalFreq(20);
  siteSync.getFreq();
  siteSync.calculateNeighbourRR();

  siteSync.setFinalDevice();

  -- Configuration 1
  cpu1.deploy(slave);
  cpu1.deploy(clkSlave);
  cpu1.deploy(app);
  cpu1.deploy(siteSync);

  cpu2.deploy(channel);
  cpu2.deploy(simClk);


/*
 -- Architecture 2
```

```
  cpu1.deploy(app);
  cpu1.deploy(clkSlave);
  cpu1.deploy(siteSync);

  cpu2.deploy(slave);
  cpu2.deploy(channel);
  cpu2.deploy(simClk);
*/

-- Architecture 3
/*
  cpu1.deploy(clkSlave);
  cpu1.deploy(app);

  cpu2.deploy(siteSync);
  cpu2.deploy(slave);
  cpu2.deploy(channel);
  cpu2.deploy(simClk);
*/

-- Architecture 4

/*
  cpu1.deploy(clkSlave);
  cpu1.deploy(app);

  cpu4.deploy(siteSync);

  cpu2.deploy(slave);
  cpu2.deploy(channel);
  cpu2.deploy(simClk);
*/

-- CPU's are started implicit
);

end Deployer
```

C.4.5 Device

```
class Device

instance variables

  -- Device ID number
  private deviceID : nat := 0;
```

```

-- Set device as a bridge
public isBridge : bool := true;

-- Frequency of the current device.
private freq : nat := 0;

-- Slave port associated to the device
public slavePort : Slave;

-- Device with no external slaves connected
-- (finalDevice = true)
public finalDevice : bool := false;
--inv finalDevice => len masterPorts = 0;

-- Current device is Grand Master -> iGM = true
-- if device isGM it should not have masters attached.
public isGM : bool := false;
inv isGM => not finalDevice;
inv finalDevice => not isGM;

-- Frequency of the master device
private masterFreq : nat := 0;

private receivedSyncEvent : bool := false;
private syncRecTS : nat := 0;

-- Neighbour rate ratio measured by the slave port
private neighbourRateRatio : real := 1;

-- Precise origin timestamp, defined by GM and
-- never changed.
private preciseOriginTS : real := 0;

-- received correction field, to be recomputed
-- before use.
private receivedCorrectionField : real := 0;

-- CumulativeRateRatio received from the previous
-- node.
private cummulativeRateRatio : real := 1;
-- Delay from master to slave port
private linkDelay : real := 10;
-- Time that has been spent in the current node
private residenceTime : real := 0;

-- Correction field calculated at the current node,
-- to be used in the current node.
private newCorrectionField : real := 0;
```

```
  -- CummulativeRateRatio calculated at the current node,
  -- to be used in the current node.
  private newCummulativeRateRatio : real := 0;

  -- Clock attached to the device
  private clk : Clock;

  private timeInfoAvailable : bool;

  public cSlave : ClockSlave := new ClockSlave();

  private followUpMessageReceived : bool := false;

  private clockNotified : bool := false;


  private readSyncEvent : bool := false;
  private readFollowUpMessage : bool := false;

operations

public setClockSlave: ClockSlave  ==> ()
setClockSlave(c) ==
  cSlave := c;

-- Actions carried out at the device per time-step
public timeStep: () ==> ()
timeStep() ==
(
  if receivedSyncEvent = false then
  (   receiveSyncEvent(); IO`print("#syncEvent#");)
  else if receivedSyncEvent and not followUpMessageReceived then
  (  receiveFollowUpMessage(); IO`print("Follow up received");)
  else if receivedSyncEvent and followUpMessageReceived then
  (  notifyClock(); IO`print("CLocks notified"););
);


-- Actions carried out at the device per time-step
public timeStep2: () ==> ()
timeStep2() ==
(
  if receivedSyncEvent = false then
  (   receiveSyncEvent();)
  else if receivedSyncEvent and
    not followUpMessageReceived then
  (  receiveFollowUpMessage(); )
  else if receivedSyncEvent and
     followUpMessageReceived and
     not clockNotified then
   (  notifyClock(); );
);

public markReadSE : () ==> ()
markReadSE() ==
  readSyncEvent := true;

public markReadFUM : () ==> ()
markReadFUM() ==
  readFollowUpMessage := true;

public readyForNext : () ==> ()
readyForNext() ==
(
  readFollowUpMessage := false;
  readSyncEvent := false;
);

public notifyClock : () ==> ()
notifyClock() ==
(
 cSlave.setTimeInfo(
    [preciseOriginTS,
    receivedCorrectionField,
    syncRecTS]);
 clockNotified := true;
);

public makeTimeAvailable : () ==> ()
makeTimeAvailable() ==
  timeInfoAvailable := true;

public getTimeInfo : () ==> seq of real
getTimeInfo() ==
  return [preciseOriginTS,
          receivedCorrectionField,
          cummulativeRateRatio];

public timeInfoRead : () ==> ()
timeInfoRead() ==
  timeInfoAvailable := false;

-- Sets the slave port that corresponds to the current device
public setSlavePort: Slave ==> ()
setSlavePort(s) ==
  slavePort := s;

-- Sets device ID
```

```
public setDeviceID: nat ==> ()
setDeviceID(id) ==
  deviceID := id
pre deviceID = 0
post deviceID <> 0;

-- Gets device ID
public getDeviceID: () ==> nat
getDeviceID() ==
  return deviceID
pre deviceID <> 0;

-- Receives the POTS
public receivePOTS: () ==> ()
receivePOTS() ==
  preciseOriginTS := slavePort.receivePOTS();

public setLocalFreq : nat ==> ()
setLocalFreq(f) ==
  freq := f;

-- Sets the frequency of the master device.
public setMasterFreq : nat ==> ()
setMasterFreq (f) ==
  masterFreq := f
pre f <> 0;

-- Get the frequency of the master device attached to
-- the slave port.
public getFreq : () ==> ()
getFreq() ==
  if (not isGM) then setMasterFreq(slavePort.getFreq());


-- Receives the follow up message sent by the master.
-- Reception is performed in one operation.
public receiveFollowUpMessage: () ==> ()
receiveFollowUpMessage() ==
(  let s : seq of real = slavePort.receiveFollowUpMessage() in
  (
    atomic (
    receivedCorrectionField := s(1);
    preciseOriginTS := s(2);
    cummulativeRateRatio := s(3); );
  if s(1) <> 0 and s(2) <>0 and s(3) <> 0 then
  followUpMessageReceived := true;
  );
);

```

```
-- Receives the cummulative rate ratio at the slave
-- port. Cummulative rate ratio is sent by the master
-- device attached to the slave port.
public receiveCRR: () ==> ()
receiveCRR() ==
  cummulativeRateRatio:= slavePort.receiveCRR();

-- Calculates the neighbour rate ratio. This calculation
-- is performed at every slave device.
public calculateNeighbourRR : () ==> ()
calculateNeighbourRR () ==
  if (not isGM) then
   neighbourRateRatio := freq / masterFreq
  else
    neighbourRateRatio := 1
pre not isGM => (freq <> 0 and masterFreq <> 0)
post neighbourRateRatio <> 0;

-- Calculates the new cummulative rate ratio to be sent
-- to the attached slave unit.
public calculateCRateRatio : () ==> ()
calculateCRateRatio () ==
  newCummulativeRateRatio := cummulativeRateRatio *
                             neighbourRateRatio
pre neighbourRateRatio <> 0 and cummulativeRateRatio <> 0;

-- Calculates the new correction field to be sent to the
-- attached slave unit.
public calculateNewCorrectionField : () ==> ()
calculateNewCorrectionField () ==
  if isGM then ()
  else (
    newCorrectionField := linkDelay + residenceTime *
    newCummulativeRateRatio + receivedCorrectionField;);

-- Receives a correction field sent by master unit.
public receiveCorrectionField : () ==> ()
receiveCorrectionField() ==
  receivedCorrectionField := slavePort.receiveCorrectionField();

-- Sets the current device as a final device.
-- A final device will not have slave devices and
-- will not be acting as a master device.
public setFinalDevice: () ==> ()
setFinalDevice() ==
  finalDevice := true
pre not isGM
post finalDevice;

```

```
-- Receives a synchronization sync event and
-- time stamps it.
public receiveSyncEvent: () ==> ()
receiveSyncEvent () ==
(
  receivedSyncEvent := slavePort.receiveSyncEvent();
  receivedSyncEvent := true;
  updateTimeRef();
);

-- Updates the time reference when the time stamp operation
-- is invoked.
public updateTimeRef: () ==> ()
updateTimeRef() ==
(
  --IO`print("\n Sync event --> timestamp generated");
  syncRecTS := slavePort.getIngressTime();
);

-- Shows the device state
public showDevice : () ==> ()
showDevice () ==
(
 IO`print("\nInfo for device: ");
 IO`print(deviceID);
 IO`print("\nIs GM:");
 IO`print(isGM);
 IO`print("\nIs final device: ");
 IO`print(finalDevice);
 IO`print("\nIs sync event received: ");
 IO`print(receivedSyncEvent);
 IO`print("\nSync event received at: ");
 IO`print(syncRecTS);
 IO`print("\nFrequency: ");
 IO`print(freq);
 IO`print("\nMaster frequency: ");
 IO`print(masterFreq);
 IO`print("\nLinkDelay: ");
 IO`print(linkDelay);

 IO`print("\nNeighbour Rate Ratio: ");
 IO`print(neighbourRateRatio);
 IO`print("\nRecCummulative Rate Ratio: ");
 IO`print(cummulativeRateRatio);

 IO`print("\nReceived CorrectionField: ");
 IO`print(receivedCorrectionField);
 IO`print("\nPrecise Origin TimeStamp: ");
 IO`print(preciseOriginTS);
);

thread
  while true do
  (
    timeStep();
  );

sync
  mutex(updateTimeRef);
  mutex(timeStep);
  mutex(notifyClock,receiveFollowUpMessage);

  mutex(timeStep,receivePOTS);
  mutex(timeStep,receiveCRR);

--  mutex(timeStep, receiveSyncEvent);

end Device
```

C.4.6 Environment

```
class Environment

types

  public timeStimulus = seq of real;

instance variables

  simSteps : nat := 3;

  timeStimuli : seq of timeStimulus := [];

  --public static timerRef : TimeStamp := new TimeStamp(5);

operations

public loadStimuli: () ==> ()
loadStimuli() ==
(
 --loadStimulus([20,30,50]);
  let i : seq of timeStimulus = [[40,20,10],[30,80,90]] in
    for all e in set {1,...,len i} do
      loadStimulus(i(e));
);
```

```

public loadStimulus: timeStimulus ==> ()
loadStimulus(s) ==
  timeStimuli := timeStimuli ^ [s];

-- Model simulation entry point
public run: () ==> ()
run() ==
(
  IO`print("\nEndpoint device receiving time corrections");

  loadStimuli();
  IO`print(timeStimuli);

  -- Simulation starts

  start(Deployer`simClk);
  start(Deployer`slave);
  start(Deployer`clkSlave);
  start(Deployer`siteSync);



   duration(0) feedPhyMessage();


  while Deployer`app.infoProvided() = false do
  (
    wait();
  );

  -- Shows the simulation results
  Deployer`siteSync.showDevice();
  Deployer`app.showReceivedTime();

  IO`print("\nModel over\n");
  IO`print("####### Unused time stimuli: ");
  IO`print(timeStimuli);

);

public feedPhyMessage: () ==> ()
feedPhyMessage() ==
(
  Deployer`channel.signalSyncEvent();
  Deployer`channel.setFollowUp(hd timeStimuli);
  timeStimuli := tl timeStimuli;
);

```

```

wait: () ==> ()
wait() ==
  skip;

end Environment
```

C.4.7 PhyMessage

```
class PhyMessage

instance variables

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  -- Frequency sent by a master
  private masterFreq : real := 0;

  private fupSet : bool := false;

operations

-- Gets the contents of the follow up message.
public getFollowUp: () ==> seq of real
getFollowUp() ==
  if not fupSet then
     return []
  else
     return [correctionField,
          preciseOriginTS,
          cummulativeRateRatio];

-- Sets the contents of the follow up message.
public setFollowUp: seq of real ==> ()
setFollowUp(s) ==
(
  setCRR(s(1));
  setCorrectionField(s(2));
  setPOTS(s(3));
  fupSet := true;
);

-- Signals a Synchronization event
public signalSyncEvent: () ==> ()
signalSyncEvent() ==
(
  --IO`print("\nMaster port --> Sync event sent");
  syncEvent := true;
  IO`print("Message in hardware buffer at: "); IO`print(time);
);

-- Check if there is a Synchronization event
public getSyncEvent: () ==> bool
getSyncEvent() ==
(
  --IO`print("\nSlave port --> Sync event received");
  return syncEvent;
);

-- ###################################################
-- Getters and setters for the different fields in the
-- Physical Message.

public setMasterFreq : real ==> ()
setMasterFreq(f) ==
  masterFreq := f;

public getMasterFreq: () ==> real
getMasterFreq() ==
  return masterFreq;

public setCRR: real ==> ()
setCRR(crr) ==
  cummulativeRateRatio := crr;

public setCorrectionField: real ==> ()
setCorrectionField(crf) ==
  correctionField := crf;

public setPOTS: real ==> ()
setPOTS(pots) ==
  preciseOriginTS := pots;

public getCField: () ==> real
getCField() ==
  return correctionField;

public getPOTS: () ==> real
```

```
getPOTS() ==
 return preciseOriginTS;

public getCRR: () ==> real
getCRR() ==
  return cummulativeRateRatio;

sync

  mutex(getFollowUp,setFollowUp);
  mutex(signalSyncEvent,getSyncEvent);

end PhyMessage
```

C.4.8 Port

```
class Port

instance variables

  -- Physical channel associated to the port
  protected channel : PhyMessage;

  -- Associated ID to the port
  protected portID : nat;

  protected clk : Clock;

operations

public timeStamp : () ==> ()
timeStamp() ==
  is not yet specified;

public timeStep : () ==> ()
timeStep() ==
  is not yet specified;

-- Change the associated channel to the slave
public changePhyChannel : PhyMessage ==> ()
changePhyChannel(ch) ==
  channel := ch;

public getPhyChannel : () ==> PhyMessage
getPhyChannel() ==
  return channel;
```

```

-- Changes the associated ID to the port
public changePortID: nat ==> ()
changePortID(id) ==
  portID := id;

end Port
```

C.4.9 Slave

```
class Slave is subclass of Port

instance variables

  ingressTime : real := 0;

  -- sync event flag. Holds true value if a sync event
  -- is put on the bus.
  private syncEvent : bool := false;

  -- Contents of the follow up message
  private preciseOriginTS : real := 0.0;
  private cummulativeRateRatio : real := 0.0;
  private correctionField : real := 0.0;

  private followUpReady : bool := false;

  public clk : Clock;

  partialTime : real := 0;


operations

public checkSync : () ==> ()
checkSync() ==
(
  duration(0) partialTime := time;
  duration(0) syncEvent := channel.getSyncEvent();
  if syncEvent = true then
   duration(0) timeStamp(); -- Time stamp
);

public checkFollowUp: () ==> ()
checkFollowUp() ==
(
  let i : seq of real = channel.getFollowUp() in
  (
    if len i = 0 then
      followUpReady := false
    else
    (
      atomic (
      correctionField := i(1);
      preciseOriginTS := i(2);
      cummulativeRateRatio := i(3););
    )
  );
);

public timeStamp : () ==> ()
timeStamp() ==
(  duration(0) ingressTime := clk.getTime();
   duration(0) IO`print("Received at time: ");
   IO`print(partialTime);
);

public getIngressTime : () ==> real
getIngressTime () ==
  return ingressTime;

-- Class constructor
public Slave: PhyMessage ==> Slave
Slave(phy) ==
  channel := phy;

-- Receives the cummulative rate ratio
public receiveCRR: () ==> real
receiveCRR() ==
  channel.getCRR();

-- Receives the correction field
public receiveCorrectionField: () ==> real
receiveCorrectionField() ==
  channel.getCField();

-- Receives the precise origin time stamp
public receivePOTS : () ==> real
receivePOTS() ==
  channel.getPOTS();

-- Receives the follow up message
-- Receives Precise Origing Time Stamp, cummulative rate ratio
-- and correction field in one invocation
public receiveFollowUpMessage: () ==> seq of real
```

```
receiveFollowUpMessage() ==
  return [correctionField,preciseOriginTS,cummulativeRateRatio];

-- Checks if there is a synchronization event
-- available
public receiveSyncEvent: () ==> bool
receiveSyncEvent() ==
  return syncEvent;

-- Gets the frequency associated to the master
public getFreq: () ==> real
getFreq() ==
  return channel.getMasterFreq();

public timeStep: () ==> ()
timeStep() ==
(
  if not syncEvent then
    checkSync()
  else
    if not followUpReady then
      checkFollowUp();
);

thread

  while true do
  (
    duration(0) timeStep();
  -- -- duration(1000) ()
  );

  --periodic(2E5,0,0,0) (timeStep);

end Slave
```

C.4.10 TimeStamp

```

class TimeStamp

values

public stepLength : nat = 1;

instance variables
```

```

currentTime   : nat   := 0;
wakeUpMap     : map nat to [nat] := {|->};
barrierCount : nat1;

operations

public TimeStamp : nat1 ==> TimeStamp
TimeStamp(count) ==
 barrierCount := count;

public WaitRelative : nat ==> ()
WaitRelative(val) ==
  WaitAbsolute(currentTime + val);

public WaitAbsolute : nat ==> ()
WaitAbsolute(val) == (
  AddToWakeUpMap(threadid, val);
  -- Last to enter the barrier notifies the rest.
  BarrierReached();
  -- Wait till time is up
  Awake();
);

BarrierReached : () ==> ()
BarrierReached() ==
(
 while  (card dom wakeUpMap = barrierCount) do
   (
    currentTime := currentTime + stepLength;
    let threadSet : set of nat = {th | th in set domwakeUpMap
             & wakeUpMap(th) <> nil and
             wakeUpMap(th) <= currentTime }
  in
   for all t in set threadSet
   do
    wakeUpMap := {t} <-: wakeUpMap;
 );
)
post forall x in set rng wakeUpMap & x = nil or x >= currentTime;

AddToWakeUpMap : nat * [nat] ==> ()
AddToWakeUpMap(tId, val) ==
   wakeUpMap := wakeUpMap ++ { tId |-> val };

public NotifyThread : nat ==> ()
NotifyThread(tId) ==
 wakeUpMap := {tId} <-: wakeUpMap;

```

```
public GetTime : () ==> nat
GetTime() ==
  return currentTime;

Awake: () ==> ()
Awake() == skip;

public ThreadDone : () ==> ()
ThreadDone() ==
 AddToWakeUpMap(threadid, nil);

sync
  per Awake => threadid not in set dom wakeUpMap;

  mutex(AddToWakeUpMap);
  mutex(NotifyThread);
  mutex(BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread);
  mutex(AddToWakeUpMap, BarrierReached);
  mutex(NotifyThread, BarrierReached);

  mutex(AddToWakeUpMap, NotifyThread, BarrierReached);

end TimeStamp
```

Appendix D

Representations and models used at different levels of abstraction

An adequate level of description should be provided at each abstraction level in order to be able to describe and analyse the relevant aspects of it. The aspects of interest at different abstraction levels vary considerably, therefore several modelling languages and methods, are required to create expressive and adequate representations at each level. Several authors [Wolf03, Waddington&06, Edwards&97] remark the fact that heterogeneous modelling is the most promising was of working with Hardware/Software systems. By using different modelling techniques in a systematic manner, clearly defining and understanding borderlines and interfaces between the representations, the best out of each modelling technology can be obtained. Besides defining the links between the models, it is responsibility of the modeller to apply the correct language at each level of abstraction, considering the kind and amount of details the model has to incorporate. For example, incorporating specific hardware details at an algorithmic level would not be relevant to discuss the system behaviour from a software perspective, further more, such a level of detail would be an inconvenient while understanding the software components of the system. On the other hand, specific hardware details are essential in a CAM model.

At an algorithmic level, languages like Unified Modelling Language (UML), Matlab or Specification Description Languages (SDL) are efficient in terms of expressiveness and precision. SystemC is performing better than the previous languages when it comes to model hardware implementations, furthermore, it is flexible and can be used across the interface level, the transaction level and the cycle accurate level. However, since it is an extension of the programming language C++, models tend to be mixed with implementation details.

In general, the lower the level of abstraction, the more details will be present. This implies more precision and complexity in the system representation. Taking as an example a hardware block, at the algorithmic level it might be represented by a black box that provides an interface to the application logic. This abstraction is isolating memory addresses from business logic through functions to read/write data to the interface. At an interface level it is relevant to know where exactly the data is placed and which procedure has to be followed in order to read/write data to the memory block. Details that were not relevant at an algorithmic level now are becoming important e.g.: concrete memory addresses, reserved areas... Figure D.1 shows part of this information and an adequate representation of it. Note that at this level time details are not relevant, details that will be incorporated in a TLM model when the current representation is refined.

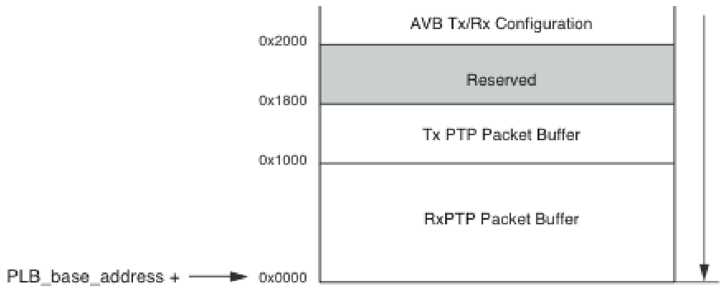


Figure D.1: Part of the memory map of an Audio Video Bridging (AVB) core interface. This memory map serves as an interface between the software and the hardware layers.

The description provided at the interface level is sufficient and precise enough for a programmer that is using the hardware, and is adequate for a hardware engineer to understand the interfaces that should be provided by the hardware layer. However a hardware engineer needs to represent more details while modelling a hardware block. In this case, what happens between clock cycles becomes relevant, and the information representation goes down to the bit level. At this level, specific Hardware Description Languages (HDL) like VHDL or Verilog are used. As an example of the level of detail that can be modelled with this languages, a VHDL representation of a tri-state buffer[1] is presented in D. This representation is useful for a hardware engineer aiming to implement the actual RTL layer. The architecture and behaviour are described at the bit level, e.g: en line 3 three bit variables are declared. The values these bits are holding during tristate operation are further specified in lines 10, 13 and 14.

```
1  entity test_tristate is
2    generic ( width : integer := 17 );
3    port ( en1, en2, en3    : in  std_ulogic;
4      inp1, inp2, inp3 : in  std_logic_vector (width downto 0);
5      tribus           : out std_logic_vector (width downto 0));
6  end test_tristate;
7
8  architecture rtl of test_tristate is
9  begin
10   tribus <= inp1 when (en1 = '1') else
11     (others => 'Z');
12   tribus <= inp2 when (en2 = '1') else
13     (others => 'Z');
14   tribus <= inp3 when (en3 = '1') else
15     (others => 'Z');
16   tribus <= ( others => 'H' );
17 end rtl;
```

It is possible to go even further and analyse the actual silicon implementation in the FPGA. In figure figure D.2, the used silicon area is shown. This representation of the system is critical for fine-tuning, floor planning and optimization. As it has been shown, multiple system views are offered by multiple modelling and representation technologies. Different representation techniques

[1]Logic device able to output a high or low logical level or present a high impedance (disconnected) state

(in the end, modelling techniques) at the implementation level are widely accepted in industry.
All of the above presented models, are providing relevant details at different levels of abstraction. Note that so far the models have been performed considering a single system. Besides this scenario, many problems are solved by systems that are communicating with other active systems, e.g. a network bridge, which implements certain business logic and communicates with other devices. This introduces an additional abstraction level, which is the Systems of Systems view. Certain communication and interaction requirements from the System of System point of view might be worth to consider while performing the Hardware/Software partitioning. Different partitioning schemes might have an impact from the distributed-real time perspective that should be considered in the development process.
Additional details on the properties the modelling languages should present will be provided in section 2.4. Further discussion on modelling technologies in the Hardware/Software co-design area will be exposed in chapter 4.

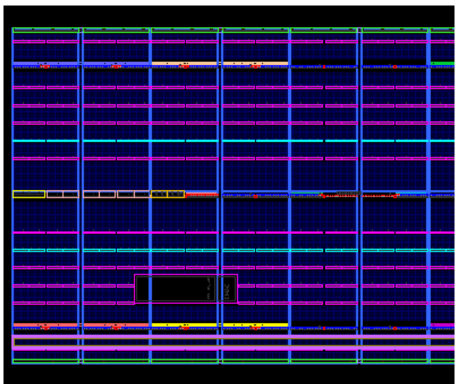


Figure D.2: Used silicon area in a programmed FPGA. Layout information can be obtained through specific tools provided by the manufacturer. Source: `xilinx.com`

Appendix E

MATLAB Based Methodology for Hardware/Software Validation

The official MATLAB based methodology is considering primarily three scenarios. In the first one,shown in figure E.1 MATLAB is used as a test bench providing on-the-fly stimulus to an HDL block, simulated in a specialized EDA tool and to Algorithms running in MATLAB. Later on, results can be analysed in the MATLAB tool.

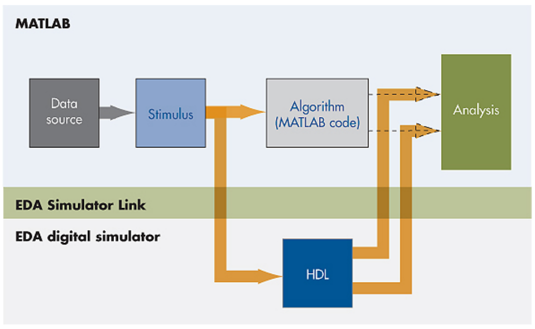


Figure E.1: MATLAB Simulink used as a testing infrastructure. Source: [Simulink]

In the second scenario, figure E.2, a MATLAB block is substituting an Hardware Definition Language block. The advantage of such an approach is that different functionality can be evaluated rapidly by implementing it in a high level programming language, without having to implement it in HDL. It must be remarked that this second simulation is taking place in the EDA simulator tool, and using MATLAB only as a functionality provider.
The last scenario, shown in figure E.3, is a combination of the previous two. Simulink is use to provide stimulus and to implement some of the HDL blocks. Part of the functionality is run in the EDA simulator and the rest in Simulink blocks. Finally, simulation results are analysed in Simulink.

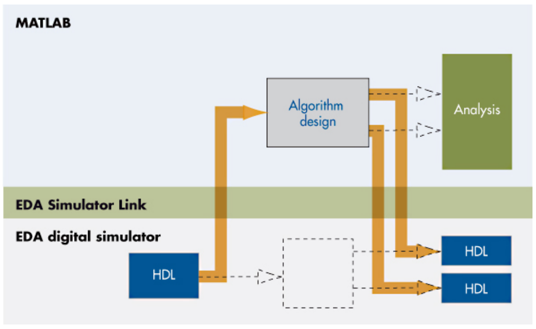


Figure E.2: MATLAB Simulink combining an algorithm block and an HDL implementation. Source: [Simulink]

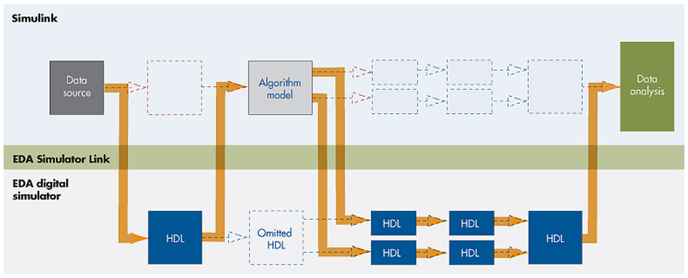


Figure E.3: Combined test approach for HDL and Simulink simulations. Source: [Simulink]