EINDHOVEN UNIVERSITY OF TECHNOLOGY

Predictable design for real-time systems

Citation for published version (APA):
Florescu, O. (2007). *Predictable design for real-time systems*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR630368

DOI:
10.6100/IR630368

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Predictable Design for Real-Time Systems

Oana Florescu

Predictable Design for Real-Time Systems

Predictable Design for Real-Time Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr.ir. C.J. van Duijn,
voor een commissie aangewezen door het College
voor Promoties in het openbaar te verdedigen op
dinsdag 4 december 2007 om 14.00 uur

door

Oana Florescu

geboren te Constanta, Roemenië

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. H. Corporaal

Copromotor:

dr.ir. J.P.M. Voeten

© Copyright 2007 by O. Florescu. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission from the copyright owner.

Printed by: Universiteitsdrukkerij Technische Universiteit Eindhoven

Cover design: Emil Onea, Romania

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Florescu, Oana

Predictable design for real-time systems / by Oana Florescu. - Eindhoven : Technische Universiteit Eindhoven, 2007.

Proefschrift. - ISBN 978-90-386-1654-4

NUR 992

Trefw.: real-time computers / systeemanalyse / software-ontwikkeling ; prototypes.

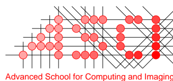
Subject headings: real-time systems / hardware-software codesign / software prototyping.

Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.

Sir Winston Churchill (1874 - 1965)

To Vali,
with all my love

This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute. This project has been partially supported by the Dutch Ministry of Economic Affairs under the Senter TS program.



This work has been carried out in the ASCI graduate school. ASCI dissertation series number 153.

The longest part of the journey is said to be the passing of the gate.
Marcus Terentius Varro (116 BC - 27 BC)

Abstract

Predictable Design for Real-Time Systems

The complexity of real-time embedded systems has motivated the research on frameworks and techniques to structure and automate their design process. Having such frameworks is especially important for embedded and safety-critical systems which are hard to design correctly within tight time-to-market constraints. Design methodologies reduce the risk of expensive design iterations by supporting designers in constructing models. Software/Hardware Engineering (SHE) is a general-purpose system-level design methodology that enables analysis of both functional correctness and performance properties. By using SHE, the designer is assisted in constructing models and applying analysis techniques with various guidelines and modelling patterns. A key feature of SHE is its foundation on formal methods, which ensures that the obtained analysis results are unambiguous. SHE also includes guidelines and techniques for automatic synthesis of real-time control software, which is again based on formal methods to ensure that properties in a model (including timing related properties) are preserved in the software realisation. This thesis contributes to both the modelling and analysis phase of the design, as well as to the correct synthesis towards an efficient software implementation of a system.

To ensure the correctness and the performance properties of real-time embedded systems, early evaluation of their properties is needed. To achieve this, we have developed a set of language-independent modelling patterns to enable easy creation of models for design space exploration. The modelling patterns cover typical real-time system components as they are considered in classical scheduling theory, such as periodic and aperiodic tasks, computation and communication resources, input event generators, and output event collectors. Furthermore, we have designed a Pattern-based system Description Language (PDL) which can be used to describe a real-time system in terms of the patterns needed and the values of their parameters. This language is easy to use and it does not require knowledge of any of the modelling languages typically used for analysis of real-time systems. Its main advantage is that it enables automatic generation of models in different modelling languages. As an example, we present an implementation of the modelling patterns in the general purpose formal modelling language called Parallel Object Specification Language (POOSL), which is underlying the SHE methodology, together with a tool that translates a PDL description into a POOSL model. Due to the expressiveness of the POOSL language, the models generated are appropriate for both hard and soft real-time systems as they enable analysis of both worst-case and average case behaviour of a system, ensuring a correct dimensioning of the final product.

To implement a concurrent real-time system from a model in a "correct" way, it is important to understand the relation between the properties of the model and of its corresponding implementation. In this thesis, we present a mathematically proved correct mechanism of determining the time deviation between a model and its corresponding realisation based on which we can predict the properties of the realisation of the system. Moreover, we have defined a notion of distance as a metric to express the observable property preservation between a model and a realisation of the system. We propose an approach to calculate an upper-bound on the size of this distance in order to predict the observable properties of the realisation based on those of the model. Furthermore, we show that this upper-bound can be decreased by imposing priority on the execution of the observable actions over the execution of the unobservable ones. Based on this result, we have extended an existing model synthesis approach to generate from a model an implementation with stronger observable property preservation.

By means of a realistic case study, we show how the contributions brought by this thesis can be applied for the model-driven design of a real-time system that ensures the control of a printer paper path. With this case study, we show how the system can be modelled using the modelling patterns, how its real-time properties can be derived and analysed and how predictions about the properties of the realisation on a target platform can be made. The synthesis of the system yields a realisation that preserves these properties and ensures a correctly running system.

Contents

Abstract **iii**

1 Introduction **1**

1.1 Embedded Real-Time Systems Design 2

1.2 Software/Hardware Engineering 4

1.2.1 Modelling and Analysis Phase 5

1.2.2 Synthesis Phase 7

1.2.3 Related Research 8

1.3 Problem Statement and Research Contributions 10

1.4 Thesis outline 11

2 Educational Example **15**

2.1 Motion Control System Description 15

2.2 High-Level System Model 16

2.3 Adding Details to the System Model 19

2.4 A More Detailed Model of the System 21

2.5 Synthesis Model of the System 24

2.6 Summary 27

3 Pattern-Based Modelling of Real-Time Systems **29**

3.1 Modelling Approaches for Real-Time Systems 30

3.2 Related Research 32

3.3 The Library of Modelling Patterns 34

3.4 UML Profile for POOSL Modelling Language 37

3.5 POOSL Modelling Patterns Library 38

3.5.1 Application Model 38

3.5.2 Platform Model 43

3.5.3 Environment Model 48

3.5.4 Mapping Model 51

3.6 Model Generation Based on Patterns 51

3.7 Summary 55

4 Analysis Approach for Dimensioning of Real-Time Systems 57

4.1 Model Analysis 58

4.2 Case Study: A Distributed In-Car Radio Navigation System 60

4.2.1 The Model of the In-Car Radio Navigation System 61

4.2.2 Analysis of the System Behaviour 63

4.3 Case Study: The Low-Level Control of a Printer Paper Path 68

4.3.1 The Model of the Paper Path Low-Level Control 68

4.3.2 Platform Dimensioning of the Paper Path Low-Level Control . 70

4.4 Summary 71

5 Proximity Between Model and Realisation 73

5.1 Preliminaries 73

5.2 Representation of System Behaviour 74

5.3 POOSL Model Synthesis Strategy 78

5.4 Determining the Proximity Between Model and Realisation 79

5.4.1 Definition of Distance Between Paths 79

5.4.2 Calculating the Distance Between Model and Realisation 84

5.5 Execution Time Accuracy Impact on Distance 87

5.6 Finite Time Computation of Distance 88

5.6.1 Preliminaries 88

5.6.2 Finite Extended Timed Labelled Transition System 91

5.7 Algorithm for Computing the Distance Between Model and Realisation 93

5.8 Simulation-Based Estimation of the Distance Between Model and Realisation 95

5.9 Summary 97

6 Predictable Real-Time Systems Synthesis 99

6.1 Real-Time Properties 99

6.1.1 Timed State Sequences 100

6.1.2 Interpretation of MTL Logic 102

6.1.3 Preservation of Properties 103

6.2 Distance Reduction from Model to Realisation 105

6.2.1 New Proximity Metric Between Model and Realisation 106

6.2.2 Reduction of a Timed Labelled Transition System 109

6.2.3 Changing Action Ordering 112

6.3 Improved POOSL Model Synthesis Strategy 113

6.4 Experimental Results 113
6.5 Related Research 117
6.6 Summary 118

7 Case Study **121**
7.1 Printer Paper Path System Description 121
7.2 Analysis Model of the Paper Path 122
7.3 Predicting Properties of the Paper Path 126
7.4 Synthesis of the Paper Path Model 129
7.5 Summary 130

8 Conclusions and Outlook **131**
8.1 Summary of Research Contributions 131
8.2 Future Research 132

A Sequences of Elements **135**

Bibliography **143**

Samenvatting **151**

Acknowledgements **153**

About the Author **155**

1 Introduction

An *embedded system* is a special-purpose information processing system completely encapsulated by the device it controls, which is also called an embedding system [25]. Usually, embedded systems have a fixed functionality and operate autonomously. Physically, this type of systems ranges from portable devices, such as MP3 players, to large stationary installations, like traffic lights or factory controllers.

Typically, embedded systems are reactive systems that are in continuous interaction with their physical environment to which they are connected through sensors and actuators. Due to the intrinsic *concurrent* nature of the different mechanical, electrical and optical devices in the environment, an embedded system is made of a large number of parallel processes to control them. Besides the concurrency, another consequence of their reactive nature is the *timeliness*. Embedded systems must run at a pace determined by their environment. The result of this is that many embedded systems must meet *real-time constraints*, i.e. they must react to stimuli within a certain time interval dictated by the environment. In safety-critical applications, such as a power plant or an aircraft control, it has to be ensured that no timing requirement is broken as this might have catastrophic consequences. These systems are called *hard real-time systems* [20]. However, there are systems, like video and audio encoders, that are considered to run correctly even though some timing requirements are occasionally broken. Although meeting the timing requirements is desirable for performance reasons, breaking them does not have catastrophic consequences for such systems. Therefore, for a cost-effective realisation of the product as perceived by the consumer [86][1], a certain percentage of allowed missed timing requirements is typically established. Such systems are called *soft real-time systems* [20].

As not only the correctness of the computations but also the timeliness of the

[1] For instance, a consumer might choose to purchase a $50 video player that happens to drop single frames under rare circumstances than a $200 system verified and certified never to drop frames.

computed results of the whole embedded system are of major concern, the design of embedded real-time systems is inherently difficult. Additionally, the increasing demand for more functionality and tighter requirements on performance, time-to-market, cost and energy consumption of the final product constrain and complicate the design even more. Therefore, the main question regarding the design of embedded real-time systems is:

How to adequately predict and guarantee the properties of the final product according to the requirements and under tight time-to-market, cost and energy consumption constraints?

The contributions brought by this thesis address several aspects of this problem as it will be shown later in this chapter. This introductory chapter presents the characteristics of embedded real-time systems design and the open issues from previous research work. We first discuss basic notions of embedded real-time systems design in section 1.1, followed by the key elements of the design methodology used throughout this work in section 1.2. The problem statement and the research contributions brought by this thesis are presented in section 1.3, whereas the thesis outline is given in section 1.4.

1.1 Embedded Real-Time Systems Design

The increasing complexity of embedded real-time systems, the growing implementation costs and the tight time-to-market constraints have motivated research on methodologies to structure and automate their design process. Such design methodologies intend to reduce and, eventually, to even eliminate, the risk of expensive design iterations by supporting designers in constructing models. A *model* is an abstract representation of a system that enables the analysis of a proposed design solution before its actual realisation into hardware and software. Often, a number of design alternatives are proposed for realising the desired functionality of the system ranging from different partitioning of the components to using different algorithms. A design is said to be *correct* when all its functional and timing-related requirements are satisfied. Other requirements, like cost, energy consumption or performance properties like latency or throughput, are considered non-functional requirements. A correct design that satisfies also the non-functional requirements is called a *feasible design* [90]. The *design process* structures the search for a feasible design.

A common approach for managing the complexity of designing embedded real-time systems is to distinguish a number of design phases. Consecutive design phases focus on finding answers to questions of how to realise the functionality such that the desired non-functional requirements are also satisfied. This implies the need for exploring different design alternatives and decide which one is more likely to lead to a final product that satisfies all the requirements [44]. *Analysis* of models helps in understanding the consequences of different decisions before getting to the expense and trouble of actually building the system according to them. By choosing a certain alternative, the model of the system is *refined* by fixing a number of details about the final product that will not be changed later in the design process. While increasing the number of details, the level of abstraction decreases and the designer gets closer to a realisation of the system. When *enough* details are fixed, the design methodology

should assist the designer in *synthesising* the hardware and/or software components of the final product.

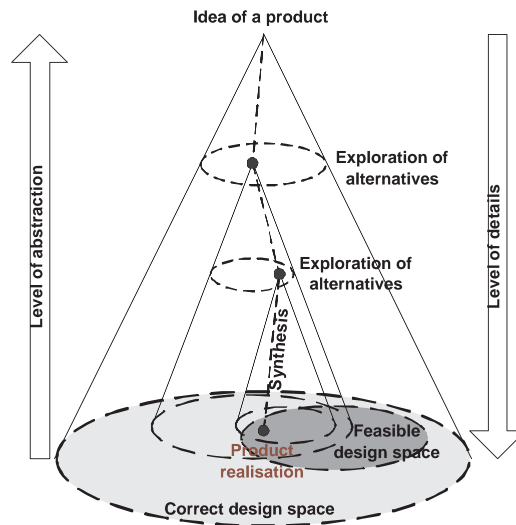


Figure 1.1: Trajectory towards a product realisation

Figure 1.1 shows the trajectory from an idea of a product towards a realisation as an abstraction pyramid like the one presented in [64]. The pyramid illustrates the process of exploring alternatives, making decisions and adding details in order to obtain a feasible design. One important aspect that the picture emphasises is that decisions made in early phases of the design process, namely high in the pyramid, can have a large impact on the final realisation of the product by pruning away a large part of the design space. For example, in [82] it is shown that wrongly choosing between a time-driven and an event-driven implementation of the control algorithms in a high-tech system, such as an airplane, an industrial plant or a copier, may strongly influence the performance and/or the cost of the final product in a negative way. Figure 1.1 visualises how the first exploration of alternatives chooses a solution that reduces substantially the accessible part of the feasible design space. By fixing more and more details of the design, the number of reachable feasible solutions is reduced to, eventually, one, which will be the synthesised product realisation.

The aim of a real-time system design is to fill the gap between requirements and realisation. Due to the increasing complexity of systems, this gap has widened. Since a code-centric, trial-and-error approach is too much time-consuming and not efficient for complex systems, designers resort to a multi-phase design process. Consecutive design phases are concerned with finding solutions for how to realise (parts of) the functionality of the system such that the requirements can be satisfied. *Predictability* of the design process refers to the capability of making design decisions at a certain design phase that will hold in later phases as well. Since the early phases involve little information about the system, it is a challenge to ensure that early design decisions are valid for the realisation of the system. Between consecutive design phases, the design process needs to support *correctness preservation*, ensuring that the properties at a certain stage are preserved in subsequent stages.

Figure 1.2: The internals of a complex real-time high-tech system [73]

The design process of real-time systems is made more difficult by two characteristics that are crucial to these systems: concurrency and timeliness. Figure 1.2 shows the internals of a large printer/copier machine from Océ Technologies BV, Netherlands, which is a complex high-tech real-time system consisting of a large number of mechanical, electrical and optical components that need to be controlled by software under different timing requirements. Due to the reactive nature of such systems and to the intrinsic concurrent nature of the controlled devices in the physical environment, real-time systems are typically composed of a set of parallel processes. Although it is possible to design a sequentialised real-time process to control all the parallel physical components in the environment, such a design would be difficult to understand, to maintain, to improve or even to extend to a whole family of products. This is especially true for systems with hundreds of concurrent processes each with its own functionality, timing constraints and intricate dependencies on other processes. Moreover, timeliness is a characteristic that contributes, besides the functionality, to the correctness of the system. The dependencies between processes in the system make very difficult to establish real-time requirements per process. Ensuring that decisions made along the design process preserve and guarantee the timing properties of a real-time system is of uttermost importance.

1.2 Software/Hardware Engineering

As mentioned earlier, the purpose of design methodologies is to structure the design process from early phases, by enabling the construction of models suitable for analysis and exploration of the design space, and towards the system realisation, by as-

sisting the synthesis of hardware and software. The Software/Hardware Engineering (SHE) is a system-level design methodology introduced in [97] and developed at the Eindhoven University of Technology. The SHE methodology distinguishes a modelling and analysis phase from a synthesis phase of the final product. The modelling and analysis phase covers the construction and the refinements of models for the purpose of analysis and exploration of the design space. The synthesis phase is concerned with the direct transformation of a model into hardware and/or software components.

1.2.1 Modelling and Analysis Phase

Figure 1.3 depicts the framework for exploring design alternatives with SHE as shown in [91], which can be applied at different levels of abstraction. The modelling and analysis phase of the design process consists of three stages: formulation, formalisation and evaluation.

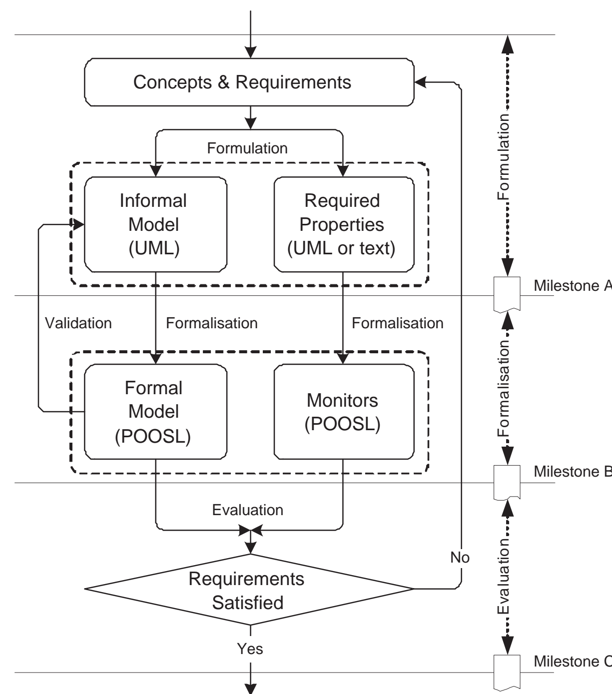


Figure 1.3: The modelling and analysis phase of the design process

The design of a system starts with brainstorm-sessions on concepts for realising the requested functionality. Additionally, a set of requirements that are to be satisfied by the final product is identified. The *formulation* stage is concerned with structuring this creative process and with documenting the results. SHE uses object-oriented analysis and design techniques in combination with several guidelines for applying them in the context of embedded real-time systems [97]. The concepts for realising the required functionality are formulated in an informal model expressed in the Uni-

fied Modelling Language (UML) [81]. Moreover, the requirements to be satisfied are formulated as required properties that represent the design issues to be addressed. The result of the formulation stage is a structured (but informal and non-executable) specification of the design concepts and requirements using schematic diagrams, like UML diagrams, possibly annotated using for example the profile for Schedulability, Performance and Time [75] or even plain text.

To enable rigorous analysis of real-time systems models, the *formalisation* stage of the SHE methodology is concerned with developing formal representations of both the informal model and the required properties delivered at milestone A. Based on a UML profile for SHE [90], the informal UML model is formalised into an executable model expressed in the formally defined Parallel Object-Oriented Specification Language (POOSL) [97]. POOSL consists of a small set of very expressive primitives able to capture concurrency, time, communication, probabilistic distribution and hierarchical structure of a system. To formally express the required properties of a system, real-time temporal logics, such as MTL [59], are used to specify real-time correctness properties. An example of a correctness property is: if two successive sensors in the paper path of a copier machine are triggered within 2 ms, the motor that drives the paper needs to be slowed down with 20%. Moreover, to specify time related performance metrics, such as long-run time-averages, like the expected occupancy of a buffer in a system, and long-run time variances, like the variance of the occupancy of a buffer, temporal reward functions [100] are used. Based on this formalisation of the required performance properties, monitors can be constructed, also in POOSL, to extend the formal model in such a way that they do not affect the properties of the system [90]. The term validation in figure 1.3 refers to the process of checking that the formal model and the monitors properly reflect the informal model and its required properties. The validated and documented formal model and monitors represent the deliverables at milestone B.

In the *evaluation* stage, the properties expressed by the monitors are actually checked against the formal model. Verification of the correctness properties is based on the model checking techniques presented in [41], whereas the evaluation of performance metrics relies on Markov chain based analysis techniques as documented in [90]. Both verification and performance analysis are made possible, as indicated in figure 1.4, by the formal semantics of POOSL, which is based on a mathematical structure called timed probabilistic labelled transition system. The formal semantics also enables unambiguous platform-independent simulation of a POOSL model guided by mathematical rules. Hence, unambiguous results are ensured in case of both exhaustive and simulation-based analysis. Moreover, the mathematical definition of the language facilitates the automation of the analysis techniques in tools. Based on the analysis results, the designer can conclude whether the requirements are satisfied. If so, the deliverable at milestone C documents the analysis results as a starting point for more detailed design which will iterate through the steps in figure 1.3 again. Otherwise, the deliverable C contains the reasons for not satisfying the requirements and the previous design concepts and requirements need to be revised.

The flow of steps for the modelling and analysis phase is rather general and can be further detailed depending on the application domain by means of additional guidelines. An example of such a guideline is the Y-chart approach from [57] that separates the model of the application from the model of the hardware platform on

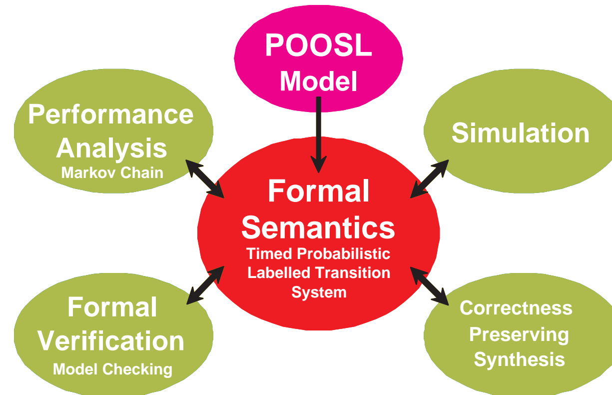


Figure 1.4: The influences of the formal semantics of POOSL

which it is executed. In [99] it is shown how this approach can be integrated in the design flow of SHE. Another example is the set of guidelines given in [90] for adequately extending POOSL models with monitors in order to evaluate various properties of a system. These guidelines include extending existing object classes with methods or attributes and creating new object classes to collect and process information. However, these guidelines assume rather good knowledge and understanding of the POOSL modelling language. Additionally, the formalisation stage of SHE is currently accomplished by hand. Hence, this process is tedious and error-prone. Therefore, more general guidelines are needed in order to alleviate and ultimately to automate the formalisation stage of SHE to enable easy and correct construction of models for real-time systems.

1.2.2 Synthesis Phase

After a number of iterations through the three steps of the modelling and analysis phase, a POOSL model can be refined to a level such that it describes the real-time system in full functional detail. At this point, the SHE methodology assists the synthesis of the final implementation according to the flow in figure 1.5. SHE uses formal correct-by-construction approaches, enabled by the formal semantics of the POOSL modelling language, to support the *synthesis* phase of the design process. This phase improves the reliability of the implementation and also reduces the design time by minimising verification and test efforts.

Currently, SHE supports only synthesis of real-time control software on single processor platforms using the techniques developed in [51] which are enabled by the formal semantics of POOSL as shown in figure 1.4. The basic idea is to execute a POOSL model in its real-time environment while interacting with this environment. POOSL models are typically closed in the sense that they often include a representation of the environment of the system. In order to synthesise the control software, this environment model together with the monitors, which may have been added previously for the purpose of analysis, must be discarded [51] as seen in figure 1.5. The interactions between the software model and the environment are

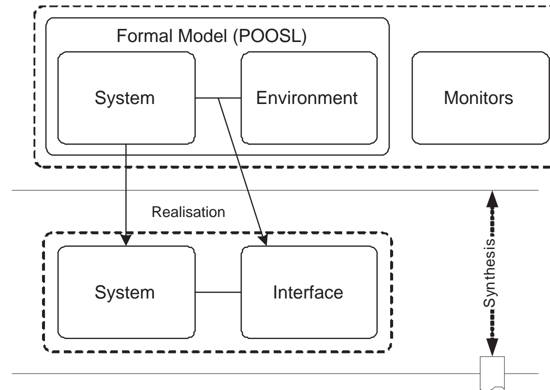


Figure 1.5: The synthesis phase of the design process

realised through device drivers that are implemented in C++ code. The deliverables of this phase in the design flow of SHE are the realised system accompanied with documentation of all interfaces to its environment.

Formally, the synthesis approach is based on a metric that characterises the distance between a model and its implementation. The required real-time properties of the system, expressed as MTL [59] formulas and satisfied by the formal model, are guaranteed to be preserved up to the distance between the model and the implementation. However, these techniques do not cover the synthesis of software with data-intensive computations. Moreover, synthesis of software on distributed platforms as well as synthesis of hardware are topics for future research.

1.2.3 Related Research

One of the main industrially applicable approaches for the design of embedded real-time systems is based on SystemC [54], which is an extension of the C++ language with classes that enable the modelling of such systems. This extension provides support for modelling concurrent behaviour, a notion of time-sequenced operations, extra data types for describing hardware, structural hierarchy and simulation support. Although this expressive language is largely used in industry and it has been adopted as an IEEE standard, SystemC does not enable rigorous, mathematically based analysis and synthesis of models. Hence, a number of design iterations are typically required to ensure that the final system fulfills the requirements, although guarantees for this cannot be provided.

In the context of design methodologies, the Unified Modelling Language (UML) [81] has been adopted as the standard facility for constructing models. UML has proved to be suitable for modelling functional aspects of a system and extensions are defined to it to provide a standardised way of denoting timing aspects for real-time systems [75]. Nevertheless, application of mathematical analysis techniques remains complicated due to the difficulty of relating formal techniques to UML diagrams [50]. Moreover, due to the lack of a standard formal semantics, a clear relation between the properties of a UML model and of its generated implementation cannot

be established. Hence, properties of the system implementation cannot be predicted from the model.

Based on UML, a couple of approaches to support the design of embedded real-time systems were developed. As an example, the Real-Time Object-Oriented Modelling (ROOM) method [84] has a platform-dependent semantics and, hence, the simulation of a model depends on the target platform on which it is executed, and an accurate analysis of the timing behaviour of a system cannot be performed [53]. The generation of the software implementation from a ROOM model is based on a virtual machine that runs on top of the target platform. The virtual machine is linked to a service library to enable the interaction of the model with the environment. Due to the platform-dependent semantics, it is not possible to predict and to guarantee appropriately the properties of the final product using this design methodology.

Another UML-based design approach, TAU2 [88] from Telelogic relies on the concept of virtual time whose progress is not directly affected by the progress of the physical time. In this way, the semantics of real-time systems models is well-defined in a platform-independent way and enables a reliable way of models analysis. However, TAU2 does not have a reliable synthesis mechanism to guarantee the preservation of the model-verified properties in the implementation. During automatic code generation, the timing expressions rely on an asynchronous timer mechanism provided by the underlying platform and thus, all expressions referring to some amount of time will refer to at least that amount of time. Timing errors are accumulated during the execution and this leads to timing and even to functional failures [53].

Rhapsody [87], the industry leading model-driven design framework for embedded real-time systems, is another design methodology from Telelogic. Rhapsody is based on SysML [76], an extension made for UML with the intention to unify the diverse modelling languages used in system engineering. Being UML-based, SysML also lacks the formal semantics, which makes Rhapsody not appropriate for rigorous analysis and mathematically proved correct synthesis.

As the expressiveness and the formal semantics of the modelling language appear to be important ingredients for the success of a design methodology for real-time systems, a number of formal frameworks have been developed based on different mathematical structures, such as timed automata [9], time process algebra [71], real-time calculus [102].

Modular Performance Analysis (MPA) [102] is a design approach based on the real-time calculus which enables the calculation of hard upper and lower bounds of the system performance. However, these bounds are in general conservative because the models are quite abstract and do not cover aspects like probabilistic distribution in the behaviour of a system. Such an aspect makes the analytical calculation of performance properties very difficult and time-consuming. Additionally, MPA is only meant for modelling and analysis phase of the design process, thus it does not provide the means to derive from a model its corresponding realisation.

TIMES [7] is a methodology for the design of embedded real-time systems based on timed automata. The important characteristics of real-time systems, such as concurrency and timing, can be captured in timed automata models which can be rigorously analysed. Because the analysis technique is based on exhaustive checking of all possible behaviours of a system, potential variations in the system behaviour

are ignored and only worst-case situations are considered. Hence, the scope of this design methodology is mainly the hard real-time systems domain. When the analysis results confirm the fulfillment of the system requirements, automatic synthesis of C-code [12] on a BrickOS [2] platform guarantees the preservation of the model properties in the implementation.

The examples that we have presented in this section as related research on design methodologies for embedded real-time systems, for which table 1.1 gives an overview, show a glimpse of the large effort that has been done in this area. More examples and a more detailed comparison of design methodologies can be found in several surveys, such as [104] and [18]. Each design approach has its own strong as well as weak points, either that they refer to the semantics of the language, to the possibility of a rigorous analysis, or to the existence of a correctness preserving synthesis mechanism. For these reasons, none of the existing design methodologies has become *the design methodology* that offers a complete flow from an idea of a product to its correct realisation for a large class of real-time systems and in a predictable way.

	SystemC	ROOM	TAU2	Rhapsody	MPA	TIMES	SHE
Expressivity	yes	no	no	no	no	yes	yes
Platform-Independent Semantics	no	no	yes	yes	yes	yes	yes
Formal Semantics	no	no	yes	no	yes	yes	yes
Rigorous Analysis	no	no	yes	no	yes	yes	yes
Correctness Preserving Synthesis	no	no	no	no	-	yes	yes
Target Realisation Language	C++	C, C++, Java	C, C++	C, C++ Java, Ada	-	C	C++

Table 1.1: Overview of several design methodologies

1.3 Problem Statement and Research Contributions

In the beginning of this introductory chapter, we have posed a problem that has been, as shown in the previous section, the topic of much research in the past years. This thesis contributes to different aspects of the Software/Hardware Engineering design methodology to offer the designer a fairly complete design flow, based on formal methods, that is able to predict and to guarantee both functionality and timing properties in the realisation of real-time systems under tight time-to-market, cost and energy consumption constraints. The research contributions brought by this thesis are the following:

Language-independent modelling patterns. To ensure the correctness and the performance properties of real-time systems, models of such systems should be derived and evaluated from early phases of the design process. To alleviate this process, we have developed a set of modelling patterns to enable easy creation of models for exploration of different design alternatives. The modelling patterns act as parameterisable templates that cover components, such as tasks, resources, input/output devices, that are typical for a large class of real-time systems. They enable the modelling of systems according to the Y-chart

approach [57]. These patterns have the potential to be used in any existing design methodology and we present an implementation of them in POOSL as a library. The POOSL library of modelling patterns enlarges the set of guidelines for the use of the SHE methodology in the design of a large class of real-time systems, covering control-dominated and control with data-intensive computations systems that can be deployed on RISC-like processors [47].

Automatic model generation. To describe a real-time system in terms of the patterns needed and the values of their parameters, we have designed a Pattern-based system Description Language (PDL). This language is easy to use and it does not require knowledge of any of the modelling languages typically used for analysis of real-time systems. Its main advantage, besides simplicity, is that it enables automatic generation of models in different modelling languages. In this thesis, we present a tool that translates a PDL description into a POOSL model based on the POOSL library of modelling patterns. Due to the expressiveness of the POOSL language, appropriate models of real-time systems can be generated and the analysis techniques associated to the SHE methodology enable predictions of the properties of the final product realisation as well as appropriate design decisions making.

Property preservation prediction. To synthesise a model of a real-time system in a "correct" and "efficient" way, it is important to understand the relation between the properties of the model and of its corresponding implementation. The techniques developed in [51] characterise the preservation of properties in the synthesis phase based on a metric to measure the distance between a model and its implementation. In this thesis, we propose a mechanism to determine this distance directly from a model, before actually going through the synthesis phase of the SHE methodology. This approach has the potential to avoid design iterations that are caused by successively obtaining large distances during synthesis.

Correct and efficient real-time model synthesis. To be able to apply the synthesis phase of the SHE methodology to a larger class of real-time systems, including those with data-intensive computations, we have refined the metric on which the techniques presented in [51] are based. The newly defined metric discriminates between what it is observed from the environment of the system and what it is not. By adopting this discrimination, the model synthesis mechanism that we have improved is able to yield real-time software implementations correctly and efficiently, strongly preserving the properties, for both control-dominated and control with data-intensive computations systems.

1.4 Thesis outline

This thesis in structured in eight chapters, as shown in figure 1.6.

Chapter 2: Educational Example. An educational example is presented in order to emphasise the issues in the design of real-time systems.

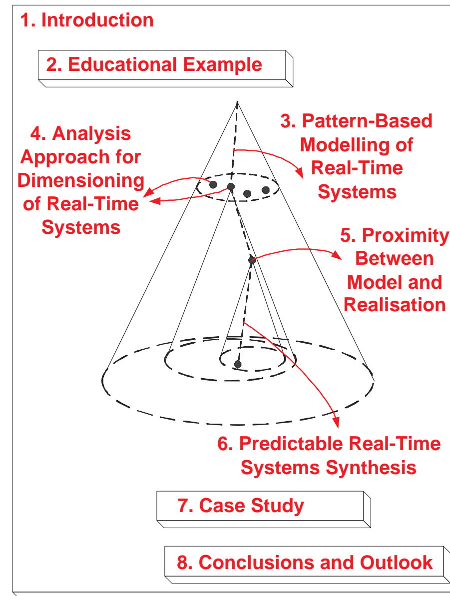


Figure 1.6: Outline of the thesis

Chapter 3: Pattern-Based Modelling of Real-Time Systems. A set of language-independent modelling patterns is presented. This chapter is based on the work entitled "Reusing Real-Time Systems Design Experience Through Modelling Patterns" that won the Best Paper Award at the 9th Forum on Specification and Design Languages (FDL) [38] and was invited as a book chapter in "Advances in Design and Specification Languages for Embedded Systems" [39].

Chapter 4: Analysis Approach for Dimensioning of Real-Time Systems. The analysis approach and the design space exploration of models for real-time systems are discussed in this chapter based on two realistic case studies. Earlier results on this topic have been published with the title "Probabilistic Modelling and Evaluation of Soft Real-Time Embedded Systems" in the proceedings of the 6th Embedded Computer Systems: Architectures, Modelling, and Simulation (SAMOS) [33].

Chapter 5: Proximity Between Model and Realisation. This chapter describes an approach for evaluating the real-time properties of a system and how strong they can be preserved in the realisation. Earlier work on this topic, "Error Estimation in Model-Driven Development for Real-Time Software", has been published in the proceedings of the 7th Forum on Specification and Design Languages (FDL) [35].

Chapter 6: Predictable Real-Time Systems Synthesis. Improvements on the efficiency of an existing model synthesis mechanism are discussed in this chapter which is based on some early ideas presented in "Property-Preservation

Synthesis for Unified Control- and Data-Oriented Models", a chapter in the book "Applications of Specification and Design Languages for SoCs" [36], and on the paper "Strengthening Property Preservation in Concurrent Real-Time Systems" published in the proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) [34].

Chapter 7: Case Study. A case study illustrating the various steps of our design methodology and the improvements brought by our contributions is presented.

Chapter 8: Conclusions and Outlook. Conclusions are drawn in this chapter and future research ideas, some of which have already been started to be applied as continuation for this work, are discussed.

In addition, appendix A provides the mathematical foundation for the formalisation presented in chapter 5.

2
Educational Example

To illustrate the steps of the model-driven design trajectory of the SHE methodology, in this chapter we present an educational case study. First, section 2.1 describes the system and its requirements. Different levels of abstraction of the model of the system, from a highly abstract to a synthesisable model, are presented in sections 2.2 to 2.5. The summary of the chapter is given in section 2.6.

2.1 Motion Control System Description

The setup of a simple motion control system that is taken as educational case study is depicted in figure 2.1. The system is made of two rotation units, each driven by a motor. The velocity of the first motor is changed from time to time while the motor is running. The second motor runs at constant velocity as long as the first one does the same. When the velocity of the first motor increases or decreases, so does the velocity of the second motor. If the velocity of the second motor gets above a certain limit, this motor stops and the first motor is communicated to do the same. Such a system is representative, for example, for the control of a part of a printer paper path where several motors must be controlled concurrently and their velocities, which depend on one another, must ensure the arrival of the sheets of paper within certain time frames at certain points.

In the setup of the system, for the interface between the computer that ensures the control and the motion system, we used a TUeDACS [1] data acquisition system that couples to the computer via the PCMCIA 20 Mbit/s serial link. It operates in real-time, without buffering of data, and provides a fast link to the computer. Moreover, the operating system running on the laptop is RTAI/Fusion Linux [5], a hard real-time extension to the Linux kernel that provides the features of an industrial-grade

real-time operating system, seamlessly accessible from the powerful and sophisticated GNU/Linux environment.

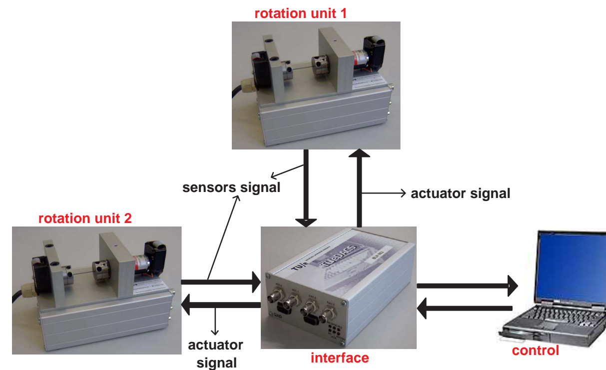


Figure 2.1: The setup of the case study

In our setup, each rotation unit is made of two masses connected through a flexible shaft. One mass is excited by a motor, while the other is excited by the rotation of the first mass via the shaft. The movement of each motor, and accordingly of each of the first masses, follows a linear law, r(t) = v * t, where the position r is given as a function of the velocity v and the current time t. The initial velocity of both motors is 20 rotations/s. When the velocity of the first motor is increased, the velocity of the second motor doubles its current value. When the velocity of the first motor is decreased, the velocity of the second motor is decreased with one third of its current value. If the velocity of the second motor increases to more than 45 rotations/s, then it stops.

The purpose of this educational case study is to obtain a realisation of the system in a model-driven way. This realisation must ensure that the velocities of the motors change as described above and the motors stop when one of them reaches a certain velocity value. Moreover, the stability of the two rotating devices must be achieved. The controller of each of the two devices was designed with the help of the control engineers from the Dynamics and Control Technology group within Eindhoven University of Technology [67]. A sampling frequency of 1000 Hz for the first motor and of 500 Hz for the second motor were chosen. Based on the work presented in [24], it could be established that a timing accuracy of 0.1 ms for each controller is sufficient to guarantee the stability of the closed loop system. Hence, the time interval between the reading of a sensor signal and the writing to an actuator is $[0.9, 1.1]$ ms for the first motor and $[1.9, 2.1]$ ms for the second motor.

2.2 High-Level System Model

To model the system, we need to identify its so-called players as it was shown in [52]. In our system, the players are the two rotation units. The first rotation unit starts rotating with some initial velocity, after which the velocity is increased and decreased

alternatively. The second rotation unit needs to adapt its velocity depending on the first rotation unit in the following way: it starts with its initial velocity which is doubled when the first unit accelerates or it is reduced with one third when the first unit decelerates.

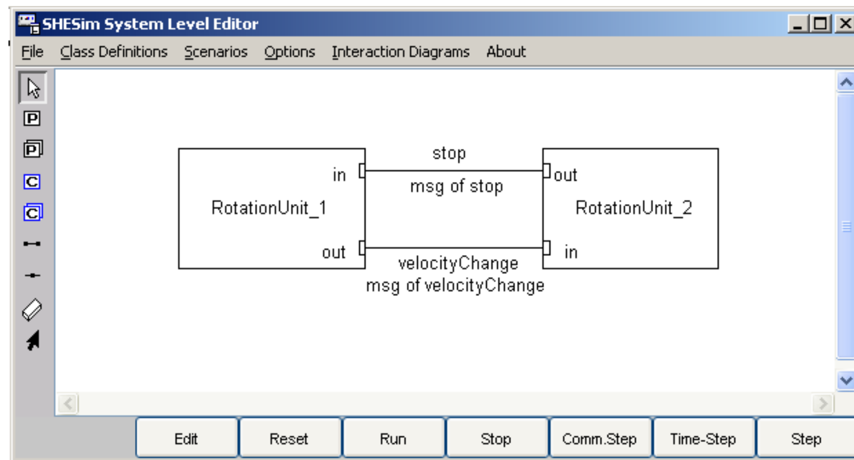


Figure 2.2: The high-level model of the case study

A high-level model of the system specified in POOSL using the SHESim environment is presented in figure 2.2. `RotationUnit_1` corresponds to the first device, whereas `RotationUnit_2` corresponds to the second device. Messages containing information referring to changes in the velocity of the first motor flow from `RotationUnit_1` to `RotationUnit_2` along a POOSL communication channel denoted with `velocityChange` in the figure. A message is sent by `RotationUnit_2` to `RotationUnit_1` along channel `stop` to notify it when the second motor stops. The POOSL specification of each of the methods of the two POOSL processes is given in figures 2.3 and 2.4 respectively.

`RotationUnit_1` needs to ensure the rotation of its corresponding motor as well as to notify `RotationUnit_2` about changes of the velocity. Moreover, it must be able to receive the notification from the second rotation unit when it stops. To specify this, we use the **par-and-rap** POOSL statement which indicates which are the two parallel activities of `RotationUnit_1` in the `Init` method, which is the initialisation method for process `RotationUnit_1`.

Each of the activities of `RotationUnit_1`, `Rotate` and `AcknowledgeStop`, is modelled as a separate method as depicted in figure 2.3. Method `Rotate` models the way the velocity of the first motor changes. In a real printer paper path, sensors would trigger when a sheet of paper passes by a certain position. Because the paper needs to have a certain timing, it can be detected if it is late or early and the velocity is increased or decreased respectively to certain values. Because in this simple example we do not have triggers from sensors, we use a simple scheme of changing the velocity of the motor: alternation of the two values the velocity can have every 2 units of time that correspond to 2 s. The velocity is first doubled and then halved al-

```
Init()()
  v := 20;
  par
    Rotate()()
  and
    AcknowledgeStop()()
  rap.

Rotate()()
  delay 2;
  [v != 0] v := v*2;
  [v != 0] out!change(v);
  delay 2;
  [v != 0] v := v/2;
  [v != 0] out!change(v);
  Rotate()().

AcknowledgeStop()()
  in?stop {v := 0}.
```

Figure 2.3: POOSL specification of the methods of `RotationUnit_1`

ternatively under the condition that the motor is running. This condition is specified using the guard $[v! = 0]$ enforcing that the speed is not zero in order to change it. The specification of method `Rotate` also shows that each time the velocity is changed, a message `change` is sent on the `out` port if the motor is not stopped. This condition is implemented using the same guard $[v! = 0]$. The message contains the new value of the motor velocity, `change(v)`. Method `AcknowledgeStop` specifies the waiting of a message `stop`. After receiving such a message, the velocity turns to zero, which means the motor stops running. The call of `AcknowledgeStop` finishes, whereas `Rotate` blocks because the condition of the guard $[v! = 0]$ is false, thus it cannot proceed. In this way, `RotationUnit_1` stops.

```
Init()()
  v := 20;
  x_prev := 0;
  par
    RotateAndChange()()
  and
    CommunicateStop()()
  rap.

RotateAndChange()() | x : Integer |
  in?change(x);
  if (x_prev < x) then v := v*2
  else v := v-v/3
  fi;
  x_prev := x;
  RotateAndChange()().

CommunicateStop()()
  [v > 45] out!stop;
  v := 0.
```

Figure 2.4: POOSL specification of the methods of `RotationUnit_2`

In a similar manner, `RotationUnit_2` needs to ensure the rotation of its motor as well as to get notifications from `RotationUnit_1` about changes of the velocity. Moreover, it must be able to send a notification to the first rotation unit when it stops. To specify this, we use the **par-and-rap** POOSL statement which indicates which are the two parallel activities of `RotationUnit_2` as indicated in the specification of the initialisation method `Init`.

The POOSL specifications of the methods `Init`, `RotateAndChange` and `CommunicateStop` are presented in figure 2.4. Method `CommunicateStop` ensures that the moment when the velocity of the second motor increases to a value larger than 45, a message `stop` is sent to the first rotation unit and the velocity becomes zero, meaning that the motor stops. Method `RotateAndChange` enables this second rotation unit to change the velocity of its motor as soon as it is notified that the first motor changed its velocity. If the velocity of the first motor has increased

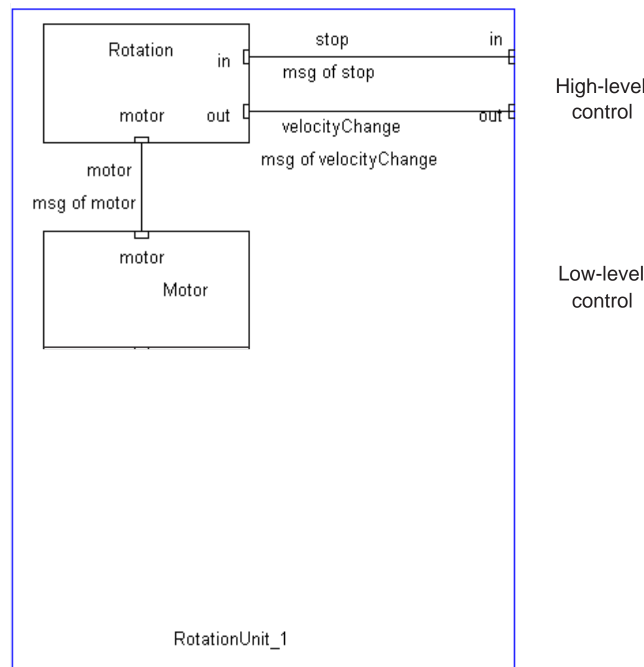


Figure 2.5: A refined model of `RotationUnit_1`

from the last communication then the velocity of the second motor doubles its current value. If the velocity of the first motor has decreased, the velocity of the second motor decreases with one third of its current value. The current value of the velocity of the first motor is kept for later comparison in the instance variable `x_prev`.

With such a model, it can be analysed if and when the system stops. The analysis shows that the first motor changes its velocity 3 times and it stops at time 6, which corresponds to 6 s after the system starts. Moreover, from the model we can also see that the second motor is able to immediately follow the changes in the velocity of the first one.

2.3 Adding Details to the System Model

A mechatronic system is built with the contributions of several disciplines: software, control and mechanical engineering. Each of these disciplines tackles problems related to different parts of the system. Moreover, they also interact with each other to fulfill the functionality of the system as a whole. Software engineering mainly addresses the high-level control of the system, which plans actions for the physical elements of the system. Control engineering deals with the low-level control of the system, which derives stable and optimal control algorithms for the physical elements. Mechanical engineering applies principles of physics to implement the

physical elements. When a high-level control unit generates an action for a physical element, the action is interpreted by a corresponding control algorithm in a low-level control unit. For example, when a high-level unit issues a command "motor 1: start to move", this action is actually connected to a low-level control loop which ensures that the physical motor starts to move according to a predefined motion profile.

To specify the interactions between different disciplines, we use a more detailed model. For design consistency, the detailed model is obtained by enhancing the high-level model of the system discussed in section 2.2, but by keeping the same observable behaviour. In this way the properties analysed for the high-level model can be preserved in the detailed model. A refinement of the model is built by splitting each of the players in two kinds of processes which correspond to the high-level control and the low-level control, as it was shown in [52]. The refinement of the `RotationUnit_1` is depicted in figure 2.5. To simplify the analysis of the interactions between different disciplines, the behaviour of the low-level control unit is specified at a discrete event level of abstraction. This allows the major interactions between disciplines to be analysed in a simple model. Moreover, the model provides also a framework for later integration of the continuous time behaviour in a straightforward way. In the rest of this section, we use `RotationUnit_1` to illustrate the refinement of the model.

```
Init()()
  v := 20;
  motor!start(v);
  par
    Rotate()()
  and
    AcknowledgeStop()()
  rap.

Rotate()()
  delay 2;
  [v != 0] v := v*2;
  [v != 0] out!change(v);
  [v != 0] motor!continueWith(v);
  delay 2;
  [v != 0] v := v/2;
  [v != 0] out!change(v);
  [v != 0] motor!continueWith(v);
  Rotate()().

AcknowledgeStop()()
  in?stop {v := 0};
  motor!stop.
```

Figure 2.6: POOSL specification of the `Rotation` behaviour in `RotationUnit_1`

The refined model of `RotationUnit_1`, depicted in figure 2.5, contains the `Rotation` POOSL process, which models the high-level control, and the `Motor` POOSL process, which models the low-level control. The behaviour of `Rotation` is an extension of the behaviour of `RotationUnit_1` that was presented in figure 2.3 by adding the interactions to the other disciplines. The refinements of `RotationUnit_1` process, as shown in figure 2.6, are made to all its methods. In method `Init`, a message `start` is sent to the low-level control on port `motor` to start the rotation of the motor. In `Rotate`, a `continueWith` command is sent to the low-level control when the motor needs to change its velocity, whereas in `AcknowledgeStop`, a message `stop` is sent when the motor has to stop.

The low-level control is modelled with the `Motor` process, whose methods specifications are depicted in figure 2.7. The `Init` method starts with the receiving of a `start` message containing the velocity with which the motor should start rotating. The discrete event behaviour of the low-level control of the motor is modelled by

```
Init()() |x :  Integer|
  motor?start(x);
  v := x;
  DiscreteBehaviour()().

DiscreteBehaviour()() |x :  Integer|
  sel
    motor?stop;
    v := 0;
  or
    motor?continueWith(x);
    v := x;
    DiscreteBehaviour()()
  les.
```

Figure 2.7: POOSL specification of the `Motor` behaviour in `RotationUnit_1`

method `DiscreteBehaviour` which either receives a `stop` message and stops the motor, or receives messages of continuation, `continueWith`, containing the value of the velocity the motor should have.

2.4 A More Detailed Model of the System

In the first two stages of the modelling activity, we could only reason about the qualitative properties of the system which are the fact that the motors change their velocities one based on the other and that the system stops eventually. Since timing information is present in the model of the high-level control units, we could also analyse at what moment in time the system stops, as shown in section 2.2. However, we also need to analyse the stability of the system. Therefore, the continuous time behaviour of the system is of interest to our analysis and hence, we need to incorporate timing information for the low-level control and to reason about the timing properties satisfied by the model.

The timing information for the low-level control is added by incorporating the control loop into the model. The control loop is a periodic activity that is needed for controlling the physical device, the actual motor. To specify the interactions between the low-level control and the physical device, we refine the model of the system by splitting each rotation unit into three kinds of processes corresponding to the high-level control, the low-level control and the physical device. Figure 2.8 depicts the refined model of `RotationUnit_1` containing `Rotation`, the high-level controller whose behaviour remains unchanged, `Motor`, the low-level controller to which we add the control loop, and `MotorActor`, the physical device. Since the physical device will not be synthesised as software and to simplify the analysis, its behaviour is specified at a discrete event level of abstraction.

The `Motor` process describing the low-level control unit incorporates the continuous time behaviour by specifying a separate activity, `ContinuousBehaviour`, in parallel with the discrete one, `DiscreteBehaviour`, as shown in figure 2.9. The process method `ContinuousBehaviour` models the control loop and the interaction of the low-level controller with the physical device. Typically, the body of a control loop contains the reading of the current position of the encoder of the motor, some calculations based on the value read which are meant to compensate for the error that occurred, a delay until the next period of time, and the sending of the position correction information to the actuator of the motor. As it can be seen in figure 2.9, we have grouped together in method `FeedbackController` all the

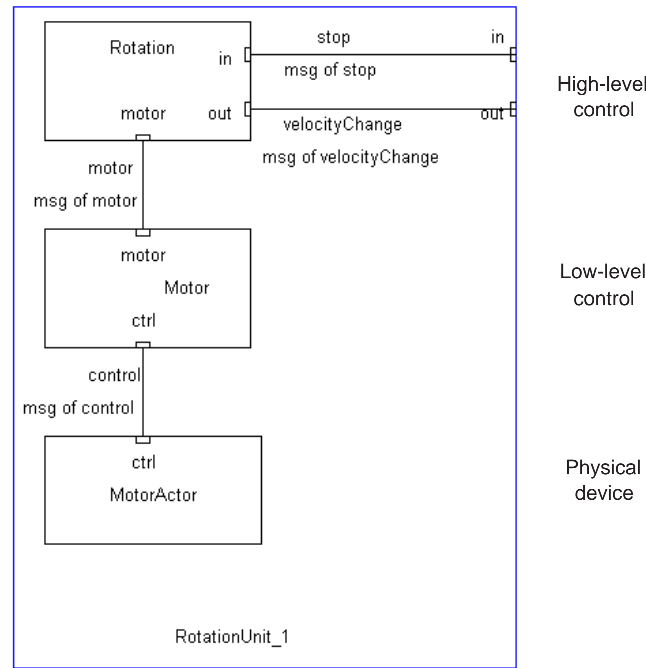


Figure 2.8: A more detailed model of `RotationUnit_1`

activities that take place in a period of time, which are the writing of the previously calculated value, the reading of a new value and the control algorithm, and we used the **par-and-rap** POOSL statement to denote that exactly after a period of time equal to 0.001 s this behaviour repeats. Since this model emphasised the structure of a typical periodic task in real-time systems, this specification represents a modelling pattern for periodic tasks. A set of such modelling patterns for different types of tasks as well as resources of real-time systems is presented in chapter 3.

Control algorithms are typically designed using commercial tools, such as Simulink [89] and 20-sim [6]. To provide a natural integration of them in the detailed model of a system specified in POOSL, they are represented by data methods. Later on, these methods will be replaced by actual algorithms during system synthesis. To model the communication to the physical device for reading the encoder and writing to the actuator of the motor, `read` and `write` messages are exchanged over the `control` channel that connects `Motor` and `MotorActor` processes. Method `controlAlgorithm(position,v)` of the data object `motor`, which is instantiated in the `Init` method, models the control algorithm that calculates the compensation of the error in the rotation of the motor. The value of the delay that elapses before the correction information is sent to the motor represents the period of the control loop that was designed by the control engineers to ensure the control stability.

The physical device, the actual motor that is rotating, is modelled by

```
Init()()
  motor?start(x);
  v := x;
  calculated_position := 0.0;
  motor := new MotorController();
  par
    DiscreteBehaviour()()
  and
    ContinuousBehaviour()()
  rap.

DiscreteBehaviour()() |x :  Integer|
  sel
    motor?stop;
    v := 0;
  or
    motor?continueWith(x);
    v := x;
    DiscreteBehaviour()()
  les.

ContinuousBehaviour()()
  par
    FeedbackController();
  and
    delay 0.001;
    ContinuousBehaviour()()
  rap.

FeedbackController()() |position :  Real|
  ctrl!write(calculated_position);
  ctrl?read(position);
  calculated_position := motor controlAlgorithm(position, v).
```

Figure 2.9: POOSL specification of the `Motor` behaviour in the detailed model

```
Init()()
  position := 0.0;
  Run()().

Run()() |new_position, position :  Real|
  ctrl?write(new_position);
  position := new_position;
  ctrl!read(position);
  Run()().
```

Figure 2.10: POOSL specification of the `MotorActor` behaviour

`MotorActor` POOSL process, presented in figure 2.10, which is a discrete event approximation of the actual behaviour of the motor. `MotorActor` receives from the low-level controller messages on port `ctrl` containing correction data for its position, `ctrl?write(new_position)`, and is able to send back messages containing its current position, `ctrl!read(position)`. Since the model of the `MotorActor` will not be synthesised in software, we specified it in a very simple way such that we can analyse the interactions with the low-level controller.

With such a model of the system, which contains timing information, it can be analysed which actions are taken at which moments in time. For example, we can determine the time interval between the reading from and the writing to a physical device, which is 1 ms for the first rotation unit and 2 ms for the second rotation unit, satisfying the timing requirements for the stability of the system. Since the actions specified in the model do not take model time, it would be interesting to determine how large the inevitable time deviation between the model and its realisation on a target platform would be. Chapter 5 presents how the value of the time deviation can be calculated from a model based on the values of the worst-case execution times of all its actions on the desired target platform. By determining the value of the time deviation from the model, we can predict how well the realisation of the system will preserve the properties analysed in the model. Nevertheless, we can also generate the realisation and measure its time deviation directly on the target platform to check if the requirements of the system are satisfied by the realisation, as we show in section 2.5.

Depending on the requirements of the system, it might be that for some real-time systems other properties might also be interesting to analyse. For example, if deadlines are specified for each task, then the schedulability of the system needs to be analysed in order to check if all tasks meet their respective deadlines. In chapter 4, we present an approach for analysing different timing aspects of a system, such as schedulability, end-to-end delay, task jitter, based on which an appropriate dimensioning of its target platform can be looked for.

2.5 Synthesis Model of the System

When the model has sufficiently many details describing the behaviour of a system, it is desirable to directly generate from such a model its implementation that can run on a certain target platform. This implementation is important to have the same properties as those analysed in the model. As we can only generate the implementation of that part of the model that represents the software, those parts of the model that represent anything else but software are removed from the model, as shown in figure 2.11.

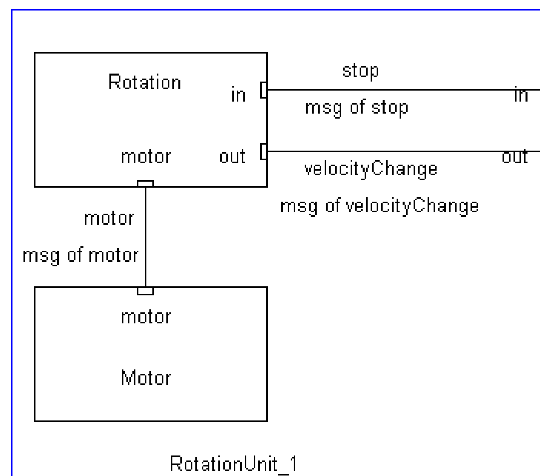


Figure 2.11: The synthesis model of $RotationUnit_1$

The interactions between software and the other disciplines, which represent its environment, need to be replaced with the actual implementation of the communication between them. In our motion control system, this communication is represented by the reading of data from the encoder of a motor, data needed by the control algorithm to calculate the correction data, and the writing of the correction data to the actuator of the motor. The reading is modelled in the continuous behaviour of the `Motor` process as receiving a message on the `control` channel from the `MotorActor` process, whereas the writing is modelled as sending a message along the same channel. As the environment of the software is removed from the model, `MotorActor` is no longer present, hence the `control` channel is also not

```
Init()()
  motor?start(x);
  v := x;
  calculated_position := 0.0;
  motor := new MotorController();
  par
    DiscreteBehaviour()()
  and
    ContinuousBehaviour(0.0)()
  rap.

DiscreteBehaviour()() |x : Integer|
  sel
    motor?stop;
    v := 0;
  or
    motor?continueWith(x);
    v := x;
    DiscreteBehaviour()()
  les.

ContinuousBehaviour()()
  par
    FeedbackController();
  and
    delay 0.001;
    ContinuousBehaviour()()
  rap.

FeedbackController()() |position : Real|
  motor write(calculated_position);
  motor read(position);
  calculated_position := motor controlAlgorithm(position, v).
```

Figure 2.12: POOSL specification of the `Motor` behaviour in the synthesis model

present. The communication to the physical device is replaced in the `Motor` process with calls to data methods as shown in figure 2.12. During synthesis, these data methods will be replaced by C++ functions, so-called primitive methods, that can be invoked from the C++ code derived from the POOSL model. The C++ implementation of the primitive method corresponding to the action of writing to the actuator is provided in figure 2.13. Besides the communication to the environment, the control algorithms that have been developed using tools specific to the control systems domain are now inserted in the synthesis model as primitive methods as well.

```
void WRITE (PD_DAS *pd)
{
  /* write the data to the actuator via TUeDACS API */
  TD_DAC_WRITE_CHAN(pd->u, pd->channel, pd->link, TD_DIRECT);
}
```

Figure 2.13: The C++ implementation of the data method WRITE

For the interface between the computer, which is running a 1.4 GHz PentiumM processor, and the motion system, we used a TUeDACS [1] data acquisition system that couples to the computer via the PCMCIA 20 Mbit/s serial link. It operates in real-time, without buffering of data, and provides a fast link to the computer. Moreover, the operating system running on the laptop is RTAI/Fusion Linux [5], a hard-real-time extension to the Linux kernel.

By using Rotalumis-RT, the tool for code generation from POOSL models, C++ code is synthesised from the model and it is linked to its corresponding primitive methods. As a model is an abstract representation of a system, actions are considered instantaneous, whereas in reality they cannot be like that. Therefore, some deviations with respect to timing occur between a model and its realisation. Measurements of the generated implementation of our case study showed a maximum

deviation of 0.221 ms from the model. Figure 2.14 shows how the measured time deviation is used to determine the time interval between the reading of the encoder and the writing to the actuator, which is $[0.558., 1.442]$ ms for the first motor and $[1.558, 2.442]$ ms for the second motor. This deviation does not comply with the timing accuracy with which the control algorithms have been designed. Hence, this realisation cannot ensure the stability of the control of the physical devices.

To avoid the trouble of synthesising a model and discovering that its realisation does not satisfy the timing requirements of the system, in chapter 5 we present an analytical approach for determining the time deviation before synthesising the model. As this time deviation is computed directly from the model, this approach can save design cycles by avoiding designers to get into the trouble of actually building a wrong system. If the time deviation is too large such that the realisation of the system cannot satisfy the timing requirements, changes of the model can be made, e.g. choosing a faster processor, and the time deviation is calculated again. This procedure is repeated until the obtained time deviation ensures that the realisation of the system satisfies the requirements.

In the analytical approach for determining the time deviation between model and realisation presented in chapter 5, no distinction is made between the actions performed by the system. A system usually has two types of actions: *observable* actions, which are actions that a user can "see" by interacting with the system, and *unobservable* actions, which are actions internal to the system that a user cannot "see". For example, in the simple motion control system considered here as an educational example, the *observable* actions are the reading of the encoders and the writing to the actuators, whereas all the other activities of the system are *unobservable* actions. Chapter 6 presents an improvement of the synthesis mechanism based on the discrimination between observable and unobservable actions in a system. This improvement is achieved by imposing higher priority on the execution of the observable actions over the unobservable ones. Correspondingly, the analytical approach for computing the time deviation before synthesising the model can give smaller values, whereas the code generation tool based on this mechanism is able to decrease the time deviation between model and realisation. Using this improved synthesis technique, which will be explained in detail in chapter 6, the time deviation for the motion control system decreased to 0.037 ms. This value ensures that the time interval between the reading of a sensor signal and the writing to an actuator is $[0.926, 1.074]$ ms for the first motor and $[1.926, 2.074]$ ms for the second motor, satisfying the timing requirements for stability.

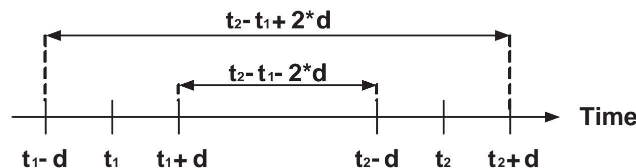


Figure 2.14: The influence of a time deviation of d units of time on the time interval between two actions that occur at t_1, respectively at t_2 in the model

2.6 Summary

This chapter has presented a simple case study to illustrate the steps taken in the model-driven design trajectory of the SHE methodology. Moreover, we have pointed out the contributions of this thesis, their position along the trajectory and their motivation. The approach starts with a simple, abstract model of the system, which is gradually refined until a complete specification that meets the requirements of the system is obtained. Using a code generation tool, C++ code is synthesised and executed on the target platform.

3
Pattern-Based Modelling of Real-Time Systems

Conservative assumptions about systems, e.g. that each task runs for its worst-case execution time, are suitable when analysing hard real-time systems. However, they are often too restrictive when analysing soft real-time systems as they lead to over-dimensioned and expensive products. For these systems a certain percentage of deadline misses is usually affordable. Hence, instead of a binary answer regarding the schedulability of such a system, a more interesting metric is the degree to which the system meets the timing requirements. For this, an appropriate model that expresses realistically the behaviour of a soft real-time system, including all its possible variations, should be built and analysed. In this chapter, we present a modelling approach that is suitable for both hard and soft real-time systems in the areas of control-dominated and control with data-intensive computations applications. This modelling approach enables analysis of both schedulability for hard real-time systems as well as the degree to which the timing requirements are met for soft real-time systems.

Furthermore, to alleviate the process of modelling, we also present a set of language-independent modelling patterns to enable easy creation of models suited for design space exploration. The modelling patterns cover typical system components as they are considered in classical scheduling theory, such as different types of tasks, resources, input event generators, and output collectors. The easiness of the modelling process is ensured by the Pattern-based system Description Language (PDL) that enables a textual specification of a system in terms of its components and their parameters.

This chapter is organised as follows. Section 3.1 introduces the issues of the currently used modelling approaches for real-time systems. Related research work is

discussed in section 3.2. The set of modelling patterns that we developed is introduced in section 3.3. Section 3.4 is a brief presentation of the UML profile for POOSL modelling language that will be used throughout the chapter, whereas section 3.5 presents each pattern and its POOSL implementation. The PDL language and the construction of a model from the patterns are discussed in section 3.6. A summary of the chapter is given in section 3.7.

3.1 Modelling Approaches for Real-Time Systems

Research in the area of real-time systems has emerged from the need to design, analyse and predict the behaviour of safety critical applications as power plant control, aircraft control or accurate printer/copier machines [65], [20]. Hence, most of the work has focussed on hard real-time systems analysis to ensure that no timing requirement is broken as this might have catastrophic consequences. The way to meet all the timing requirements and thus, to ensure that the system is schedulable, is to make conservative assumptions about the system, e.g. that each task runs for its worst-case execution time. However, these assumptions are too conservative for soft real-time systems, which are considered to run correctly even if some timing requirements are occasionally broken [21]. For systems like audio and video encoders, it is more interesting to know the degree to which a system meets its timing requirements rather than the binary answer that the system is schedulable or not.

The most common, industrially applied, approach for the design of soft real-time systems is to consider the system as having hard constraints and to analyse its worst-case behaviour, such as shown in [102]. This approach is usually taken because the analysis techniques for hard real-time systems are very well established and widely known. Its obvious drawback is that it leads to an over-dimensioned and expensive product. Another approach is to distinguish among the tasks of the system the "more important" from the "less important" ones. The level of importance of a task is usually given by its occurrence rate or by the nature of the tasks in the system that depend on it. In such a system, high priorities are assigned to the important tasks and low priorities to the less important ones, whereas the execution time of each task is assumed to be constant and equal to its worst-case value [20]. This approach ensures the schedulability of the important tasks while providing *good* average response times for the less important ones. The main reason for both these approaches of not being suitable for soft real-time systems is that they cannot properly take into account all kinds of variations and that they always assume worst-case situations.

Usually, the execution time of a task depends on a large number of factors. For example, the amount of input data to be processed in a task at a time determines the instruction load imposed on the platform by the task at that time. Moreover, the amount and type of data may vary, as it is for example the case for differently coded MP3 audio samples, as well as the data occurrence rate, such as input events triggered by sensors placed in the environment. All these varying factors introduce variation in the execution time of the task. Another factor that influences the task execution time is the type and throughput of the processing unit on which the task runs. Additionally, cache memory behaviour, pipeline stalls, network load and operating system overhead are other platform-dependent influencing factors that may also

vary. Considering the intricate influences of all these factors on the execution time of a task, an analysis based on the worst-case behaviour leads to an over-dimensioned and expensive system a customer might not want to pay for [86]. For instance, a consumer might prefer to purchase a $50 video player that happens to drop single frames under rare circumstances over a $200 system verified and certified never to drop frames.

Based on the general-purpose formally defined modelling language POOSL, we present in this chapter a modelling approach which is appropriate for both hard and soft real-time systems. Thanks to the expressiveness of the language, many of the possible variations that may occur in a real-time system can be taken into account, such as for example the tasks input data, which can be modelled to be periodic or sporadic, with or without jitter, the instruction load, described with an arbitrary probability distribution function, as well as the platform-dependent factors. A model with fixed, worst-case values, is a special case of the stochastic model that we present here. Although an analysis approach based on a worst-case situation model yields a product that is guaranteed to fulfil all the timing requirements, it is likely that the product is expensive and the occurrence of its worst-case behaviour has a small probability. When the timing requirements are not hard, it is more appropriate to consider for analysis a model of the realistic, average case behaviour of the system, which may lead to a cheaper architecture.

Additionally, based on the observation that system models are built from similar components, in this chapter we present an easy way to build models that enable the exploration of the design space of real-time systems. The easiness of model construction is achieved by using a set of *modelling patterns*. These patterns represent a simple and highly abstract way to specify parameterisable components of real-time systems. The idea of deriving patterns was driven by the fact that real-time systems are composed from typical components like tasks, resources and input/output devices. Each time the design of a new product starts, such commonly encountered components are designed from scratch although similar systems were developed in the past. The modelling patterns represent frozen knowledge derived from past experience. They give guidance to build models for new systems and alleviate the modelling process.

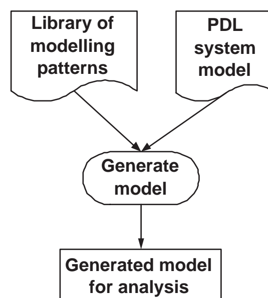


Figure 3.1: Pattern-based model generation

To enable easy use of the patterns we conceived the Pattern-based system Description Language (PDL) to specify models based on the modelling patterns. This

language is independent from the target modelling language and lets one indicate the patterns that are needed to develop a system model and the values of their parameters. Based on an implementation of the modelling patterns in a certain target language and the PDL specification of a system, a model aimed for analysis can be automatically generated in that target modelling language as shown in figure 3.1.

3.2 Related Research

The concept of patterns has been long known as solutions to commonly encountered design problems. The famous publication on design patterns [40] discusses software design patterns, focussing for example on how to create a factory of objects, or how to iterate over a set of elements. With the development of real-time systems, design patterns were also needed for dealing with issues like concurrency, resource sharing, or distribution, which are discussed in [28]. In a similar manner, modelling patterns are solutions for modelling commonly encountered system components. As an example, in [45], the authors propose such patterns to deal with the complexity of Petri nets models of systems by reusing structures expressing expert modelling experience at a higher level of abstraction than the basic elements.

The patterns presented in this chapter alleviate the process of modelling embedded real-time systems. Each of the typical components of these systems, such as tasks, schedulers and resources, is characterised by a set of parameters that capture its important aspects as they are perceived in classical scheduling theory [20]. For each such component we have developed a pattern which is a parameterisable template. In this chapter we present an implementation of these patterns in the formally defined modelling language POOSL.

Models of embedded systems are built to analyse their properties and to make design decisions. Analysis of system models is hence an important phase in the design of a new product. The use of techniques for performance analysis and design space exploration enables prediction about the properties of the final product and of the meeting of its requirements. In the past decades, an impressive amount of work has been carried out in the area of schedulability analysis for meeting hard real-time requirements (e.g. [65], [20], [17]) focussing on worst-case execution and making conservative assumptions about the system. However, as it is also shown in [90] and [66], not so much work addresses the analysis of systems which are still considered correct even with occasional failures of meeting some timing requirements.

To make models more suitable for systems with less restrictive timing constraints, in [21] the authors propose an elastic task model that enables tasks with periodic behaviour to change their execution rate to provide different quality of service. Although this is a relaxation from the Liu and Layland [65] type of model, systems typically exhibit more variations than just in their execution rate. Moreover, the periodic task model is not suitable for systems that are triggered by events from the environment as these are aperiodic by nature.

In [13], the authors extend the classical rate monotonic scheduling policy of [65] with an admittance controller to handle tasks with stochastic execution times. Their

approach is limited though to rate monotonic analysis and assumes the presence of an admission controller at run-time. In [8], the focus is on how to schedule both hard and soft real-time tasks on the same processor. The performance analysis method proposed there is used for assessing the constant bandwidth server scheduling policy and is restricted to the scope of their assumptions (uni-processor architecture, constant bandwidth server scheduling policy and combined hard and soft real-time tasks with stochastic parameters).

In [61], Lehoczky models task sets as a Markovian process. The advantage of this approach is that it is applicable to arbitrary scheduling policies. The process state space is the vector of lead-time (the time left until the deadline). Because this space is potentially infinite, the author analyses it in heavy traffic conditions, which reduces the state space. However, the heavy traffic theory fails to apply smoothly to real-time systems. One of the causes is that the heavy traffic phenomenon can be observed only for processor load close to 1, which leads to large (infinite) queues of ready tasks and to systems with large latency. This leads to a large deadline miss ratio and limits the applicability of such an approach in real-time systems.

In [58], the authors considered applications implemented on multi-processor platforms and modelled them as queueing networks. The underlying mathematical model is the continuous time Markov chain. The authors restrict the task execution times to exponentially distributed ones which reduces the complexity of the analysis. The tasks are considered to be scheduled according to a FIFO policy.

In [66], the authors present an analytical approach for obtaining the expected deadline miss ratio of an application. Considering that task execution times are given as generalised probability distribution functions, their approach is limited to periodic tasks, uni-processor architectures and non-preemptive scheduling policies.

The analysis techniques discussed so far are based on analytical computation of the performance of a system and they are exhaustive in the sense that all possible behaviours of the system are taken into account. To avoid the state space explosion problem, the models they can handle have a number of restrictions as it was shown: either the tasks are periodic, or their execution times are exponentially distributed, or the scheduling is limited to a certain policy. For this reason, these techniques target specific applications and are not suitable for a larger class of systems that include soft real-time systems as well.

On the other hand, there are also analysis techniques based on simulation of models. They allow the investigation of a *limited* number of all the possible behaviours of the system. To be able to interpret the results obtained, the simulation-based techniques typically require their models to be amenable to mathematical analysis as it is shown in [90]. An example of such a simulation method is Spade [64] that uses Kahn process networks [55] which is a timeless computational model suitable for media-processing application domain. A separate model of the target architecture is built based on a library of generic parameterisable building blocks. The application and the architecture models are co-simulated using the trace-driven simulation technique and performance metrics are collected. These metrics include resources utilisations, amount of workload on processors, number of deadlines missed, or amount of data to be transferred over the busses. Different mappings of application onto architecture may yield different results and the best mapping and the most

suitable architecture can be chosen. Nevertheless, in this methodology, as well as in other related methodologies like Artemis [77] and Pamela [98], the model of the system is deterministic and assumes worst-case execution times of tasks in the application model. Possible run-time variations caused by cache and pipeline effects or operating system overhead, as well as data-dependent execution time of tasks are abstracted from. Hence, over-dimensioning of the system might occur because of these reasons. A similar modelling approach is taken in Metropolis [15] which is a system design environment based on a meta-model with formal semantics that enables simulation, formal analysis and synthesis of models. A separate specification, which abstracts from time, of the functional and architecture models is realised and the mapping between the two enables analysis of performance metrics. Aspects like timing or energy consumption are analysed based on annotations of events that occur in the system. As these annotations are fixed, variations of various aspects of the model caused by input jitter, system latencies or operating system overhead cannot be taken into account.

To overcome the above restrictions, we propose a modelling approach that enables construction of realistic models applicable to a large class of systems and which is based on the concepts of the formally defined modelling language POOSL [78]. Both periodic and event-driven tasks can be modelled such that their execution times may depend on the input data according to a given probability distribution, which can be selected from normal, uniform, discrete, exponential and Bernoulli distributions. Moreover, any scheduling policy, from both preemptive and non-preemptive categories, can be chosen for scheduling tasks onto the resources of the target multiprocessor architecture. The variations in the execution times of tasks caused by cache or pipeline effects, or operating system overhead, can also be taken into account. As the application area for this type of modelling includes embedded systems that interact with physical devices, the environment is modelled as an approximation of its continuous time behaviour. Due to the semantics of the language, analytical computation of the properties of a real-time system is *possible* [90]. However, the type of models that we shall present, though compact, incorporate stochasticity. Next to the traditional state space explosion problem, this is an additional reason for which the exhaustive analysis is sacrificed in favour of simulation-based estimations. Nevertheless, the accuracy of the obtained results can be determined.

3.3 The Library of Modelling Patterns

One of the approaches for modelling a real-time system in order to analyse its performance and to explore its design space is the Y-chart scheme, introduced in [57]. This scheme makes a distinction between applications (the required functional behaviour) and platforms (the infrastructure used to perform this functional behaviour). Moreover, as real-time systems are typically reactive systems, meaning that there is a continuous interaction with the outside world, in [37] we added the model of the environment to the Y-chart scheme, as depicted in figure 3.2. The design space can be explored by evaluating different mappings of application onto platform.

To reduce the amount of time needed to construct models for design space ex-

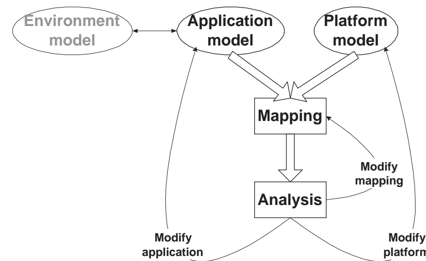


Figure 3.2: Y-chart scheme

ploration of real-time systems, we propose a set of *modelling patterns* that are highly abstract models of components commonly encountered in such systems. Such components are: tasks, resources and input/output devices. Each component is characterised by a set of parameters. These patterns enable the specification of a system by indicating what its components and the values of their parameters are, independent of a target modelling language. Moreover, the patterns allow automatic generation of a model into a specific language provided an implementation of them in that language, enabling the analysis of the system to benefit from the strengths of the techniques associated to that particular language. The modelling patterns act like templates that can be applied in many different situations by setting the appropriate values of their parameters. These modelling patterns emerged from previous experience with modelling of real-time systems (see [37], [32], [96]). The library of patterns contains templates for different types of tasks, resources, schedulers and input and output devices, which are presented in table 3.1 and reflect the modelling approach assumed by the classical scheduling analysis [20]. The table shows the Y-chart component to which each of these patterns belongs, the name of the pattern and its parameters. A description of each of the modelling patterns and their parameters is given in section 3.5 together with their implementation in POOSL.

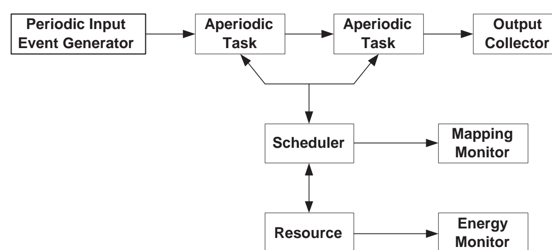


Figure 3.3: Example of system model built from patterns and their interconnections

As an example of how the patterns are used for modelling a real-time system, a simplified version of a radio system in which a user turns a knob to change the audio volume is considered. Figure 3.3 shows the modelling patterns required to model the system and their interconnections. The application part is made of two tasks that are supposed to handle the events coming from the environment and to ensure the changing in the volume. The tasks are mapped onto a resource and they are dispatched on it by a scheduler. The patterns used for modelling the system are:

System	Pattern Name	Parameters Names	and Explanations
Application Model	Periodic Task	T	period
		D	deadline
		Offset	initial offset
		BCLoad	best-case processor load
		WCLoad	worst-case processor load
		LoadDistrib	load distribution
		Iterations	number of execution iterations
		Priority	task priority
		Latency	activation latency
		OutEvent	output to another task
	Aperiodic Task	Trigger	triggering event
		D	deadline
		BCLoad	best-case processor load
		WCLoad	worst-case processor load
		LoadDistrib	load distribution
		Priority	task priority
		Latency	activation latency
		OutEvent	output to another task
Platform Model	Computation/Communication Resource	Throughput	bits/intructions per time unit
		InitialLatency	latency induced by platform effects
		FixedLatency	if the latency has a fibed value or not
		IdlePower	idle power consumption
		NominalPower	nominal power consumption
		Monitored	the energy is monitored or not
	Scheduler	Policy	scheduling policy
		Monitored	the scheduling is monitored or not
	Mapping Monitor	Accuracy	analysis desired accuracy
	Energy Monitor	-	
Environment Model	Periodic Input Event Generator	Type	event type
		Size	bit size of the generated information
		T	generation period
		Offset	initial offset
		Jitter	maximum jitter
	Sporadic Input Event Generator	Type	event type
		Size	bit size of the generated information
		Stream	generation distribution
		LowT	lower-bound of the distribution
		UpT	upper-bound of the distribution
	Output Collector	Type	accepted event type
		Accuracy	analysis desired accuracy

Table 3.1: Modelling patterns

- a periodic input event generator that models the knob;
- two communicating aperiodic tasks, one triggered by the events from the environment and the second one sending the output to the environment;
- an output collector that models the speaker from which the user can notice the changes in the audio volume;
- a scheduler to dispatch the ready task onto the resource;
- a mapping monitor to analyse the schedulability of the system;
- a resource on which the application runs;
- an energy monitor to analyse the energy consumption.

The arrows in the picture indicate the interconnections of the components in the model in order to analyse various properties such as schedulability, deadline miss ratio, or energy consumption.

To enable an easy specification of a model in terms of the patterns needed, independent of a target modelling language, we have developed the Pattern-based system Description Language (PDL). This language is based on the Extensible Markup

```
<system>
    <environment>
        <PeriodicInputEvent Name="KNOB” Type="KNOB” Size=”1” T=”1/32" Offset="0" Jitter=”0">
        <OutputCollector Name="SPEAKER” Type="EVENT” Accuracy=”0.95">
    </environment>
    <application>
        <AperiodicTask Name="TASK1" Trigger=”KNOB” D="6" BCLoad=”90" WCLoad=”110"
            LoadDistrib=”Uniform” Priority=”1" Latency=”0.1" OutEvent=”EVENT”>
        <AperiodicTask Name="TASK2" Trigger=”EVENT” D="6" BCLoad="90" WCLoad=”120"
            LoadDistrib=”Uniform” Priority=”2" Latency=”0.2" OutEvent=”EVENT”>
    </application>
    <platform>
        <Resource Name="CPU" InitialLatency="0.1" FixedLatency= “false” Throughput="1000"
                IdlePower=”0"  NominalPower=”0.0069” Monitored=”true”>
            <Scheduler Policy=”EDF" Monitored=”true”>
            <MappingMonitor Accuracy=”0.95">
        </Resource>
    </platform>
    <mapping>
        <map TaskName="TASK1" ResourceName="CPU">
        <map TaskName="TASK2" ResourceName="CPU">
    </mapping>
</system>
```

Figure 3.4: PDL specification of the simplified radio system

Language (XML) [4] notation to indicate which are the necessary patterns for modelling the system and to set their parameters. The structure and the semantics of the XML file is based on the Y-chart scheme parts, environment, application, platform and mapping as it can be seen in figure 3.4 that shows the PDL model of the simplified version of the radio. The semantics of the language is explained in more detail in section 3.6.

Besides the fact that the modelling patterns encapsulate experience with modelling of real-time systems that can be used in future products development, they also offer the access of designers to modelling languages and analysis tools and techniques with which they do not have much experience. The modelling patterns can be implemented in different modelling languages and provided an automatic model generation tool, a designer can benefit from the expressiveness of that language and the strengths of its analysis techniques in the design of the product.

3.4 UML Profile for POOSL Modelling Language

The Parallel Object-Oriented Specification Language (POOSL) lies at the core of the Software/Hardware Engineering (SHE) system-level design methodology. Introduced in [97] as an object-oriented extension of the Calculus of Communicating Systems (CCS) [68], POOSL has been later extended with real-time in [41] and probabilities in [93] to become a very expressive formal modelling language. The language contains a set of powerful primitives to describe concurrency, probabilistic distributions, synchronous communication, timing and functional features of a system into a single executable model. Its formal semantics is based on timed labelled transition systems [83]. This mathematical structure guarantees a unique and unambiguous

interpretation of POOSL models. Hence, POOSL is suitable for specification and, subsequently, verification of correctness and evaluation of performance for real-time systems.

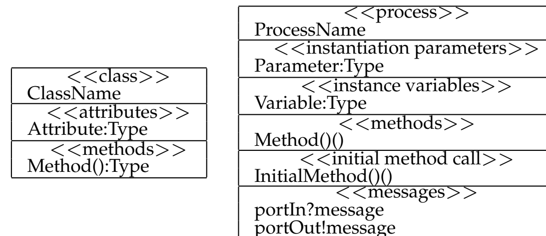


Figure 3.5: UML class vs. POOSL process class diagram

POOSL consists of a *process* part and a *data* part. The process part is used to specify the behaviour of active components in the system, called processes. The data part is based on traditional concepts of sequential object-oriented programming and it is used to specify the information that is generated, exchanged, interpreted or modified by the active components. With the acknowledgment of UML [74] as the standard modelling language, a UML profile for SHE has been presented in [90] that slightly adapts the UML notations for exploitation in the context of the SHE methodology. As an example, figure 3.5 presents the relation between the UML class and the POOSL process class symbols. The name compartment of the class symbol for a POOSL process class is stereotyped with «process» instead of «class» like in UML. The «attributes» of a process class include «instantiation parameters» and «instance variables». The former allow parameterising the behaviour of a process at instantiation, whereas the latter represent the process internally needed variables. The behaviour of each process is described by «methods». The start behaviour of a process is specified by its «initial method call». The last compartment of the process class symbol expresses the fact that in hardware/software systems the exchange of information is not always directly related to invoking certain behaviour. «messages» can be sent, using **!**, and received, using **?**, by different communicating processes in the system.

3.5 POOSL Modelling Patterns Library

The description of each of the modelling patterns presented in table 3.1 is given in the rest of this section together with its POOSL implementation. The section is organised based on the Y-chart scheme parts: the application part model is presented in subsection 3.5.1, the platform part model is discussed in subsection 3.5.2, the environment model is shown in subsection 3.5.3 and the model of the mapping in subsection 3.5.4.

3.5.1 Application Model

The application part of the Y-chart models the functional behaviour of a real-time embedded system which is implemented through a number of tasks that may com-

municate with each other. A task is a piece of code that can be executed many times with different input data. Task activations can be periodic (time-driven), occurring at regular intervals equal to the task period, or aperiodic (event-driven), waiting for the occurrence of a certain event. There are three types of uncertainties that may affect a task: the activation latency, the release jitter and the output jitter. These types of uncertainties are depicted in figure 3.6 with respect to a reference time which represents the time shown by a perfect clock. We define and explain the causes of each uncertainty:

- The *activation latency* represents the latency between the moment when the task should be ready for execution and the moment the task actually becomes ready for execution. This latency is caused, for example, by the inaccuracies of the processor clock that might drift from the reference time because of temperature variations. For event-driven tasks, the performance of the runtime system, which cannot continuously monitor the environment for events, influences the moment when the task becomes ready.
- The *release jitter* is the latency between the moment when the task is ready for execution and the moment when the scheduler dispatches it for execution. This latency is caused by the operating system overhead and the interference of other tasks that, depending on the scheduling mechanism, may impede the newly activated task to start immediately its execution.
- The *output jitter* is the variation between the earliest and the latest moment when a task finishes its execution. The output jitter is caused by the cumulated interference of other tasks in the system, the scheduling mechanism that may allow preemption of the executing task, the variation of the activation latency and even of the execution time of the task itself, which may depend on the input data.

In classical real-time scheduling theory [20], the release jitter and, to some extent, the output jitter[1] can be computed, but the activation latency is usually ignored, assuming a perfect platform. However, there are systems, like for example those that realise the low-level control of mechanical devices, for which the effect of this latency might be significant because their timing requirements are in the same range of values as the activation latency. Therefore, the task modelling patterns that we present in this chapter overcome this problem and take it into account.

Besides time uncertainties, figure 3.6 shows also an important parameter for real-time tasks: the deadline. The deadline represents the moment in time before which the task is supposed to finish its execution. For critical systems like power plants or aircrafts control, missing a deadline might lead to a catastrophe, whereas for an audio/video encoder/decoder this is not the case and a certain ratio of misses is usually allowed.

The model of a task incorporates its activation, whether it is periodical or event-driven, the latencies that might occur, the communication with other tasks in the system and the deadline that needs to be met. The details of the computations that

[1] In classical scheduling theory, the output jitter can be computed without taking into account possible variations of the execution time of a task nor its dependencies on the input data.

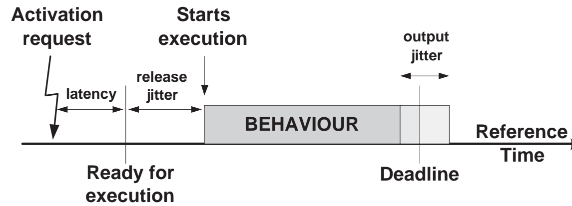


Figure 3.6: Real-time task parameters

are performed by the task are abstracted from and only the load, which is the number of instructions that need to be performed in the computation, is specified. The two task patterns that we have conceived are visualised using the POOSL process class diagrams in figure 3.7. The parameters, as specified in table 3.1, can be found in the «`instantiation parameters`» compartment of each pattern.

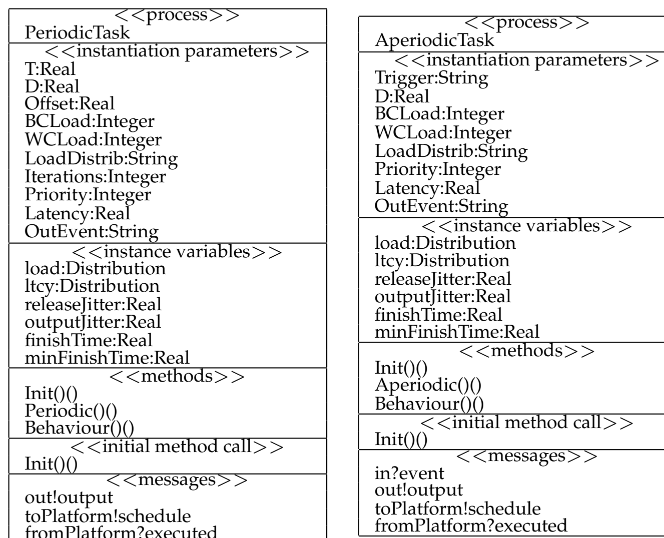


Figure 3.7: Application modelling patterns

Periodic Task. The periodic task pattern is to be used whenever an independent periodic task is required. Its parameters are the period `T`, the relative deadline `D`, the initial `Offset` in case the task does not start at time zero, the processor load, which represents a certain distribution, selected with `LoadDistrib`, between a best-case (`BCLoad`) and a worst-case (`WCLoad`) value of the number of instructions the task imposes on a target processor and which can be obtained based on previous experience, the number of `Iterations` for the case the task is not infinitely running, the `Priority`, the activation `Latency`, and potentially an `OutEvent` in case the result of the computations of the current task has to be sent to another task in the system.

Aperiodic Task. The aperiodic task pattern should be applied for the specification of a task triggered by an event from the environment or by a message from another task in the system. Its parameters are the `Trigger` of the task, the relative deadline `D`, the `LoadDistrib` between `BCLoad` and `WCLoad`, the `Priority`, the activation `Latency`, and potentially an `OutEvent` in case the result of the computations of the current task has to be sent to another task in the system.

```
Init()()
  load := new (Distribution) ofType(LoadDistrib) withParameters(BCLoad, WCLoad);
  ltcy := new (Distribution) ofType("Uniform") withParameters(0, Latency);
  outputJitter := 0;
  minFinishTime := 0;
  delay Offset;
  Periodic()().
```

Figure 3.8: POOSL specification of the `Init` method for the periodic task

Each of these two patterns has three methods. The method that is called to initialise the instance variables and to start the behaviour of a task is `Init`. The POOSL specification of the `Init` method for the periodic task pattern is given in figure 3.8. The data class object `load` is an instantiation of the data class representing the type of distribution specified by `LoadDistrib`. A uniform distribution is desired as a model of the activation latency because any value in the chosen interval has the same probability of occurrence. Hence, the data object `ltcy` is instantiated from the class representing a uniform distribution in $[0, \texttt{Latency}]$. In the ideal situation, the activation `Latency` is zero and no latency would be introduced in the activation of the task. If $\texttt{Latency} > 0$, a value in $[0, \texttt{Latency}]$ is generated based on the uniform distribution and the actual activation moment drifts from the reference time with $\pm\texttt{Latency}$. Hence, for a periodic task, the actual activation of the task happens somewhere in the interval $[0, \texttt{Latency}] \cup [n * \texttt{T} - \texttt{Latency}, n * \texttt{T} + \texttt{Latency}]$, $\texttt{Latency} < \texttt{T}$, $n \in \mathbb{N}^+$. The other instance variables, which are meant for the monitoring of the task, are set to zero.

Besides the method that is initially called, two more methods are specified. One is called `Periodic`, respectively `Aperiodic`, and contains the specification of the task according to the type of triggering it has, such as waiting for the next period, respectively for the next incoming event, to be activated. The other method, `Behaviour`, contains the specification of the actual computation the task needs to perform. In the templates provided with our library, the specification of the actual behaviour of a task is empty for two reasons. The first one is that it depends on the application what a task is supposed to compute, hence the designer who is using this library has to supply the right specification if needed. The second reason is that for the type of analysis we are interested in at a high-level of abstraction, which will be discussed in chapter 4, the actual computation performed by a task is not important and it can be left out.

In the POOSL implementation of the patterns library that we present in this chapter, the desired timing behaviour of a task can be completely decoupled from its actual timing behaviour. For a periodic task (see figure 3.9.a), the **par-and-rap** POOSL statement indicating parallel composition in `Periodic`, is used to decouple the task period from its real activation moment. The **par** branch is used to execute the actual `Behaviour`, possibly with activation latency, while the **and** branch is used to determine the next period by delaying exactly `T` units of time and then recursively calling

```
Periodic()() |lat : Real|
  if (Iterations != 0) then
   par
     lat := 2*ltcy sample();
     delay T-Latency+lat;
     Behaviour(D+Latency, lat)()
   and
     delay T;
     if (Iterations != -1) then
       Iterations := Iterations-1 fi;
     Periodic()()
   rap
  fi.
```

a) Periodic task

```
Aperiodic()() |lat : Real, ev : Event|
  in?event(ev |
          ev getEventType() = Trigger);
  par
    par
      lat := ltcy sample()
      delay lat;
      Behaviour(D, lat, ev)(ev)
    and
      delay D
     rap;
    if OutEvent != "" then
       out!output(ev) fi
  and
    Aperiodic()()
  rap.
```

b) Aperiodic task

Figure 3.9: POOSL specification of the application modelling patterns

itself. The actual deadline of the current activation of the task is $\texttt{D} + \texttt{Latency} - \texttt{lat}$ and is given split as two parameters to the `Behaviour` method call. Furthermore, the execution of the periodic task is modelled to be finite (if $\texttt{Iterations} > 0$) or infinite (if $\texttt{Iterations} = -1$).

The event-driven tasks are activated at the arrival of a message of type `Trigger` on the port `in` (see figure 3.9.b). For this reason, there is no need to express a certain number of iterations if the execution of an aperiodic task is not infinite, as it is blocked/stopped anyway waiting for an event to occur. Task activation latency is possible to occur because it might take a while before the system acknowledges the occurrence of an event in the environment and wakes up the corresponding task. Usually, an aperiodic task is required to output the result of its computation (`out!output`) before some deadline `D`. If `Behaviour` does not finish by that time, the output is postponed, causing output jitter. During simulation, the designer is informed about such situations and the output jitter for each type of task can be calculated.

```
Behaviour(deadline:Real, latency:Real)() |req : Request, tstart, tstop : Real|
  tstart := currentTime;
  req := new Request();
  req setReleaseTime(tstart);
  req setReleaseJitter(-1);
  req setDeadline(deadline-latency);
  req setLoad(load sample());
  req setPriority(Priority);
  toPlatform!schedule(req);
  fromPlatform?executed();
  if OutEvent != "" then out!output(new Event() setType(OutEvent)) fi;
  tstop := currentTime;
  releaseJitter := req getReleaseJitter();
  finishTime := tstop - tstart;
  if (minFinishTime > finishTime) then minFinishTime := finishTime fi;
  outputJitter := finishTime - minFinishTime.
```

Figure 3.10: POOSL specification of the `Behaviour` method of the periodic task

As mentioned above, for analysis purposes, the specification of the `Behaviour` of a task does not need to contain the actual computations the task needs to execute, but mainly to enable analysis of the instruction load imposed on the platform and the schedulability of the task. Therefore, the implementation of the `Behaviour` method for the periodic task may look like the one shown in figure 3.10. A timestamp is taken when the task is ready to start its execution and a `Request` data object is built encapsulating the characteristics of the task: release time, deadline, load, which is taken as a *`sample`* from its probability distribution, and priority. The release jitter is currently set to -1 to indicate that the task is not yet dispatched to a resource. To

model the load distribution, the POOSL library offers uniform, normal, exponential, discrete uniform and Bernoulli distributions [90]. In a model of a hard real-time system, the values of both `BCLoad` and `WCLoad` are equal to the worst-case number of instructions the task may have. The `Request` data class is discussed in more detail in subsection 3.5.2. The platform is informed of the need to execute the newly created request and the notification of finished execution is waited for. If an event should be produced at the end of the execution in order, for example, to trigger another task in the system, a message is sent on the `out` port containing a data object `Event` which will be discussed in subsection 3.5.3. Once the execution is finished, the output jitter is calculated, based on the initialisation of `minFinishTime` to zero in the `Init` method, and it can be used for analysis.

3.5.2 Platform Model

The platform on which tasks run can be described as a collection of computation and communication resources that are able to provide the capacity to perform the desired functional behaviour. The modelling patterns we have conceived for describing the platform part of the Y-chart model of a system provide a unified way of specifying communication and computation resources by exploiting their common characteristics. This modelling approach is possible at a high-level of abstraction as there is no large conceptual difference between a processor and a bus: they both receive requests, execute them (either by transferring the bits of a message or executing the instructions of an algorithm) and send back a notification on completion. As a resource is typically shared, a scheduler is needed to arbitrate the access to a resource.

Figure 3.11 visualises the POOSL process class diagrams of the platform modelling patterns: the `Scheduler`, the `Resource`, the `MappingMonitor` and the `EnergyMonitor`.

Scheduler. The `Scheduler` modelling pattern is to be used for the modelling of a simple operating system that ensures the scheduling of the tasks on a resource according to a certain scheduling policy. The scheduler has two parameters, one that represents the name of the scheduling policy desired to be used on a certain resource, and the other one that selects if the scheduling is monitored or not. The scheduler also has two methods, `Init`, to instantiate the scheduling policy based on the `Policy` name, and `Schedule`, to receive and dispatch scheduling requests.

Resource. The `Resource` modelling pattern is to be used when a computation (RISC-like processor [47]) or communication (bus) component needs to be modelled in the system. A resource is characterised by a `Throughput`, which is the number of instructions a processor can execute per time unit or the transfer bit rate on a bus, an `InitialLatency`, which models the operating system overhead and hardware induced delays due to cache misses and/or pipeline stalls or the communication protocol overhead, a boolean parameter called `FixedLatency` to select a fix or variable initial latency, a `NominalPower` and an `IdlePower` consumption, and a boolean parameter called `Monitored` to select if the energy consumption of the resource has to be monitored or not. At the initialisation of the system, method `Init` is called for each resource to instantiate the `overhead` data object as a uniform distribution in

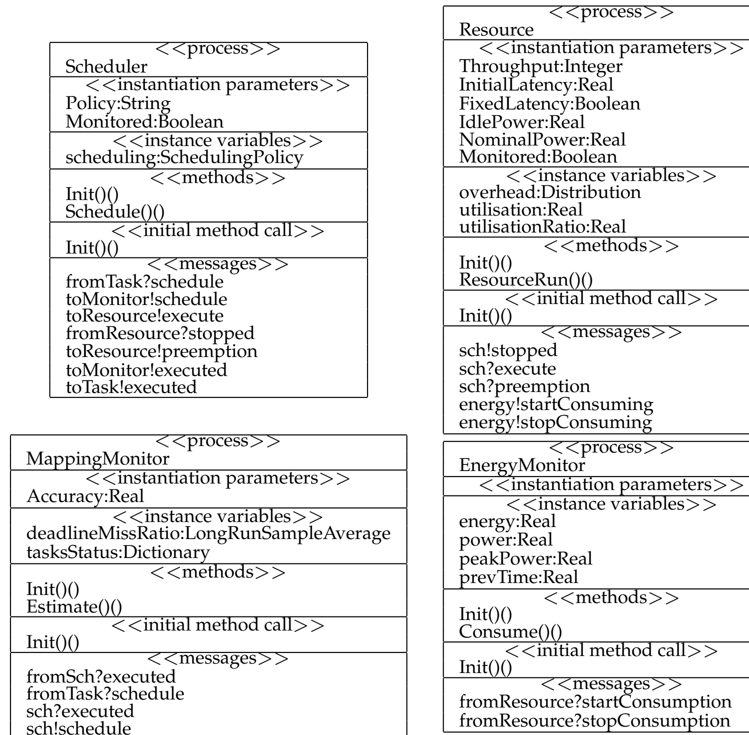


Figure 3.11: Platform modelling patterns

[0, `InitialLatency`] if `FixedLatency` is *false*, to set the utilisation to zero and to send the value of the `IdlePower` to an energy monitor if the energy consumption is monitored. This means that as soon as the system starts its execution, even though there might be resources not used yet, they consume a certain amount of energy already.

Mapping Monitor. The `MappingMonitor` modelling pattern is to be used when one is interested in the analysis of how often deadlines are missed on a certain resource. The parameter of this pattern is the desired `Accuracy` in the estimation of the deadline miss ratio.

Energy Monitor. The `EnergyMonitor` modelling pattern is to be used when one is interested in monitoring the energy consumption in a certain configuration of the platform of the system. This monitor models a battery from which energy in drained. There is only one such monitor in a system and all the resources must be connected to it. The pattern has no parameter.

In the rest of this subsection, the POOSL implementation of each of the platform modelling patterns is presented in detail.

In order to schedule the requests on a shared resource, one of the available scheduling policies must be instantiated. The scheduling policies are implemented as POOSL data classes that derive the `SchedulingPolicy` abstract class as shown

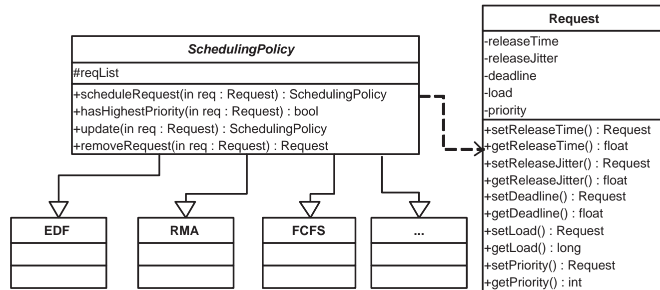


Figure 3.12: UML diagram for scheduling policies

```
scheduleRequest(req : Request): SchPolicy | i, j : Integer |
  i := 1;
  while(req getDeadline() > reqList get(i) getDeadline()) do
    i:=i+1
  od;
  reqList insert(i, req);
  return self.
```

Figure 3.13: EDF scheduling policy

in figure 3.12. The available policies include both preemptive (e.g earliest deadline first, rate monotonic) and non-preemptive (e.g. first come first served) scheduling algorithms known from classical scheduling theory [20]. The methods of each of these data classes require as parameter a data object of type `Request`, containing the information needed for scheduling: release time, deadline, and priority. Additionally, a `Request` object has two more parameters: the load of the request (number of instructions of a task / length of a message) and the release jitter that is computed at the time the request is dispatched to a resource. Such an object is created by the `Behaviour` method of each task and sent to the scheduler at each task activation.

The method that ensures the actual scheduling of a task is *`scheduleRequest(req)`*. Its specification for the earliest deadline first (EDF) policy is given in figure 3.13. The current request `req` is placed in the list `reqList` of already scheduled requests based on the value of its deadline.

The method `hasHighestPriority(req)` returns *true* or *false* if the current request has the highest priority or not. In case of a non-preemptive scheduling, the method *`hasHighestPriority`*`(req)` returns *false* if there is a task already being executed. Upon preemption of a request, the *`update(req)`* method updates the information about the `req` in the scheduler list. *`removeRequest(req)`* is called upon completion of a request to remove it from the scheduler list and returns the next request to be scheduled if there is one or *nil* otherwise.

Figure 3.14 shows the POOSL specification of the `Schedule` method of the scheduler pattern. The data object `scheduling` is an instance variable instantiated from one of the scheduling policies data classes according to the name `Policy` set as parameter of the pattern. The core of the scheduler pattern relies on the non-deterministic choice between receiving scheduling requests from newly activated tasks or messages to be sent (the outer **sel** branch) and notifications from the platform about completed requests (the **or** branch). The newly activated request is scheduled according to the chosen policy by calling the *`scheduleRequest(req)`*.

```
Schedule()() | req, oldreq :  Request |
  sel
    fromTask?schedule(req);
    scheduling scheduleRequest(req);
    if (scheduling hasHighestPriority(req) == true) then
        sel
          toResource!execute(req);
          if (req getReleaseJitter() == -1) then
            req setReleaseJitter(currentTime - req getReleaseTime()) fi
        or
          toResource!preemption;
          fromResource?stopped(oldreq);
          toResource!execute(req);
          if (req getReleaseJitter() == -1) then
            req setReleaseJitter(currentTime - req getReleaseTime()) fi;
          scheduling update(oldreq)
        les
    fi;
    Schedule()()
  or
    fromResource?stopped(oldreq);
    toTask!executed;
    if (Monitored == true) then toMonitor!executed(oldreq) fi;
    req := scheduling removeRequest(oldreq);
    if (req != nil) then toResource!execute(req) fi;
    Schedule()()
  les.
```

Figure 3.14: POOSL scheduling pattern method specification

If `req` has the current highest priority, it is sent to the resource for being immediately handled (the inner **sel** branch). As the resource might already be running another request, the corresponding **or** branch models the situation when the old request is preempted and rescheduled (*`update(oldreq)`*). As soon as a request is dispatched to a resource, its release jitter is computed if it just starts its execution. In the outer **or** branch, the scheduler receives completed requests from the resource and removes them from the ready list by calling *`removeRequest(oldreq)`*, which also returns the next scheduled request, if there is one.

Figure 3.15 presents the model of the `ResourceRun` method of the resource (which can be a RISC-like CPU or a bus) which receives execution requests from the scheduler. Immediately after receiving a new request, if the resource is supposed to be monitored for energy consumption (`Monitored` is *true*), it announces the energy monitor that it will start consuming more energy according to the difference between the `NominalPower` and the `IdlePower` given as parameters. Before the actual execution of the task, the resource has an initial latency that models, in case of a CPU, the context switch that proceeds the execution of a newly scheduled task for saving the status of the previous task and loading the current task, and in case of a bus, the time it takes for the first bit of the message to be transferred, which depends mostly on the communication protocol used. For an average case analysis of the system, a uniform distribution `overhead` between zero and `InitialLatency` is considered to model the operating system overhead and the possible delays induced by hardware effects. For a worst-case analysis of the system, the parameter `FixedLatency` needs to be set to *true* and the fixed value of the `InitialLatency` is used. It is important to note that the initial latency of a bus usually has a constant value, thus the value of `FixedLatency` parameter should be *true* to model a bus. After the initial delay, the resource lets the time pass according to the execution time associated to the request. The execution time is computed based on the load of the request (representing either the number of instructions of a task or the length of a message) and the `Throughput` of the resource, which is the second parameter of the pattern.

The core concept behind the presented modelling pattern for a resource is the

```
ResourceRun()() | req:  Request, loadLeft, tstart, tstop :  Integer |
  sch?execute(req);
  if (Monitored == true) then energy!startConsuming(NominalPower-IdlePower) fi;
  if (FixedLatency == true) then delay InitialLatency
  else delay overhead sample() fi;
  tstart := currentTime;
  abort
     delay req getLoad() / Throughput
  with sch?preemption;
  if (Monitored == true) then energy!stopConsuming(nominalPower-idlePower) fi;
  tstop := currentTime;
  utilisation := utilisation + (tstop - tstart);
  utilisationRatio := utilisation / currentTime;
  loadLeft := req getLoad() - (tstop - tstart) * Throughput;
  req setLoad(loadLeft);
  sch!stopped(req);
  ResourceRun()().
```

Figure 3.15: POOSL resource pattern method specification

possibility of the language to express the breaking of the execution, needed if the scheduling mechanism allows preemption. In POOSL, this can be modelled with the **abort** statement. Once a `preemption` message is received by the resource, the execution of the current request is stopped and its remaining execution time (actually the remaining load) is computed and updated (*`req setLoad(loadLeft)`*) and the request is sent back to the scheduler. Preemption is usually the case for computation resources, and less common for communication. Nevertheless, as preemptions and their associated latencies (like context switches) might have a large influence, for example on the finishing time and the output jitter of a task, they must be taken into account. As soon as the current request finishes the execution or it is preempted, a `stopConsuming` message is sent to the energy monitor if the energy needs to be monitored to announce the drop in energy consumption from `NominalPower` to `IdlePower`. Moreover, the `utilisation` of the resource, which is the total amount of time the resource was active, is updated, as well as the `utilisationRatio`, which represents the ratio of the active time of the resource from the total time the system was active.

The energy monitor pattern presented in figure 3.11 has no parameters. It models a battery that receives `startConsumption` and `stopConsumption` notifications from the resources in the system. This pattern has two methods. `Init` is the method called at the initialisation of the POOSL process `EnergyMonitor` to set to zero all the instance variables and call the `Consume` method. The POOSL implementation of the latter method is given in figure 3.16. The method `Consume` models the receiving of announcements from each of the communication and computation resources in the system of either starting or stoping using a certain amount of power. As soon as a `startConsuming` message is received, the value of the already consumed energy is updated based on the amount of time elapsed from the last change in the power drained and the value of this power. The level of power consumption is increased with the value requested by the calling resource and, if needed, the value of the `peakPower` is updated. When a `stopConsuming` message is received, the value of the consumed energy is updated as well as the value of the currently drained power.

The last pattern that belongs to the platform model in table 3.1 is the mapping monitor, visualised as a POOSL process class diagram in figure 3.11. To decide whether a certain mapping is suitable or not, a mapping monitor can track the missed deadlines and calculate the deadline miss ratio for each resource in the system. Based on such information, a bottleneck resource might be identified in order to

```
Consume()() | p :  Real |
  sel
    fromResource?startConsuming(p);
    energy := (currentTime - prevTime) * power + energy;
    power := power + p;
    if (power > peakPower) then peakPower := power fi;
    prevTime := currentTime
  or
    fromResource?stopConsuming(p);
    energy := (currentTime - prevTime) * power + energy;
    power := power - p;
    prevTime := currentTime
  les;
  Consume()().
```

Figure 3.16: POOSL energy monitor pattern method specification

```
Estimate()() |req :  Request, deadline :  Real|
  sch?executed(req);
  deadline := req getDeadline();
  if (deadline < currentTime) then
    deadlineMissRatio rewardRC(1, true);
  else
    deadlineMissRatio rewardRC(0, false)
  fi;
    deadlineMissRatio log()
  if (Accuracy != 0) then
    if (deadlineMissRatio accurate() == false)
    then Estimate()() fi
  fi.
```

Figure 3.17: POOSL mapping monitor pattern method specification

replace it or to change the mapping. The parameter of the mapping monitor pattern is the desired accuracy of the results. This parameter is useful when it is desired for the simulations to be run until the accuracy of the results is obtained. Moreover, the pattern has two methods: `Init` and `Estimate`. The method `Init` initialises a data object `deadlineMissRatio` for estimating the deadline miss ratio. This object is an instantiation of the `LongRunSampleAverage` data class that provides support for analysing performance metrics during simulations, as described in [90]. The method `Estimate`, depicted in figure 3.17, models that the mapping monitor receives notifications from the scheduler with the executed requests and checks for each of them if the deadline was met. To estimate the probability of missing a deadline for the system under analysis, the data method *`rewardRC`* is called with parameter 1 to register the occurrence of deadline miss events. Otherwise, the method *`rewardRC`* is called with parameter 0 if the deadline is met. If a certain accuracy of the estimations is desired, then the value of the `Accuracy` parameter differs from zero and the simulation of the model is run until the data method *`accurate()`* returns *true*, meaning that the estimation of the deadline miss ratio is accurate.

3.5.3 Environment Model

The model of the environment is composed of input event generators and output collectors, as shown in figure 3.18. The input event generators model the generation of environmental `Events` having an arrival pattern which can be chosen among periodic, periodic with jitter, or sporadic with a certain distribution of occurrence. These events trigger the activation of tasks in the application model and are collected by output collectors which model the output devices in the environment. The collector accepts events of a certain `Type` exiting the system and monitors their end-to-end delay. If a certain accuracy of the simulation-based analysis results is desired, its value should be set in the instantiation parameter `Accuracy`.

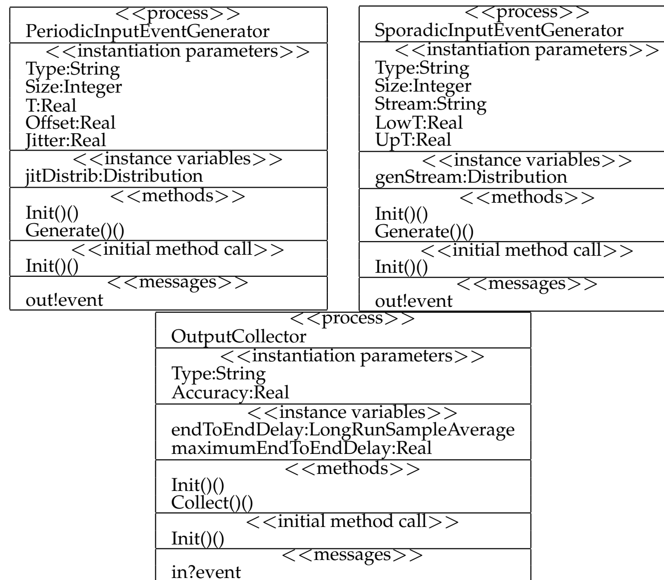


Figure 3.18: Environment modelling patterns

In the rest of this subsection, we will present first the data class `Event` and then the modelling patterns needed for modelling the environment and their POOSL specifications.

An `Event` is a data object generated by an input event generator and fed to the application part of the Y-chart model. As tasks triggered by events typically perform some computations and send the results as messages to other tasks, in our model we consider that an object `Event` travels in this way through the system collecting timestamps when it is generated, `releaseTime`, and when it is collected, `finishTime`, such that the end-to-end delay can be determined. Besides the timestamps, an `Event` also has a `type` based on which a task coupled to a number of input generators can choose the events it accepts. Moreover, an `Event` might have a certain `size` if it is supposed to carry information through the system.

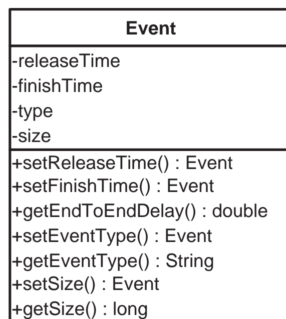


Figure 3.19: UML diagram for `Event` data class

For the modelling patterns to describe the environment of a system, we have conceived two types of input event generators, periodic (with and without jitter) and sporadic, and an output collector.

Periodic Input Event Generator. The `PeriodicInputEventGenerator` modelling pattern is to be used for modelling a device in the environment that periodically generates input events for the application part of the system. The periodic input generator resembles in some sense the periodic task. To obtain a perfectly periodic input stream, the value of `Jitter` must be set to zero. Otherwise, based on `jitDistrib`, which is an instance of a uniform distribution between zero and `Jitter`, a certain variation with maximum ±`Jitter` around each period `T` is generated. This models certain inaccuracies of the physical devices which may cause these variations. A new `Event` is created with the same `type` as the `Type` of the event generator and the release time timestamp. If the generation of events starts with a certain `Offset`, a value different than zero must be specified for this parameter.

Sporadic Input Event Generator. The `SporadicInputEventGenerator` modelling pattern is to be used for modelling a device that generates input events sporadically. In case of a sporadic input event generator, a minimum inter-arrival time between events is ensured by the fact that the `genStream` distribution has this value as its `LowT`, whereas the `UpT` can be set to infinity. By sampling the generation distribution, a minimum `LowT` units of times is ensured between two consecutively generated events.

The POOSL specification of how the events are generated by the `PeriodicInputEventGenerator` and by the `SporadicInputEventGenerator` can be seen in figure 3.20.

```
Generate()() |j :  Real|
  par
    j := 2*jitDistrib sample();
    delay T-Jitter+j;
    out!event(new Event()
        setEventType(Type)
        setReleaseTime(currentTime))
  and
    delay T;
    Generate()()
  rap.
```

a) Generate method for `PeriodicInputGenerator`

```
Generate()()
  while (true) do
    delay genStream sample();
    out!event(new Event()
        setEventType(Type)
        setReleaseTime(currentTime))
  od.
```

b) Generate method for `SporadicInputGenerator`

Figure 3.20: POOSL specification of the input generators

Output Collector. The `OutputCollector` models a device in the environment that collects messages from the application. The POOSL specification of the behaviour of the output collector is presented in figure 3.21. The `endToEndDelay` data object is an instance of the `LongRunSampleAverage` performance monitor data class and it evaluates the average end-to-end delay by collecting the value of the end-to-end delay of each event that arrives at the `OutputCollector`. This evaluation, performed using the data method *`rewardBM`*, uses the batch-means technique to estimate the average value (see [90]). The *`accurate`* method checks if the currently obtained accuracy of the estimation meets the desired `Accuracy` set as parameter of the pattern. Whenever an event of the desired `Type` is collected, its finishing time is timestamped and its calculated end-to-end delay contributes to the estimation of

the end-to-end delay in the system for that type of events. Moreover, the maximum value of this delay is also collected. The output collector finishes, and so does the simulation, when the desired accuracy of the results is obtained.

```
Collect()() |ev : Event|
  while((endToEndDelay accurate() == false) do
    in?event(ev | ev getEventType() = Type);
    ev setFinishTime(currentTime);
    NominalEndToEndDelay rewardBM(ev getEndToEndDelay());
    if (ev getEndToEndDelay() > maximumEndToEndDelay) then
      maximumEndToEndDelay := ev getEndToEndDelay()
    fi
  od;
  NominalEndToEndDelay log().
```

Figure 3.21: POOSL specification of the output collector

3.5.4 Mapping Model

The mapping stage of a Y-chart based model is realised in POOSL using communication channels that directly link POOSL processes that represent tasks to POOSL processes that model resources. According to the desired mapping in the system, each task is linked to a certain processor. As communicating tasks may be mapped onto different CPUs, the messages that are sent from one to the other must be transferred over a communication bus. In order to completely decouple the application from the architecture, in such a situation, a so-called communication task is inserted in the model and it is mapped onto the bus, as indicated by `TASKB` in the example from figure 3.22. This task behaves like an infinite buffer at the application level, getting the message from the sending task and making it available to the receiving task. The behaviour method of this type of task consists in communicating to the underlying resource, the bus, the size of the message and the deadline for its transmission, and waiting for the confirmation of this being done. In fact this is the same behaviour the software tasks are doing, only that they send to CPUs the number of instructions they need to execute. That is why a CPU and a bus can be modelled with the same pattern at the level of abstraction assumed in this work. To model the communication task, an aperiodic task is used. If this task is automatically inserted in the model, its deadline is set to zero in order to get high priority in case a priority-based scheduling is used for the bus. Adding this task to the application model does not change the model of computation. In a usual task-to-task communication on the same CPU, the sending task does not block if the receiving one is always able to receive the message. As an aperiodic task is modelled to be able to receive any message the moment it becomes available, this so-called communication task does not affect the sending task. On the other hand, a receiving task is able to retrieve the message from the buffer as soon as the transfer over the bus has finished, without any further stalling.

3.6 Model Generation Based on Patterns

To build a model of a real-time system in order to analyse it and to explore its design space, its components that correspond to the modelling patterns described in the previous section must be identified as well as their parameters. The use of the Pattern-

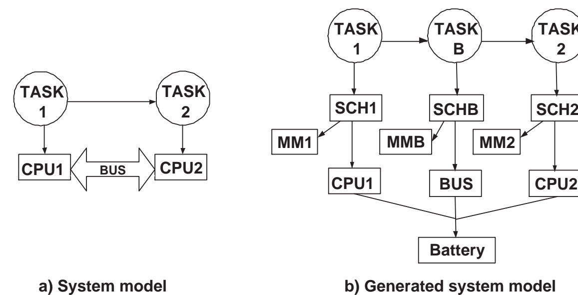


Figure 3.22: System model vs. generated model based on modelling patterns

based system Description Language (PDL) enables easy construction of models. The modelling of a system based on identifying the necessary patterns and setting their parameters is alleviated by the Extensible Markup Language (XML) [4]-based syntax of PDL. The structure and the semantics of the XML file is defined based on the Y-chart scheme parts, application, platform, environment and mapping, and covers the names and the parameters of the patterns. Figure 3.23 shows an example of such a model for a simple producer-consumer-like system shown in figure 3.22.a.

The application part of the system is identified with the `<application>` and `</application>` tags and contains instantiations of the periodic and aperiodic tasks patterns. To instantiate a task, a tag named either `PeriodicTask` or `AperiodicTask` is specified. Within such a tag, all the parameters of that type of task are enumerated and a value is given to each of them.

The platform model is delimited by the `<platform>` and `</platform>` tags and a list of the resources needed falls in between using a tag named `Resource`. For each declared resource, its scheduler and, if required, its mapping monitor are specified. If the scheduler parameter `Monitored` is set to *false* and a mapping monitor is specified, or if the scheduler parameter `Monitored` is set to *true* but a mapping monitor is not specified, an error is signaled by the model generation tool.

The mapping of the application on the platform is specified between `<mapping>` and `</mapping>` tags. Each task, identified based on the name of the object that instantiates it, is mapped onto a resource, which is identified also based on its corresponding object name.

When two tasks that communicate with each other are mapped onto different resources, the communication task between them, which can be automatically generated by the tool if it is not specified in the application model, is mapped on the bus that links the two required resources, as figure 3.22.b shows. `TASKB` is the name of the object that instantiates the communicating task that acts like a buffer and handles the transmission of the message over the `BUS`.

Moreover, we can model the environment of a system in case there are tasks that are triggered by events outside the application model. The tags used to specify this part of the Y-chart model are `<environment>` and `</environment>`. Each generator of events in the environment is specified to generate events of a certain `Type`. The matching between an input generator and an aperiodic task triggered

```
<system>
    <application>
        <PeriodicTask Name="TASK1" T=”6" D="6" Offset=”0" BCLoad=”90"
            WCLoad=”110" LoadDistrib=”Uniform” Iterations="10" Priority=”2"
            Latency=”0.1" OutEvent=”EVENT”>
        <AperiodicTask Name="TASK2" Trigger=”EVENT” D="6" BCLoad="90"
            WCLoad=”110" LoadDistrib=”Uniform” Priority=”1" Latency=”0.2">
    </application>
    <platform>
        <Resource Name="CPU1" InitialLatency="0.1" FixedLatency=”false”
            Throughput="1000" IdlePower=”0" NominalPower=”0.0069”
            Monitored=”true”>
            <Scheduler Policy=”EDF" Monitored=”true”>
            <MappingMonitor Accuracy=”0.95">
        </Resource>
        <Resource Name="CPU2" InitialLatency="0.1" FixedLatency=”false”
            Throughput="1000" IdlePower=”0" NominalPower=”0.0045”
            Monitored=”true”>
            <Scheduler Policy="EDF" Monitored=”true”>
            <MappingMonitor Accuracy=”0.95">
        </Resource>
        <Resource Name="BUS" InitialLatency="0.1" FixedLatency=”true”
            Throughput="1000" IdlePower=”0" NominalPower=”0”
            Monitored=”true”>
            <Scheduler Policy="FCFS" Monitored=”true”>
            <MappingMonitor Accuracy=”0.95">
        </Resource>
    </platform>
    <mapping>
        <map TaskName="TASK1" ResourceName="CPU1">
        <map TaskName="TASK2" ResourceName="CPU2">
    </mapping>
</system>
```

Figure 3.23: Producer-consumer system model specification based on modelling patterns

by the events generated by it is based on this `Type` which is a string identifying the name of the event.

The steps made by the model generation tool in order to obtain a model in a desired target language, for which there exists an implementation of the modelling patterns, from the model specified in the Pattern-based system Description Language (PDL) are shown in figure 3.24. First, from the PDL model the tool identifies which of the patterns from the library are needed and instantiates the required objects. For the communication tasks which are not present in the PDL model, it is checked whether instantiations of their corresponding patterns are required and what their parameters would be. Once all the necessary objects are instantiated, the model in the desired target language is ready. In case the target language is POOSL, a model in the format needed for the fast simulation engine, Rotalumis, is obtained. Figure 3.25 gives the graphical representation of the generated POOSL model for the producer-consumer-like system presented in figure 3.22.

The analysis of the POOSL model may lead to changes in the application model, like splitting or combining of tasks, in the platform model, such as choosing faster

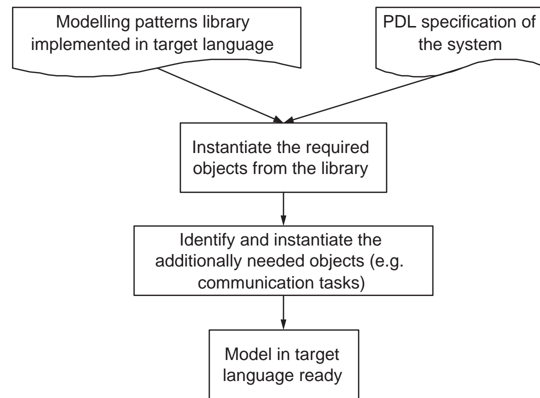


Figure 3.24: Flow for generation of a model from patterns

or slower resources, or even in the mapping, by choosing a different assignment of tasks onto resources. Any of these possible changes can be done easily at the level of the PDL model, by adjusting the values of the parameters of patterns, adding or removing model components, or changing the mapping. This activity takes only minutes and the POOSL model can be immediately generated for the new configuration of the system. In chapter 4, by means of two case studies, we present guidelines for how to explore the design space exploration of an embedded real-time system based on such a model.

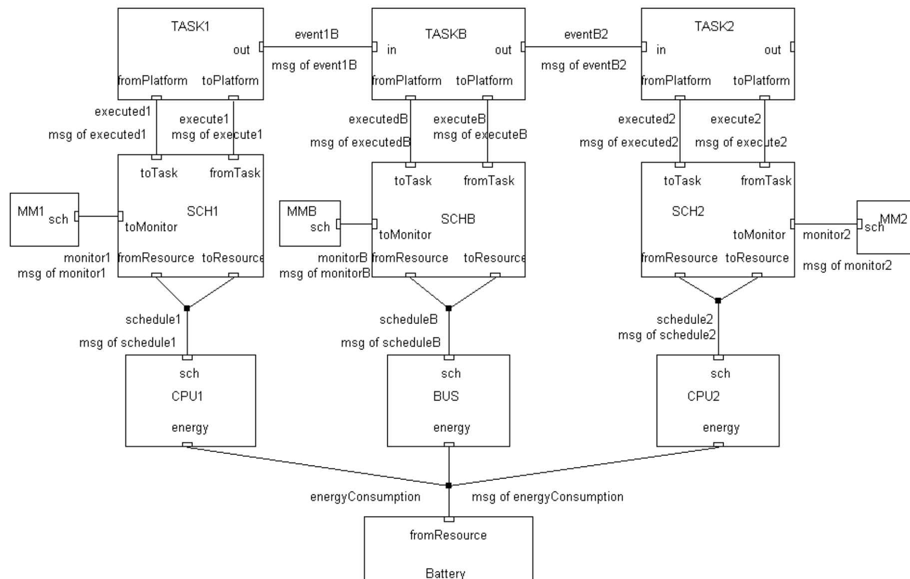


Figure 3.25: Generated POOSL model for the producer-consumer-like system

3.7 Summary

In this chapter, we have presented a set of modelling patterns suitable for modelling real-time systems. We have discussed each of these patterns and we have provided an implementation of them in the general-purpose formally defined Parallel Object-Oriented Specification Language (POOSL). The use of such patterns enables the automatic generation of models in different modelling languages. Hence, the patterns have the potential of offering the designers access to tools and techniques with which they are not familiar and enable them to use their strengths in modelling and analysing different properties of the system.

The patterns that we presented in this chapter were designed for soft real-time systems, as they capture many of the variations that appear in the behaviour of a system, from the load of a task to the varying overhead induced by the operating system and the hardware effects. However, these patterns can be used for the modelling of hard real-time systems as well, enabling worst-case analysis. This is possible by using the same values for both best-case and worst-case load of a task, and fixed, worst-case values for the initial latency of the resources. An example of such a model is given in chapter 4. An important observation we need to make is that the kind of modelling that we present in this thesis assumes that the tasks are implemented in software and we do not deal with hardware-mapped tasks.

Regarding the completeness of the modelling patterns library, as the case studies in chapter 4 show, these patterns can be used for applications in the areas of control-dominated systems as well as in control with data-intensive computations applications. The patterns show to be general enough since we were able to use the same patterns for case studies in different application domain areas. Extensions to the library are being worked on [92] to make the patterns also applicable to multimedia applications, for which a scenario-based analysis may be required [42], in which tasks may have multiple inputs and/or outputs, or they might be deployed on more complex platforms (e.g. using VLIW processors [31] and network-on-chip [85]). Moreover, a modelling pattern for a CPU separated from the one for a bus could be developed in order to provide designers with a model that captures in more detail the characteristics and the behaviour of a processor (e.g. pipelining or multiple instruction issue).

4

Analysis Approach for Dimensioning of Real-Time Systems

In the previous chapter, we have presented a modelling approach, based on modelling patterns, that can be applied for both hard and soft real-time systems in the areas of control-dominated and control with data-intensive computations applications. We have also presented an implementation of the patterns in POOSL that serves as input for the automatic model generation tool. In this chapter, we discuss the analysis approach for a suitable dimensioning of a system based on the generated POOSL model. Figure 4.1 depicts the steps of the modelling and analysis approach that enable design space exploration of a system. The first part of the diagram, namely the generation of the model is the same as the one depicted in figure 3.1. The diagram is extended with activities regarding the analysis of the model and decisions with respect to changes in the model of the system, such as values of parameters (e.g. for increasing or decreasing the performance of a resource), different mapping of tasks onto resources, or different scheduling policies, for exploration of the design space.

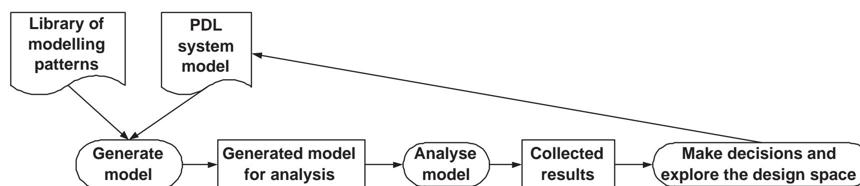


Figure 4.1: Pattern-based model generation and analysis flow

The chapter is structured as follows. Section 4.1 describes the analysis approach and the properties that can be analysed on a POOSL model built as presented in chapter 3 and discusses how design space exploration can be realised. Two case

studies are presented in this chapter. Section 4.2 presents the analysis and dimensioning of a soft real-time system, whereas section 4.3 presents the analysis of a hard real-time system. A summary of the chapter is given in section 4.4.

4.1 Model Analysis

In this section, we present the system properties that can be analysed from a POOSL model built using on the modelling patterns shown in chapter 3. By using a `MappingMonitor`, presented in section 3.5.2, for the mapping of tasks on each of the resources, CPUs and buses, in the system, the property that the application is schedulable on the chosen platform can be checked. The deadlines missed are monitored and the deadline miss ratio is estimated. A system is fully schedulable if no deadline is missed. If deadlines are missed, at the end of the simulation, the deadline miss ratio is checked against the requirements. If the requirements are fulfilled, the application-to-platform mapping is acceptable.

For control-dominated systems, such as systems that control the behaviour of physical devices in the environment, requirements might be set with respect to release and output jitter of task. For this, the `Behaviour` method of each task monitors these parameters, in the way presented in figure 3.10, and they can be checked against the requirements.

For soft real-time systems with streams of messages flowing through the application, an important property to analyse is the end-to-end delay of a message that traverses the system. Usually, the end-to-end delay is given as a requirement, and the design of the system must ensure its satisfaction. For this, the `OutputCollector`, presented in subsection 3.5.3, is used. It contains a `LongRunSampleAverage` performance monitor to estimate the average value of the end-to-end delay of the system. Moreover, it also keeps the maximum value of the end-to-end delay encountered during simulation. If the end-to-end delay requirements are met, then the given configuration is a good candidate for the realisation of the system.

Although it is desired to run simulations of models until the accuracy of the estimation of each desired property is achieved, sometimes it might take a very long time to obtain that. Since a number of parameters are monitored, like the deadline miss ratio for each resource and the end-to-end delay, it might be that some of them are estimated with their desired accuracy, but the others not. As an example, a deadline miss might be a rare event and a very long simulation needs to be performed to estimate this parameter with its desired accuracy. A solution to save simulation time is the following. The simulation is run until the desired accuracy of the end-to-end delay of the system is obtained. All the values obtained for the end-to-end delay are plotted in a histogram showing for each value how many times it appeared during simulation. This histogram is fitted into a known distribution, most likely the normal distribution because of the central limit theorem [3], as shown in the case study in section 4.2. Based on the mathematical rules corresponding to this distribution, it can be determined what the likeliness of deadline misses is.

Applying known types of distribution curves over the obtained simulation results provides an abstract model of the timing behaviour of the software part of a

complex system. It can be further used as input for multi-disciplinary models that are aiming at trade-offs across the disciplines involved in complex machines, like mechanical, electrical, optical and software engineering.

Besides schedulability and end-to-end delay, other important properties of a system are the resource utilisation, which is monitored by the `ResourceRun` method of each computation and communication resource, and the energy consumption, which is monitored by the `Consume` method of the energy monitor pattern and which might be given as a requirement of the system. Even if the energy consumption is not a requirement but an objective, if all the task deadlines are met and the end-to-end delay of the system is within the requirements but the resources are used very little, it is likely that a platform of less performance might still meet the functional and non-functional requirements while reducing the cost, usually a critical aspect of a system. However, the cost depends on both the price of the resource components of the system as well as on their induced energy consumption. Hence, it might be that cheap resources are used, but which consume considerable amounts of energy, whereas somewhat more expensive components might compensate by consuming less energy. Hence, a trade-off between cost and energy consumption is required among the configurations analysed to ensure the system meets all the other non-functional and functional requirements. The steps for analysing and dimensioning of soft real-time systems are summarised in figure 4.2.

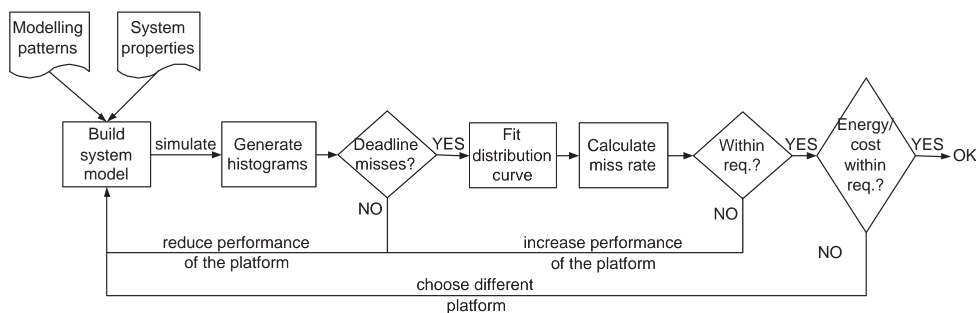


Figure 4.2: Flow of the steps in the analysis approach for soft real-time systems

In this chapter, we present the analysis approach by means of two case studies. Extensions of the technique to ensure automatic design space exploration are also possible and they are left as future research. Work in this direction has already been done at the University of Limerick, Ireland, [72] in the area of design space exploration for network processors. The authors incorporate in their analysis multi-objective evolutionary algorithms in order to automatically explore the design space. Such techniques can also be incorporated in our approach to automatically generate and analyse the design space for soft real-time systems.

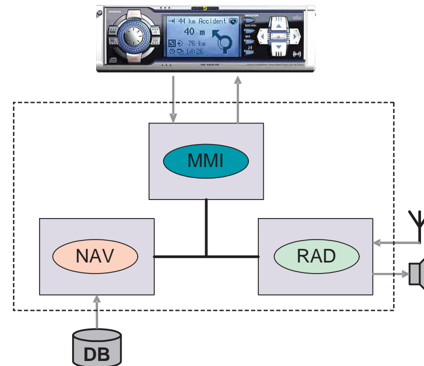


Figure 4.3: Distributed in-car radio navigation system

4.2 Case Study: A Distributed In-Car Radio Navigation System

The first case study is inspired by a system architecture definition study for a distributed in-car radio navigation system. Such a system usually executes a number of concurrent applications that share a common platform. Each of the applications typically has a number of individual performance requirements that need to be met by the platform. During the system definition phase, several candidate platform architectures might be proposed and need to be evaluated. Typical questions to answer are: (i) does the platform meet the performance requirements of all the applications; (ii) how robust is the platform with respect to changes in the application and/or input data parameters; (iii) is there a platform with cheaper and less powerful components to save cost but still meet the performance criteria of all applications?

The purpose of this case study is to show how our proposed analysis approach can be applied in practice, the type of results that can be obtained and what their accuracy is, as well as the suitability of our technique for soft real-time systems. The in-car radio navigation system has been previously modelled and analysed in [102] using Modular Performance Analysis, and in [46] using timed automata. As both these techniques are suited for hard real-time systems, the analysis results for dimensioning the system were over-conservative, as it was discussed in [32]. Since this system is not a safety critical system, but it allows 5% deadline miss ratio, it was very interesting for us to see the results that the average case analysis of the system can reveal.

We first describe in subsection 4.2.1 the system components and its model based on the modelling patterns presented in chapter 3. The analysis and the dimensioning of the platform on which the system should run are discussed in subsection 4.2.2.

4.2.1 The Model of the In-Car Radio Navigation System

An overview of the system is depicted in figure 4.3. It has three clusters of functionality: the man-machine interface (MMI) handles the interaction with the user; the navigation functionality (NAV) deals with route-planning and navigation guidance; the radio (RAD) is responsible for basic tuner and volume control, as well as receiving traffic information from the network. For this system, three possible application scenarios are identified: the change volume (VOL) scenario allows users to change the volume; the change address (ADDR) scenario enables route planning by looking up addresses in the maps stored in the database; in the handle network messages (TMC) scenario the system needs to handle the navigation messages received from the network. Each of these scenarios is described by a UML message sequence diagram, shown in figures 4.4, 4.5 and 4.6. These diagrams are annotated with information about the event rates, message sizes and task loads. The order of magnitude of the numbers shown in the diagrams is realistic. During the design, the system architect tries to improve their accuracy by using for example better estimation technique on details of the design, such as worst-case execution time analysis or by performing measurements on existing similar systems.

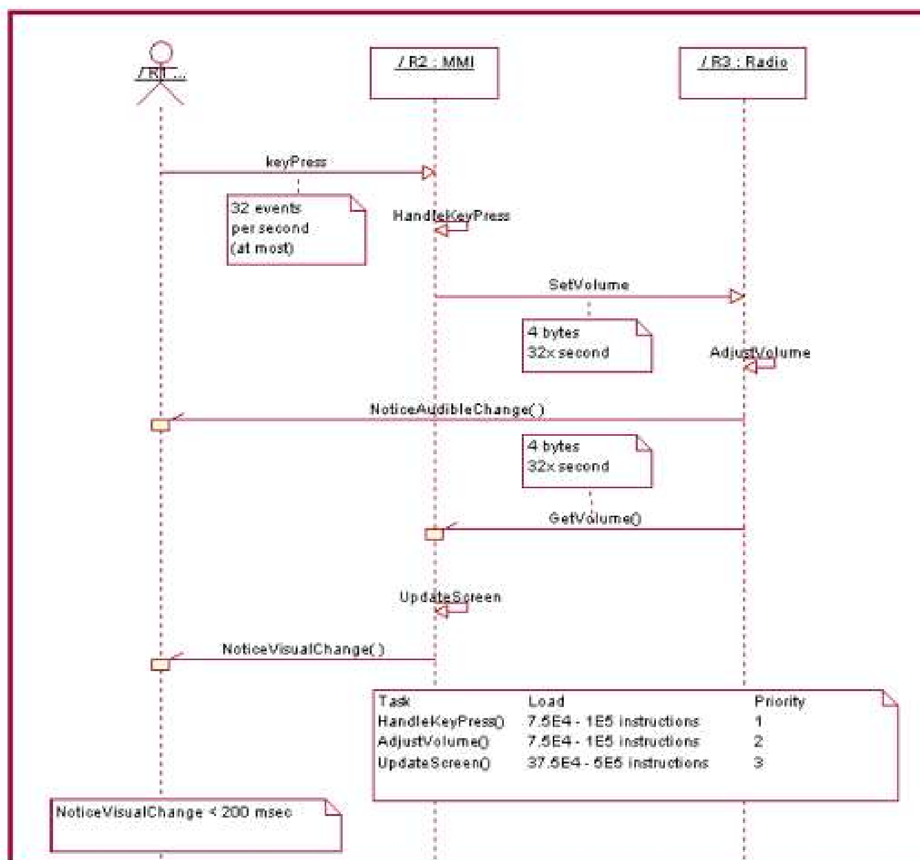


Figure 4.4: Change volume scenario

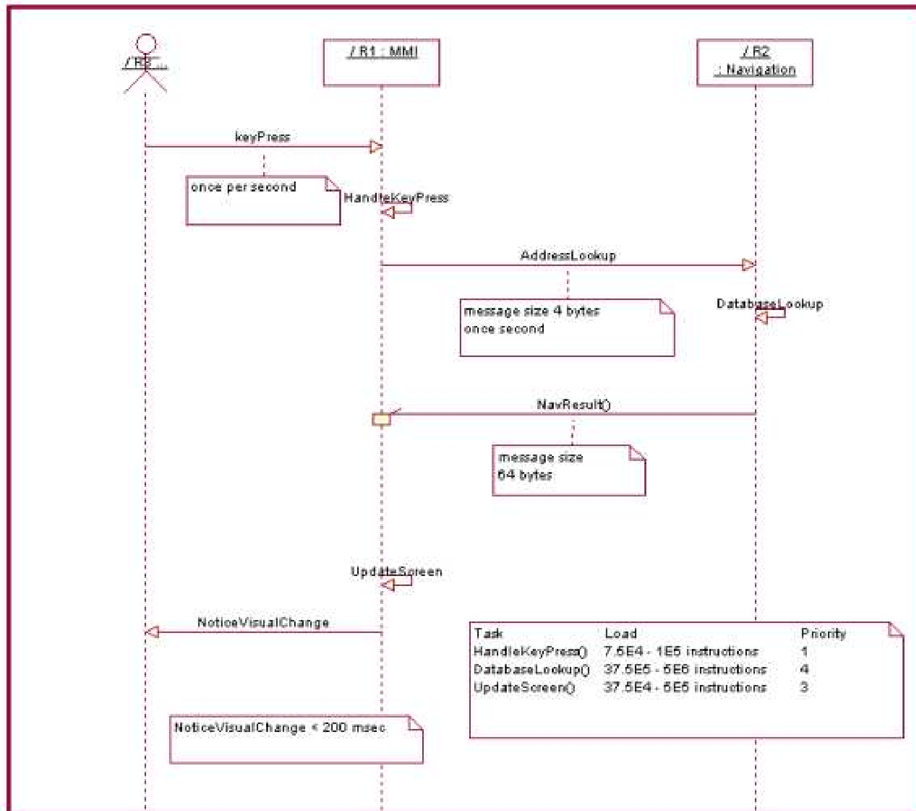


Figure 4.5: Change address scenario

The scenarios of the in-car radio navigation system have the property that they can occur in parallel. Navigation messages received from the network may be processed while the user changes the volume or enters an address to look it up in the database. However, VOL and ADDR scenarios cannot occur in parallel because both of them share a common resource, the rotary button.

Figure 4.7 presents five potential platforms for the deployment of the system. These platforms are similar to those typically used for automotive applications. The figure shows the architecture of each of the five platforms, together with the capacities of the communication and computation resource units and their nominal power consumption. The order of magnitude of these numbers is correct as they are taken from the data sheets of several commercially available automotive CPUs.

The information that we have at this moment is enough to build the model of the system. For the application part, we only need to use the `AperiodicTask` pattern as all tasks are triggered by either a system input event or a message coming from another task. To build the PDL description of the system, we derive the values of the modelling patterns parameters from the information provided by the UML sequence diagrams. As we did not have any information about how the loads of the tasks vary, we assumed that the tasks loads vary uniformly between the given

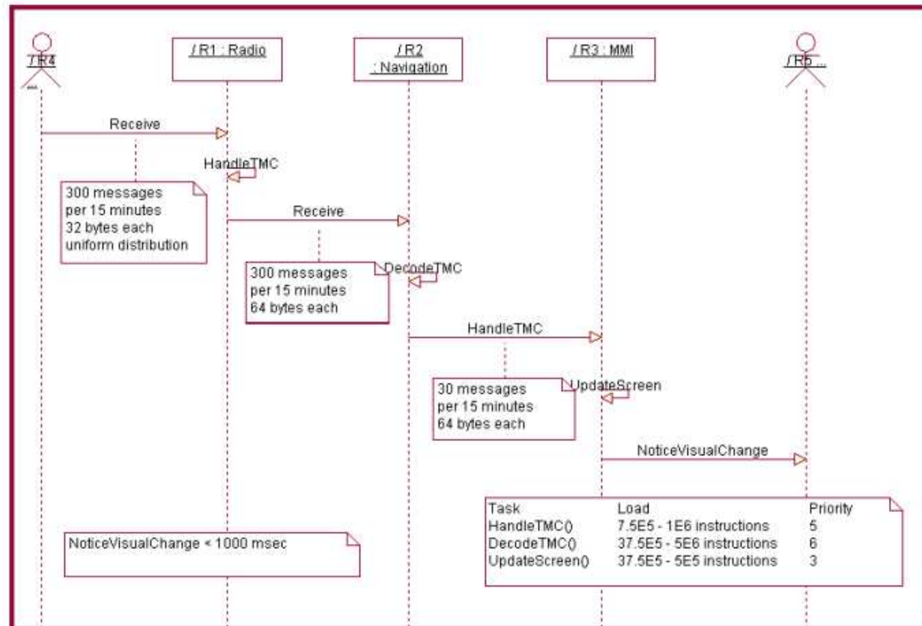


Figure 4.6: Handle TMC messages scenario

best-case and worst-case values. This probability distribution is especially difficult to analyse because it generates a large state space, which makes it very interesting. In [90], it is shown that for small models, analytical computation of the model properties is possible. However, this is not the case in our system as the in-car navigation system is too complex to be analysed in this way. The input event streams were modelled as periodic with jitter, the value of the jitter being considered as half of the period value to ensure a variation that is large enough. Moreover, for each task that is triggered by an event from the environment, we assumed to have an activation latency of 10% from the event rate in order to cover for typical latencies propagated in such systems. Furthermore, for the modelling of the platform part of the system, we used all the modelling patterns presented in table 3.1: `Resource`, `Scheduler`, `MappingMonitor` and `EnergyMonitor`. For the computational resources we considered to have a variational initial latency. As this latency models the operating system overhead as well as architecture characteristics effects (e.g. cache, pipeline) that by far are not constant, we considered a uniform variation with a worst-case value of 1000 instructions translated into execution time based on the MIPS rate. The order of magnitude of this value is in-line with the characteristics of the platforms and the typical operating systems that can run on such processors. The scheduling policy of each resource is preemptive priority-based.

4.2.2 Analysis of the System Behaviour

After deriving the POOSL model of the in-car radio navigation system, we have run simulations and analysed the results. Figure 4.8 shows the maximum value of the

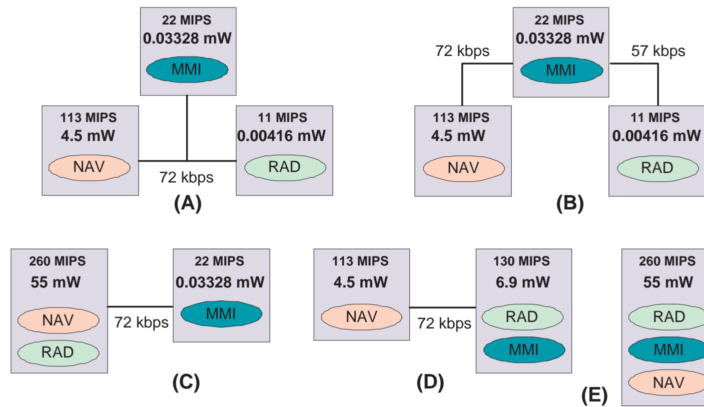


Figure 4.7: Platforms proposed for analysis

end-to-end delay observed during simulations on each of the proposed platforms for each configuration of scenarios, whereas a similar chart is shown in figure 4.9 for the estimated average value of the end-to-end delay. In all the simulations run, no deadline miss was reported, which is also proved by the obtained maximum delays that are smaller than the requirements. The maximum end-to-end delay of ADDR scenario obtained on platform A has the same value as the one for platform B as ADDR uses only NAV and MMI processors which are linked on both architectures with the same type of bus. The maximum end-to-end delay for VOL scenario is only a little larger on B than on A because the bus over which the messages are transferred between MMI and RAD is slower. In case of the TMC scenario, the even larger value of the end-to-end delay on B is explained by the need of an additional task that runs on MMI to ensure communication between NAV and RAD. When two scenarios are run in parallel, the same observations hold for platforms A and B. With respect to platforms C, D and E, the values obtained for the end-to-end delay of each combination of scenarios are much smaller which leads us to the conclusion that the platforms are over-dimensioned.

With respect to the estimations of the average end-to-end delays, we need to mention that we have run continuously our simulations until an accuracy of 95% of the results was obtained. For most of the scenarios, the simulations length is given in tenth of minutes. But when scenarios with timing requirements in different ranges, like VOL that gets an event every 1/32 of a second and TMC that gets an event every three seconds, run concurrently, the length of the simulation was measured in hours.

Another property of the system that can be checked during the simulations is the resources utilisation in each of the architecture configurations proposed. The results depicted in table 4.1 confirm our assumptions about the over-dimensioned platforms. The MMI processor, which is the same in platforms A, B and C is the most heavily used, up to 89% in platform B when an additional communication task needs to run on it. The other processors are less utilised, especially in platforms C, D and E where their MIPS rate is quite high. Moreover, we checked the power consumption of the system in each of the possible configurations. Figure 4.10 visualises the power

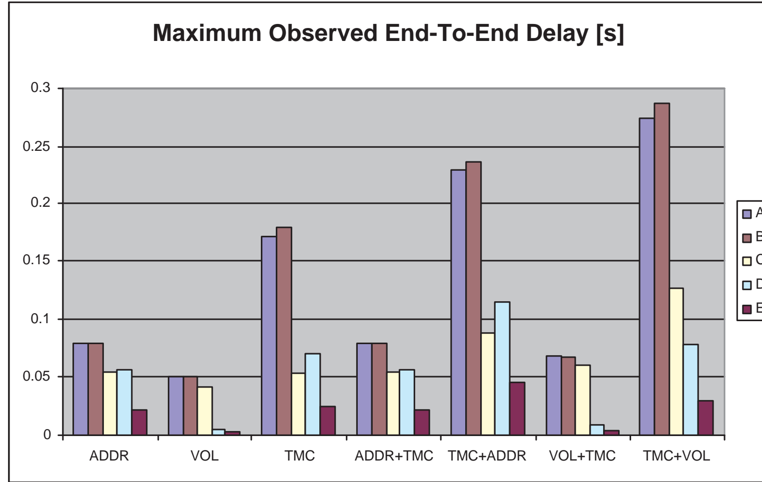


Figure 4.8: Comparison of maximum end-to-end delay on all platforms

consumption normalised to the largest value obtained for all possible scenarios and platforms configurations. It turned out that C, D and E are also the most power hungry platforms.

Arch.	CPU1	CPU2	CPU3	BUS1	BUS2
A	MMI: 88%	NAV: 6%	RAD: 32%	3%	-
B	MMI: 89%	NAV: 6%	RAD: 32%	1%	4%
C	NAV+RAD: 20%	MMI: 88%	-	3%	
D	NAV: 6%	MMI+RAD: 18%	-	1%	-
E	NAV+RAD+MMI: 10%	-	-	-	-

Table 4.1: Worst-case resources utilisations in each platform configuration

As a soft real-time system that allows 5% deadline miss ratio, the in-car radio navigation system might be deployed on a cheaper architecture, that would also consume less energy, while still providing the desired quality of service. However, if a future development of a line of products of this kind is foreseen, a dimensioning of the system suitable for the current product might not be that desirable. Usually it is better in such cases to fix a platform on which also the next versions of the application could be deployed as well. Hence, this platform needs to be performant enough to accommodate the features of the coming products.

Nevertheless, in this subsection, we show how a designer can dimension a system for the current requirements of the application. Due to the large difference between the requirements and the observed maximum end-to-end delay of each scenario, as well as based on the insight in the power consumption and utilisation of the resources, we can reason about possible platform performance reductions to reduce costs.

As the previous experiments indicate platform A as the most utilised and still consuming little power, we have used it as an example for how it can be dimensioned to suit even better the requirements of the system:

- **MMI** - The utilisation of this processor is quite high, namely 88%. The periods

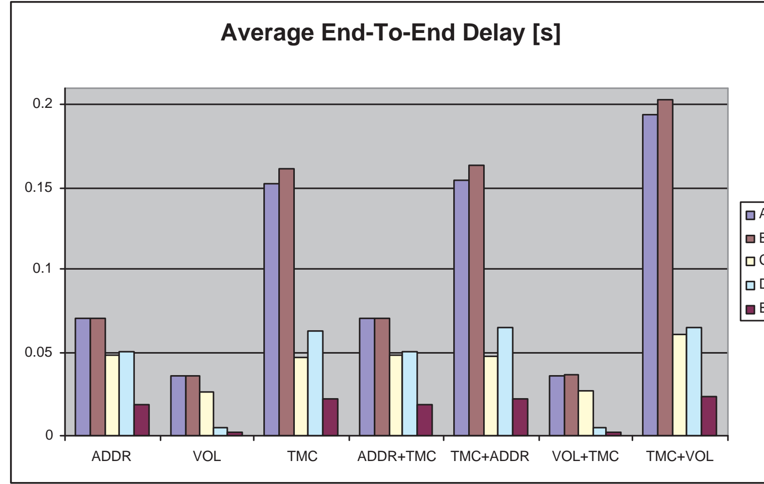


Figure 4.9: Comparison of average end-to-end delay on all platforms

Measured scenario	Other active scenario	Average end-to-end delay [*ms*]	Maximum observed end-to-end delay [*ms*]
ADDR	-	134.12	270.81
VOL	-	45.31	55.92
TMC	-	318.59	361.13
ADDR	TMC	134.12	270.81
TMC	ADDR	349.71	496.03
VOL	TMC	46.57	74.73
TMC	VOL	838.32	1056.06

Table 4.2: End-to-end delays for combinations of scenarios on platform A

and loads of the tasks mapped on this processor do not leave much room for the decrease of its capacity.

- **NAV** - This processor is used 6%. The simulation results for scenarios ADDR and TMC showed a difference of 120 ms and 730 ms respectively between the worst-case delays obtained and the requirements. This leads to the conclusion that considerable reduction of the capacity of this CPU can be achieved. We assume a reduction to 40 MIPS.

- **RAD** - The utilisation of this processor is 32%. The analysis showed a difference of 150 ms for VOL between the maximum delay and the deadline. As there is a large potential for capacity reduction, we reduce this CPU to 5 MIPS.

- **BUS** - The utilisation of the bus is very low, only 3%. Therefore, we do not expect the communication to be a bottleneck once the performance of the CPUs is reduced. However, as we did not have details about the communication link used and its power consumption, reducing the throughput of the bus would influence only the end-to-end delay.

With this new configuration for architecture A, we resumed our simulations and the results obtained for the average and maximum end-to-end delays are given in table 4.2. By the time the simulations were stopped because of achieving the desired accuracy of the estimations for the end-to-end delays, which was of 95 %, the

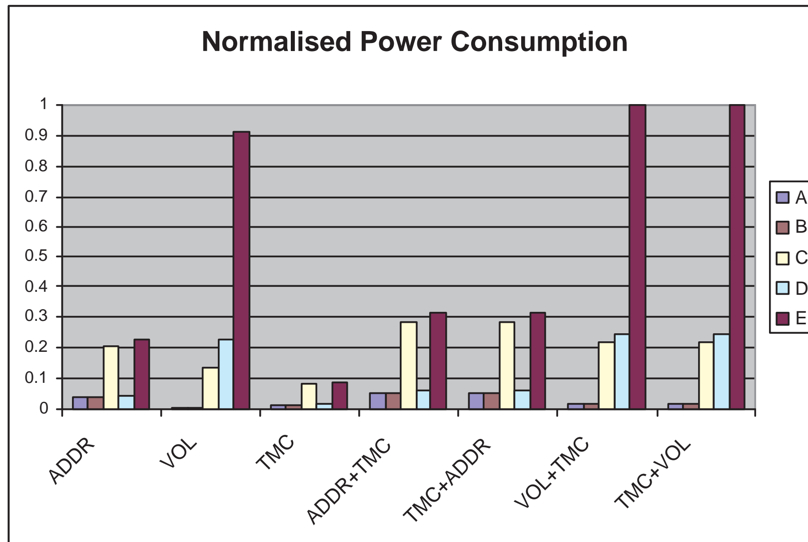


Figure 4.10: Normalised power consumption for all combinations of scenarios

monitoring of the deadline miss ratio did not achieve its accuracy. To obtain accurate estimations of them, the simulation time should have been very long, thus we stopped it without these estimates to meet their accuracy criterion. We decided to stop simulations based on the accuracy of the average delay as the deadline miss is not a frequent event. To estimate with more precision its value and to determine if the system meets the requirements, we have plotted all the end-to-end delay values obtained during the simulation as a distribution histogram. Each histogram, as the one depicted in figure 4.11, shows on the horizontal axis the rate of occurrence of each value. According to the central limit theorem [3] in probability theory, due to the uniformly distributed loads of the tasks and to the fact that tasks in different scenarios are independent, the sum of their delays, which is the end-to-end delay of a scenario, has approximately a normal distribution. Therefore, over the distribution histogram obtained from a simulation, a normal distribution curve is fitted. The curved line in figure 4.11 shows such a distribution curve fitted over the histogram of the TMC scenario. The mean value parameter (μ) of the resulted normal distribution is 838, which represents the estimated average end-to-end delay. The resulted standard deviation (σ^2) is 3953. From such curves, the rate of deadline misses can be deduced, based on their characteristics. For example, the deadline for TMC, which is 1000 ms, can be found between two and three standard deviations from the mean value. Thus, the probability of missing the deadline is less than 5%, which means the requirements are met. By accepting misses of deadlines, the designer of a system should also think of ways to back-up such situations. For our case study, this could mean leaving the radio volume as it was or dropping out the network message that could not be handled timely.

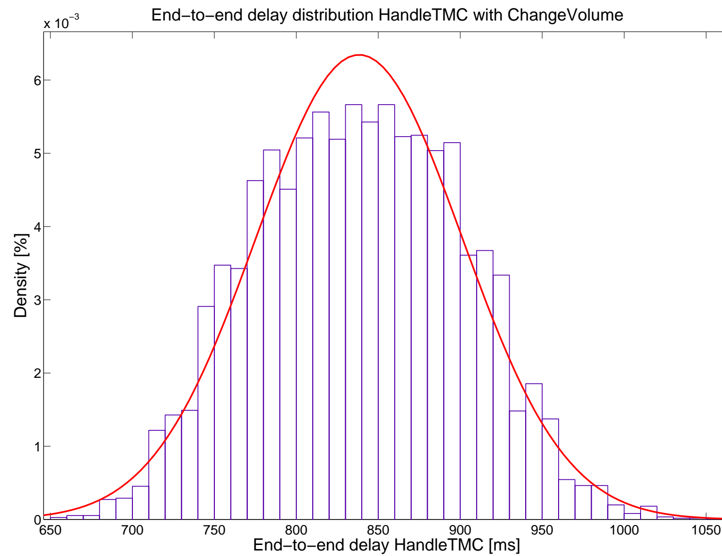


Figure 4.11: Handle TMC scenario end-to-end results fitted in a normal distribution

4.3 Case Study: The Low-Level Control of a Printer Paper Path

The second case study is inspired by a system architecture exploration for the control of the paper path of a printer. The high-level view of the system model, visualised using the SHESim tool, is given in figure 4.12. Printing requests come from the user and arrive at the high-level control (HLC) of the machine which computes which activities need to take place and when in order to accomplish the user request. The HLC tasks activate the tasks representing the low-level control (LLC) of the physical components of the paper path, like motors, sensors and actuators.

As HLC tasks are soft real-time, whereas LLC tasks, whose model is presented in figure 4.13, are hard real-time, a rather natural solution was to consider a distributed architecture. LLC can be assigned to dedicated processor(s) and connected through a network to the general-purpose processor that runs HLC. Under these circumstances, the *problem* was mainly *to find an economical architecture for LLC*, whose task parameters are shown in table 4.3.

4.3.1 The Model of the Paper Path Low-Level Control

The main activity of LLC software is to process the stream of data read from sensors and feed it back in the system. For the models of the periodic tasks of type T1, T3 and T4, considering the characteristics of the network links, we took into account an activation latency of up to 10% of their period. Both tasks of type T2 and T5 are event-driven. Tasks T2 are activated based on notifications from HLC, whereas

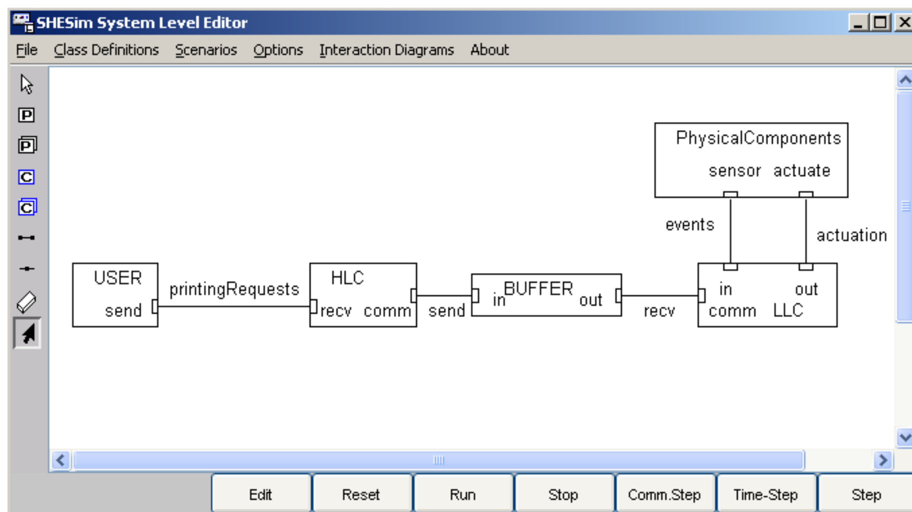


Figure 4.12: High-level printer control POOSL model

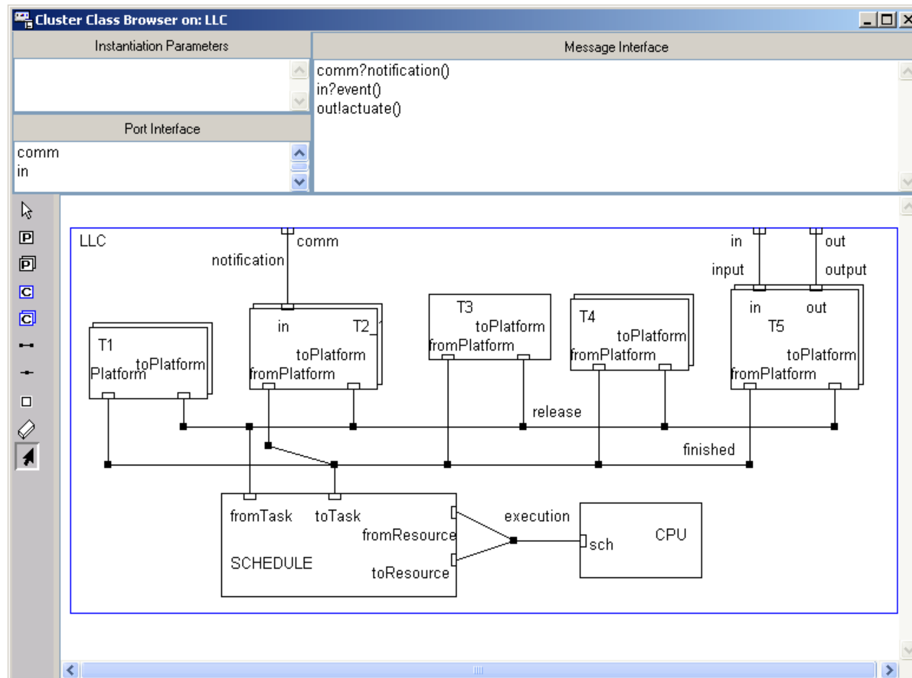


Figure 4.13: POOSL LLC model

Task type	No. of Instantiations	Load	T [ms]	D [ms]
T1	3	3200	2	2
T2	8	1200	-	2
T3	1	2000	2	2
T4	3	800	0.66	0.1
T5	4	160	-	0.064

Table 4.3: LLC task parameters

tasks T5 are triggered by events from the physical components in the environment. Therefore, a model of the *PhysicalComponents* was needed, for which we considered event streams with a uniform distribution in $[1, 20]$ ms.

4.3.2 Platform Dimensioning of the Paper Path Low-Level Control

Given the frequency of events and the task execution times, we have analysed three commercially available low-end processors, a 40 MIPS, a 20 MIPS and a 10 MIPS, and compared their utilisations under different schedulers. Figure 4.14 presents the results obtained using the earliest deadline first scheduling algorithm. Although the 10 MIPS processor seems to be used the most efficiently (close to its maximum capacity), the analysis of the model showed that there were missed deadlines; thus this processor is not a good candidate since the system is a hard real-time one. For the other two candidate processors, all deadlines are met and there were no deadlocks detected in the system. Due to the fast execution engine Rotalumis, tens of hours of system behaviour could be covered in less than one minute simulation time. Moreover, the analysis of the model gave the values of the maximum release jitter, respectively output jitter of the tasks (for the simulation on the 20 MIPS processor they are shown in table 4.4) which could be checked against the expected margins of errors of the environment control design.

Although the patterns presented in chapter 3.3 were developed mainly for modelling of soft real-time systems, this case study shows that they can successfully be applied for the modelling of hard real-time systems as well. Moreover, the tools and the analysis techniques associated to the POOSL modelling language are able to handle the analysis of such systems as well.

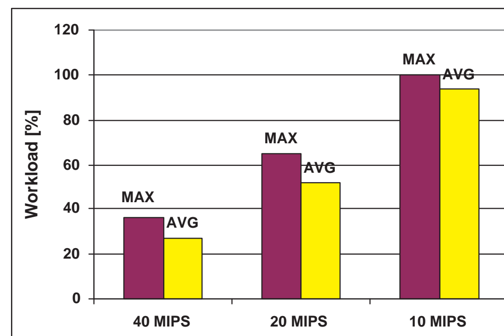


Figure 4.14: CPU workload comparison

Task type	Release jitter [ms]	Output jitter [ms]
T1	0.466	1.852
T2	0.466	1.852
T3	0.414	1.884
T4	0.042	0.128
T5	0.472	1.094

Table 4.4: Tasks jitter on the 20 MIPS processor

4.4 Summary

In this chapter, we have presented an analysis approach for dimensioning real-time systems in the areas of control-dominated and control with data-intensive computations applications. Two industrial case studies, the low-level control of a printer paper path and an in-car radio navigation system, are presented and the results of their analysis are discussed.

The in-car radio navigation system is a soft real-time system that has been previously analysed in [102] and [46] using techniques suitable for hard real-time systems. By using the modelling patterns presented in chapter 3.3 and the analysis tools associated to the POOSL language, we have successfully performed an average case analysis of the system. The results obtained helped us in finding a better dimensioning of the target platform, reducing the performance of the original platform with more than 50%.

Although the modelling and analysis approach that we have discussed is this thesis is targeted mainly towards soft real-time systems, with the case study of the low-level control of a printer paper path, we have shown that it can successfully be applied to hard real-time systems as well. By using our proposed analysis technique, we were able to determine a suitable platform for the deployment of the low-level control of the paper path that ensures the meeting of the deadlines and low task jitter.

5

Proximity Between Model and Realisation

A key aspect in the predictable design of real-time systems is to understand the relation between the properties of a model and of its corresponding implementation. After presenting in the previous chapters some means for easily building appropriate models of real-time systems and for analysing their performance properties, in this chapter we introduce an approach to predict the properties of a realisation based on those of the model.

The chapter is organised as follows. Section 5.1 introduces some mathematical notations that are needed throughput the chapter. Section 5.2 presents the mathematical model used for representation of real-time systems behaviour. The strategy for synthesising a POOSL model is discussed in section 5.3. In section 5.4, we present an approach for determining the proximity between a model and its realisation based on the exact execution times of its actions. Section 5.5 discusses the impact of the accuracy of the values of the execution times on the calculated distance, whereas section 5.6 provides the necessary and sufficient conditions such that this calculation can be done in finite time. The algorithm for calculating the distance is given in section 5.7 and a corresponding simulation approach to estimate it is presented in section 5.8. Section 5.9 gives the summary of the chapter.

5.1 Preliminaries

In this section, we define a notion of sequence of elements from a given set that we shall use throughout the chapter.

Definition 5.1 *Given a set of elements V, we denote with $\overline{v} \in V^n$ a sequence of n elements $\overline{v} = v_1 v_2 \ldots v_n$ where $v_1, v_2, \ldots v_n \in V$. The number of elements of the sequence represents*

the length of the sequence. The i-th element of the sequence $\overline{v} \in V^n$ is given by v_i, where $1 \leq i \leq n$. The empty sequence is denoted with $<>$, whereas the sequence made by a single element $u \in V$ is denoted with $<u>$.

Definition 5.2 *Given two sequences $\overline{v} \in V^n$ and $\overline{w} \in V^m$, their concatenation, denoted with $\overline{v}.\overline{w}$, is a sequence $\overline{u} \in V^{n+m}$ where $u_i = v_i$ for $1 \leq i \leq n$ and $u_{n+i} = w_i$ for $1 \leq i \leq m$.*

5.2 Representation of System Behaviour

To properly reason about the properties of a system, an abstract, mathematical model of it is needed. A common abstract representation of the behaviour of embedded systems is a graph structure regarding a system as an entity having some internal state, represented by a node. This entity is able, depending on its state, to engage in transitions, represented by edges, leading to other states (nodes). A graph structure defines a set of traces, each of them representing a possible execution of the system. Both the nodes and the edges of such a graph can be annotated with specific information to facilitate the analysis of the system properties [69].

There are two ways to annotate a graph structure such that it represents the untimed behaviour of a system. One is to annotate the nodes (the states) with atomic propositions. The other is to annotate the edges (the state transitions) with actions. A graph structure with node annotations is often formalised as a Kripke structure [60], whereas a graph structure with edge annotations is formalised as a labelled transition system [68]. A comparison of these formalisations can be found in [26]. Such a graph structure provides information about the sequences of states or state transitions of the system, but nothing about the time at which transitions take place.

To analyse the timing behaviour of a system based on a graph structure, timing information has to be attached to it. A few examples include timed Muller automata [9], which annotate edges with timing constraints, timed automata [10], which annotate nodes with timing constraints, and timed labelled transition systems, which consider the progress of time as a special type of action and correspondingly annotate edges. An example of timed labelled transition system is timed CCS [22].

As the semantics of the POOSL modelling language is based on timed labelled transition systems, in the following we present the characteristics of this mathematical structure in this context needed to understand the remainder of this thesis.

Definition 5.3 *A* timed labelled transition system *is a 6-tuple:*

$$\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$$

consisting of a countable set $\mathcal{S}$ of states, an initial state s_I, a set Act of actions, a positive time domain T^+ which is included in the set of real positive numbers $\mathbb{R}^+ \cup \{\infty\}$, a family of action transition relations $\xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S}$, and a family of time transition relations $\xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S}$.

An action transition $s_1 \xrightarrow[Act]{a} s_2$ denotes a possible transition of the system from state s_1 to state s_2 by performing action a. A time transition $s_1 \xrightarrow[T^+]{t} s_2$ denotes a transition of the system from state s_1 to state s_2 by letting an amount $t > 0$ of time pass. The time transition relation $s_1 \xrightarrow[T^+]{t} s_2$ holds if in state s_1 the model can delay for t units of time and after that it behaves as state s_2. Time transitions follow the concepts presented in [101]. The basic idea is that a process only has time transitions for maximal durations of time it wants to delay, which is called *maximal progress*, where (for reasons of compositionality) it is implicitly understood that it is also willing to delay for shorter durations of time. In the remainder, we often leave out the subscript identifying the transition relation as being the action transition relation or the time transition relation. It will be clear from the label (being from either Act or T^+) which relation is intended.

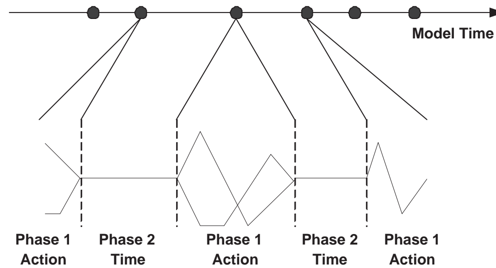


Figure 5.1: Two phase execution model

As concurrency is an important aspect of real-time systems, the interleaving semantics has been adopted in many formal frameworks to model simultaneous actions. In this model, parallel execution of two actions, $a||b$, is represented by a non-deterministic choice between two sequential actions a followed by b (i.e. $a; b$), and b followed by a (i.e. $b; a$). The interleaving semantics facilitates sequential analysis of concurrent behaviours [14]. Under the choice of interleaving semantics, the timing behaviour of a concurrent system is often formalised by a two phase execution model based on action urgency [70]. As depicted in figure 5.1, the state of the system changes either by asynchronously executing simultaneous atomic actions without passage of time (phase 1), or by letting time pass synchronously for all the components of the system when no action can be performed (phase 2). As soon as an action becomes available, the first phase is resumed.

Example 5.1 *For a system consisting of two independent parallel components P (figure 5.2.a) and Q (figure 5.2.b), assuming that the environment is always willing to participate in the communication with these components, the timed labelled transition system of the system $P||Q$ is shown in figure 5.2.c. Actions $in1$, $in2$, $out1$ and $out2$ represent communication with the environment, whereas $computation1$ and $computation2$ represent some internal calculations. In the resulting timed labelled transition system, all possible ordering of the executions of actions are taken into account, e.g. it can be either $in1$, $computation1$, $in2$, $computation2 \ldots$, or $in1$, $in2$, $computation1$, $computation2 \ldots$ and so on. Note that*

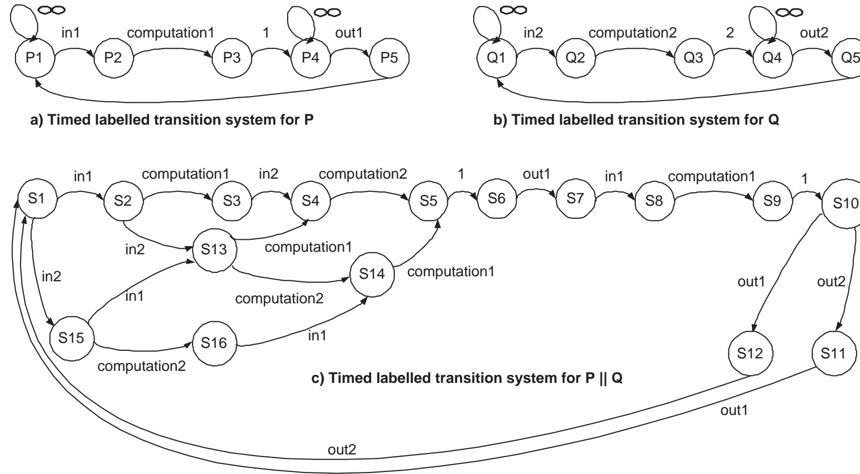


Figure 5.2: Timed labelled transition systems

due to action urgency, as component P outputs its result every 1 *unit of time, whereas component Q needs to output its result every* 2 *units of time, P performs twice its behaviour while Q performs once.*

For POOSL, process execution trees (PETs) are used for representing the timed labelled transition system of the model. The state of each POOSL process is represented by a tree structure, where each leaf is a statement and internal nodes represent compositions of their children. For example, figure 5.3.a shows the POOSL specification of the system consisting of the two parallel processes P and Q whose timed labelled transition systems are depicted in figure 5.2. The PETs of $P||Q$ are shown in figure 5.3.b. During the evolution of the system, the PETs send *action requests* and/or time *delay requests* to the scheduler. The PET scheduler, whose behaviour is described by the algorithm in figure 5.4, asynchronously grants all eligible atomic actions until there are no other actions available at the current model time moment. The internal state of each PET is dynamically changed according to the choices made by the PET scheduler and new requests may be sent to the scheduler. When no action is possible, based on the shortest delay request, time passes synchronously for all PETs until an action becomes eligible again. The value of the variable $modelTime$ is updated based on the value of the granted delay. More details about PETs can be found in [94]. The correctness of PET with respect to the semantics of the POOSL language was formally proved in [41].

Definition 5.4 *A* path *through a timed labelled transition system $\mathcal{T}$ is a 2-tuple:*

$$\mathcal{P} = (\overline{s}, \overline{\gamma})$$

where:

- $\overline{s} \in \mathcal{S}^n$ *is a finite sequence of states* $s_1, s_2, \ldots s_n$*;*

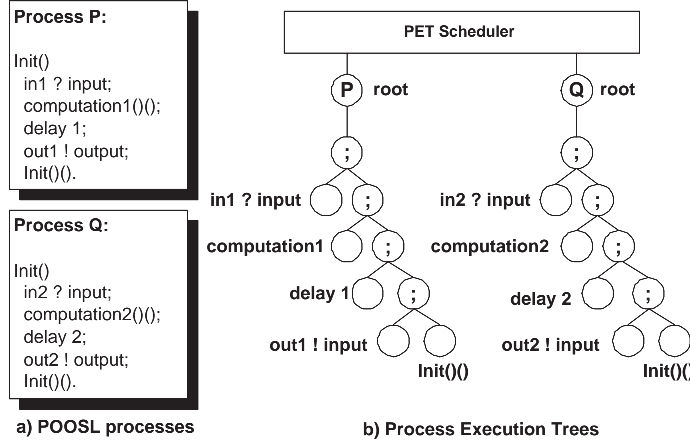


Figure 5.3: The $P||Q$ system in POOSL

```
PETSCHEDULER(LIST actions, LIST delays)
1  modelTime ← 0
2  while (true)
3      do while (actions NOTEMPTY())
4              do actions GETASYNCHRONOUSLY()->GRANT();
5          if (delays NOTEMPTY())
6            then modelTime ← modelTime + delays GETSMALLEST()->AMOUNTOFTIME();
7                  continue;
8            else DEADLOCK();
9                  return
```

Figure 5.4: The PET scheduler

- $\overline{\gamma} \in (Act \cup T^{+})^{n-1}$ *is a finite sequence of transitions* $\gamma_1, \gamma_2, \ldots \gamma_{n-1}$

with the property that $\forall i$, $1 \leq i \leq n$, $s_i \xrightarrow{\gamma_i} s_{i+1}$. *The length of the path is the length of its sequence of states.*

For the convenience of reading, a path can also be written in the form:

$$\mathcal{P} = s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} s_3 \ldots s_{n-1} \xrightarrow{\gamma_{n-1}} s_n.$$

Example 5.2

$$\begin{array}{l} s_1 \xrightarrow{in1} s_2 \xrightarrow{computation1} s_3 \xrightarrow{in2} s_4 \xrightarrow{computation2} s_5 \xrightarrow{1} s_6 \\ \xrightarrow{out1} s_7 \xrightarrow{in1} s_8 \xrightarrow{computation1} s_9 \xrightarrow{1} s_{10} \xrightarrow{out1} s_{12} \xrightarrow{out2} s_1 \end{array} \tag{5.1}$$

is a path through the timed labelled transition system presented in figure 5.2.c. Another

possible path is

$$\begin{aligned} &s_1 \xrightarrow{in1} s_2 \xrightarrow{in2} s_{10} \xrightarrow{computation1} s_4 \xrightarrow{computation2} s_5 \xrightarrow{1} s_6 \\ &\xrightarrow{out1} s_7 \xrightarrow{in1} s_8 \xrightarrow{computation1} s_9 \xrightarrow{1} s_{10} \xrightarrow{out2} s_{11} \xrightarrow{out1} s_1 \end{aligned} \tag{5.2}$$

Definition 5.5 *Given a timed labelled transition system $\mathcal{T}$, a path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ is a cycle through the transition system if $\overline{s}$ is of the form $\overline{s} = s_1 s_2 \dots s_n$ where $s_1 = s_n$.*

Definition 5.6 *Given a timed labelled transition system $\mathcal{T}$, a path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ is a simple cycle through the transition system if $\overline{s}$ is of the form $\overline{s} = s_1 s_2 \dots s_n$ where $s_1 = s_n$ and $\forall i, j$, $1 \leq i < n$ and $1 \leq j < n$, $s_i \neq s_j$ for $i \neq j$.*

Example 5.3 *Both paths 5.1 and 5.2 are simple cycles through the transition system from figure 5.2.c.*

Definition 5.7 *Given a timed labelled transition system $\mathcal{T}$, the behaviour of the system is the set $\mathcal{B}(\mathcal{T})$ of all possible paths through $\mathcal{T}$ starting from the initial state s_I, $\mathcal{B}(\mathcal{T}) = \{\mathcal{P} \mid \mathcal{P} = (\overline{s}, \overline{\gamma})$, where $s_1 = s_I\}$.*

5.3 POOSL Model Synthesis Strategy

In this section, we present a model synthesis approach for models expressed in the Parallel Object-Oriented Specification Language (POOSL). Rotalumis is a tool that takes a POOSL model and generates the executable code for the target platform. As discussed in section 5.2, the timed labelled transition system of each process in a POOSL model is represented using process execution trees (PETs). Each PET in the model is directly translated into a C++ structure whose behaviour is the same as described above. As a result, the generated implementation exhibits exactly the same behaviour as the model, if interpreted in model time domain. On the other hand, the implementation of a system needs to interact with the outside world and its behaviour has to be interpreted in physical time domain. Since the progress of model time is monotonicly increasing, which is consistent with the progress of physical time, the action order observed in model time domain is consistent with that in physical time. That is, the scheduler of PETs ensures that the implementation always has the same event order as observed in the POOSL model.

To obtain the same (or similar) quantitative timing behaviour in the physical time as in the model time, the PET scheduler synchronises the model time with the physical time during execution, as shown in figure 5.5. Considering the overhead of the PET scheduler, this ensures that an execution path observed in physical time domain is always as close as possible to a path observed in model time. As shown in [51], the smaller the distance in time between the corresponding paths in model and in realisation, the stronger the properties are preserved. Hence, it is very important to be able to determine at design time the size of this distance to be able to predict the properties of the final system.

```
PETSCHEDULERROTALUMIS(LIST actions, LIST delays)
 1  modelTime ← 0
 2  startTime ← READPHYSICALTIME()
 3  while (true)
 4      do while (actions NOTEMPTY())
 5              do actions GETASYNCHRONOUSLY()->GRANT();
 6          if (delays NOTEMPTY())
 7            then modelTime ← modelTime + delays GETSMALLEST()->AMOUNTOFTIME();
 8                 /* synchronisation between model and physical time */
 9                 if modelTime > READPHYSICALTIME() - startTime
10                   then wait_until modelTime == READPHYSICALTIME() - startTime;
11                 continue;
12            else DEADLOCK();
13                 return
```

Figure 5.5: The PET scheduler in Rotalumis

5.4 Determining the Proximity Between Model and Realisation

As a model is an approximation of the system realisation with respect to time, we will characterise the proximity between a model and its corresponding realisation as a distance in time. For this, in subsection 5.4.1 we define the distance between a path in the model and its corresponding path in the realisation based on a time labelling. Moreover, in subsection 5.4.2 we propose a way to easily calculate the distance along all the paths in the system in order to determine the proximity between the model and its realisation.

5.4.1 Definition of Distance Between Paths

In order to determine the timing distance between a path in the model and the corresponding path taken in the realisation of the system, we need to first characterise a path in time.

Definition 5.8 *A* labelling in model time *of a path* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *from a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{\xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{\xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$ *is a 3-tuple:*

$$\mathcal{L}_{\mathcal{M}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{\gamma})$$

where:

- $\overline{s} \in \mathcal{S}^n$ *is a finite sequence of states starting with the initial state of the transition system,* $s_1 = s_I$*;*
- $\overline{t_M} \in {\mathbb{R}^+}^n$ *is a finite sequence of model time labels;*
- $\overline{\gamma} \in (Act \cup T^+)^{n-1}$ *is a finite sequence of transitions*

with the following properties:

- $\forall i, 1 \leq i < n, s_i \xrightarrow{\gamma_i} s_{i+1}$;
- $t_{M_1} = 0$;
- $\forall i, 1 \leq i < n, t_{M_{i+1}} = t_{M_i}$ *if* $\gamma_i \in Act$;
- $\forall i, 1 \leq i < n, t_{M_{i+1}} = t_{M_i} + \gamma_i$ *if* $\gamma_i \in T^+$.

A labelling of a path in model time describes the timing behaviour of the system in model time. The label of a state is the model time at which the transition leaving from it occurs. For the easiness of reading, we sometimes use the following notation:

$$\mathcal{L}_{\mathcal{M}}(\mathcal{P}) = s_1(t_{M_1}) \xrightarrow{\gamma_1} s_2(t_{M_2}) \ldots s_{n-1}(t_{M_{n-1}}) \xrightarrow{\gamma_{n-1}} s_n(t_{M_n})$$

Example 5.4 *If we consider path 5.1 from example 5.2, the timing behaviour of the system based on this path is given by the following labelling:*

$$\begin{aligned} s_1(0) \xrightarrow{in1} s_2(0) \xrightarrow{computation1} s_3(0) \xrightarrow{in2} s_4(0) \xrightarrow{computation2} s_5(0) \\ \xrightarrow{1} s_6(1) \xrightarrow{out1} s_7(1) \xrightarrow{in1} s_8(1) \xrightarrow{computation1} s_9(1) \\ \xrightarrow{1} s_{10}(2) \xrightarrow{out1} s_{12}(2) \xrightarrow{out2} s_1(2) \end{aligned} \tag{5.3}$$

Because a model is an approximation of the actual timing behaviour of a system, in reality it is not possible to execute action transitions in zero time as formalised in the semantics of the model. To take this into account, we define a function on the set of action transitions of the system that gives the value of the execution time of each action on a target platform. Moreover, we assume that the execution time of an action incorporates the overhead of the execution engine that executes the model of the system in real-time. This overhead is induced by the PETs who need to change their internal states according to the choices made by the scheduler.

Definition 5.9 *Given a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$*, let*

$$ExecTime : Act \rightarrow \mathbb{R}^+$$

be a function that associates to each action transition labelled $\gamma \in Act$ *a positive real number representing the exact amount of time it takes to execute the action transition on a target platform.*

Using the function that gives the execution time of actions in a system, we can now define a labelling of a path based on both the model time and the physical time. By labelling each state of a path with both the model time at which a transition can be made from it and the physical time at which the transition is actually made, one can describe the timing behaviour of the realisation of a system. By characterising the timing behaviour of both the model and the realisation of a system, we will be able to define and to determine the distance in time between them.

Definition 5.10 *Given a path* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *from a timed labelled transition system* $\mathcal{T}$*, a labelling of it in model time* $\mathcal{L}_{\mathcal{M}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{\gamma})$ *and a function* $ExecTime : Act \to \mathbb{R}^+$*, the* labelling *of the path* $\mathcal{P}$ in both model and physical time *is a 4-tuple:*

$$\mathcal{L}^{ExecTime}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$$

where:

- $\overline{s} \in \mathcal{S}^n$ *is a finite sequence of states starting with the initial state of the transition system* $s_1 = s_I$*;*
- $\overline{t_M} \in \mathbb{R}^{+^n}$ *is a finite sequence of model time labels;*
- $\overline{t_P} \in \mathbb{R}^{+^n}$ *is a finite sequence of physical time labels;*
- $\overline{\gamma} \in (Act \cup T^+)^{n-1}$ *is a finite sequence of transitions*

with the following properties:

- $\forall i,\ 1 \leq i \leq n,\ s_i \xrightarrow{\gamma_i} s_{i+1}$*;*
- $t_{M_1} = 0$*;*
- $\forall i,\ 1 \leq i < n,\ t_{M_{i+1}} = t_{M_i}$ *if* $\gamma_i \in Act$*;*
- $\forall i,\ 1 \leq i < n,\ t_{M_{i+1}} = t_{M_i} + \gamma_i$ *if* $\gamma_i \in T^+$*;*
- $t_{P_1} = 0$[1]*;*
- $\forall i,\ 1 \leq i < n,$ *if* $\gamma_i \in Act$ *then* $t_{P_{i+1}} = t_{P_i} + ExecTime(\gamma_i)$*;*
- $\forall i,\ 1 \leq i < n,$ *if* $\gamma_i \in T^+$ *then* $t_{P_{i+1}} = t_{P_i}$ *if* $t_{M_{i+1}} < t_{P_i}$*;*
- $\forall i,\ 1 \leq i < n,$ *if* $\gamma_i \in T^+$ *then* $t_{P_{i+1}} = t_{M_{i+1}}$ *if* $t_{M_{i+1}} \geq t_{P_i}$.

From the definition, it is obvious that the model time label is either the same or lags behind the physical time label, namely that $t_{M_i} \leq t_{P_i}$. The labelling in physical time of a path is based on the synthesis mechanism for POOSL models presented in section 5.3. The value of the physical time label for a state depends on the type of transition that leads to that state. If it is an action transition, the physical time label is determined based on the label of the previous state and the execution time of the action transition, $t_{P_{i+1}} = t_{P_i} + ExecTime(\gamma_i)$. Therefore, it is clear that it will be larger than its corresponding model time label. If it is a time transition, based on the synthesis mechanism for POOSL models, a synchronisation of the model time with the physical time is realised. If the time delay is not large enough to compensate for the execution times of the actions that happened after the last time transition in the path, it means that $t_{P_i} > t_{M_{i+1}}$. Then, no time delay is taken in physical time and the physical time label of the state is copied from the previous state, $t_{P_{i+1}} = t_{P_i}$. If the time delay is large enough to compensate for the execution times of the actions that

[1]For sake of simplicity but without loss of generality, we consider that a physical clock starts counting when the execution of the system begins.

happened after the last time transition in the path, it means that $t_{P_i} \leq t_{M_{i+1}}$. Then, an amount of time delay is taken in physical time until model time and physical time are synchronised. The physical time label of the state is set to the value of the model time label of the same state, $t_{P_{i+1}} = t_{M_{i+1}}$.

For the easiness of reading, we often use the following notation for a path labelled with both model time and physical time:

$$\mathcal{L}^{\text{ExecTime}}_{\mathcal{MP}}(\mathcal{P}) = s_1(t_{M_1}, t_{P_1}) \xrightarrow{\gamma_1} s_2(t_{M_2}, t_{P_2}) \ldots s_{n-1}(t_{M_{n-1}}, t_{P_{n-1}}) \xrightarrow{\gamma_{n-1}} s_n(t_{M_n}, t_{P_n})$$

Example 5.5 *For path 5.3 of example 5.4 which is labelled in model time, we assume the following execution times for the different actions:*

- $ExecTime(in1) = 0.01$;
- $ExecTime(in2) = 0.01$;
- $ExecTime(computation1) = 0.12$;
- $ExecTime(computation2) = 0.16$;
- $ExecTime(out1) = 0.01$;
- $ExecTime(out2) = 0.01$.

The labelling in both model time and physical time that results is:

$$\begin{aligned} & s_1(0,0) \xrightarrow{in1} s_2(0,0.01) \xrightarrow{computation1} s_3(0,0.13) \xrightarrow{in2} s_4(0,0.14) \\ & \xrightarrow{computation2} s_5(0,0.30) \xrightarrow{1} s_6(1,1) \xrightarrow{out1} s_7(1,1.01) \xrightarrow{in1} s_8(1,1.02) \\ & \xrightarrow{computation1} s_9(1,1.14) \xrightarrow{1} s_{10}(2,2) \xrightarrow{out1} s_{12}(2,2.01) \xrightarrow{out2} s_1(2,2.02) \end{aligned} \tag{5.4}$$

Based on the labelling in both model time and physical time, we can now define the distance between model and realisation along a path.

Definition 5.11 *Given a path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ of length n through a timed labelled transition system $\mathcal{T}$ and its labelling $\mathcal{L}^{\text{ExecTime}}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ based on a function $ExecTime : Act \rightarrow \mathbb{R}^+$, the* distance between the timing behaviour of the model and of the realisation along this path *is given by:*

$$d^{\text{ExecTime}}(\mathcal{P}) = \max_{1 \leq i \leq n} (t_{P_i} - t_{M_i}).$$

Example 5.6 *The distance between model and realisation along path 5.4 in example 5.5 is:*

$$\begin{aligned} d^{\text{ExecTime}}(\mathcal{P}) = \max\{&0-0,\ 0.01-0,\ 0.13-0,\ 0.14-0,\ 0.30-0,\ 1-1,\ 1.01-1, \\ &1.02-1,\ 1.14-1,\ 2-2,\ 2.01-2,\ 2.02-2\} = \max\{0,\ 0.01,\ 0.13,\ 0.14, \\ &0.30,\ 0,\ 0.01,\ 0.02,\ 0.14,\ 0,\ 0.01,\ 0.02\} = 0.30 \end{aligned}$$

However, the labelling with both model time and physical time is not very convenient for calculating the distance between model and realisation because both the model time and the physical time keep increasing. Therefore, we define a new labelling of a path in which the difference between the values of the model time label and the physical time label of a state are connected immediately.

Definition 5.12 *Given a path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ through a timed labelled transition system $\mathcal{T}$ and a function $ExecTime : Act \to \mathbb{R}^+$, a* time difference labelling *of it is a 3-tuple:*

$$\mathcal{L}^{ExecTime}(\mathcal{P}) = (\overline{s}, \overline{d}, \overline{\gamma})$$

where:

- $\overline{s} \in \mathcal{S}^n$ *is a finite sequence of states starting with the initial state of the transition system* $s_1 = s_I$*;*
- $\overline{d} \in \mathbb{R}^{+n}$ *is a finite sequence of labels;*
- $\overline{\gamma} \subseteq (Act \cup T^+)^{n-1}$ *is a finite sequence of transitions*

where

- $d_1 = 0$*;*
- $\forall i$, $1 \leq i < n$, *if* $\gamma_i \in Act$ *then* $d_{i+1} = d_i + ExecTime(\gamma_i)$*;*
- $\forall i$, $1 \leq i < n$, *if* $\gamma_i \in T^+$ *then* $d_{i+1} = d_i \dot{-} \gamma_i$*, where* $\dot{-}$ *is the monus function [62] which is defined as* $d_{i+1} = \max(d_i - \gamma_i, 0)$*.*

Example 5.7 *The time difference labelling of path 5.1 from example 5.2 using the execution times of actions from example 5.5 is:*

$$\begin{aligned} s_1(0) &\xrightarrow{in1} s_2(0.01) \xrightarrow{computation1} s_3(0.13) \xrightarrow{in2} s_4(0.14) \\ &\xrightarrow{computation2} s_5(0.30) \xrightarrow{1} s_6(0) \xrightarrow{out1} s_7(0.01) \xrightarrow{in1} s_8(0.02) \\ &\xrightarrow{computation1} s_9(0.14) \xrightarrow{1} s_{10}(0) \xrightarrow{out1} s_{12}(0.01) \xrightarrow{out2} s_1(0.02) \end{aligned} \tag{5.5}$$

Lemma 5.1 *Given a path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ of length n through a timed labelled transition system $\mathcal{T}$ and its time difference labelling $\mathcal{L}^{ExecTime}(\mathcal{P}) = (\overline{s}, \overline{d}, \overline{\gamma})$ based on a function $ExecTime : Act \to \mathbb{R}^+$, the distance between the timing behaviour of the model and of the realisation along this path is:*

$$d^{ExecTime}(\mathcal{P}) = \max_{1 \leq i \leq n} (d_i).$$

Proof Let $\mathcal{L}^{\text{ExecTime}}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ be the labelling of path $\mathcal{P}$ in both model time and physical time. Using definitions 5.10 and 5.12, we are going to show by induction that $\forall i$, $1 \leq i \leq n$, $d_i = t_{P_i} - t_{M_i}$.

Case 1. Let $i = 1$. Then $d_i = 0 = t_{P_i} - t_{M_i}$.

Case 2. Let $i > 1$ and assume $d_i = t_{P_i} - t_{M_i}$. Then

- if $\gamma_i \in Act$, $d_{i+1} = d_i + ExecTime(\gamma_i) = t_{P_i} - t_{M_i} + ExecTime(\gamma_i) = t_{P_{i+1}} - t_{M_{i+1}}$;
- if $\gamma_i \in T^+$,

$$d_{i+1} = d_i \dot{-} \gamma_i = \begin{cases} d_i - \gamma_i, & \text{if } d_i > \gamma_i \\ 0, & \text{if } d_i \leq \gamma_i \end{cases}$$

$$= \begin{cases} t_{P_i} - t_{M_i} - \gamma_i, & \text{if } t_{P_i} - t_{M_i} > \gamma_i \\ 0, & \text{if } t_{P_i} - t_{M_i} \leq \gamma_i \end{cases}$$

$$= \begin{cases} t_{P_i} - t_{M_i} - \gamma_i, & \text{if } t_{P_i} > t_{M_i} + \gamma_i \\ 0, & \text{if } t_{P_i} \leq t_{M_i} + \gamma_i \end{cases}$$

$$= \begin{cases} t_{P_i} - t_{M_{i+1}}, & \text{if } t_{P_i} > t_{M_{i+1}} \\ t_{M_{i+1}} - t_{M_{i+1}}, & \text{if } t_{P_i} \leq t_{M_{i+1}} \end{cases}$$

$$= \begin{cases} t_{P_{i+1}} - t_{M_{i+1}}, & \text{if } t_{P_i} > t_{M_{i+1}} \\ t_{P_{i+1}} - t_{M_{i+1}}, & \text{if } t_{P_i} \leq t_{M_{i+1}} \end{cases}$$

Then $d_{i+1} = t_{P_{i+1}} - t_{M_{i+1}}$.

□

Considering the time difference labelling of states along a path in a system, the time deviation between model and realisation is given by the label with the largest value observed along a path.

Definition 5.13 *Given a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$ *and a function* $ExecTime : Act \to \mathbb{R}^+$, *the* distance between the timing behaviour of the model and that of the realisation *is given by:*

$$d^{ExecTime}(\mathcal{T}) = \sup\{d^{ExecTime}(\mathcal{P}) \mid \mathcal{P} \textit{ is a path of } \mathcal{T} \textit{ starting in } s_I\}.$$

The distance between a model and its realisation represents an upper-bound of all the distances along all possible paths $\mathcal{P}$ starting in the initial state of the system, $\mathcal{P} \in \mathcal{B}(\mathcal{T})$. As there can be infinitely many possible paths, we need to use the supremum instead of the maximum for the definition of the distance. This distance represents the key ingredient in determining the relation between the properties analysed in the model and their preservation in the realisation. In [51], it is shown that the properties of the model are preserved (also called weakened) in the realisation up to their distance $d^{\text{ExecTime}}(\mathcal{T})$. Hence, the calculation of the distance between a model and its realisation onto a certain target platform is crucial for predicting the properties of the realisation before actually building it.

5.4.2 Calculating the Distance Between Model and Realisation

Typically a timed labelled transition system has infinitely many paths. Therefore, lemma 5.1 for determining the distance between a model and a realisation along a path is not directly suitable to compute this distance for all the paths in the system.

To efficiently compute this distance, we define an extended timed labelled transition system based on which we can determine the distance between model and realisation without identifying all the individual paths. This extended timed labelled transition system is built using the technique of time difference labelling of states given by definition 5.12.

Definition 5.14 *Given a timed labelled transition system*

$$\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$$

and a function $ExecTime : Act \to \mathbb{R}^+$, the extended timed labelled transition system *is a 6-tuple:*

$$\mathcal{E}^{ExecTime}(\mathcal{T}) = ((\mathcal{S} \times \mathbb{R}^+), (s_I, 0), Act, T^+, \{ \xrightarrow[Act]{a}{}^{\mathcal{E}} \subseteq (\mathcal{S} \times \mathbb{R}^+) \times (\mathcal{S} \times \mathbb{R}^+) \mid$$

$$a \in Act\}, \{ \xrightarrow[T^+]{t}{}^{\mathcal{E}} \subseteq (\mathcal{S} \times \mathbb{R}^+) \times (\mathcal{S} \times \mathbb{R}^+) \mid t \in T^+\}).$$

Action transition $(s_1, d_1) \xrightarrow[Act]{a}{}^{\mathcal{E}} (s_2, d_2)$ holds iff:

- $s_1 \xrightarrow[Act]{a} s_2$ *in* $\mathcal{T}$;
- $d_2 = d_1 + ExecTime(a)$.

Time transition $(s_1, d_1) \xrightarrow[T^+]{t}{}^{\mathcal{E}} (s_2, d_2)$ holds iff:

- $s_1 \xrightarrow[T^+]{t} s_2$ *in* $\mathcal{T}$;
- $d_2 = d_1 \dot{-} t$.

The reader may have noticed that, according to the definition, the state space $\mathcal{S} \times \mathbb{R}^+$ of an extended timed labelled transition system is not countable. However, it is not difficult to prove that the collection of states that are reachable from the initial state $(s_I, 0)$ is countable [100].

Example 5.8 *The extended timed labelled transition system for the example in figure 5.2.c is depicted in figure 5.6. As a remark, it is possible that multiple copies of a state from the original transition system to appear in the extended timed labelled transition system with different time difference labels, as it is the case with states $s_1, s_2, s_3, s_4, s_5, s_{13}, s_{14}, s_{15}, s_{16}$ in figure 5.6.*

Using the extended timed labelled transition system, the following theorem overcomes the problem of taking each individual path of the system into account for determining the distance between a model and its realisation from definition 5.13.

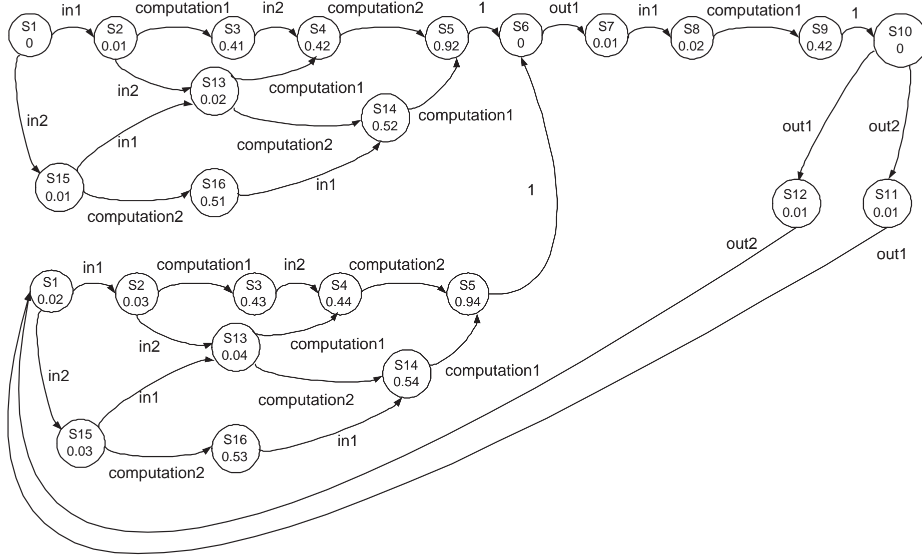


Figure 5.6: Extended timed labelled transition system

Theorem 5.2 *Let $\mathcal{T}$ be a timed labelled transition system whose initial state is s_I and let $\mathcal{E}^{ExecTime}(\mathcal{T})$ be its corresponding extended timed labelled transition system whose initial state is $(s_I, 0)$. Then:*

$$d^{ExecTime}(\mathcal{T}) = \sup\{d \mid (s,d) \textit{ is a state in } \mathcal{E}^{ExecTime}(\mathcal{T}) \textit{ reachable from } (s_I, 0)\}.$$

Proof By definition 5.13,

$$d^{\text{ExecTime}}(\mathcal{T}) = \sup\{d^{\text{ExecTime}}(\mathcal{P}) \mid \mathcal{P} \text{ is a path of } \mathcal{T} \text{ starting in } s_I\}.$$

Therefore, we need to show that

$$\sup\{d \mid (s,d) \text{ is a state in } \mathcal{E}^{\text{ExecTime}}(\mathcal{T}) \text{ reachable from } (s_I,0)\} = \\ \sup\{d^{\text{ExecTime}}(\mathcal{P}) \mid \mathcal{P} \text{ is a path of } \mathcal{T} \text{ starting in } s_I\}.$$

For any extended state (s,d) reachable from $(s_I, 0)$ in $\mathcal{E}^{\text{ExecTime}}(\mathcal{T})$, there exists a path $\mathcal{E}^{\text{ExecTime}}(\mathcal{P}) = (\overline{(s,d)}, \overline{\gamma})$ of length n such that $(s,d)_1 = (s_I, 0)$ and $(s,d)_n = (s,d)$. By the construction of the extended timed labelled transition system, there exists a corresponding path $\mathcal{P}$ in $\mathcal{T}$ from s_I to s such that the time difference label of s is d. From lemma 5.1, $d \leq d^{\text{ExecTime}}(\mathcal{P})$. Hence

$$\sup\{d \mid (s,d) \text{ is a state in } \mathcal{E}^{\text{ExecTime}}(\mathcal{T}) \text{ reachable from } (s_I,0)\} \leq \\ \sup\{d^{\text{ExecTime}}(\mathcal{P}) \mid \mathcal{P} \text{ is a path of } \mathcal{T} \text{ starting in } s_I\}.$$

The other way around, for any path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ starting in the initial state s_I of $\mathcal{T}$, there exists a corresponding path $\mathcal{E}^{\text{ExecTime}}(\mathcal{P}) = (\overline{(s,d)}, \overline{\gamma})$ in $\mathcal{E}^{\text{ExecTime}}(\mathcal{T})$ starting

in $(s_I, 0)$. Then, for any state s reachable from s_I through a path $\mathcal{P}$ in $\mathcal{T}$ there exists a corresponding extended state (s, d) in $\mathcal{E}^{\text{ExecTime}}(\mathcal{T})$ reachable from $(s_I, 0)$ through path $\mathcal{E}^{\text{ExecTime}}(\mathcal{P})$. Since $d^{\text{ExecTime}}(\mathcal{P})$ is equal to the time difference label of one of the states in $\overline{s}$, we have that

$$\{d^{\text{ExecTime}}(\mathcal{P}) \mid \mathcal{P} \text{ is a path of } \mathcal{T} \text{ starting in } s_I\} \subseteq$$
$$\{d \mid (s, d) \text{ is a state in } \mathcal{E}^{\text{ExecTime}}(\mathcal{T}) \text{ reachable from } (s_I, 0)\}.$$

Hence

$$\sup\{d^{\text{ExecTime}}(\mathcal{P}) \mid \mathcal{P} \text{ is a path of } \mathcal{T} \text{ starting in } s_I\} \leq$$
$$\sup\{d \mid (s, d) \text{ is a state in } \mathcal{E}^{\text{ExecTime}}(\mathcal{T}) \text{ reachable from } (s_I, 0)\}.$$

In this way, we proved that

$$d^{\text{ExecTime}}(\mathcal{T}) = \sup\{d \mid (s, d) \text{ is a state in } \mathcal{E}^{\text{ExecTime}}(\mathcal{T}) \text{ reachable from } (s_I, 0)\}.$$

□

Based on theorem 5.2, the distance between a model and its realisation is the upper-bound on the value of all the labels of states reachable from the initial state in the extended timed labelled transition system.

5.5 Execution Time Accuracy Impact on Distance

In section 5.4, we have presented an approach for determining the distance between a model and its realisation using the exact execution times of actions on the desired target platform. However, in practice, we do not have such values, and most of the time we need to resort to (sometimes rough estimations of) the worst-case execution times that can be obtained using methods like [63] and [43]. In the rest of this section, we show that by using the worst-case execution time of actions, the distance between model and realisation increases compared to the situation when the exact execution times of actions are used. This aspect is important because if by using the worst-case execution times of actions the distance decreased, the actual distance between model and realisation would be larger than the one calculated. Hence, the properties of the realisation would not be correctly predicted from the model.

Theorem 5.3 *Given a timed labelled transition system $\mathcal{T}$ and two functions $ExecTime_1 : Act \rightarrow \mathbb{R}^+$ and $ExecTime_2 : Act \rightarrow \mathbb{R}^+$ with $ExecTime_1(\gamma) = ExecTime_2(\gamma)$ for any $\gamma \in Act \setminus \{a\}$ and $ExecTime_1(a) < ExecTime_2(a)$ for some $a \in Act$, then $d^{ExecTime_1}(\mathcal{T}) \leq d^{ExecTime_2}(\mathcal{T})$.*

Proof Assume there exists some $s, s' \in \mathcal{S}$ such that $s \xrightarrow{a} s'$ and on any path from s_I to s there is no occurrence of a. Then there exists some d_1, d_1' such that $(s, d_1) \xrightarrow{a}_{\mathcal{E}} (s', d_1')$ in $\mathcal{E}^{\text{ExecTime}_1}(\mathcal{T})$ and some d_2, d_2' such that $(s, d_2) \xrightarrow{a}_{\mathcal{E}} (s', d_2')$ in $\mathcal{E}^{\text{ExecTime}_2}(\mathcal{T})$. Since a does not occur in the path from s_I to s, then by definition 5.14 $d_1 = d_2$. Moreover, $d_1' = d_1 + ExecTime_1(a)$ and $d_2' = d_2 + ExecTime_2(a) = d_1 +$

$ExecTime_2(a) > d_1 + ExecTime_1(a) = d_1^{'}$. Hence, by theorem 5.2, $d^{\text{ExecTime}_1}(\mathcal{T}) = \sup\{d \mid (s,d)$ is a state in $\mathcal{E}^{\text{ExecTime}_1}(\mathcal{T})$ reachable from $(s_I, 0)\} \leq \sup\{d \mid (s,d)$ is a state in $\mathcal{E}^{\text{ExecTime}_2}(\mathcal{T})$ reachable from $(s_I, 0)\} = d^{\text{ExecTime}_2}(\mathcal{T})$.

□

Corollary 5.4 *Given an extended timed labelled transition system, the largest distance between model and realisation is obtained when the execution time of each action is equal to its worst-case execution time.*

This result gives us the possibility of safely using the worst-case execution times of actions, which are typically available at least as estimations, for calculating an upper-bound on the distance from model to realisation. Hence, we are not restricted to determining exact execution times of actions which in general is not possible. Moreover, we are sure that the value obtained for the distance is not an optimistic one and in fact, the real value will always be smaller.

5.6 Finite Time Computation of Distance

By theorem 5.2, we have that the distance between a model and its realisation is an upper-bound on the labels of all the states in the extended timed labelled transition system. If the timed labelled transition system has infinitely many states, then obviously, its extended timed labelled transition system also has infinitely many states and the distance cannot be computed in finite time. However, when the timed labelled transition system is finite in the number of states and transitions, it is not immediately obvious if its corresponding extended timed labelled transition system is also finite. In the rest of this section, we show that when the timed labelled transition system is finite, which is generally the case for real-time systems, if the resulting extended timed labelled transition system is infinite, then the distance between model and realisation is infinite. We also provide the necessary and sufficient conditions for the extended timed labelled transition system to be finite and the computation of the distance to be done in finite time.

We first present in subsection 5.6.1 some mathematical notations and results needed to understand the main result presented in subsection 5.6.2. Then, we prove a theorem which is the foundation for building an algorithm to calculate the distance between model and realisation.

5.6.1 Preliminaries

Definition 5.15 *Given a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{\xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{\xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$ *and a function* $ExecTime : Act \to \mathbb{R}^+$, *let* $L : Act \cup T^+ \to \mathbb{R}$ *be a function on the labels of the transitions in* $\mathcal{T}$ *such that:*

$$L(\gamma) = \begin{cases} ExecTime(\gamma), & \gamma \in Act; \\ -\gamma, & \gamma \in T^+. \end{cases}$$

Let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a path through $\mathcal{T}$. The sequence of numbers along the path, *denoted by $\overline{\mathcal{P}} \in \mathbb{R}^{n-1}$, is defined as $\overline{\mathcal{P}} = <L(\gamma_1)> . <L(\gamma_2)> \ldots <L(\gamma_{n-1})>$.*

Example 5.9 *For the execution time function defined in example 5.5 and path 5.1 of example 5.2, the sequence of numbers along it is:*

$$<0.01>.<0.12>.<0.01>.<0.16>.<-1>.<0.01>.<0.01>.$$
$$<0.12>.<-1>.<0.01>.<0.01>$$

Definition 5.16 *Given a sequence $\overline{x} \in \mathbb{R}^n$, the sum operation over the sequence $\sum \overline{x}$ is defined recursively as follows:*

- $\sum <> = 0$
- $\sum(\overline{x}. <y>) = \max(\sum \overline{x} + y, 0)$, $y \in \mathbb{R}$

Lemma 5.5 *Given two sequences $\overline{x} \in \mathbb{R}^n$ and $\overline{y} \in \mathbb{R}^m$, the sum of the elements of the concatenated sequence $\overline{x}.\overline{y} \in \mathbb{R}^{n+m}$ is $\sum(\overline{x}.\overline{y}) = \sum(<\sum \overline{x}> .\overline{y})$.*

Proof Please find the proof in appendix A lemma A.3.

□

Lemma 5.6 *Given a timed labelled transition system $\mathcal{T}$, let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a path of length n in $\mathcal{T}$ and $\overline{\mathcal{P}}$ be the sequence of numbers along $\mathcal{P}$. Given a function $ExecTime : Act \to \mathbb{R}^+$, let $\mathcal{E}^{\text{ExecTime}}(\mathcal{T})$ be the extended timed labelled transition system where d_1 is the label of state s_1, respectively d_n of s_n. Then, $d_n = \sum(<d_1> .\overline{\mathcal{P}})$.*

Proof By the construction of $\mathcal{E}^{\text{ExecTime}}(\mathcal{T})$, for any $1 < i \leq n$ we have:

$$d_{i+1} = \begin{cases} d_i + ExecTime(\gamma_i) & \text{if } \gamma_i \in Act \\ d_i \dot{-} \gamma_i = \max(d_i - \gamma_i, 0) & \text{if } \gamma_i \in T^+ \end{cases}$$

$$= \{\textit{by definition } 5.15\} \begin{cases} d_i + \mathcal{P}_i & \text{if } \mathcal{P}_i \geq 0; \\ \max(d_i + \mathcal{P}_i, 0) & \text{if } \mathcal{P}_i < 0. \end{cases}$$

Hence, $d_{i+1} = \max(d_i + \mathcal{P}_i, 0)$ for all $1 < i \leq n$. Moreover, by definition 5.16, we have that $d_{i+1} = \sum(<d_i> . <\mathcal{P}_i>)$. By substitution and using lemma 5.5, we obtain $d_n = \sum(<d_1> .\overline{\mathcal{P}})$.

□

Lemma 5.7 *Given $\overline{x} \in \mathbb{R}^n$ and $a \in \mathbb{R}$, then $\sum(<a> .\overline{x}) \leq a + \sum \overline{x}$ if $a \geq 0$.*

Proof Please find the proof in appendix A lemma A.6.

□

Theorem 5.8 *Let $\overline{x} \in \mathbb{R}^n$ and let $\overline{y} \in \mathbb{R}^n$ be any permutation of $\overline{x}$ such that for some m, $1 \leq m \leq n+1$, $t_i \geq 0$ for all $1 \leq i < m$ and $t_i < 0$ for all $m \leq i \leq n$. Then $\sum \overline{y} \leq \sum \overline{x'}$ for any permutation $\overline{x}'$ of $\overline{x}$.*

Proof Please find the proof in appendix A theorem A.9.

$\square$

Definition 5.17 *For a sequence $\overline{x} \in \mathbb{R}^n$, the smallest value of the sum over any permutation of $\overline{x}$ is denoted with $\downarrow\overline{x}$.*

Theorem 5.9 *Let $\overline{x} \in \mathbb{R}^n$ and let $\overline{y} \in \mathbb{R}^n$ be any permutation of $\overline{x}$ such that for some m, $1 \leq m \leq n+1$, $t_i < 0$ for all $1 \leq i < m$ and $t_i \geq 0$ for all $m \leq i \leq n$. Then $\sum \overline{y} \geq \sum \overline{x'}$ for any permutation $\overline{x}'$ of $\overline{x}$.*

Proof Please find the proof in appendix A theorem A.10.

$\square$

Definition 5.18 *For a sequence $\overline{x} \in \mathbb{R}^n$, the largest value of the sum over any permutation of $\overline{x}$ is denoted with $\uparrow\overline{x}$.*

Theorem 5.10 *Consider a timed labelled transition system $\mathcal{T}$ such that $\downarrow\overline{\mathcal{C}} = 0$ for every simple cycle $\mathcal{C}$ in $\mathcal{T}$. Then for any $a \geq 0$ and any cycle $\mathcal{Q}$ in $\mathcal{T}$, the following are true:*

1. $\downarrow\overline{\mathcal{Q}} = 0$;
2. $\sum(<a> .\overline{\mathcal{Q}}) \leq \max(a, \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\})$;
3. *for any prefix $\overline{\mathcal{Q}'}$ of $\overline{\mathcal{Q}}$,* $\sum(<a> .\overline{\mathcal{Q}'}) \leq \max(a, \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}) + \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}$.

Proof Please find the proof in appendix A theorem A.12.

$\square$

Lemma 5.11 *Let L be a finite set of rational numbers, $L \in \mathbb{Q}$, and let S be any set of finite sequences of numbers from L. Then the set $K = \{\sum \overline{x} \mid \overline{x} \in S\}$ is finite iff $\sup(K)$ is a finite number.*

Proof *"$\Rightarrow$"*

If K is finite, then it is obvious that there exists some $m \in \mathbb{Q}^+$ such that $\sum \overline{x} \leq m$ for all $\overline{x} \in S$. Hence, $\sup(K) = m$, meaning that K is bounded.

"$\Leftarrow$"

Assuming K is bounded, there exists some $m \in \mathbb{Q}^+$ such that $\sum \overline{x} \leq m$ for all $\overline{x} \in S$. Moreover, as any number in L is from $\mathbb{Q}$, for any $l \in L$ we can write $l = \frac{p}{q}$. Because L is finite, there exists $N \in \mathbb{N}$ the least common multiplier of all denominators q such that $N \cdot l \in \mathbb{Z}$ for all $l \in L$.

We define $K^N = \{N \cdot \sum \overline{x} \mid \overline{x} \in S\}$. By construction, K^N is a set of natural numbers because $N \cdot \sum \overline{x} \in \mathbb{N}$ for each $\overline{x} \in S$.

We build a function $F : K \to K^N$ such that $F(\sum \overline{x}) = N \cdot \sum \overline{x}$. Because for any $\sum \overline{x_1}, \sum \overline{x_2} \in K$, we have that $\sum \overline{x_1} \neq \sum \overline{x_2}$, then $F(\sum \overline{x_1}) \neq F(\sum \overline{x_2})$, thus F

is bijective. Since K is bounded, K^N is also bounded and $\sup(K^N) = N \cdot m$. But by construction K^N is a set of natural numbers and there exists a finite number of natural numbers smaller or at most equal to $N \cdot m$.

Hence, K^N is finite and because of the one to one mapping of its elements to the elements of K, K is also a finite set.

□

As an observation, lemma 5.11 is not true if the set L contains also numbers from $\mathbb{R} \setminus \mathbb{Q}$. To show that, let us assume a set $L = \{\Pi, -1\}$ where $\Pi \in \mathbb{R} \setminus \mathbb{Q}$. Moreover, let $S = \{\overline{x}_i \mid i \in \mathbb{N}\}$, where $\overline{x}_1 = <\Pi> . <-1> . <-1> . <-1>$ and $\overline{x}_i = <\Pi> \ldots <\Pi> . <-1> \ldots <-1>$ such that $<\Pi>$ occurs i times and $< -1 >$ occurs $\lfloor i\Pi \rfloor$ times. By construction, each $\overline{x}_i$ is a finite sequence whose length is $i + \lfloor i\Pi \rfloor$. Thus, S is an infinite set of finite sequences of numbers from L. Moreover, $\sum \overline{x}_i = i\Pi - \lfloor i\Pi \rfloor$, hence $K = \{\sum \overline{x}_i \mid \overline{x}_i \in S\} = \{i\Pi - \lfloor i\Pi \rfloor \mid i \in \mathbb{N}\}$. Because $i\Pi - \lfloor i\Pi \rfloor \in (0, 1)$ for any $i \in \mathbb{N}$, K is bounded.

Now we are going to show that K is infinite. Assume that K is finite and hence there exists some i and j, $i > j$ such that $i\Pi - \lfloor i\Pi \rfloor = j\Pi - \lfloor j\Pi \rfloor$. So we have that $(i-j)\Pi = \lfloor i\Pi \rfloor - \lfloor j\Pi \rfloor$. As it can be seen, $\lfloor i\Pi \rfloor - \lfloor j\Pi \rfloor \in \mathbb{N}$, whereas $(i-j)\Pi \in \mathbb{R} \setminus \mathbb{Q}$. This means that K is an infinite set.

5.6.2 Finite Extended Timed Labelled Transition System

In this subsection we provide the necessary and sufficient conditions for an extended timed labelled transition system to be finite. We show that if a finite timed labelled transition system has a simple cycle for which the sum over the sequence of numbers along it is positive, then the corresponding extended timed labelled transition system is infinite and the distance between model and realisation is also infinite.

Theorem 5.12 *Let $\mathcal{T}$ be a timed labelled transition system which is finite in the number of states and transitions and let $ExecTime : Act \to \mathbb{Q}^+$ be the function for the execution times of actions. Then, the extended timed labelled transition system $\mathcal{E}^{ExecTime}(\mathcal{T})$ is infinite iff there exists a simple cycle $\mathcal{C}$ in $\mathcal{T}$ such that $\downarrow \overline{\mathcal{C}} > 0$.*

Proof "$\Rightarrow$"

Assume that for any simple cycle $\mathcal{C}$ in $\mathcal{T}$, $\downarrow \overline{\mathcal{C}} = 0$. Moreover, assume that the extended timed labelled transition system $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$ is infinite. We shall prove the theorem by deriving a contradiction.

Since $\mathcal{T}$ is finite, there exists at least one state that appears with infinitely many different time difference labels in $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$. Let S be the set of finite sequences along all the paths in $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$ starting in its initial state. By lemma 5.6, the set $K = \{\sum \overline{x} \mid \overline{x} \in S\}$ is the set of state labels in $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$. Since $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$ is infinite, K is infinite as there are infinitely many state labels in $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$ and by lemma 5.11, K is unbounded.

Assume any infinite path $\mathcal{P}$ through $\mathcal{E}^{\mathrm{ExecTime}}(\mathcal{T})$. Then it contains a state s infinitely many times. Consider $s_{(1)}$ the first occurrence of state s in $\mathcal{P}$. Then the

sequence of states until the first occurrence of s in the path is of the form:

$$s_1 \dots s_1 s_2 \dots s_2 s_3 \dots s_m s_{(1)}$$

where for any $i \neq j$, $s_i \neq s_j \neq s$, and $s_j \dots s_j$ represents a sequence of zero or more states that appear between the first and the last occurrence of s_j in $\mathcal{P}$. If for some $1 \leq j \leq m$ we have $s_j s_{i_j} s_{i_j+1} \dots s_{i_j+n_j} s_j$, then for any $i_j \leq l \leq i_j + n_j$ and for any $k < j$, $s_l \neq s_k$. By definition 5.15, the sequence along this path is $\overline{\mathcal{Q}} = \overline{\mathcal{Q}_1}.\overline{u_1}.\overline{\mathcal{Q}_2}.\overline{u_2} \dots \overline{\mathcal{Q}_m}.\overline{u_m}$ where for any $1 \leq k \leq m$ $\overline{\mathcal{Q}_k}$ corresponds to path $\mathcal{Q}_k$ with state sequence $s_k s_{i_k} \dots s_{i_k+n_k} s_k$, $\overline{u_1}$ corresponds to path u_1 with state sequence $s_1 s_2$, for any $1 < k < m$ $\overline{u_k}$ corresponds to path u_k with state sequence $s_k s_{k+1}$, and $\overline{u_m}$ corresponds to path u_{m+1} with state sequence $s_m s_{(1)}$. By construction, $\mathcal{Q}_k$ is a cycle and because for any simple cycle $\mathcal{C}$ in $\mathcal{T}$, $\downarrow \overline{\mathcal{C}} = 0$, then theorem 5.10 can be applied. Moreover, $\overline{u} = \overline{u_1}.\overline{u_2}.\dots.\overline{u_m}$.

Let $\overline{\mathcal{Q}'}$ be any prefix of $\overline{\mathcal{Q}}$. Then $\overline{\mathcal{Q}'} = \overline{\mathcal{Q}_1}.\overline{u_1}.\overline{\mathcal{Q}_2}.\overline{u_2} \dots \overline{u_{k-1}}.\overline{\mathcal{Q}'_k}$ for some $\mathcal{Q}'_k$ a prefix of the cycle $\mathcal{Q}_k$. Then $\sum \overline{\mathcal{Q}'} = \sum(\overline{\mathcal{Q}_1}.\overline{u_1}.\overline{\mathcal{Q}_2}.\overline{u_2} \dots \overline{u_{k-1}}.\overline{\mathcal{Q}'_k}) \leq \{by\ theorem\ 5.9\} \sum(\overline{u_-}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \sum(\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+})$, where $\overline{u_-}$ represents the sequence of negative numbers from $\overline{u}$ and $\overline{u_+}$, respectively, the sequence of non-negative numbers. By theorem 5.10(2) and taking $a = 0$, $\sum \overline{\mathcal{Q}'} \leq \sum(<\max(0, \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\})> .\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \sum(<\max(0, \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\})> .\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+})$. Let us denote with $m = \sum\{\uparrow \overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}$. Then $\sum \overline{\mathcal{Q}'} \leq \sum(<\max(0, m)> .\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) = \sum(<m> .\overline{\mathcal{Q}_2} \dots \overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \{by\ induction\} \sum(<m> .\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \{by\ theorem\ 5.10(3)\} \sum(<\max(m, m) + m> .\overline{u_+}) = \sum(<2m> .\overline{u_+}) \leq \{by\ lemma\ 5.7\} 2m + \sum \overline{u_+}$. $\sum \overline{u_+}$ is finite because by construction, $\sum \overline{u_+} \leq |\mathcal{S}| \cdot \max\{ExecTime(\gamma) \mid \gamma \in Act\}$, where $|\mathcal{S}|$ is the number of states in $\mathcal{T}$. We denote with $M = \sum \overline{u_+}$. Hence, $\overline{\mathcal{Q}'} \leq 2m + M$ which means that the value of the label of any state in path $\mathcal{Q}'$ is bounded.

As s occurs infinitely often in $\mathcal{P}$, assume $s_{(i)}$ is its i-th occurrence, $i > 1$. We denote with $\overline{z}$ the sequence of numbers along the path from $s_{(1)}$ to $s_{(i)}$ which is a cycle. Then, the labels of states along this path are given by $\sum(\overline{\mathcal{Q}_1}.\overline{u_1}.\overline{\mathcal{Q}_2}.\overline{u_2} \dots \overline{\mathcal{Q}_m}.\overline{u_m}.\overline{z'})$, where $\overline{z'}$ is any prefix of $\overline{z}$. Then $\sum(\overline{\mathcal{Q}_1}.\overline{u_1}.\overline{\mathcal{Q}_2}.\overline{u_2} \dots \overline{\mathcal{Q}_m}.\overline{u_m}.\overline{z'}) \leq \sum(<2m + M> .\overline{z'}) \leq \{by\ theorem\ 5.10(3)\} \max(2m + M, m) + m \leq 3m + M$. Hence, the label of any state $s_{(i)}$ is bounded by $3m + M$. Since we showed above that K is unbounded, we have just proved that any element of K is actually bounded, thus we have a contradiction.

"$\Leftarrow$"

If $\mathcal{C} = (\overline{s}, \overline{\gamma})$ is a simple cycle in $\mathcal{T}$ with the sequence of states of the form $s_1 s_2 \dots s_k s_1$ and the sequence of transition labels $\gamma_1 \gamma_2 \dots \gamma_k$, then there exists a path in $\mathcal{E}^{\text{ExecTime}}(T)$ such that

$$(s_1, d_1) \xrightarrow{\gamma_1}_{\mathcal{E}} (s_2, d_2) \xrightarrow{\gamma_2}_{\mathcal{E}} \dots (s_k, d_k) \xrightarrow{\gamma_k}_{\mathcal{E}} (s_1, d_{k+1}).$$

By lemma 5.6, $d_{k+1} = \sum(<d_1> .\overline{\mathcal{C}})$. By theorem 5.8, we have that the smallest value of this sum is obtained when all the non-negative numbers are on the left

side of the sequence and the negative ones on the right. By the construction of $\overline{\mathcal{C}}$ from definition 5.15, this translates into having all the action transitions one after each other at the beginning of the path and the time transitions one after each other in the second half of the path. Thus, the smallest value of d_{k+1} is obtained when there exists $1 \leq m \leq k+1$ such that for all $1 \leq j < m$, $\gamma_j \in Act$ and for all $m \leq j \leq k$, $\gamma_j \in T^+$ for which we can write $\overline{\mathcal{C}} = \overline{\mathcal{C}_{Act}}.\overline{\mathcal{C}_{T^+}}$. Then $d_{k+1} = \sum(<d_1> .\overline{\mathcal{C}}) = \sum(<d_1> .\overline{\mathcal{C}_{Act}}.\overline{\mathcal{C}_{T^+}}) = \{by\ lemma\ 5.5\} \sum(<\sum(<d_1> .\overline{\mathcal{C}_{Act}})> .\overline{\mathcal{C}_{T^+}})$. Since by construction $\overline{\mathcal{C}_{Act}}$ is made of non-negative numbers, then $\sum(<d_1> .\overline{\mathcal{C}_{Act}}) = d_1 + \sum_{j=1}^{m-1} ExecTime(\gamma_j)$. Moreover, since $\downarrow \overline{\mathcal{C}} > 0$, by definitions 5.16 and 5.17 we have that $\sum_{j=1}^{m-1} ExecTime(\gamma_j) > \sum_{j=m}^{k} \gamma_j$.

By definition 5.16, $d_{k+1} = \sum <d_1 + \sum_{j=1}^{m-1} ExecTime(\gamma_j)> .\overline{\mathcal{C}_{T^+}}) = d_1 + \sum_{j=1}^{m-1} ExecTime(\gamma_j) - \sum_{j=m}^{k} \gamma_j > d_1$. Thus, no matter the order of the transitions in the simple cycle $\mathcal{C}$, in the extended timed labelled transition system $d_{k+1} > d_1$ always and state s_1 appears infinitely many times with different labels. Hence, the extended timed labelled transition system is infinite.

□

Based on this theorem, we have the means to check if we are able to traverse the extended timed labelled transition system and determine the size of the distance between model and realisation. If a simple cycle of positive cost is found in the original transition system, then we have an infinite extended timed labelled transition system and the distance between model and realisation is infinite.

5.7 Algorithm for Computing the Distance Between Model and Realisation

To calculate the distance between a model and its realisation on a target platform, given the timed labelled transition system and the worst-case execution times of each of its actions, the algorithm in figure 5.7 can be used. The algorithm is based on theorems 5.2, 5.3 and 5.12. The input of the method CALCULATEDISTANCEMODELTOREALISATION is the initial state of a finite timed labelled transition system. The algorithm assumes that any state of the transition system has incorporated into it the transitions leaving from it and any transition knows its source state and its destination state. Moreover, an action transition has incorporated into it the worst-case execution time of the action it represents. The output of the algorithm is the distance between the model and the realisation of the system, which is denoted with ϵ.

By calling method EXISTSSIMPLECYCLEWITHPOSITIVECOST, the algorithm first identifies all the simple cycles in the system and checks if the execution times of actions along each such cycle are not compensated by the time delays. In [80] it is shown that the complexity of finding all cycles in a graph is $O((n+l)(c+1))$ where n is the number of nodes in the graph, l is the number of edges and c is the number of cycles. In the worst-case, in [80] it is shown that a graph can have $c = \sum_{i=2}^{n} \binom{n}{i} (i-$

1)! > $(n-1)!$, where $\binom{n}{i}$ represents the number of subsets of i elements from the set of n elements. As the cost of every simple cycle is computed and in the worst-case the length of a cycle is n the number of states in the transition system, the complexity of EXISTSSIMPLECYCLEWITHPOSITIVECOST is $O((n+l)n(c+1))$.

```
CALCULATEDISTANCEMODELTOREALISATION(STATE s_I)
 1  if EXISTSSIMPLECYCLEWITHPOSITIVECOST() == true
 2     then ε ← ∞
 3     else ε ← 0
 4          statesSet ← {(s_I, 0)}
 5          while (statesSet.HASUNVISITEDSTATES())
 6              do (s, d) ← statesSet.GETANUNVISITEDSTATE()
 7                 (s, d).VISITED()
 8                 for each transition t leaving from s in T
 9                     do if t is action transition
10                           then ε ← max(ε, d + t.wcet)
11                                statesSet ← statesSet ∪ {(t.destination, d + t.wcet)}
12                           else statesSet ← statesSet ∪ {(t.destination, d ∸ t.duration)}
13  return ε
```

Figure 5.7: Algorithm to calculate distance between model and realisation

If such a simple cycle with positive cost exists, then the distance between model and realisation denoted with ϵ is inifinite. In case no such cycle is found, the distance between model and realisation is initialised with zero. The set $statesSet$ of states of the extended timed labelled transition system starts being built from the initial state of the original transition system with label zero. Afterwards, as long as there are unvisited elements in $statesSet$, checked with HASUNVISITEDSTATES method, each one is considered. GETANUNVISITEDSTATE is a heuristic method that returns such an unvisited state. All the transitions departing from the corresponding state in the original transition system is taken into account. Each transition is assumed to contain information about its duration, if it is a time transition, respectively worst-case execution time, if it is an action transition, and the destination state in the original system. A state in the extended transition system is created from the destination state and the duration, respectively worst-case execution time, of the transition. A set union operation is performed between the set containing the new state and the set of states already built. If the state is indeed a new state, the union operation also sets it as an unvisited state. The value of ϵ is updated based on the labels of the newly discovered states of the extended timed labelled transition system. When there are no more states to visit, the algorithm returns the value found for the distance between model and realisation as the maximum of all the state labels. For the system given as example in figure 5.2.c, the result returned by the algorithm is 0.32.

If the extended timed labelled transition system is finite, the complexity of the algorithm for calculating the distance between a model and its realisation is $O(n+l+l^2)$, where n is the number of states in the original timed labelled transition system and l is the number of its transitions. This complexity is due to the fact that we take all the states and transitions of the original transition system and for each transition we might create a new state in the extended transition system. Hence we have l more states from which we have at most l transitions. Because there is no simple cycle of positive cost, we have at most one copy

of each state along any simple cycle. Hence, the overall complexity of the algorithm CALCULATEDISTANCEMODELTOREALISATION is $O((n+l)n(c+1)+n+l+l^2)$.

5.8 Simulation-Based Estimation of the Distance Between Model and Realisation

Due to the complexity of the algorithm presented in section 5.7, it might be the case that the extended timed labelled transition system is finite but it takes a very long time (e.g. several hours) to compute the distance between the model and its realisation. Therefore, we have conceived a corresponding simulation approach in order to estimate this distance.

The mechanism that we propose involves a modelling approach similar to the one presented in chapter 3. As we are interested in the synthesis of the software part of a system and in the strength of the preservation of its properties, the Y-chart-based model is chosen because of its separation of the application, which is the counterpart of the timed labelled transition system, the platform and the environment parts. This separation allows us to take directly into account the action transitions in the model of the software of the system as they influence the distance from model to realisation.

The application part of the system is modelled as tasks that communicate with each other and that may be triggered by events from the environment. These tasks correspond to the concurrent processes from which the system is made and for which we showed in section 5.2 how the timed labelled transition system is built. Where applicable, namely for data-intensive kind of systems, the modelling patterns presented in section 3.5.1 can be used, with a little adaptation as seen in figure 5.8. However, the simulation-based approach for the estimation of the distance between model and realisation is applicable to any kind of system, even if the modelling patterns do not apply. As shown in the adapted specification of the aperiodic task in figure 5.8, the main ingredient of the approach is the decoration of any action that in the final product is executed on a processor. The decoration consists in building a `Request` data object that incorporates information, such as the load imposed on the processor, about the specific action. The `Request` object is sent to the processor and in this way the execution of the action is simulated. An important remark is that braces, { and }, are used to ensure that once the action is executed in the model, the request is also sent to the platform in an atomic operation.

The platform part models the execution of the actions specified in the application model and that in the final product are computations executed on a processor. As the synthesis approach presented in section 5.3 is possible only on a single processor, the platform model contains one computational resource and no communication resource. The specification of each of the methods of the POOSL process class `Resource` is shown in figure 5.9. This class differs from the modelling pattern presented in section 3.5.2. From the semantics of the model, the execution of actions is atomic and any interleaving of parallel actions is allowed. Hence, a scheduler, as the one described in section 3.5.2, is not needed for scheduling the actions that occur at the same time in the model. Moreover, the model of the processor does not need to take into account the energy consumption, as that is not the goal of the analysis, nor

```
Aperiodic()() |lat : Real, ev : Event, req : Request|
  {in?event(ev | ev getEventType() = Trigger);
  req := new Request("recv");
  cpu!execute(req)};
  par
    par
      lat := ltcy sample()
      delay lat;
      Behaviour(D, lat, ev)(ev)
    and
      delay D
    rap;
    {out!output(ev);
    req := new Request("send");
    cpu!execute(req) }
  and
    Aperiodic()()
  rap.
```

Figure 5.8: Model of an aperiodic task in the application part

the operating system overhead or effects caused by for example cache or pipeline, because they are assumed to be incorporated in the worst-case execution time.

```
Init()() | |
  epsilon := 0;
  buffer := new Queue init();
  par
    ReceiveRequests()()
  and
    HandleRequests()()
  rap.

ReceiveRequests()() | req: Request |
  task?execute(req);
  buffer put(req);
  ReceiveRequests()().

HandleRequests()() | req: Request |
  req := buffer remove();
  delay req getLoad() / throughput;
  if epsilon < currentTime - req getRequestTime()
    then epsilon := currentTime - req getRequestTime() fi;
  task!stopped(req)
  HandleRequests()().
```

Figure 5.9: Model of the processor

The model of a computation resource differs also in other ways from the one specified as a modelling pattern. An instance variable `epsilon` is added for keeping the estimation of the distance between the application model and its realisation. `epsilon` is set to zero in the initialisation of the process object, which is done by the `Init` method, and its value is updated during the simulation of the model. In order to preserve the timing semantics of the application part of the system, the resource part of the model is supposed to not interfere with the moments in time when the actions should occur. Because the resource is supposed to simulate the execution of these actions, it has to be able to receive requests even when it is busy with another action. Therefore, the parallel composition ensures that the resource is doing both receiving of requests, `ReceiveRequest`, and handling/executing the requests, `HandleRequest`. Once an action transition is taken in the application part, its execution request is also received by the resource and the behaviour of the task is

not affected by the synchronous communication. A buffer is declared as a queue in which (potentially infinitely many) requests are placed and from which they are retrieved by the `HandleRequest` method for their execution in order to establish the distance between the application model and its realisation on the target platform. After the execution of each action, the value of `epsilon` is updated; when the simulation stops it will contain the maximum timing deviation obtained between the model time and the physical time along the path taken by the simulation engine.

An important remark is that if the timed labelled transition system is deterministic, hence only one path is possible through it, and it is finite, the result of the estimation obtained from simulation is the same as the one obtained by analytical computations, as it was the case of the example shown throughout this chapter. Otherwise, the value of the estimation of the distance is smaller or at most as large as the distance calculated using the analytical approach.

5.9 Summary

In this chapter, we have presented an analytical approach for determining the distance between a model and its realisation on a target platform before actually obtaining this realisation. As the distance can be obtained from the model of the system and based on estimations of the worst-case execution times of its actions, this approach saves design cycles, avoiding designers to get into the trouble of actually building a wrong system. The value of this distance is important as it is a measure of how well the properties specified and analysed in the model can be preserved in the realisation. Since in this chapter we proved when the distance between model and realisation can be computed in finite time assuming that the worst-case execution times of actions are rational numbers, as future research we aim to investigate if the same result holds when these are real numbers.

6
Predictable Real-Time Systems Synthesis

An important ingredient of a design methodology for real-time systems is the synthesis step to obtain a "correct" realisation of a system from a model that is analysed and verified. In this chapter, we present an improvement on the efficiency of the model synthesis approach in the Software/Hardware Engineering design methodology.

The chapter is organised as follows. Section 6.1 discusses the formalisation of real-time properties and property preservation. Section 6.2 presents an approach for reducing the distance between a model and its realisation. The improvement of the POOSL model synthesis strategy based on this approach is discussed in section 6.3. Experimental results are presented in section 6.4, whereas related research is given in section 6.5. A summary of the chapter is given in section 6.6.

6.1 Real-Time Properties

The target of the design process of a real-time system is to obtain a product that possesses certain properties that meet the system requirements. The purpose of the analysis is to show what the properties of the system are and to check if they are preserved along the different stages of the design process. Properties are often formalised in certain mathematical frameworks. Temporal logics are a class of mathematical frameworks widely used for the formalisation of real-time properties [16]. Based on the trace structures on which the logics can be applied, temporal logics are classified into linear-time, such as LTL [56], and branching-time logics, such as CTL [29] and CTL* [23]. These logics are used for the formalisation of qualitative properties, referring to the states of a system or the ordering of the actions observed.

As timing is a critical aspect of real-time systems, qualitative temporal logics

were extended by attaching time bounds to express quantitative real-time properties. Such quantitative real-time properties include minimal, maximal and exact time-distance between two actions, and periodicity of an action. Examples of temporal logics to formalise qualitative real-time properties are MITL [10], which is a time-bounded extension of LTL, and RTCTL [30], which is real-time CTL. Detailed surveys on extending temporal logics with time-bounds can be found in [10] and [11].

In this thesis, the analysis of real-time properties does not consider branching structures. Following the work presented in [51], we use a linear-time temporal logic for their formalisation, namely MTL [59]. MTL formulas have the following syntactic forms:

$$\varphi ::= p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \cup_I \varphi_2 \mid \varphi_1 \vee_I \varphi_2 \tag{6.1}$$

where p is an atomic proposition and the time-bound I is an interval of non-negative reals that takes one of the following forms: $\varnothing$, $[a,a]$, $[a,b]$, $[a,b)$, $(a,b]$, (a,b), $[a,\infty)$ and (a,∞), where $a < b$ for $a, b \in \mathbb{R}^+$. MTL formulas are interpreted over sequences of timed states, which can be derived from the paths defined in chapter 5.

In the rest of this section, we first define the encoding of a timed state sequence from a path in subsection 6.1.1. Then we briefly present the interpretation of MTL formulas in subsection 6.1.2, whereas subsection 6.1.3 discusses the prediction of properties preservation along timed labelled transition systems.

6.1.1 Timed State Sequences

Definition 6.1 *A* complete path *through a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{\xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{\xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$ *is a 2-tuple:*

$$\mathcal{P} = (\overline{s}, \overline{\gamma})$$

where:

- $\overline{s}$ *is a finite or countably infinite sequence of states* $s_1, s_2, \ldots$*;*
- $\overline{\gamma}$ *is a finite or countably infinite sequence of transitions* $\gamma_1, \gamma_2, \ldots$

with the property that for any $i \geq 1$ *with* $s_i \in \overline{s}$*, either there exists* $\gamma_i \in Act \cup T^+$ *and* $s_{i+1} \in \mathcal{S}$ *such that* $s_{i+1} \in \overline{s}$*,* $\gamma_i \in \overline{\gamma}$ *and* $s_i \xrightarrow{\gamma_i} s_{i+1}$ *or* s_i *is a state from which the path cannot be extended anymore, and then it is the last state in* $\overline{s}$*.*

A finite path is a complete path that cannot be extended anymore.

Definition 6.2 *Let* $Prop$ *be a set of atomic propositions. A state of a system can be interpreted as a subset of* $Prop$ *which contains precisely the propositions that hold in that state.*

Definition 6.3 *A* timed state sequence *is a 2-tuple:*

$$\tau = (\overline{\sigma}, \overline{I})$$

where:

- *$\overline{\sigma}$ is a finite or countably infinite sequence of states $\sigma_i \subseteq Prop$;*
- *$\overline{I}$ is a finite or countably infinite sequence of non-negative time intervals, where I_i is of the form $[l(I_i), r(I_i))$ or $[l(I_i), r(I_i)]$ or $[l(I_i), \infty)$ and it represents the duration of state σ_i.*

For two consecutive states σ_i and σ_{i+1}, their corresponding time intervals I_i and I_{i+1} are adjacent, meaning that $r(I_i) = l(I_{i+1})$.

A trace in the behaviour of a system can be represented by either a timed state sequence or a path. Since the formalisation of real-time properties preservation which we use in this thesis is based on timed state sequences [51], we need to encode a timed state sequence from a path, as it is shown in [26]. The following definition gives this encoding.

Definition 6.4 *Given a complete path $\mathcal{P} = (\overline{s}, \overline{\gamma})$ through a timed labelled transition system $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$, a time labelling $\mathcal{L}_T(P) = (\overline{s}, \overline{t}, \overline{\gamma})$ and a set of atomic propositions $Prop = \{P_\gamma \mid \gamma \in Act\}$ where P_γ is the atomic proposition "γ is observed", a timed state sequence $\tau_{(\mathcal{P},T)} = (\overline{\sigma}, \overline{I})$ encoded from $\mathcal{P}$ has the following properties:*

- *$\overline{\sigma}$ is a finite or countably infinite sequence of states such that:*
  - *for any $i \geq 1$ $\sigma_i = \varnothing$ iff $\gamma_i \in T^+$, where $\varnothing$ is a state at which no atomic proposition is observed (i.e. no action takes place);*
  - *for any $i \geq 1$ $\sigma_i = \{P_{\gamma_i}\}$ iff $\gamma_i \in Act$;*
  - *when $\mathcal{P}$ is finite, $\sigma_n = \varnothing$ where n is the length of $\mathcal{P}$;*
- *$\overline{I}$ is a finite or countably infinite sequence of non-negative time intervals such that:*
  - *for any $i \geq 1$ $I_i = [t_i, t_{i+1})$ iff $\gamma_i \in T^+$;*
  - *for any $i \geq 1$ $I_i = [t_i, t_{i+1}]$ iff $\gamma_i \in Act$;*
  - *when $\mathcal{P}$ is finite, $I_n = [t_n, \infty)$ where n is the length of $\mathcal{P}$.*

An important remark we need to make is that, by construction, the states of a timed state sequence do not correspond to the states of the timed labelled transition system, but to its action transitions. The reason for this is that the real-time properties of a system are related to the action transitions. Another remark is that because of *maximal progress* for time transitions in a timed labelled transition system, there are no two adjacent states σ_i and σ_{i+1} in $\overline{\sigma}$ such that $\sigma_i = \sigma_{i+1} = \varnothing$.

Example 6.1 *For cycle 5.1 from example 5.2 and the set of atomic propositions $Prop = \{P_{in1}, P_{in2}, P_{out1}, P_{out2}, P_{computation1}, P_{computation2}\}$, the timed state sequence derived is:*

$$\begin{aligned}\tau_{(\mathcal{P},M)} = (\{P_{in1}\}, [0,0])(\{P_{computation1}\}, [0,0])(\{P_{in2}\}, [0,0])(\{P_{computation2}\}, [0,0])\\ (\varnothing, [0,1))(\{P_{out1}\}, [1,1])(\{P_{in1}\}, [1,1])(\{P_{computation1}\}, [1,1])\\ (\varnothing, [1,2))(\{P_{out1}\}, [2,2])(\{P_{out2}\}, [2,2])\ldots\end{aligned}$$

From the labelling in both model and physical time of the same cycle using the execution time function from example 5.5 and the set of atomic propositions $Prop = \{P_{in1}, P_{in2}, P_{out1}, P_{out2}, P_{computation1}, P_{computation2}\}$*, the timed state sequence that can be derived based on the physical time labels is:*

$$\tau_{(\mathcal{P},P)} = (\{P_{in1}\}, [0, 0.01])(\{P_{computation1}\}, [0.01, 0.13])(\{P_{in2}\}, [0.13, 0.14])$$
$$(\{P_{computation2}\}, [0.14, 0.30])(\varnothing, [0.30, 1))(\{P_{out1}\}, [1, 1.01])(\{P_{in1}\}, [1.01, 1.02])$$
$$(\{P_{computation1}\}, [1.02, 1.14])(\varnothing, [1.14, 2))(\{P_{out1}\}, [2, 2.01])(\{P_{out2}\}, [2.01, 2.02])\ldots$$

As states may have attached singular time intervals (a time interval with a single point), at any moment in time the system may be in a number of different states, $\sigma_{k+1}, \sigma_{k+2} \ldots \sigma_{k+m} \subseteq Prop$. We denote with $\tau(t, i), t \in \mathbb{R}^+, i > 0$ the state σ_{k+i} which is i-th state among $\sigma_{k+1}, \sigma_{k+2} \ldots \sigma_{k+m}$.

6.1.2 Interpretation of MTL Logic

The interpretation of MTL formulas over a timed state sequence as shown in [51] is given below. If $\tau = (\overline{\sigma}, \overline{I})$ is a timed state sequence and $t \in I_j$ in $\overline{I}$, then $(\tau, \langle t, i\rangle)$ represents a suffix of τ. This suffix is made of a sequence of states and a sequence of corresponding time intervals. The state sequence is formed by the states in $\overline{\sigma}$ starting with some σ_j which is the i-th state among the states in which the system can be at time t. The sequence of intervals starts with $[t, r(I_j))$ where $r(I_j)$ is the right-side of I_j.

Definition 6.5 *Let φ be an MTL formula and let τ be a timed state sequence derived from a path through a timed labelled transition system. The interpretation of φ over $(\tau, \langle t, i\rangle)$ is given as follows:*

- $(\tau, \langle t, i\rangle) \models p$ *iff* $p \in \tau(t, i)$;
- $(\tau, \langle t, i\rangle) \models \neg p$ *iff* $p \notin \tau(t, i)$;
- $(\tau, \langle t, i\rangle) \models \varphi_1 \vee \varphi_2$ *iff* $(\tau, \langle t, i\rangle) \models \varphi_1$ *or* $(\tau, \langle t, i\rangle) \models \varphi_2$;
- $(\tau, \langle t, i\rangle) \models \varphi_1 \wedge \varphi_2$ *iff* $(\tau, \langle t, i\rangle) \models \varphi_1$ *and* $(\tau, \langle t, i\rangle) \models \varphi_2$;
- $(\tau, \langle t, i\rangle) \models \varphi_1 \cup_I \varphi_2$ *iff there exists some* $t_2 \in I$ *and some* j *such that* $(\tau, \langle t{+}t2, j\rangle) \models \varphi_2$ *and for any* $0 \leq t_1 < t_2$ $(\tau, \langle t + t_1, k\rangle) \models \varphi_1$ *for all possible* k;
- $(\tau, \langle t, i\rangle) \models \varphi_1 \vee_I \varphi_2$ *iff for any* $t_2 \in I$ *and some* j*, either* $(\tau, \langle t + t_2, j\rangle) \models \varphi_2$ *or there exists some* $0 \leq t_1 < t_2$ $(\tau, \langle t + t_1, k\rangle) \models \varphi_1$ *for all possible* k.

We use $(\tau, \langle l(I_1), 1\rangle) \models \varphi$ *(in short* $\tau \models \varphi$*) to denote that the timed state sequence* τ *satisfies the MTL formula* φ.

This definition shows that the interpretation of $\neg$, $\vee$ and $\wedge$ is the same as in traditional logics [79]. The satisfaction of formula $\varphi_1 \cup_I \varphi_2$ means that up to some time $t_2 \in I$, φ_1 is satisfied, and afterwards φ_2 is. The satisfaction of formula $\varphi_1 \vee_I \varphi_2$

true	$T \equiv p \vee \neg p$
false	$F \equiv p \wedge \neg p$
eventually	$\Diamond_I \varphi \equiv T \cup_I \varphi$
always	$\Box_I \varphi \equiv F \vee_I \varphi$
implication	$\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$

Table 6.1: Syntactic abbreviations

means that either for all $t_2 \in I$, φ_2 is satisfied, or there is some point in time when φ_1 is. In case I is $[0, \infty)$, we omit the time-bound I of the operator. Table 6.1 presents some syntactic abbreviations commonly used.

Example 6.2 *Assume that the system in figure 5.2.c has the following set of atomic propositions: $Prop = \{P_{in1}, P_{in2}, P_{out1}, P_{out2}, P_{computation1}, P_{computation2}\}$, where we denote with P_γ the atomic proposition that action "γ is observed". Then, using the execution time function from example 5.5, some MTL properties of the system are:*

$$\Box(P_{in1} \rightarrow \Diamond_{[0.80,1]} P_{out1}); \tag{prop1}$$

$$\Box(P_{in1} \rightarrow \Diamond_{[0,0.17]} P_{computation1}); \tag{prop2}$$

$$\Box(P_{in2} \rightarrow \Diamond_{[1.85,2]} P_{out2}); \tag{prop3}$$

$$\Box(P_{in2} \rightarrow \Diamond_{[0,0.13]} P_{computation2}). \tag{prop4}$$

6.1.3 Preservation of Properties

Since a real-time system can be viewed as a set of timed state sequences, the satisfaction of its real-time properties can be formally defined using the linear trace interpretation of MTL. The formal definition follows.

Definition 6.6 *Let φ be a real-time property expressed as an MTL formula. A real-time system satisfies φ iff for all timed state sequences τ in the behaviour of the system, $\tau \models \varphi$.*

Definition 6.7 *Let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a complete path through a timed labelled transition system, let $\mathcal{L}_{\mathcal{M}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{\gamma})$ be a model time labelling of it and φ be an MTL formula. Then $\mathcal{P}$ satisfies φ in model time, denoted with $\mathcal{P} \models^M \varphi$, iff $\tau_{(\mathcal{P},M)} \models \varphi$.*

Definition 6.8 *Let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a complete path through a timed labelled transition system, and given a function $ExecTime : Act \rightarrow \mathbb{R}^+$ let $\mathcal{L}^{ExecTime}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ be the labelling of $\mathcal{P}$ in both model and physical time and let φ be an MTL formula. Then $\mathcal{P}$ satisfies φ in physical time, denoted with $\mathcal{P} \models^P \varphi$, iff $\tau_{(\mathcal{P},P)} \models \varphi$.*

Given an MTL formula φ, we denote with φ_ϵ an ϵ-weakening of it. In [51], a weakening function over MTL formulas $R^\epsilon(\varphi)$ is defined as follows.

Definition 6.9 *A weakening function $R^\epsilon : MTL \rightarrow MTL$, where $\epsilon \in \mathbb{R}^+$, is recursively defined as:*

- $R^\epsilon(p) = p$;

- $R^\epsilon(\neg p) = \neg p$;
- $R^\epsilon(\varphi_1 \vee \varphi_2) = R^\epsilon(\varphi_1) \vee R^\epsilon(\varphi_2)$;
- $R^\epsilon(\varphi_1 \wedge \varphi_2) = R^\epsilon(\varphi_1) \wedge R^\epsilon(\varphi_2)$;
- $R^\epsilon(\varphi_1 \cup_I \varphi_2) = R^\epsilon(\varphi_1) \cup_{I'} R^\epsilon(\varphi_2)$ *where* $l(I^{'}) = \max(0, l(I) - \epsilon)$ *and* $r(I^{'}) = r(I) + \epsilon$;
- $R^\epsilon(\varphi_1 \vee_I \varphi_2) = R^\epsilon(\varphi_1) \vee_{I'} R^\epsilon(\varphi_2)$ *where* $l(I^{'}) = l(I) + \epsilon$, $r(I^{'}) = r(I) - \epsilon$ *and if* $l(I^{'}) > r(I^{'})$ *then* $I^{'} = \varnothing$.

Definition 6.10 *Let* $\tau_1 = (\overline{\sigma}_1, \overline{I}_1)$ *and* $\tau_2 = (\overline{\sigma}_2, \overline{I}_2)$ *be two timed state sequences. The distance between them is defined as:*

$$\mathcal{D}(\tau_1, \tau_2) = \begin{cases} \sup_{i \geq 1}(|l(I_{1i}) - l(I_{2i})|), & \text{if } \overline{\sigma}_1 = \overline{\sigma}_2 \\ \infty, & \text{otherwise.} \end{cases} \tag{6.2}$$

where $l(I_i)$ *denotes the left-hand side of an interval* I_i.

Based on this distance metric, in [51] it is shown that if τ_1 satisfies a property φ, $\tau_1 \models \varphi$, then τ_2 satisfies a $2 * \mathcal{D}(\tau_1, \tau_2)$-weakening of φ, denoted with $\tau_2 \models \varphi_{2*\mathcal{D}(\tau_1,\tau_2)}$. However, the distance metric between corresponding paths in the model and in the realisation of a system is the same as the distance metric between the timed state sequences derived from those paths as we are going to show here. Based on this we are able to establish a property preservation relation between model and realisation along a complete path through a timed labelled transition system.

Lemma 6.1 *Given a complete path* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *through a timed labelled transition system* $\mathcal{T}$ *and a function* $ExecTime : Act \to \mathbb{R}^+$, *let* $\mathcal{L}^{ExecTime}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ *be the labelling of* $\mathcal{P}$ *in both model and physical time. Then* $\mathcal{D}(\tau_{(\mathcal{P},M)}, \tau_{(\mathcal{P},P)}) = d^{ExecTime}(\mathcal{P})$.

Proof By definition 6.4 of the encoding of a complete path into a timed state sequence, $\tau_{(\mathcal{P},M)} = (\overline{\sigma}_M, \overline{I}_M)$ and $\tau_{(\mathcal{P},P)} = (\overline{\sigma}_P, \overline{I}_P)$ have the same sequence of states. Hence, $\overline{\sigma}_M = \overline{\sigma}_P$. Moreover, for any $i \geq 1$, if $\gamma_i \in Act$ then $I_{Mi} = [t_{Mi}, t_{Mi+1}]$ and $I_{Pi} = [t_{Pi}, t_{Pi+1}]$, whereas if $\gamma_i \in T^+$ then $I_{Mi} = [t_{Mi}, t_{Mi+1})$ and $I_{Pi} = [t_{Pi}, t_{Pi+1})$. Then[1]

$$\mathcal{D}(\tau_{(\mathcal{P},M)}, \tau_{(\mathcal{P},P)}) = \sup_{i\geq 1}(|l(I_{Mi}) - l(I_{Pi})|) = \sup_{i \geq 1}(|t_{Mi} - t_{Pi}|) = \\ = \sup_{i\geq 1}(t_{Pi} - t_{Mi}) = d^{\text{ExecTime}}(\mathcal{P}).$$

□

Theorem 6.2 *Let* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *be a complete path through a timed labelled transition system. Given an execution time function* $ExecTime : Act \to \mathbb{R}^+$ *and an MTL formula* φ, *then* $\mathcal{P} \models^P \varphi_{2*d^{ExecTime}(\mathcal{P})}$ *if* $\mathcal{P} \models^M \varphi$.

[1] Since the path may be infinite, the max in definition 5.11 is replaced with sup.

Proof By definition 6.7, $\mathcal{P} \models^M \varphi$ implies that $\tau_{(\mathcal{P},M)} \models \varphi$. By the property preservation between timed state sequences proved in [51], $\tau_{(\mathcal{P},P)} \models \varphi_{2*\mathcal{D}(\tau_{(\mathcal{P},M)},\tau_{(\mathcal{P},P)})}$. Since by lemma 6.1, $\mathcal{D}(\tau_{(\mathcal{P},M)},\tau_{(\mathcal{P},P)}) = d^{\text{ExecTime}}(\mathcal{P})$, we have that $\tau_{(\mathcal{P},P)} \models \varphi_{2*d^{\text{ExecTime}}(\mathcal{P})}$ which means, by definition 6.8, that $\mathcal{P} \models^P \varphi_{2*d^{\text{ExecTime}}(\mathcal{P})}$. □

With this theorem, we have shown that the results regarding property preservation between timed state sequences are applicable to paths as well. If a real-time property is satisfied along a path in a timed labelled transition system, a two times the distance between model and realisation along the path weakening of the property is satisfied by the corresponding path in the realisation of the system. Based on this result, we can also prove the property preservation between model and realisation for a timed labelled transition system.

Definition 6.11 *Let $\mathcal{T}$ be a timed labelled transition system and φ be an MTL formula. Then $\mathcal{T}$ satisfies φ in model time, denoted with $\mathcal{T} \models^M \varphi$, iff for any complete path $\mathcal{P}$ from $\mathcal{T}$, $\mathcal{P} \models^M \varphi$.*

Definition 6.12 *Let $\mathcal{T}$ be a timed labelled transition system and φ be an MTL formula. Then $\mathcal{T}$ satisfies φ in physical time, denoted with $\mathcal{T} \models^P \varphi$, iff for any complete path $\mathcal{P}$ from $\mathcal{T}$, $\mathcal{P} \models^P \varphi$.*

Theorem 6.3 *Let $\mathcal{T}$ be a timed labelled transition system. Given an execution time function $ExecTime : Act \rightarrow \mathbb{R}^+$ and an MTL formula φ, then $\mathcal{T} \models^P \varphi_{2*d^{\text{ExecTime}}(\mathcal{T})}$ if $\mathcal{T} \models^M \varphi$.*

Proof The proof is similar to the one for theorem 6.2. □

Based on this result, given a timed labelled transition system, we can derive its properties and we can predict the preservation of them between model and realisation based on the distance between the timing behaviour of the system in the model and in the realisation.

6.2 Distance Reduction from Model to Realisation

Usually, the real-time properties of a system are related to what a user can see by interacting with it. The actions performed by the system that a user can "see" are called *observable* actions. To express the property preservation between model and realisation, we need to consider the observable behaviour of the system which is defined with respect to an observer. Depending on what properties are of interest, at a coarse level of granularity, an observer can "see" the inputs and the outputs of the whole system. At a finer level of granularity, properties related to the communication taking place between the components of the system might also be interesting. In that case, these communication actions are considered observable. However, typically the computations that different components of the system perform are considered internal, and thus not observable. In this work, we assume that based on the properties of interest for a system, the designer defines which actions are considered observable and respectively unobservable.

Example 6.3 *Among the properties presented in example 6.2, properties (prop1) and (prop3) are real-time properties referring to observable actions of the system. Actions $computation1$ and $computation2$ are internal computations and they cannot be observed from the outside of the system. If we consider the timed labelled transition system in figure 5.2.c and the execution time function given in example 5.5, based on the value of the calculated distance between model and realisation, we can predict that the weakening of the properties in the realisation is 0.32. However, when we look carefully at the ordering of the observable and the unobservable actions in each path of the system, we notice that before any time transition there is always an unobservable action. Since these unobservable actions are taken into account in the calculation of the distance between model and realisation, they in fact weaken the prediction of the observable properties of the system.*

The distance metric together with the synthesis approach that we have presented so far in this thesis do not distinguish between the observable and the unobservable actions, although the unobservable actions might weaken the prediction of the property preservation. In this section, we present an approach to make better predictions about the preservation of properties.

In subsection 6.2.1, we introduce a new metric to characterise the preservation of observable properties. Subsection 6.2.2 discusses a reduction of a timed labelled transition system that enables stronger properties preservation in the realisation of a system, whereas subsection 6.2.3 presents an approach for generating such a reduced timed labelled transition system.

6.2.1 New Proximity Metric Between Model and Realisation

In the model synthesis approach presented in chapter 5 section 5.3, the proximity metric considers that all actions, both observable and unobservable, affect the prediction of the properties preservation between model and realisation. Both types of actions are considered equally important and the synthesis mechanism treats them accordingly. Thus, it is possible that the observable properties get unnecessarily weakened. Therefore, we define a new proximity metric based on the separation of observable and unobservable actions to express observable property preservation.

Definition 6.13 *Let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a complete path through a timed labelled transition system whose set of actions is $Act = ObsAct \cup UnobsAct$, where $ObsAct$ is the set of observable actions and $UnobsAct$ is the set of unobservable actions. Given the labelling of $\mathcal{P}$ in both model and physical time $\mathcal{L}^{\mathit{ExecTime}}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ based on a function $ExecTime : Act \to \mathbb{R}^+$, the* observable distance between the timing behaviour of the model and of the realisation along this path *is given by:*

$$d^{*\mathit{ExecTime}}(\mathcal{P}) = \sup_{1 \leq i, \gamma_i \in ObsAct} (t_{P_{i+1}} - t_{M_{i+1}}).$$

Example 6.4 *The observable distance between model and realisation along cycle 5.1 in example 5.2, based on the execution time function from example 5.5, is:*

$$d^*(\mathcal{P}) = \sup\{0.01\ ,\ 0.14\ ,\ 0.01\ ,\ 0.02\ ,\ 0.01\ ,\ 0.02\ ,\ 0.03\ ,\ 0.16\ ,\ 0.01\ ,\ 0.02\ ,\ \ldots\} = 0.16$$

By definition, the observable distance along a path is a measure of how much the finishing time of the execution of observable actions deviates in the realisation from the model. This metric has the potential of improving the prediction of the observable property preservation as we show below.

Lemma 6.4 *Given a complete path $\mathcal{P}$ through a timed labelled transition system and an execution time function $ExecTime : Act \to \mathbb{R}^+$, the observable distance between the timing behaviour of the model and of the realisation along this path is at most as large as the distance given by definition 5.11:*

$$d^{*ExecTime}(\mathcal{P}) \leq d^{ExecTime}(\mathcal{P}).$$

Proof The proof comes directly from the definitions of the distance and of the observable distance, based on the observation that the observable actions in a timed labelled transition system are a subset of the set of all actions.

□

In order to use the observable distance metric for the prediction of the observable properties preservation, we need to define a new encoding of a timed state sequence from a complete path.

Definition 6.14 *Let $\mathcal{P} = (\overline{s}, \overline{\gamma})$ be a complete path through a timed labelled transition system whose set of actions is $Act = ObsAct \cup UnobsAct$, where $ObsAct$ is the set of observable actions and $UnobsAct$ is the set of unobservable actions. Given a time labelling $\mathcal{L}_\mathcal{T}(\mathcal{P}) = (\overline{s}, \overline{t}, \overline{\gamma})$ and a set of atomic propositions $Prop = \{P_\gamma \mid \gamma \in ObsAct\}$ where P_γ is the atomic proposition "γ is observed", an* observable timed state sequence *$\tau^*_{(\mathcal{P},T)} = (\overline{\sigma}, \overline{I})$ encoded from $\mathcal{P}$ has the following properties:*

- *$\overline{\sigma}$ is a finite or countably infinite sequence of states such that:*
  - *for any $i \geq 1$ $\sigma_i = \{P_{\gamma_k}\}$ iff there exists some k with $\gamma_k \in ObsAct$ and for any $l < k$ with $\gamma_l \in ObsAct$ there exists some $j < i$ such that $\sigma_j = \{P_{\gamma_l}\}$;*
  - *$\sigma_1 = \varnothing$, where $\varnothing$ is a state at which no atomic proposition is observed, iff there exists some j such that $\gamma_1, \gamma_2 \ldots \gamma_j \in T^+ \cup UnobsAct$ and either $\gamma_{j+1} \in ObsAct$ or the length of $\overline{\gamma}$ is j;*
  - *for any $i > 1$ $\sigma_i = \varnothing$, where $\varnothing$ is a state at which no atomic proposition is observed, iff*
    - * *there exists some k with $\sigma_{i-1} = \{P_{\gamma_k}\}$ and some j such that $\gamma_{k+1}, \gamma_{k+2} \ldots \gamma_{k+j} \in T^+ \cup UnobsAct$ and either $\gamma_{k+j+1} \in ObsAct$ or the length of $\overline{\gamma}$ is $k + j$;*
    - * *there exists some k such that $\sigma_{i-1} = \{P_{\gamma_k}\}$ and for all $j > k$ $\gamma_j \in T^+ \cup UnobsAct$;*
- *$\overline{I}$ is a finite or countably infinite sequence of non-negative time intervals such that:*
  - *for any $i \geq 1$ $I_i = [t_k, t_{k+1}]$ iff there exists some k with $\gamma_k \in ObsAct$;*
  - *$I_1 = [t_1, t_{j+1}]$ iff there exists some j such that $\gamma_1, \gamma_2 \ldots \gamma_{j-1} \in T^+ \cup UnobsAct$, $\gamma_j \in UnobsAct$ and $\gamma_{j+1} \in ObsAct$;*

- $I_1 = [t_1, t_{j+1})$ *iff there exists some* j *such that* $\gamma_1, \gamma_2 \dots \gamma_{j-1} \in T^+ \cup UnobsAct$, $\gamma_j \in T^+$ *and* $\gamma_{j+1} \in ObsAct$;
- $I_1 = [t_1, \infty)$ *iff all* $\gamma_j \in \overline{\gamma}$, $\gamma_j \in\in T^+ \cup UnobsAct$;
- *for any* $i > 1$ $I_i = [t_{k+1}, t_{k+j+1}]$ *iff there exists some* k *and* j *such that* $\gamma_k, \gamma_{k+j+1} \in ObsAct$, $\gamma_{k+1}, \gamma_{k+2} \dots \gamma_{k+j-1} \in T^+ \cup UnobsAct$ *and* $\gamma_{k+j} \in UnobsAct$;
- *for any* $i > 1$ $I_i = [t_{k+1}, t_{k+j+1})$ *iff there exists some* k *and* j *such that* $\gamma_k, \gamma_{k+j+1} \in ObsAct$, $\gamma_{k+1}, \gamma_{k+2} \dots \gamma_{k+j-1} \in T^+ \cup UnobsAct$ *and* $\gamma_{k+j} \in T^+$;
- *for some* $i \geq 1$ $I_i = [t_{k+1}, \infty)$ *iff there exists some* k *such that* $\gamma_k \in ObsAct$ *and for all* $j > k$, $\gamma_j \in T^+ \cup UnobsAct$.

Example 6.5 *For cycle 5.1 from example 5.2 and the set of atomic propositions* $Prop = \{P_{in1}, P_{in2}, P_{out1}, P_{out2}\}$*, the observable timed state sequence derived from it is:*

$$\tau^*_{(\mathcal{P},M)} = (\{P_{in1}\}, [0,0])(\varnothing, [0,0])(\{P_{in2}\}, [0,0])(\varnothing, [0,1))(\{P_{out1}\}, [1,1])(\{P_{in1}\}, [1,1])$$
$$(\varnothing, [1,2))(\{P_{out1}\}, [2,2])(\{P_{out2}\}, [2,2]) \dots$$

Lemma 6.5 *Given a complete path* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *through a timed labelled transition system* $\mathcal{T}$ *and a function* $ExecTime : Act \to \mathbb{R}^+$*, let* $\mathcal{L}^{ExecTime}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ *be the labelling of* $\mathcal{P}$ *in both model and physical time. Then* $\mathcal{D}(\tau^*_{(\mathcal{P},M)}, \tau^*_{(\mathcal{P},P)}) = d^{*ExecTime}(\mathcal{P})$.

Proof The proof comes from definitions 6.10, 6.13 and 6.14.

□

Since we are interested in the preservation of the observable properties, we need to define the notion of a path satisfying an observable property based on the encoding of the path into an observable timed state sequence.

Definition 6.15 *Let* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *be a complete path through a timed labelled transition system, let* $\mathcal{L}_{\mathcal{M}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{\gamma})$ *be the model time labelling of it and* φ *be an MTL formula. Then* $\mathcal{P} \models^M \varphi$ *iff* $\tau^*_{(\mathcal{P},M)} \models \varphi$.

Definition 6.16 *Let* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *be a complete path through a timed labelled transition system, and given a function* $ExecTime : Act \to \mathbb{R}^+$ *let* $\mathcal{L}^{ExecTime}_{\mathcal{MP}}(\mathcal{P}) = (\overline{s}, \overline{t_M}, \overline{t_P}, \overline{\gamma})$ *be the labelling of* $\mathcal{P}$ *in both model and physical time and let* φ *be an MTL formula. Then* $\mathcal{P} \models^P \varphi$ *iff* $\tau^*_{(\mathcal{P},P)} \models \varphi$.

Theorem 6.6 *Let* $\mathcal{P} = (\overline{s}, \overline{\gamma})$ *be a complete path through a timed labelled transition system. Given an execution time function* $ExecTime : Act \to \mathbb{R}^+$ *and an MTL formula* φ*, then* $\mathcal{P} \models^P \varphi_{2*d^{*ExecTime}(\mathcal{P})}$ *if* $\mathcal{P} \models^M \varphi$.

Proof The proof follows from definitions 6.15, 6.16 and lemma 6.5.

□

Definition 6.17 *Given a timed labelled transition system* $\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\})$ *and a function* $ExecTime : Act \rightarrow \mathbb{R}^+$*, the observable distance between the timing behaviour of the model and that of the realisation is given by:*

$$d^{*ExecTime}(\mathcal{T}) = \sup\{d^{*ExecTime}(\mathcal{P}) \mid \mathcal{P} \text{ is a complete path of } \mathcal{T} \text{ starting in } s_I\}.$$

Theorem 6.7 *Let* $\mathcal{T}$ *be a timed labelled transition system. Given an execution time function* $ExecTime : Act \rightarrow \mathbb{R}^+$ *and an MTL formula* φ*, then* $\mathcal{T} \models^P \varphi_{2*d^{*ExecTime}(\mathcal{T})}$ *if* $\mathcal{T} \models^M \varphi$*.*

Proof The proof is similar to the one for theorem 6.6.

□

By lemma 6.4 and theorem 6.7, we obtain that the observable distance metric introduced in this section enables a better prediction of the observable properties preservation in the context of the current synthesis approach.

6.2.2 Reduction of a Timed Labelled Transition System

In this subsection, we are going to show how the preservation of the observable real-time properties of a system can be strengthened in its realisation by building a reduced timed labelled transition system. We first define a reduced timed labelled transition system and then we prove that the observable properties are preserved stronger by the realisation of this transition system compared to the original one.

Definition 6.18 *Given a timed labelled transition system*

$$\mathcal{T} = (\mathcal{S}, s_I, Act, T^+, \{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\}),$$

a reduced timed labelled transition system is a 6-tuple:

$$\mathcal{R}(\mathcal{T}) = (\mathcal{S}^{\mathcal{R}}, s_I^{\mathcal{R}}, Act^{\mathcal{R}}, T^{+\mathcal{R}}, \{ \xrightarrow[Act^{\mathcal{R}}]{a}{}^{\mathcal{R}} \subseteq \mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mid a \in Act^{\mathcal{R}}\},$$
$$\{ \xrightarrow[T^{+\mathcal{R}}]{t}{}^{\mathcal{R}} \subseteq \mathcal{S}^{\mathcal{R}} \times \mathcal{S}^{\mathcal{R}} \mid t \in T^{+\mathcal{R}}\})$$

where $s_I^{\mathcal{R}} = s_I$*,* $\mathcal{S}^{\mathcal{R}} = \mathcal{S}$*,* $Act^{\mathcal{R}} \subseteq Act$*,* $T^+ = T^{+\mathcal{R}}$*,* $\xrightarrow[Act^{\mathcal{R}}]{a}{}^{\mathcal{R}} \subseteq \xrightarrow[Act]{a}$ *, and* $\xrightarrow[T^{\mathcal{R}+}]{t}{}^{\mathcal{R}} \subseteq \xrightarrow[T^+]{t}$ *, such that for any* $s \in \mathcal{S}$ *with* $\gamma_1, \gamma_2 \ldots \gamma_n \in Act \cup T^+$ *transitions departing from* s *in* $\mathcal{T}$ *there exists at least one transition* $\gamma \in Act^{\mathcal{R}} \cup T^{+\mathcal{R}}$ *departing from* s *in* $\mathcal{R}(\mathcal{T})$ *such that* $\gamma = \gamma_i$ *for some* $1 \leq i \leq n$*.*

In other words, a reduced timed labelled transition system keeps at least one transition departing from each state in the original timed labelled transition system. By construction, the relation between a reduced timed labelled transition

system and its corresponding original timed labelled transition system is a strong timed simulation [68]. It means that for any $s_1, s_2 \in \mathcal{S}^\mathcal{R}$ with $s_1 \xrightarrow{\gamma}{}^{\mathcal{R}} s_2$, where $\gamma \in Act^\mathcal{R} \cup T^{+\mathcal{R}}$, we also have $s_1 \xrightarrow{\gamma} s_2$. Hence, any path in the reduced timed labelled transition system is a path in the original timed labelled transition system. Moreover, there exists at least one complete path in $\mathcal{T}$ that can be found in $\mathcal{R}(\mathcal{T})$.

Theorem 6.8 *Given a timed labelled transition system $\mathcal{T}$, let $\mathcal{R}(\mathcal{T})$ be its reduced timed labelled transition system and let φ be an MTL formula. Then $\mathcal{R}(\mathcal{T}) \models^M \varphi$ if $\mathcal{T} \models^M \varphi$.*

Proof By definition, $\mathcal{T} \models^M \varphi$ iff for all paths $\mathcal{P}$ in $\mathcal{T}$, $\mathcal{P} \models^M \varphi$. By construction of the reduced timed labelled transition system, any path $\mathcal{P}_\mathcal{R}$ in $\mathcal{R}(\mathcal{T})$ is also a path in $\mathcal{T}$. Hence, for all paths $\mathcal{P}_\mathcal{R}$, $\mathcal{P}_\mathcal{R} \models^M \varphi$ which implies that $\mathcal{R}(\mathcal{T}) \models^M \varphi$.

□

Since we showed that a reduced timed labelled transition system $\mathcal{R}(\mathcal{T})$ satisfies the same properties as the original one $\mathcal{T}$, we can also establish a relation between the prediction of property preservation based on $\mathcal{T}$ and the one based on $\mathcal{R}(\mathcal{T})$. Given an execution time function $ExecTime : Act \rightarrow \mathbb{R}^+$ and an MTL formula φ, $\mathcal{T} \models^M \varphi$ implies that $\mathcal{T} \models^P \varphi_\epsilon$ where $\epsilon = 2 * d^{*\text{ExecTime}}(\mathcal{T})$. Moreover, by theorem 6.8, $\mathcal{R}(\mathcal{T}) \models^M \varphi$, which implies that $\mathcal{R}(\mathcal{T}) \models^P \varphi_{\epsilon_\mathcal{R}}$ where $\varphi_{\epsilon_\mathcal{R}} = 2 * d^{*\text{ExecTime}}(\mathcal{R}(\mathcal{T}))$. Since by construction the set of all paths in $\mathcal{R}(\mathcal{T})$ is a subset of all paths in $\mathcal{T}$, $d^{*\text{ExecTime}}(\mathcal{R}(\mathcal{T})) \leq d^{*\text{ExecTime}}(\mathcal{T})$. Hence, $\epsilon_\mathcal{R} \leq \epsilon$, as visualised in figure 6.1.

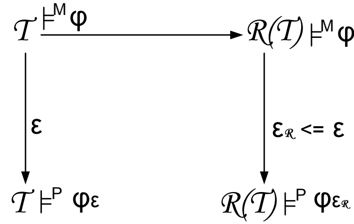


Figure 6.1: Strengthening prediction of property preservation

In order to increase the chances of obtaining a smaller distance between a model and its realisation, we can build the reduced timed labelled transition system in the following way. Given a timed labelled transition system

$$\mathcal{T} = (\mathcal{S}, s_I, Act = ObsAct \cup UnobsAct, T^+,$$
$$\{ \xrightarrow[Act]{a} \subseteq \mathcal{S} \times \mathcal{S} \mid a \in Act\}, \{ \xrightarrow[T^+]{t} \subseteq \mathcal{S} \times \mathcal{S} \mid t \in T^+\}),$$

the reduced timed labelled transition system is a 6-tuple:

$$\mathcal{R}(\mathcal{T}) = (\mathcal{S}^\mathcal{R}, s_I^\mathcal{R}, Act^\mathcal{R} = ObsAct^\mathcal{R} \cup UnobsAct^\mathcal{R}, T^{+\mathcal{R}},$$
$$\{ \xrightarrow[Act^\mathcal{R}]{a}{}^\mathcal{R} \subseteq \mathcal{S}^\mathcal{R} \times \mathcal{S}^\mathcal{R} \mid a \in Act^\mathcal{R}\}, \{ \xrightarrow[T^{+\mathcal{R}}]{t}{}^\mathcal{R} \subseteq \mathcal{S}^\mathcal{R} \times \mathcal{S}^\mathcal{R} \mid t \in T^+\})$$

where $s_I^{\mathcal{R}} = s_I$, $\mathcal{S}^{\mathcal{R}} = \mathcal{S}$, $Act^{\mathcal{R}} \subseteq Act$, $T^+ = T^{+\mathcal{R}}$, $\xrightarrow[Act^{\mathcal{R}}]{a}{}^{\mathcal{R}} \subseteq \xrightarrow[Act]{a}$, and $\xrightarrow[T^{\mathcal{R}+}]{t}{}^{\mathcal{R}} \subseteq \xrightarrow[T^{+}]{t}$, such that for any $s \in \mathcal{S}$ with $\gamma_1, \gamma_2 \dots \gamma_n \in Act \cup T^+$ transitions departing from s in $\mathcal{T}$:

- if some $\gamma_i, \gamma_{i+1} \dots \gamma_{i+m} \in ObsAct$ with $1 \leq i \leq n - m$, then $\gamma_i, \gamma_{i+1} \dots \gamma_{i+m} \in ObsAct^{\mathcal{R}}$ are transitions departing from s in $\mathcal{R}(\mathcal{T})$;
- else, if some $\gamma_i, \gamma_{i+1} \dots \gamma_{i+m} \in UnobsAct$ with $1 \leq i \leq n - m$, then $\gamma_i, \gamma_{i+1} \dots \gamma_{i+m} \in UnobsAct^{\mathcal{R}}$ are transitions departing from s in $\mathcal{R}(\mathcal{T})$;
- else, $n = 1$ and $\gamma_1 \in T^+$, then γ_1 is a transition departing from s in $\mathcal{R}(\mathcal{T})$.

With this construction, we impose priority of the execution of the observable actions over the unobservable actions, which can lead to a smaller time deviation between model and realisation, as shown in the example below.

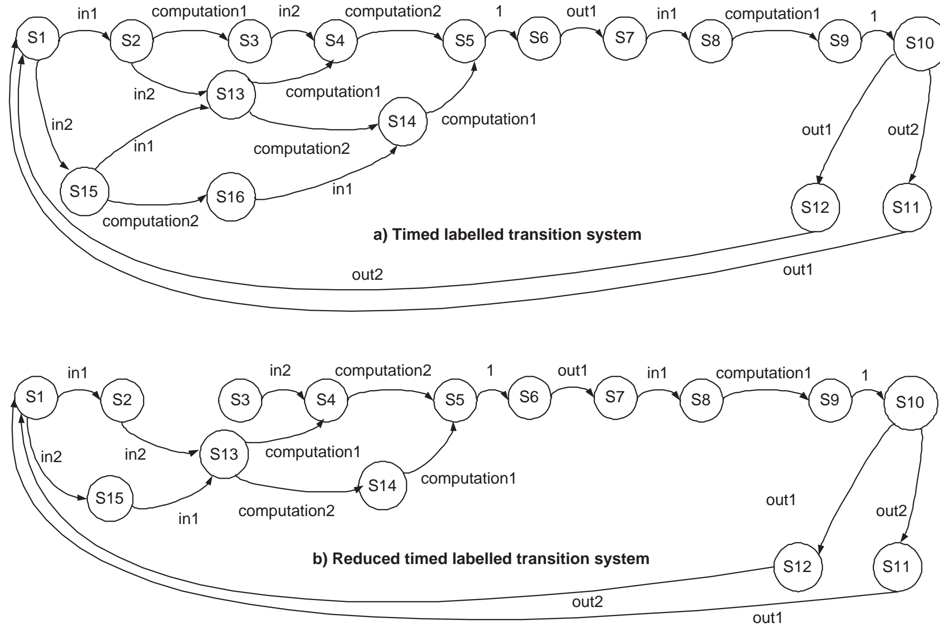


Figure 6.2: Example of building a reduced timed labelled transition system

Example 6.6 *For the timed labelled transition system from figure 5.2.c, depicted again in figure 6.2.a, the corresponding reduced timed labelled transition system is given in figure 6.2.b. The observable distance between model and realisation along the original timed labelled transition system based on the execution time function from example 5.5 is 0.20, whereas the observable distance along the reduced timed labelled transition system using the same function is 0.04.*

6.2.3 Changing Action Ordering

In this subsection, we present an algorithm that, given a timed labelled transition system, builds a reduced timed labelled transition system based on the observations made in the previous subsection. For this reduced timed labelled transition system we have shown that it might enable a stronger property preservation between model and realisation.

As already discussed in chapter 5, process execution trees (PETs) are used for representing the timed labelled transition system of a model in POOSL. The state of each POOSL process is represented by a tree structure, where each leaf is a statement and internal nodes represent compositions of their children. During the evolution of the system, the PETs send *action requests* and/or time *delay requests* to the scheduler. To discriminate between observable and unobservable actions, the PETs send *observable action requests* and *unobservable action requests*. By imposing priority on the execution of the observable actions, the PET scheduler, whose behaviour is described by the algorithm in figure 6.3, asynchronously grants all eligible atomic observable actions. When there are no more observable actions available, unobservable actions are granted until an observable one becomes eligible or until there are no more unobservable actions either. In this way, the scheduler ensures action urgency based on a two-level priority scheme. The internal state of each PET is dynamically changed according to the choices made by the PET scheduler and new requests may be sent to the scheduler. When no action of any kind is possible, based on the shortest delay request, time passes synchronously for all PETs until an action becomes eligible again. The value of the variable $modelTime$, which keeps track of the model time, is updated based on the value of the granted delay.

```
PETSCHEDULER(LIST observableActions, LIST unobservableActions, LIST delays)
 1  modelTime ← 0
 2  while (true)
 3      do while (observableActions NOTEMPTY())
 4              do observableActions GETASYNCHRONOUSLY()->GRANT();
 5          if (unobservableActions NOTEMPTY())
 6            then unobservableActions GETASYNCHRONOUSLY()->GRANT()
 7            else
 8                  if (delays NOTEMPTY())
 9                    then modelTime ← modelTime + delays GETSMALLEST()->AMOUNTOFTIME();
10                    else DEADLOCK();
11                         return
```

Figure 6.3: The PET scheduler

From such a reduced timed labelled transitions system, we can build the corresponding extended timed labelled transition system in the way shown in definition 5.14. From this one, we can calculate the observable distance between model and realisation as the upper-bound on the labels of the states, reachable from the initial state, in which the system can arrive by performing an observable action transition. With a small modification, the algorithm presented in figure 5.7 to determine the distance between model and realisation can be used to calculate the observable distance as well. The change consists in updating the value of ϵ only when observable action transitions are encountered in the transition system. The algorithm is

```
CALCULATEOBSERVABLEDISTANCEMODELTOREALISATION(STATE s0)
 1  if EXISTSELEMENTARYCYCLEWITHPOSITIVECOST() > 0
 2     then ε ← ⊥
 3     else ε ← 0
 4          statesSet ← ∅
 5          statesSet ← statesSet ∪ {(s0, 0)}
 6          while (statesSet.HASUNVISITEDSTATES())
 7             do (s, d) ← statesSet.GETANUNVISITEDSTATE()
 8                for each transition t leaving from s in T
 9                    do if t is observable action transition
10                          then ε ← max(ε, d + t.EXECTIME)
11                       if t is action transition
12                          then statesSet ← statesSet ∪ {(t.destination, d + t.wcet)}
13                          else statesSet ← statesSet ∪ {(t.destination, d ∸ t.duration)})
14  return ε
```

Figure 6.4: Algorithm to calculate observable distance between model and realisation

depicted in figure 6.4 and has the same complexity as the one in figure 5.7. For the system given in figure 6.2.a, the algorithm determined that the size of the observable distance between model and realisation is 0.04.

6.3 Improved POOSL Model Synthesis Strategy

Based on the mechanism of strengthening the observable property preservation that we have discussed in section 6.2, we have extended the PET scheduler in Rotalumis such that it can cope differently with observable and unobservable actions and it can enforce urgency on the execution of observable actions. With a proper design time annotation of the model of a real-time system, observable and unobservable action requests that arrive at the scheduler are placed into different lists. Whenever observable actions are available for execution, they are granted by the PET scheduler. When no observable action is eligible, an unobservable action request is granted, and then the observable action requests list is checked again. The algorithm for the extended PET scheduler of Rotalumis is presented in figure 6.5. When no action request of any kind is available, the smallest time transition in the system is granted. Model time passes synchronously for all PETs, whereas physical time is synchronised with the model time. When some action becomes eligible again, the algorithm is resumed. In this way, a realisation of the system is built by always choosing the path that contains eligible observable actions in front of unobservable actions. Besides generating a realisation of the system, the algorithm also determines the value of the observable distance, which is updated in the variable called *epsilon*.

6.4 Experimental Results

To illustrate the improvement that can be obtained in the strength of property preservation of a system by using the approach proposed in this chapter, we considered a very simple motion control system that is an instantiation of the timed labelled tran-

```
PETSCHEDULERROTALUMIS(LIST observableActions, LIST unobservableActions, LIST delays)
 1  modelTime ← 0
 2  startTime ← READPHYSICALTIME()
 3  epsilon ← 0
 4  while (true)
 5      do while (observableActions NOTEMPTY())
 6              do observableActions GETASYNCHRONOUSLY()->GRANT();
 7         if (an observableAction was granted) && (epsilon < READPHYSICALTIME() - modelTime)
 8           then epsilon ← READPHYSICALTIME() − modelTime
 9         if (unobservableActions NOTEMPTY())
10           then unobservableActions GETASYNCHRONOUSLY()->GRANT()
11           else
12                 if (delays NOTEMPTY())
13                   then modelTime ← modelTime + delays GETSMALLEST()->AMOUNTOFTIME()
14                         /* synchronisation between model and physical time */
15                         if modelTime > READPHYSICALTIME() - startTime
16                           then wait_until modelTime == READPHYSICALTIME() - startTime;
17                         continue
18                   else  DEADLOCK()
19                         return
```

Figure 6.5: The PET scheduler in Rotalumis

sition system shown in figure 5.2. The system is made of two independent rotation units that do not interact with each other and whose controllers, designed by control engineers, run at different frequencies, 1000 Hz, respectively 500 Hz. The setup of the system is the same as the one for the educational example in chapter 2, as depicted in figure 6.6.

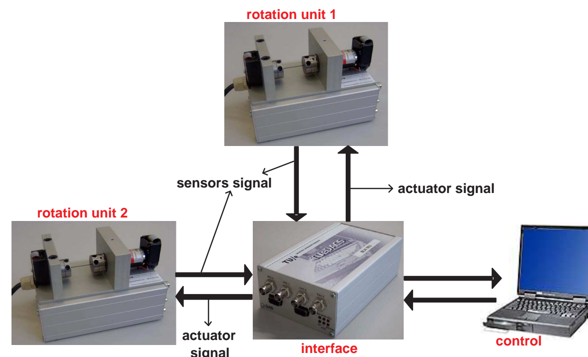


Figure 6.6: The motion system setup

An analysis model of the motion system is shown in figure 6.7. The control part of the system is made of two parallel processes, `MotorController_1` and `MotorController_2`, whereas the environment consists of `Motor_1` and `Motor_2`. The code shown in figure 6.7 is the POOSL model for each `MotorController`. The method `controlAlgorithm` models the actual control algorithm for a motor. From the design of the system, `ControllerPeriod`, which is an instantiation parameter of each process, is 1 ms for the first motor and 2 ms for the second one.

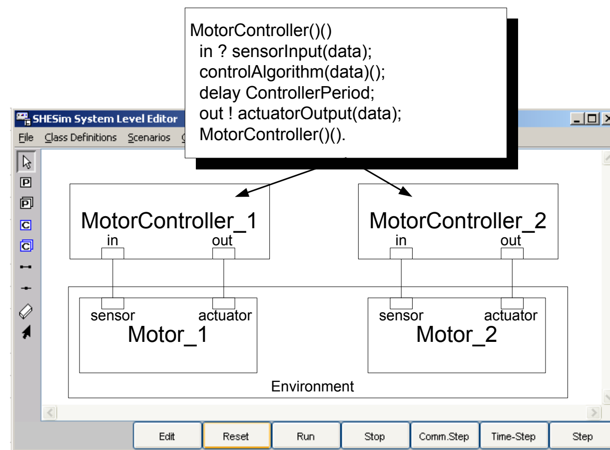


Figure 6.7: Model of the system

As discussed in the educational example, to ensure the stability of the control of the two motors system, two required real-time properties must be satisfied. For the first motor, the message `actuatorOutput` must *always* be sent between 0.9 and 1.1 ms after the message `sensorInput` is received. The MTL formula corresponding to this property is:

$$\Box(p_1 \rightarrow \Diamond_{[0.9,1.1]} q_1)$$

where p_1 represents the receiving of `sensorInput` message and q_1 the sending of `actuatorOutput` message for the first motor. For the second motor, with similar notations, the property that needs to be satisfied is:

$$\Box(p_2 \rightarrow \Diamond_{[1.9,2.1]} q_2).$$

Since the model is very simple, we could manually check that the control part in the model satisfies $\Box(p_1 \rightarrow \Diamond_{[1,1]} q_1)$ and $\Box(p_2 \rightarrow \Diamond_{[2,2]} q_2)$.

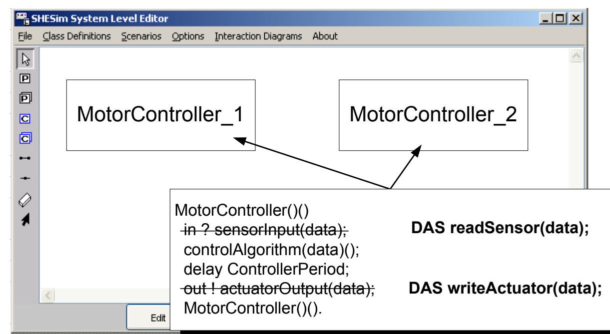


Figure 6.8: Synthesis model of the system

To enable automatic generation of the implementation of the control part of the

system from the model, a synthesis model was developed. In this model, the environment part was removed and all the communication with the environment was replaced with a synthesisable interface. The model presented in figure 6.8 shows how the communication with the environment model was replaced with calls to the `readSensor` and `writeActuator` methods from a data class called DAS (Data Acquisition System). This data class provides only virtual methods for the synthesis model. Its actual implementation for the communication with the real motors via the TUeDACS device is provided with Rotalumis in C++ code (figure 6.9 shows as an example the `readSensor` method). The implementation of the interface should not be blocking because its timing behaviour can affect the deviation between model and implementation.

```
void READSENSOR ()
{
  /* read the sensor data via TUeDACS API */
  TD_ENC_READ_CHAN(&pd->y, pd->channel, pd->link, TD_DIRECT);
}
```

Figure 6.9: The C++ implementation of READSENSOR

First, we have synthesised the model of the motor controllers using the synthesis approach presented in section 5.3, which treats with no distinction the observable and the unobservable actions. We have measured the time deviations obtained in the implementation for several hours of continuous behaviour and the maximum deviation was 0.213 ms. This implies that the weakening of the real-time properties in the realisation of the system is 0.426 ms and these properties are:

$$\Box(p_1 \rightarrow \Diamond_{[0.574,1.426]} q_1);$$
$$\Box(p_2 \rightarrow \Diamond_{[1.574,2.426]} q_2).$$

Thus, they do not meet the requirements. This large weakening explains the unstable behaviour of the system that could be noticed during execution. A possible solution to obtain a smaller time deviation would be to use a processor of a higher frequency.

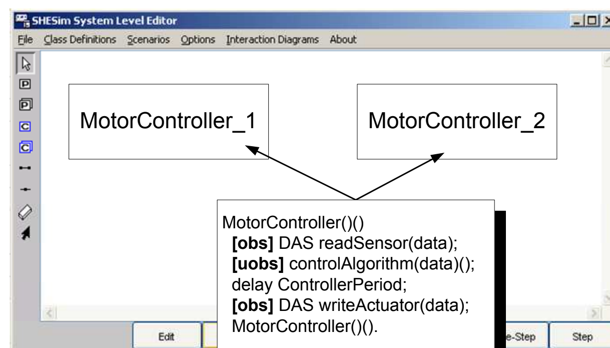


Figure 6.10: Annotated synthesis model

The solution that we have chosen for decreasing the distance was to use in a second experiment the synthesis approach presented in this chapter. We have labelled each action in the model, as shown in figure 6.10, such that Rotalumis could identify which is considered observable or unobservable. Actions like reading from the sensor and writing to the actuator are observable activities of the system, whereas the control algorithm computation is unobservable from outside the system. During several hours of continuous behaviour, the maximum obtained observable distance between the model and the generated implementation was 0.037 ms. Thus the properties satisfied by the implementation are:

$$\Box(p_1 \rightarrow \Diamond_{[0.926,1.074]} q_1);$$
$$\Box(p_2 \rightarrow \Diamond_{[1.926,2.074]} q_2).$$

They fulfill the requirements. Hence, the control strategy designed is implementable using the proposed approach.

From this experiment, we could observe how significant the impact of the new synthesis approach on the strength of property preservation may be. In our case, the time deviation decreased with 80% as the computations were much more time-intensive than the reading and the writing actions.

6.5 Related Research

The timing semantics based on the two phase execution framework that treats system functionality and time progress in an orthogonal way, namely, system actions are timeless (without any time progress) and the time progress is actionless (without preforming any action), is commonly adopted in design languages. Variations of this semantics have been used in different formal frameworks to model and analyse real-time systems, such as timed automata [9], time process algebra [71] and real-time transition systems [49]. Recently, this semantics is also integrated into design languages, such as SDL-2000 supported by TAU2 [88], or POOSL supported by SHE [41]. However, in current practice, the automatic generation of implementations from models lacks sufficient support to bridge the gap between the timing semantics of the modelling language and of the implementation language. As a result, the properties of the implementation cannot be deduced from the properties of the model. As an example, in the automatic implementation of an SDL-2000 model, the timing expressions rely on an asynchronous timer mechanism provided by the underlying platform. Hence, all expressions referring to some amount of time will refer to *at least* that amount of time. Timing errors are accumulated during execution, and this leads to timing failures and even functionality failures [53].

As shown in this thesis, the automatic generation of an implementation from a POOSL model overcomes the timing issue by the synchronisation of the model time with the physical time. By keeping an upper-bound on the time deviation between model and implementation, properties of the implementation can be predicted from the properties of the model.

One of the formal approaches widely used for modelling and analysis of real-time systems is timed automata. TIMES [7] is a tool for design of real-time systems models

based on timed automata that can describe concurrency and synchronisation of periodic, sporadic, preemptive or non-preemptive real-time tasks with or without precedence constraints. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable (all associated tasks can be computed within their required deadlines). From such a real-time system model, the TIMES compiler generates a scheduler, based on fixed priority assignment, and computes the worst-case response time for all tasks. For this, it relies on synchrony hypothesis, assuming that the time for handling system functions on the target platform can be ignored compared with the execution times and deadlines of tasks which are considered pre-specified. This issue may be overcome by integrating the ϵ-hypothesis in their code generation. Moreover, the type of real-time properties TIMES focusses on refer to deadlines of tasks, which is at a coarser level of granularity than the properties we are are looking at for preserving in the implementation.

A relaxation of the synchrony hypothesis is proposed in [27] where the notion of Almost ASAP semantics was introduced for timed automata. This semantics allows a certain time-bound, which is a parameter of the model, for the reaction of a system, preventing the need for instantaneous reaction to events as imposed by the synchrony hypothesis. The Almost ASAP semantics is useful when modelling real-time controllers and its idea is in-line with the ϵ-hypothesis that we use. It was shown that control strategies modelled with this semantics are robust and implementable. However, our approach differs from theirs in the sense that we look for an implementation that has the smallest time deviation from the model, whereas they set a requirement on the deviation and give a solution that satisfies it.

In [48], a programming model for real-time embedded controllers called Giotto is presented. Giotto is a methodology designed specifically for embedded control software development. Giotto is an embedded software model that can be used to specify a solution to a given control problem independent from a target platform. However, it is closer to executable code than to a mathematical model. Giotto is restricted to periodic non-preemptive real-time control tasks. For model synthesis, worst-case execution times of all tasks on the target CPU have to be provided, together with a jitter tolerance of the model. The Giotto compiler determines a schedule of the tasks that realises the execution on the target platform conforming to the tolerance. Moreover, Giotto can be seen as an intermediary step between mathematical models like timed automata and real execution code.

Compared with Giotto, our approach is fully based on a mathematical structure, the timed labelled transition system, and on a metric to express property preservation. During automatic code generation and execution, the time deviation (which is equivalent with the jitter tolerance) between the implementation and the model is determined for that specific target platform.

6.6 Summary

In this chapter, we have shown how the strength of observable property preservation for concurrent real-time systems can be improved. We defined a notion of distance that abstracts away from the internal unobservable actions. This distance

was used as a metric to express the strength of observable property preservation between model and realisation. We proved that an implementation in which observable action transitions can be taken before unobservable transitions has a smaller distance to the model than any other implementation of the same model. Moreover, we have incorporated this result into an existing predictable development method. By the means of a motion control system case study, we showed that the proposed approach improved the strength of property preservation with 80% and succeeded in generating an implementation that fulfilled the requirements.

With the discrimination between observable and unobservable actions that we introduced in this chapter, we open up for future research the possibility of using various scheduling policies, especially preemptive ones, for the execution of the unobservable actions. In order to do this, we need to formalise the dependencies between actions in a system and to rigourously define how these dependencies can be exploited for further improving the efficiency of the synthesis phase.

7
Case Study

In this chapter we present the model-driven design of the control of a printer paper path. The purpose of this case study is to illustrate the application of the various contributions brought by this thesis on a realistic system. Section 7.1 describes the setup of the system and discusses the system requirements. Section 7.2 presents the analysis model of the system based on modelling patterns, whereas section 7.3 deals with the prediction of the real-time properties preservation in the realisation of the system. Section 7.4 presents the synthesis of the printer paper path model and a summary of the chapter is given in section 7.5.

7.1 Printer Paper Path System Description

An experimental paper path of a printer has been designed and built at the Dynamics and Control Technology group within Eindhoven University of Technology [67]. A detailed description of the design for the setup is given in [103]. The setup of the system is depicted in figure 7.1. It consists of a paper input tray connected to a motor, denoted with PIM, and a paper path with five pinches, each connected via a gear belt to a motor, denoted with M1 to M5. Whereas in real printer paper paths aluminium plates are used to guide the sheets through the paper path, in the experimental setup thin steel wires are used to prevent the sheets from dropping out of the path and to be able to visually observe the sheets when they are transported. The paper path is designed for A4 sheets travelling in landscape orientation from left to right. There are four optical sensors, denoted with S1 to S4, installed along the path, each of which can be used to detect the presence of a sheet of paper. The setup is connected to a PC-based control system. This system consists of a 1.8 GHz Pentium4 host computer running RTAI/Fusion Linux [5] and three TUeDACS USB I/O devices [95].

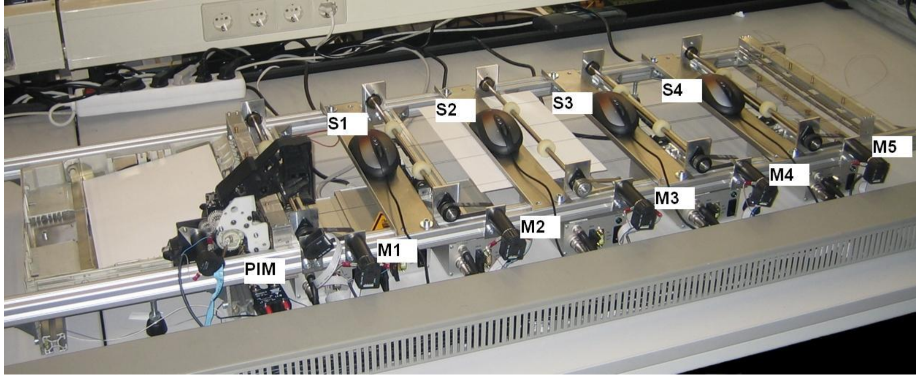


Figure 7.1: Photo of the experimental paper path setup

The experiment considered for case study is to drive sheets of paper through the paper path realising a throughput of 12 pages per minutes, which is a typical throughput for home and office printers. To achieve such a transportation rate, the velocity of a sheet of paper was calculated to be 70 mm/s and the distance between two travelling sheets is 140 mm. These parameters ensure that a sheet of paper exits the paper path every 5 s. When a late sheet is detected along its transportation, the velocity of the transportation is increased to 80 mm/s. A sheet is considered late if the distance between it and the previous sheet is more than 210 mm. The controllers of the motors were designed by control engineers [19] and a sampling frequency of 500 Hz was chosen, which is ten times higher than the typical motor control bandwidths. Based on the work presented in [24], it could be established that a timing accuracy of 0.1 ms for each controller is sufficient to guarantee the stability of the closed loop system. Hence, the time interval between the reading of a sensor signal and the writing to an actuator is required to be within $[1.9, 2.1]$ ms for each motor.

The purpose of this case study is to use a model-driven approach to obtain the software part of this system. The software implementation must ensure a correctly running system based on the design made by the control and mechanical engineers. This means that the stability of the system must be ensured as well as the desired throughput must be achieved without paper jams. Moreover, given the configuration of the setup, the use of only one optical sensor, the first one in the row, S1, for the detection of a late sheet was considered enough in order to correct the velocity of a late sheet by accelerating the last four motors in the paper path.

7.2 Analysis Model of the Paper Path

In order to analyse the properties of the system, we need to construct a model of it and hence, we first identified its players in a similar way we did in chapter 2. In this system, there is only one player, the so-called paper path controller, that ensures the control of the whole paper path and coordinates the activities of all devices. Hence, we had to go a step further and build a model of the system that takes into account

the various disciplines involved in such a system, software, control and mechanical engineering, as depicted in figure 7.2. A high-level controller supervises the low-level controllers and checks for every sheet of paper that passes by the optical sensor if it is in time or it is late. In case the sheet is late, it sets the velocities of motors M2, M3, M4 and M5 to a higher value to compensate for the lateness of the sheet. There are six low-level controllers in the system, one for each motor in the paper path, PIM, M1, M2, M3, M4, and M5. They are provided with information by the high-level controller regarding the velocity they should run with. The environment models the physical devices that are controlled by the low-level controllers, namely the six motors, and the optical sensor which is used by the high-level control to check the presence of sheets of paper.

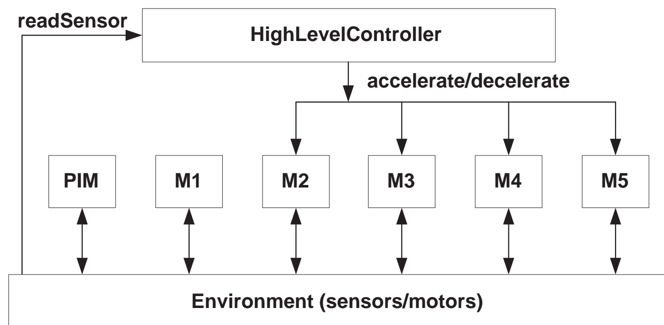


Figure 7.2: Refined model of the paper path system

The high-level controller makes decisions based on the timing of the sheets of paper. When sensor S1 triggers because it detects a sheet of paper, the high-level controller can determine if the sheet is late or in time. In the synthesis model of the system, the environment would be removed and the communication to it would be replaced with primitive data methods. Hence, a primitive method implemented in C++ would actually perform the reading from the optical sensor. Because of the synthesis mechanism that transforms the whole model into a single process running on a target processor, this method must be non-blocking. Therefore, the detection of a sheet passing by the sensor must be based on polling the information from the sensor. When the sensor detects a sheet, it will trigger. Hence, in the implementation of the system we must make sure that while a sheet passes by the sensor, the high-level controller tries to read at least once the sensor information. Moreover, to detect as early as possible that a sheet is going to arrive late, it must read the sensor often enough. Based on this reasoning, we decided to implement the high-level controller as a task that periodically polls the sensor and makes decisions based on the information it gets, setting the right values of the velocities of motors 2, 3, 4 and 5. Hence, the periodic task modelling pattern, presented in section 3.5.1, can be directly used for the model of the high-level controller.

Regarding the low-level controllers, as it can be seen in figure 7.2, PIM and M1 are independent tasks that control their corresponding motors. M2, M3, M4 and M5 can receive commands from the high-level controller for accelerating or decelerating. In essence, all these controllers are identical except for their reference velocities. Hence, we decided to model them in the same way and to implement the control algorithms

such that they read their corresponding velocities from some shared memory area to which the high-level controller has writing access. Thus, there will be no explicit interaction between the high-level controller and the low-level controllers. This decision implies that the low-level controllers can also be modelled as periodic tasks using the corresponding modelling pattern.

Table 7.1 presents the values of the periodic task modelling pattern parameters for each instantiation. As it takes 3 s for a sheet of paper to cross the optical sensor and a late sheet must be detected as early as possible, we have chosen that the high-level controller polls the optical sensor every 0.5 s. The initial offset of the high-level controller task is equal to the time is takes for the first sheet to arrive at the sensor. The low-level controllers start immediately, without any offset, and their period is 2 ms because the control algorithms were designed for a sampling frequency of 500 Hz. The loads imposed by the computations of each of these tasks on the platform are determined based on measurements of their worst-case execution times on the 1.8 GHz Pentium4 and on the assumption of a throughput of 2 billions of instructions per second for the processor. Moreover, since we run our experiments in a laboratory with a stable environment, with no sudden changes in the temperature, we assumed no drifts of the processor clock, thus no activation latency for the periodic tasks.

Task	Period [*ms*]	Deadline [*ms*]	Offset [*ms*]	Load [*instr.*]	Iterations	Latency [*ms*]	Priority
HighLevelController	500	500	2285	1100000	-1	0	1
PIM	2	2	0	480000	-1	0	2
M1	2	2	0	480000	-1	0	2
M2	2	2	0	480000	-1	0	2
M3	2	2	0	480000	-1	0	2
M4	2	2	0	480000	-1	0	2
M5	2	2	0	480000	-1	0	2

Table 7.1: Parameters of tasks

As mentioned earlier, the target platform for our experiments is a Pentium4 processor at 1.8 GHz. To model it, we have used the resource modelling pattern presented in section 3.5.2. The value of the throughput of the resource was taken as 2 billions of instructions per second. For our analysis, we assumed to have only this application running on the machine since this would be the case in a real printer. The initial latency of the resource, which models the context switch time between tasks, was considered zero because of the POOSL synthesis mechanism that creates only one task from the entire application.

To analyse schedulability of the system, a scheduler has also been modelled for dispatching the tasks onto the resource. The scheduling policy chosen was non-preemptive priority-driven, as it can be seen in figure 7.3. Since the requirement of the system is to have no paper jams and the design realised by control engineers must ensure that, we had to check if the system is schedulable or not on the desired target platform. The simulation of the model showed that all deadlines of the system are met. This means that all the tasks are able to accomplish their computations before their respective deadline. This information is helpful for a coarse-grain reasoning about the properties of the system. However, we need to determine if the system can actually run correctly, meeting the timing requirements that can ensure the stability of the motors. For this, we need a finer-grain analysis of the system, looking closer to the actions that each task has to perform.

```
<system>
    <application>
        <PeriodicTask Name="HighLevelController" T=”500" D="500" Offset=”2285"
            BCLoad=”1100000" WCLoad=”1100000" LoadDistrib=”Uniform”
            Iterations=”-1" Priority=”1" Latency=”0">
        <PeriodicTask Name="PIM" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
        <PeriodicTask Name="M1" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
        <PeriodicTask Name="M2" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
        <PeriodicTask Name="M3" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
        <PeriodicTask Name="M4" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
        <PeriodicTask Name="M5" T=”2" D="2" Offset=”0" BCLoad=”480000"
            WCLoad=”480000" LoadDistrib=”Uniform” Iterations=”-1"
            Priority=”2" Latency=”0">
    </application>
    <platform>
        <Resource Name="CPU" InitialLatency="0" FixedLatency=”true”
            Throughput="2000000000" IdlePower="0" NominalPower=”0”
            Monitored=”false”>
            <Scheduler Policy=”NPPrio" Monitored=”true”>
    </Resource>
    </platform>
    <mapping>
        <map TaskName="HighLevelController" ResourceName="CPU">
        <map TaskName="PIM" ResourceName="CPU">
        <map TaskName="M1" ResourceName="CPU">
        <map TaskName="M2" ResourceName="CPU">
        <map TaskName="M3" ResourceName="CPU">
        <map TaskName="M4" ResourceName="CPU">
        <map TaskName="M5" ResourceName="CPU">
    </mapping>
</system>
```

Figure 7.3: Paper path model specification based on modelling patterns

7.3 Predicting Properties of the Paper Path

The analysis model of the controllers discussed in section 7.2 is based on the periodic task modelling pattern that describes a task in a very abstract way, incorporating its type of activation, its load and its timing requirements. To check what are the properties of the system and to analyse how strongly they can be preserved in the implementation, we need more details about the behaviour of each task. For example, we need to identify what are the actions in the behaviour of a task and to distinguish which are observable and which are unobservable.

Figure 7.4 depicts the timed labelled transition system of the high-level controller. The transition system reflects the periodic task pattern in the sense that it shows to have a periodic behaviour with a period of 500 ms. Moreover, it shows that the activity realised by the high-level controller and modelled in an abstract way with the `Behaviour` method of the modelling pattern is that of checking the optical sensor and making decisions with respect to the velocity with which the motors should run.

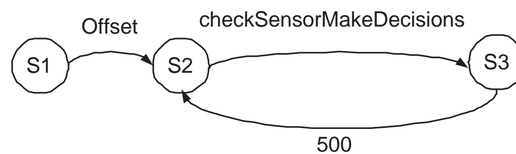


Figure 7.4: Timed labelled transition system of the high-level controller

Figure 7.5 depicts the timed labelled transition system of a low-level controller. This transition system also reflects the periodic task pattern in the sense that it shows a periodic behaviour with a period of 2 units of time, representing 2 ms which is the period of all the low-level controllers. It means that this transition system is the same for each of the low-level controllers, PIM, M1, M2, M3, M4 and M5. The behaviour of each low-level controller is decomposed into three main actions. The action of writing represents the writing to an actuator of a pre-computed value that is supposed to correct the position of the motor. The action of reading represents the reading of the current position of the motor from its encoder. Both reading and writing actions represent communication to the environment which is supposed to happen at precise moments in order to ensure the correctness and the stability of the control algorithm. The third action performed by a low-level controller is the actual control algorithm.

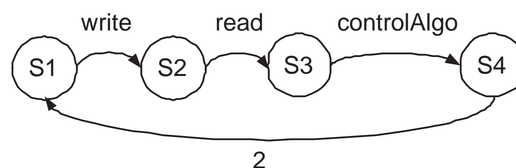


Figure 7.5: Timed labelled transition system of a low-level controller

Based on the timed labelled transition systems of the high-level and the low-level

controllers, the timed labelled transition system corresponding to the paper path control can be derived. Although each of the transition systems has a small number of states, due to the interleaving of the parallel actions, the resulting timed labelled transition system has one million of states. Therefore, we cannot show it in a figure. Nevertheless, from it we can derive the properties of interest, whose preservation we want to obtain as strong as possible in the realisation of the system. Using the MTL formulas notation, these properties are:

$$\begin{gathered}\Box(read_{PIM} \rightarrow \Diamond_{[2,2]} write_{PIM});\\ \Box(read_1 \rightarrow \Diamond_{[2,2]} write_1);\\ \Box(read_2 \rightarrow \Diamond_{[2,2]} write_2);\\ \Box(read_3 \rightarrow \Diamond_{[2,2]} write_3);\\ \Box(read_4 \rightarrow \Diamond_{[2,2]} write_4);\\ \Box(read_5 \rightarrow \Diamond_{[2,2]} write_5);\end{gathered}$$

meaning that a read action is always eventually followed by a corresponding write action after exactly 2 ms.

Action	Worst-case execution time [us]
checkSensorMakeDecisions	550
read	3
write	3
controlAlgo	234

Table 7.2: Worst-case execution times of actions

To predict the properties of the realisation of the paper path on the target platform, we need to determine the largest possible time deviation between the model and its realisation on our desired target platform. Since there is no cycle with positive cost in the transition system, we used the algorithm presented in section 5.7. The calculation of the time deviation is based on the measurements on the target platform of the worst-case execution times of all the actions given in table 7.2. Because of the large number of states in the timed labelled transition system, it took several hours to determine the largest possible time deviation, which is 1.990 ms. This means that the properties of the realisation of the system would be:

$$\begin{gathered}\Box(read_{PIM} \rightarrow \Diamond_{[0,5.98]} write_{PIM});\\ \Box(read_1 \rightarrow \Diamond_{[0,5.98]} write_1);\\ \Box(read_2 \rightarrow \Diamond_{[0,5.98]} write_2);\\ \Box(read_3 \rightarrow \Diamond_{[0,5.98]} write_3);\\ \Box(read_4 \rightarrow \Diamond_{[0,5.98]} write_4);\\ \Box(read_5 \rightarrow \Diamond_{[0,5.98]} write_5).\end{gathered}$$

Because the system is deterministic, the simulation-based estimation of the time deviation, which only investigates one path of the system, also gives the same value but in much shorter time (in the order of minutes). The properties predicted for the realisation of the system do not satisfy the timing requirements that can ensure stability. However, when we computed the time deviation between model and realisation, we considered that all the actions of the system are observable and hence those that are actually unobservable weaken the prediction of properties preservation.

Figure 7.6 depicts the timed labelled transition system of the high-level controller in which the action of checking the sensor and making decisions is labelled as an unobservable action, i.e. [uobs], as it is not important at what moment precisely it takes place, but it needs to be finished before the deadline of the task, which is the beginning of the next period. Moreover, figure 7.7 depicts the timed labelled transition system of a low-level controller in which its actions are also labelled. The action of writing as well as the action of reading represent communication to the environment which is supposed to happen at precise moments in order to ensure the correctness and the stability of the control algorithm. Hence, they are labelled as observable, i.e. [obs]. The control algorithm is considered an internal computation as its only timing requirement is to be finished before the next period.

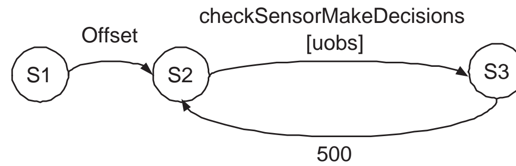


Figure 7.6: The timed labelled transition system of the high-level controller

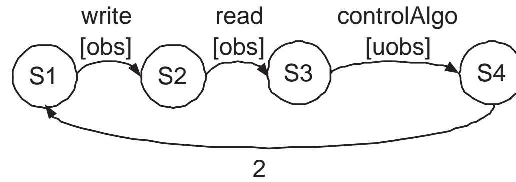


Figure 7.7: The timed labelled transition system of a low-level controller

By constructing the reduced timed labelled transition system using this discrimination between observable and unobservable actions, the number of states in the resulting system is smaller by a factor of five compared to the original system. By using the algorithm presented in figure 6.4, the value determined to be the largest possible time deviation between model and realisation is 0.036 ms. This means that the properties of the realisation of the system would be:

$$\begin{aligned}
\Box(read_{PIM} &\rightarrow \Diamond_{[1.928,2.072]} write_{PIM});\\
\Box(read_1 &\rightarrow \Diamond_{[1.928,2.072]} write_1);\\
\Box(read_2 &\rightarrow \Diamond_{[1.928,2.072]} write_2);\\
\Box(read_3 &\rightarrow \Diamond_{[1.928,2.072]} write_3);\\
\Box(read_4 &\rightarrow \Diamond_{[1.928,2.072]} write_4);\\
\Box(read_5 &\rightarrow \Diamond_{[1.928,2.072]} write_5).
\end{aligned}$$

These properties satisfy the requirements of the system and the realisation was found suitable to ensure the stability of the system.

```
double vPIM, v1, v2, v3, v4, v5;
int pages = 0, counter = 0;
void PDM_checkSensorMakeDecisions ()
{
   /* read from the sensor device */
   READ(fd, buf, 1);
   /* if the sensor did not trigger, buf[0] == 'n' or else buf[0] == 'y' */
   /* counter keeps track for how long the sensor did not trigger */
   if (buf[0] =='n') counter++;
   /* when a new page is detected, the number of pages passing by is incremented */
   if (buf[0] == 'y' && counter > 0)
   {
    counter = 0;
    pages++;
   }
   /* if more than 3 s have passed by with no paper then the motors are accelerated */
   if (counter >= 6 && pages >= 2)
   {
    v2 = V_HIGH;
    v3 = V_HIGH;
    v4 = V_HIGH;
    v5 = V_HIGH;
    pages = 0;
   }
   else
   /* the deceleration is done after a number of pages passed by */
   if(pages >= 2)
   {
    v2 = V_NORMAL;
    v3 = V_NORMAL;
    v4 = V_NORMAL;
    v5 = V_NORMAL;
    pages = 0;
   }
}
```

Figure 7.8: The C++ implementation of the data method CHECKSENSORMAKEDECISIONS

7.4 Synthesis of the Paper Path Model

To synthesise the model of the paper path, the actual implementation of the methods that realise the communication with the environment, as well as the control and the decision making algorithms has to be provided in C++. These so-called primitive methods are invoked as data methods from the POOSL model.

As an example, figure 7.8 presents the implementation of the decision making algorithm for the high-level controller. The method first checks the sensor if it currently detects a sheet or not. If no sheet is detected, then a counter is incremented to determine for how long there was no sheet detected. The distance between two sheets of paper travelling through the paper path should be 140 mm. Since the velocity of the transportation is 70 mm/s, it means that in the ideal situation the distance in time between two consecutive sheets is 2 s. If the value of the counter is 6, then no sheet has been detected in a time interval of $[2.5, 3]$ s since the last sheet has passed, as it is explained in figure 7.9. In this case, motors M2, M3, M4 and M5 are accelerated in order to compensate for the lateness of the sheet. Besides detecting what is the distance in time from the last observed sheet, we also take into account the number of pages that have been seen. To avoid changing the velocities of the motors

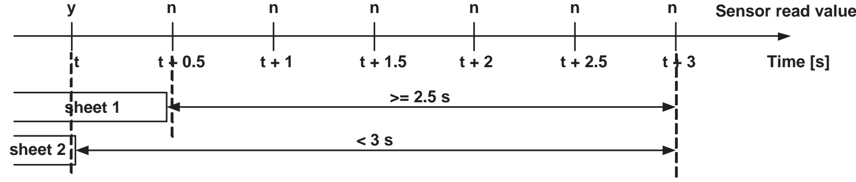


Figure 7.9: Sheet lateness interval

too often, we allow them to run with constant velocities for as long as at least two pages travel through the system.

The model of the paper path has been synthesised using the improved synthesis mechanism presented in chapter 6. While the experiment was running, the time deviation was recorded by the synthesis tool. The largest value obtained as the time deviation was 0.023 ms. This value is smaller than the one computed in section 7.3 because in reality the execution times of actions did not always have the worst-case value. Hence, the properties satisfied by the realisation of the system are:

$$\Box(read_{PIM} \rightarrow \Diamond_{[1.954,2.046]} write_{PIM});$$
$$\Box(read_1 \rightarrow \Diamond_{[1.954,2.046]} write_1);$$
$$\Box(read_2 \rightarrow \Diamond_{[1.954,2.046]} write_2);$$
$$\Box(read_3 \rightarrow \Diamond_{[1.954,2.046]} write_3);$$
$$\Box(read_4 \rightarrow \Diamond_{[1.954,2.046]} write_4);$$
$$\Box(read_5 \rightarrow \Diamond_{[1.954,2.046]} write_5).$$

With this case study we have shown that the control strategy of the printer paper path is implementable using our model-driven design approach.

7.5 Summary

In this chapter, we have presented the model-driven design of a printer paper path. With this case study we have demonstrated the application of the contributions brought by this thesis to a realistic system. We have used the modelling patterns to obtain a high-level model of the system and to analyse its schedulability. Moreover, we have estimated the distance between the model and the realisation of the system on the target platform based on measurements of the worst-case execution times of its actions. Using the synthesis approach presented in chapter 6, we managed to obtain an implementation of the software part of the system that strongly preserves the properties analysed in its model.

8
Conclusions and Outlook

The continuously increasing complexity of the embedded real-time systems demands a model-driven design approach that is able to predict and to guarantee the properties of the final product according to the system requirements and under tight time-to-market, cost and energy consumption constraints. This thesis brings contributions that address several aspects of this problem in the context of the Software/Hardware Engineering design methodology. These contributions represent a step towards a complete design flow, based on formal methods, for the predictable design for real-time system. In this chapter, the contributions of this work are summarised and ideas for future research are presented.

8.1 Summary of Research Contributions

To ensure the correctness and the performance properties of real-time embedded systems, early evaluation of their properties is needed. To achieve this, we have developed a set of modelling patterns that cover typical components of real-time systems as they are seen in classical scheduling theory [20]. The modelling patterns act as parameterisable templates that cover components, such as tasks, resources, input/output devices, that are typical for a large class of real-time systems, as shown in chapter 3. The use of these patterns alleviates the process of deriving models for exploration of different design alternatives. We have shown in chapter 4 that these patterns can be successfully applied for modelling systems from the areas of control-dominated and control with data-intensive computations applications. The POOSL library containing the implementation of the modelling patterns enlarges the set of guidelines for the use of the SHE methodology in the design of real-time systems.

Since the modelling patterns encapsulate real-time systems modelling experi-

ence, we have designed the Pattern-based system Description Language (PDL) to enable easy description of systems without the knowledge of a certain target modelling language. The language is presented in chapter 3 where we also introduce a model generation tool that takes as input a PDL model and a library containing the implementation of the patterns in a certain modelling language, and it outputs the model in that language. Due to the expressiveness of the POOSL language, appropriate models of real-time systems can be generated and the analysis techniques associated to the SHE methodology enable predictions of the properties of the final product realisation as well as appropriate design decisions making, as shown in chapter 4.

Because a model is just an approximation of a real system with respect to time, it is important to know how large the time deviation between the model and its implementation would be in order to understand the relation between the properties of the model and of the implementation. In chapter 5 of this thesis, we have presented and mathematically proved correct a mechanism to calculate from a model this time deviation based on worst-case execution times of actions. This approach has the potential of avoiding design iterations that are caused by successively obtaining large deviations during synthesis. Moreover, we have also developed a matching simulation-based approach for determining the time deviation between model and realisation that can be used when the algorithm for the actual calculation takes too long due to its large complexity.

To be able to ensure the preservation of the observable properties stronger for systems that contain data-intensive computations which are typically unobservable, in chapter 6 we have refined the metric that gives the distance between a model and its realisation. The refined metric discriminates between the actions that are observed from the environment of the system and those that are not. By adopting this discrimination, we have made the model synthesis mechanism more efficient. The synthesis step of the SHE methodology is now able to yield real-time software implementations preserving stronger the properties for both control and data-intensive systems.

By means of a case study, in chapter 7 we show how the contributions brought by this thesis can be applied for the model-driven design of a system that ensures the control of a printer paper path. With this case study we show how the system can be modelled using the modelling patterns, how its real-time properties can be derived and analysed and how predictions about the properties of the realisation on a target platform can be made. The synthesis of the system yields a realisation that preserves these properties and ensures a correctly running system.

8.2 Future Research

We have identified several interesting topics that are worth further research:

Extending the applicability of the pattern-based modelling technique. As it is currently, the library of modelling patterns can be used for modelling of control-dominated and control with data-intensive computations real-time systems and for analysing aspects like their end-to-end delay, schedulability

or energy consumption. However, there are systems in which instead of requirements for the end-to-end delay of messages travelling through the system, specific requirements are given with respect to the throughput, namely the number of messages that exit the system per time unit. This is the case for example for streaming-oriented applications, such as audio/video decoders that are supposed to deliver a certain number of decoded frames per second. To analyse such systems, an adaptation of the output collector needs to be done such that it can monitor either the end-to-end delay or the throughput based on the necessities of the system.

A possible extension of the modelling patterns library is the addition of a memory unit pattern. Many of the streaming-oriented systems require large memory space as they operate on large data (e.g. the size of a frame for MPEG 4 video decoder is 4 MB). For such systems it is very important to analyse the memory needed at runtime such that a good design of the memory layout and an appropriate choice of sizes and locations of the memory units can be made. Hence, a model of a memory monitor needs to be incorporated in the library in order to make it suitable for the analysis of this type of systems as well. The memory monitor would keep track of the storage space required by various components in the system. Moreover, it would enable a more accurate analysis of the end-to-end delay and of the schedulability of the system since the memory access latencies can be explicitly taken into account.

Another possible extension of the library of modelling patterns is the addition of a pattern modelling a network-on-chip [85] resource. Such a resource may contain various processors, memory units, network links and interconnections and usually requires complex schedulers. As the complexity of integrating systems keeps growing, a network-on-chip provides enhanced performance and scalability in comparison with other communication architectures, such as shared buses. For this reason, applications nowadays are more and more chosen to be deployed on such architectures. In order to be able to model and to analyse such a system, a modelling pattern that appropriately captures the characteristics of a network-on-chip needs to be developed.

Automatic design space exploration. Since we have already automated the modelling step of a model-driven design trajectory, a natural next step would be the automatic design space exploration of the models created. One possibility is the use of multi-objective evolutionary algorithms, the way it is shown in [72] for the case of design space exploration of network processors. The addition of an automatic design space exploration step increases the potential of the modelling and analysis approach presented in this thesis of being applicable for large-scale industrial systems due to its easiness of usage and accuracy of the results based on which appropriate design trade-offs can be made.

Incorporation of scheduling in the synthesis mechanism. Based on the semantics of a model, the existing synthesis approach within the SHE methodology considers all actions as atomic and executes them accordingly. With the discrimination between observable and unobservable actions that we introduced in chapter 6, we open up for future research the possibility of using preemptive scheduling policies for the execution of the unobservable actions. These actions are typically internal computations that might take a large amount of time and

that have requirements for their deadline. It might be that during the execution of a computation, some observable action, independent from the computation, becomes ready (e.g. two processes become ready for communication). However, according to the current approach, the execution of the observable action has to wait for the finishing of the computation, even though they are independent and hence the computation could be preempted in order to allow the execution of the observable action. As future research, it is an interesting challenge to formalise the independence between actions in the system and to incorporate the preemption of a computation for the atomic execution of an observable action.

Synthesis of a model for a distributed platform. The current synthesis approach is able to generate the realisation of a real-time system on a single processor platform using a central scheduler for all the processes of the system. When a multi-processor architecture is desired for the deployment of the system, the proposed technique may be inefficient or even unable to ensure the timing and the ordering of actions as verified in the model. Hence, a technique for automatic and correct synthesis of real-time systems on distributed architectures remains an interesting research challenge.

Sequences of Elements

This appendix presents the proofs for the mathematical results in chapter 5.

Definition A.1 *Given a sequence $\overline{x} \in \mathbb{R}^n$, the sum operation over the sequence $\sum \overline{x}$ is defined recursively as follows:*

- $\sum <> = 0$
- $\sum(\overline{x}.<y>) = \max(\sum \overline{x} + y, 0)$, $y \in \mathbb{R}$

Lemma A.1 *The sum operation for the elements of a sequence of numbers has the following properties:*

1. *It is commutative if the numbers have the same sign: if $x \cdot y \geq 0$, then $\sum(<x>.<y>) = \sum(<y>.<x>)$.*
2. *It is not commutative if the numbers have different signs: if $x \cdot y < 0$, then $\sum(<x>.<y>) \neq \sum(<y>.<x>)$.*
3. *It corresponds to the monus operation (see [62]): if $x \geq 0$ and $y < 0$, then $\sum(<x>.<y>) = x \dot{-} |y|$.*

Proof

1. Given $x, y \in \mathbb{R}$ with $x \cdot y \geq 0$, there are two cases.

   Case 1: Let $x \geq 0$ and $y \geq 0$. Then

   $\sum(<x>.<y>) = \max(\sum <x> + y, 0) = \max(x + y, 0) = x + y$ and $\sum(<y>.<x>) = \max(\sum <y> + x, 0) = \max(y + x, 0) = x + y$.

Case 2: Let $x < 0$ and $y < 0$. Then
$\sum(<x>.<y>) = \max(\sum<x>+y, 0) = \max(0 + y, 0) = 0$ and $\sum(<y>.<x>) = \max(\sum<y> +x, 0) = \max(0 + x, 0) = 0$.

2. Given $x, y \in \mathbb{R}$ with $x \cdot y < 0$, there are two cases.

   Case 1: Let $x > 0$ and $y < 0$. Then

$$\sum(<x>.<y>) = \max(\sum<x>+y, 0) =$$
$$= \max(x + y, 0) = \begin{cases} x - |y| & \text{if } x > |y|; \\ 0 & \text{if } x \leq |y|. \end{cases}$$

   and

$$\sum(<y>.<x>) = \max(\sum<y>+x, 0) = \max(0 + x, 0) = x$$

   Clearly, $x \neq 0$ and $x \neq x - |y|$.

   Case 2: Let $x < 0$ and $y > 0$. The proof is similar.

3. Given $x, y \in \mathbb{R}$ with $x \geq 0$ and $y < 0$, then $\sum(<x>.<y>) = \max(\sum<x> +y, 0) = \max(\sum<x>-|y|, 0) = x \dot{-} |y|$.

□

Lemma A.2 *The sum of the elements of a sequence $\overline{x}$ is a non-negative number, $\sum \overline{x} \geq 0$.*

Proof The proof is by induction on the length of $\overline{x}$ by using the definition A.1 of the sum of its elements.

□

Lemma A.3 *Given two sequences $\overline{x} \in \mathbb{R}^n$ and $\overline{y} \in \mathbb{R}^m$, the sum of the elements of the concatenated sequence $\overline{x}.\overline{y} \in \mathbb{R}^{n+m}$ is $\sum(\overline{x}.\overline{y}) = \sum(<\sum \overline{x}>.\overline{y})$.*

Proof We prove this lemma by induction on the length of $\overline{y}$.

Case 1. Let $\overline{y} =<>$. Then
$\sum(\overline{x}.\overline{y}) = \sum(\overline{x}.<>) = \sum \overline{x} = \sum<\sum \overline{x}>= \sum(<\sum \overline{x}>.<>)$.

Case 2. Let $\overline{y} = \overline{y'}.<a>$ and assume that $\sum(\overline{x}.\overline{y'}) = \sum(<\sum \overline{x}>.\overline{y'})$. Then
$\sum(\overline{x}.\overline{y}) = \sum(\overline{x}.\overline{y'}.<a>) = \max(\sum(\overline{x}.\overline{y'}) + a, 0) = \max(\sum(<\sum \overline{x}>.\overline{y'}) + a, 0) = \sum(<\sum \overline{x}>.\overline{y'}.<a>) = \sum(<\sum \overline{x}>.\overline{y})$.

□

Lemma A.4 *Given a sequence $\overline{x} \in \mathbb{R}^n$, then $\sum(<0>.\overline{x}) = \sum \overline{x}$.*

Proof We prove the lemma by induction on the length of $\overline{x}$.

Case 1. Let $\overline{x} =<>$. Then
$\sum(<0>.\overline{x}) = \sum(<0>.<>) = \sum<0>= \max(\sum<> +0, 0) = 0 = \sum<>= \sum \overline{x}$.

Case 2. Let $\overline{x} = \overline{x'}.<x>$ and assume $\sum(<0>.\overline{x'}) = \sum \overline{x'}$. Then

$\sum(<0>.\overline{x}) = \sum(<0>.\overline{x'}.<x>) = \{by\ lemma\ A.3\} \sum(<\sum(<0>.\overline{x'})>.<x>) = \{by\ induction\} \sum(<\sum \overline{x'}>.<x>) = \{by\ lemma\ A.3\} \sum(\overline{x'}.<x>) = \sum \overline{x}.$

□

Lemma A.5 *Given $\overline{x} \in \mathbb{R}^n$ and $a, b \in \mathbb{R}$, then $\sum(<a>.\overline{x}) \leq \sum(<b>.\overline{x})$ if $a \leq b$.*

Proof We prove this lemma by induction on the length of $\overline{x}$.

Case 1. Let $\overline{x} =<>$. Then

$\sum(<a>.\overline{x}) = \sum(<a>.<>) = \sum<a> = \sum(<>.<a>) = \max(\sum<> +a, 0) = \max(a, 0)$.

Similarly, $\sum(<b>.\overline{x}) = \max(b, 0)$. Because we have $a \leq b$, $\max(a, 0) \leq \max(b, 0)$.

Case 2. Let $\overline{x} = \overline{x'}.<x>$ and assume that $\sum(<a>.\overline{x'}) \leq \sum(<b>.\overline{x'})$. Then

$\sum(<a>.\overline{x}) = \sum(<a>.\overline{x'}.<x>) = \max(\sum(<a>.\overline{x'}) + x, 0) \leq_{by\ induction} \max(\sum(<b>.\overline{x'}) + x, 0) = \sum(<b>.\overline{x})$. Thus, $\sum(<a>.\overline{x}) \leq \sum(<b>.\overline{x})$.

□

Lemma A.6 *Given $\overline{x} \in \mathbb{R}^n$ and $a \in \mathbb{R}$, then $\sum(<a>.\overline{x}) \leq a + \sum \overline{x}$ if $a \geq 0$.*

Proof We prove this lemma by induction on the length of $\overline{x}$.

Case 1. Let $\overline{x} =<>$. Then

$\sum(<a>.\overline{x}) = \sum(<a>.<>) = \sum<a> = \max(\sum<> +a, 0) = a \leq a + 0 = a + \sum<> = a + \sum \overline{x}$.

Case 2. Let $\overline{x} = \overline{x'}.<b>$ and assume that $\sum(<a>.\overline{x'}) \leq a + \sum \overline{x'}$. Then

$\sum(<a>.\overline{x}) = \sum(<a>.\overline{x'}.<b>) = \max(\sum(<a>.\overline{x'}) + b, 0) \leq \{\textit{by induction}\} \max(a + \sum \overline{x'} + b, 0) \leq \{\textit{because } a \geq 0\} a + \max(\sum \overline{x'} + b, 0) = a + \sum(\overline{x'}.<b>) = a + \sum \overline{x}$.

□

Lemma A.7 *Let $\overline{x} \in \mathbb{R}^n$ and $a, b \in \mathbb{R}$. Then $\sum(\overline{x}.<a>.<b>.\overline{y}) \leq \sum(\overline{x}.<b>.<a>.\overline{y})$ if $a > 0$ and $b \leq 0$.*

Proof From the definition of the sum over a sequence, $\sum(<\sum \overline{x}>.<a>.<b>) = \max(\sum(<\sum \overline{x}>.<a>) + b, 0) = \max(\max(\sum \overline{x} + a, 0) + b, 0)$. Since $\sum \overline{x} \geq 0$ and $a > 0$, we have $\sum(<\sum \overline{x}>.<a>.<b>) = \max(\sum \overline{x} + a + b, 0)$.

Using a similar reasoning, we get $\sum(<\sum \overline{x}>.<b>.<a>) = \max(\sum(<\sum \overline{x}>.<b>) + a, 0) = \max(\max(\sum \overline{x} + b, 0) + a, 0)$. Because $b \leq 0$, we have two cases.

Case 1. If $\sum \overline{x} + b \leq 0$, then $\sum(<\sum \overline{x}>.<b>.<a>) = \max(a, 0) = a$ and $\sum \overline{x} + b + a \leq a$. Thus, $\max(\sum \overline{x} + a + b, 0) \leq \max(a, 0)$. From this we have that $\sum(<\sum \overline{x}>.<a>.<b>) \leq \sum(<\sum \overline{x}>.<b>.<a>)$.

Case 2. If $\sum \overline{x} + b > 0$, then $\sum(<\sum \overline{x}>.<b>.<a>) = \max(\sum \overline{x} + a + b, 0) = \sum(<\sum \overline{x}>.<a>.<b>)$.

Hence, in any case $\sum(<\sum\overline{x}>.<a>.<b>) \leq \sum(<\sum\overline{x}>.<b>.<a>)$.

We now have $\sum(\overline{x}.<a>.<b>.\overline{y}) = \{by\ lemma\ A.3\} \sum(<\sum\overline{x}>.<a>.<b>.\overline{y}) = \sum(<\sum(<\sum\overline{x}>.<a>.<b>)>.\overline{y}) = \{by\ lemma\ A.5\ and\ the\ result\ above\} \sum(<\sum(<\sum\overline{x}>.<b>.<a>)>.\overline{y}) = \{by\ lemma\ A.3\} \sum(\overline{x}.<b>.<a>.\overline{y})$

□

Lemma A.8 *Let* $\overline{x} \in \mathbb{R}^n$ *and* $a, b \in \mathbb{R}$. *Then* $\sum(\overline{x}.<a>.<b>.\overline{y}) = \sum(\overline{x}.<b>.<a>.\overline{y})$ *if* $a \geq 0$ *and* $b \geq 0$ *or* $a < 0$ *and* $b < 0$.

Proof Case 1. Let $a \geq 0$ and $b \geq 0$. Using the same line of reasoning as in the proof of lemma A.7, $\sum(<\sum\overline{x}>.<a>.<b>) = \max(\sum\overline{x} + a + b, 0)$ and $\sum(<\sum\overline{x}>.<b>.<a>) = \max(\sum\overline{x} + b + a, 0)$. Thus $\sum(\overline{x}.<a>.<b>.\overline{y}) = \sum(\overline{x}.<b>.<a>.\overline{y})$.

Case 2. Let $a < 0$ and $b < 0$. Then $\sum(<\sum\overline{x}>.<a>.<b>) = \max(\max(\sum\overline{x} + a, 0) + b, 0)$ and $\sum(<\sum\overline{x}>.<b>.<a>) = \max(\max(\sum\overline{x} + b, 0) + a, 0)$. We distinguish 4 subcases.

Subcase 1. Let $\sum\overline{x} + a < 0$ and $\sum\overline{x} + b < 0$. Then $\sum(<\sum\overline{x}>.<a>.<b>) = \max(b, 0) = 0$ and $\sum(<\sum\overline{x}>.<b>.<a>) = \max(a, 0) = 0$. Hence $\sum(\overline{x}.<a>.<b>.\overline{y}) = \sum(\overline{x}.<b>.<a>.\overline{y})$ by using lemma A.5 twice.

Subcase 2. Let $\sum\overline{x} + a < 0$ and $\sum\overline{x} + b \geq 0$. Then $\sum(<\sum\overline{x}>.<a>.<b>) = \max(b, 0) = 0$ and $\sum(<\sum\overline{x}>.<b>.<a>) = \max(\sum\overline{x} + b + a, 0)$. Because $\sum\overline{x} + a < 0$, we have $\sum\overline{x} + a + b < b < 0$ and thus $\sum(<\sum\overline{x}>.<b>.<a>) = 0$. Hence $\sum(\overline{x}.<a>.<b>.\overline{y}) = \sum(\overline{x}.<b>.<a>.\overline{y})$ by using lemma A.5 twice.

Subcase 3. Let $\sum\overline{x} + a \geq 0$ and $\sum\overline{x} + b < 0$. This subcase is similar to subcase 2.

Subcase 4. Let $\sum\overline{x} + a \geq 0$ and $\sum\overline{x} + b \geq 0$. This subcase is similar to subcase 1.

□

Theorem A.9 *Let* $\overline{x} \in \mathbb{R}^n$ *and let* $\overline{y} \in \mathbb{R}^n$ *be any permutation of* $\overline{x}$ *such that for some* m, $1 \leq m \leq n + 1$, $t_i \geq 0$ *for all* $1 \leq i < m$ *and* $t_i < 0$ *for all* $m \leq i \leq n$. *Then* $\sum\overline{y} \leq \sum\overline{x'}$ *for any permutation* $\overline{x}'$ *of* $\overline{x}$.

Proof Let $\overline{x'}$ be any permutation of $\overline{x}$. It is easy to see that $\overline{y}$ can be derived from $\overline{x'}$ by repeatedly swapping two consecutive elements. Either two elements of the same sign are swapped, keeping the sum invariant according to lemma A.8, or a negative and a non-negative number are swapped such that the negative number is moved to the right. In the latter case, the sum is decreased according to lemma A.7. Hence $\sum\overline{y} \leq \sum\overline{x'}$.

□

By theorem A.9 and definition 5.17, $\downarrow\overline{x} = \sum\overline{y}$ where $\overline{y} \in \mathbb{R}^n$ is any permutation of $\overline{x}$ such that for some m, with $1 \leq m \leq n + 1$, $t_i \geq 0$ for all $1 \leq i < m$ and $t_i < 0$ for all $m \leq i \leq n$. Notice that $\downarrow\overline{x}$ is uniquely defined in this way. Because if $\overline{y}$ and $\overline{y'}$ are both permutations satisfying the conditions above, then by theorem A.9, $\sum\overline{y} \leq \sum\overline{y'}$ and $\sum\overline{y'} \leq \sum\overline{y}$.

Theorem A.10 *Let $\overline{x} \in \mathbb{R}^n$ and let $\overline{y} \in \mathbb{R}^n$ be any permutation of $\overline{x}$ such that for some m, $1 \leq m \leq n+1$, $t_i < 0$ for all $1 \leq i < m$ and $t_i \geq 0$ for all $m \leq i \leq n$. Then $\sum \overline{y} \geq \sum \overline{x'}$ for any permutation $\overline{x}'$ of $\overline{x}$.*

Proof The proof of this theorem is similar to the proof of theorem A.9.

□

By theorem A.10 and definition 5.18, $\uparrow\overline{x}$ is uniquely given by $\sum \overline{y}$ where $\overline{y} \in \mathbb{R}^n$ is any permutation of $\overline{x}$ such that for some m, with $1 \leq m \leq n+1$, $t_i < 0$ for all $1 \leq i < m$ and $t_i \geq 0$ for all $m \leq i \leq n$. Moreover, we have that $\downarrow\overline{x} \leq \sum \overline{x} \leq \uparrow\overline{x}$.

Lemma A.11 *Let $\overline{x} \in \mathbb{R}^n$ be a sequence with $\downarrow\overline{x} = 0$. Then for all $a \geq 0$, $\sum(<a>.\overline{x}) \leq \max(a, \sum \overline{x})$.*

Proof Let $\overline{x_1}$ and $\overline{x_2}$ be two sequences such that $\overline{x} = \overline{x_1}.\overline{x_2}$. Then $\overline{x_1}$ is called a prefix of $\overline{x}$. From the definition of the sum over a sequence of numbers, $\sum(<a>.\overline{x_1}) \geq 0$ for any prefix $\overline{x_1}$ of $\overline{x}$. To prove the lemma, we distinguish two cases.

Case 1. Assume $\sum(<a>.\overline{x_1}) > 0$ for every prefix $\overline{x_1}$ of $\overline{x}$. Then the sum $\sum(<a>.\overline{x})$ actually corresponds to the arithmetic sum and since $\downarrow\overline{x} = 0$, $\sum(<a>.\overline{x}) \leq a \leq \max(a, \sum \overline{x})$.

Case 2. Otherwise $\sum(<a>.\overline{x_1}) = 0$ for some prefix $\overline{x_1}$ of $\overline{x}$. By lemma A.5, $\sum(<0>.\overline{x_1}) \leq \sum(<a>.\overline{x_1}) = 0$ and from lemma A.4, $\sum(<0>.\overline{x_1}) = \sum \overline{x_1}$. Hence $\sum \overline{x_1} = 0$.

But then $\sum(<a>.\overline{x}) = \sum(<a>.\overline{x_1}.\overline{x_2}) = \sum(<\sum(<a>.\overline{x_1})>.\overline{x_2}) = \sum(<0>.\overline{x_2}) = \sum(<\sum \overline{x_1}>.\overline{x_2}) = \sum(\overline{x_1}.\overline{x_2}) = \sum \overline{x} \leq \max(a, \sum \overline{x})$.

□

Theorem A.12 *Consider a timed labelled transition system $\mathcal{T}$ such that $\downarrow\overline{\mathcal{C}} = 0$ for every simple cycle $\mathcal{C}$ in $\mathcal{T}$. Then for any $a \geq 0$ and any cycle $\mathcal{Q}$ in $\mathcal{T}$, the following are true:*

1. $\downarrow\overline{\mathcal{Q}} = 0$;
2. $\sum(<a>.\overline{\mathcal{Q}}) \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ *is a simple cycle in* $\mathcal{Q}\})$;
3. *for any prefix* $\overline{\mathcal{Q}'}$ *of* $\overline{\mathcal{Q}}$, $\sum(<a>.\overline{\mathcal{Q}'}) \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ *is a simple cycle in* $\mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ *is a simple cycle in* $\mathcal{Q}\}$.

Proof We prove the theorem by induction on the complexity of the cycle $\mathcal{Q}$.

Case 1. Assume $\mathcal{Q}$ is a simple cycle. From the hypothesis of the theorem, $\downarrow\overline{\mathcal{Q}} = 0$, thus 1 holds.

Further, by lemma A.11, $\sum(<a>.\overline{\mathcal{Q}}) \leq \max(a, \sum \overline{\mathcal{Q}}) \leq \max(a, \uparrow\overline{\mathcal{Q}}) \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ is a simple cycle in $\mathcal{Q}\})$. Thus 2 holds.

By lemma A.6, $\sum(<a>.\overline{\mathcal{Q}'}) \leq a + \sum \overline{\mathcal{Q}'} \leq a + \uparrow\overline{\mathcal{Q}'} \leq a + \uparrow\overline{\mathcal{Q}} \leq \max(a, \uparrow\overline{\mathcal{Q}}) + \uparrow\overline{\mathcal{Q}} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ is a simple cycle in $\mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}$ is a simple cycle in $\mathcal{Q}\}$. Hence 3 also holds.

Case 2. Assume $\mathcal{Q}$ is a cycle in which the sequence of states from the transition system is of the form $ss_1s_2\ldots s_kss_{k+1}\ldots s_ms$ with $s \neq s_i$ for any $1 \leq i \leq m$. Then $\overline{\mathcal{Q}}$ can be written as $\overline{\mathcal{Q}} = \overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}$ where $\overline{\mathcal{Q}_1}$ corresponds to path $\mathcal{Q}_1$ with state sequence $ss_1s_2\ldots s_ks$ and $\overline{\mathcal{Q}_2}$ corresponds to path $\mathcal{Q}_2$ with state sequence $ss_{k+1}\ldots s_ms$.

By construction, $\overline{\mathcal{Q}_1}$ and $\overline{\mathcal{Q}_2}$ both correspond to cycles. Thus, by induction, $\downarrow\overline{\mathcal{Q}_1} = 0$ and $\downarrow\overline{\mathcal{Q}_2} = 0$. But then we also have that $\downarrow\overline{\mathcal{Q}} = 0$ and hence 1 holds.

To show 2 holds, we have $\sum(<a>.\overline{\mathcal{Q}}) = \sum(<a>.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}) = \sum(<\sum(<a>.\overline{\mathcal{Q}_1})>.\overline{\mathcal{Q}_2}) \leq \{by\ induction\}\sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\})>.\overline{\mathcal{Q}_2}) \leq \max(\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\}), \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_2\}) \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q})$. Hence 2 also holds.

To prove 3, we need to observe that for a prefix $\overline{\mathcal{Q}'}$ of $\overline{\mathcal{Q}}$ there are two cases: either $\overline{\mathcal{Q}'}$ is a prefix of $\overline{\mathcal{Q}_1}$ or there exists some $\overline{u}$ such that $\overline{\mathcal{Q}_1}.\overline{u} = \overline{\mathcal{Q}'}$.

If $\overline{\mathcal{Q}'}$ is a prefix of $\overline{\mathcal{Q}_1}$, then $\sum(<a>.\overline{\mathcal{Q}'}) \leq \{by\ induction\}\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}\}$. Hence 3 holds.

If $\overline{\mathcal{Q}_1}.\overline{u} = \overline{\mathcal{Q}'}$, then $\overline{u}$ is actually a prefix of $\overline{\mathcal{Q}_2}$. Then $\sum(<a>.\overline{\mathcal{Q}'}) = \sum(<a>.\overline{\mathcal{Q}_1}.\overline{u}) = \{by\ lemma\ A.3\}\sum(<\sum(<a>.\overline{\mathcal{Q}_1})>.\overline{u}) \leq \{by\ induction\}\sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\})>.\overline{u}) \leq \{by\ induction\}\max(\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_1\}), \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_2\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}_2\} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C}\text{ is a simple cycle in }\mathcal{Q}\}$. Hence 3 holds.

Case 3. Assume $\mathcal{Q}$ is a cycle in which the sequence of states from the transition system is of the form:

$$st_1\ldots t_1t_2\ldots t_2t_3\ldots t_m\ldots t_ms$$

where for any $i \neq j$, $t_i \neq t_j \neq s$, and $t_j\ldots t_j$ represents a sequence of zero or more states that appear between the first and the last occurrence of t_j in $\mathcal{Q}$. If for any $1 \leq j \leq m$ we denote with $t_js_{i_j}s_{i_j+1}\ldots s_{i_j+n_j}t_j$, then for any $i_k \leq l \leq i_k + n_k$ and $k < j$, $s_l \neq t_j$.

Then $\overline{\mathcal{Q}} = \overline{u_1}.\overline{\mathcal{Q}_1}.\overline{u_2}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_{m+1}}$ where $\overline{u_1}$ corresponds to path u_1 with state sequence st_1, for any $1 \leq k \leq m$ $\overline{\mathcal{Q}_k}$ corresponds to path $\mathcal{Q}_k$ with state sequence $t_ks_{i_k}\ldots s_{i_k+n_k}t_k$, for any $1 < k \leq m$ $\overline{u_k}$ corresponds to path u_k with state sequence $t_{k-1}t_k$, and $\overline{u_{m+1}}$ corresponds to path u_{m+1} with state sequence t_ms. By construction, $\mathcal{Q}_k$ is a cycle and u is a simple cycle where $\overline{u} = \overline{u_1}.\overline{u_2}.\ldots\overline{u_{m+1}}$. Hence, $\downarrow\overline{Q_k} = 0$ for all $1 \leq k \leq m$ and $\downarrow\overline{u} = 0$.

$\downarrow\overline{\mathcal{Q}} = \downarrow(\overline{u_1}.\overline{\mathcal{Q}_1}.\overline{u_2}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_{m+1}}) = \downarrow(\overline{u_1}.\overline{u_2}\ldots\overline{u_{m+1}}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}) = \downarrow(\overline{u}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}) \leq \sum(<\downarrow\overline{u}>.<\downarrow\overline{\mathcal{Q}_1}>.<\downarrow\overline{\mathcal{Q}_2}>\ \ldots\ <\downarrow\overline{\mathcal{Q}_m}>) \leq \{by\ lemma\ A.6\} \downarrow\overline{u} + \sum(<\downarrow\overline{\mathcal{Q}_1}>.<\downarrow\overline{\mathcal{Q}_2}>\ \ldots\ <\downarrow\overline{\mathcal{Q}_m}>) \leq \ldots \leq \downarrow\overline{u} + \downarrow\overline{\mathcal{Q}_1} + \downarrow\overline{\mathcal{Q}_2} + \ldots + \downarrow\overline{\mathcal{Q}_m} = 0$. Hence, $\downarrow\overline{Q} \leq 0$. By lemma A.2, $\downarrow\overline{Q} = 0$, thus 1 holds.

To prove 2, by theorem A.10 we have that $\sum(<a>.\overline{\mathcal{Q}}) \leq \sum(<a>.\overline{u_-}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_+})$ where $\overline{u_-}$ represents the sequence of all the negative numbers from $\overline{u}$ and $\overline{u_+}$ represents the sequence of all the non-negative numbers from $\overline{u}$. Thus, $\sum\overline{u_-} = 0$ and $\sum\overline{u_+}$ is the arithmetic sum of all the numbers in the sequence. Moreover, we denote with $\mathcal{Q}'$ the cycle with $\overline{\mathcal{Q}'} = \overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}$.

Because by lemma A.6, $\sum(<a>.\overline{u_-}) \leq a + \sum \overline{u_-} = a$, we have $\sum(<a> .\overline{u_-}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) = \{by\ lemma\ A.3\} \sum(<\sum(<a>.\overline{u_-})>.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) \leq \sum(<a>.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) \leq \{by\ induction\ and\ by\ lemmas\ A.3 and\ A.5\} \sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\})>.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) \leq \sum(<\max(\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\}), \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_2\})>.\overline{\mathcal{Q}_3}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) \leq \sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\mathcal{Q}_2\})>.\overline{\mathcal{Q}_3}\ldots\overline{\mathcal{Q}_m}.\overline{u_+}) \leq \ldots \leq \sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}'\})>.\overline{u_+}) \leq \{because\ \sum\overline{u_+} \leq \uparrow\overline{u}\ and\ by\ lemma\ A.6\} \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}'\}) + \uparrow\overline{u} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}'\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } u\} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\})$. Hence 2 also holds.

To prove 3, if the prefix of $\overline{\mathcal{Q}}$ is of the form $\overline{\mathcal{Q}'} = \overline{u_1}.\overline{\mathcal{Q}_1}.\overline{u_2}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_k}.\overline{u'_{k+1}}$, the proof is similar to the one for case 2. If the prefix of $\overline{\mathcal{Q}}$ is of the form $\overline{\mathcal{Q}'} = \overline{u_1}.\overline{\mathcal{Q}_1}.\overline{u_2}.\overline{\mathcal{Q}_2}\ldots\overline{u_k}.\overline{\mathcal{Q}'_k}$, where $\mathcal{Q}'_k$ is a prefix of the cycle $\mathcal{Q}_k$, then $\sum(<a>.\overline{\mathcal{Q}'}) = \sum(<a>.\overline{u_1}.\overline{\mathcal{Q}_1}.\overline{u_2}.\overline{\mathcal{Q}_2}\ldots\overline{u_k}.\overline{\mathcal{Q}'_k})$. Let $\overline{u} = \overline{u_1}\ldots\overline{u_k}$. By shifting all the negative numbers from $\overline{u}$ to the left and all the non-negative numbers to the right, we obtain by theorem A.10 that $\sum(<a>.\overline{\mathcal{Q}'}) \leq \sum(<a>.\overline{u_-}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+})$ where $\overline{u_-}$ represents the sequence of negative numbers from $\overline{u}$ and $\overline{u_+}$, respectively, the sequence of non-negative numbers.

Because by lemma A.6, $\sum(<a>.\overline{u_-}) \leq a + \sum\overline{u_-} = a$, we have $\sum(<a>.\overline{u_-}.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) = \{by\ lemma\ A.3\} \sum(<\sum(<a> .\overline{u_-})>.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \sum(<a>.\overline{\mathcal{Q}_1}.\overline{\mathcal{Q}_2}\ldots\overline{\mathcal{Q}_{k-1}}.\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \{by\ induction\} \sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\mathcal{Q}_2\ldots\mathcal{Q}_{k-1}\})> .\overline{\mathcal{Q}'_k}.\overline{u_+}) \leq \sum(<\max(\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\mathcal{Q}_2\ldots\mathcal{Q}_{k-1}\}), \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k\}> .\overline{u_+}) \leq \sum(<\max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\mathcal{Q}_2\ldots\mathcal{Q}_k\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k\})>.\overline{u_+}) \leq \{because\ \sum\overline{u_+} \leq\uparrow \overline{u}\ and\ by\ lemma\ A.6\} \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_1\mathcal{Q}_2\ldots\mathcal{Q}_k\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k\} + \uparrow\overline{u} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k\} + \uparrow\overline{u} \leq \{simple\ cycle\ u\ does\ not\ occur\ in\ \mathcal{Q}_k\} \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}_k u\} \leq \max(a, \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}) + \sum\{\uparrow\overline{\mathcal{C}} \mid \mathcal{C} \text{ is a simple cycle in } \mathcal{Q}\}$. Hence 3 also holds.

□

Bibliography

[1] TUeDACS, TU/e Data Acquisition and Control System, 2006. http://www.tuedacs.nl/.

[2] BrickOS, 2007. http://brickos.sourceforge.net/.

[3] Central Limit Theorem, 2007. http://en.wikipedia.org/wiki/Central_limit_theorem.

[4] Extensible Markup Language, 2007. http://www.w3.org/XML/.

[5] RTAI - the RealTime Application Interface for Linux, 2007. https://www.rtai.org/.

[6] The 20-sim Homepage, 2007. http://www.20sim.com/.

[7] TIMES Tool, 2007. http://www.timestool.com/.

[8] L. Abeni and G. Buttazzo. QoS Guarantee Using Probabilistic Deadlines. In: *Proceedings of the 11th IEEE Euromicro Conference on Real-Time Systems*, pp. 242–249. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[9] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126 (2): pp. 183–235, 1994.

[10] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In: *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pp. 74–106. Springer, Dordrecht, NL, 1992.

[11] R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104 (1): pp. 35–77, 1993.

[12] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In: *Proceedings of the Formal Modeling and Analysis of Timed Systems Workshop*, pp. 60–72. Springer, Dordrecht, NL, 2003.

[13] A. Atlas and A. Bestavros. Statistical Rate Monotonic Scheduling. In: *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 123–132. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[14] J.C.M. Baeten. The Total Order Assumption. In: *Proceedings of the First North American Process Algebra Workshop*, pp. 231–240. Springer, Dordrecht, NL, 1993.

[15] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36 (4): pp. 45–52, 2003.

[16] P. Bellini, R. Mattolini, and P. Nesi. Temporal Logics for Real-Time System Specification. *ACM Computing Surveys*, 32 (1): pp. 12–42, 2000.

[17] E. Bini, G.C. Buttazzo, and G. Buttazzo. A Hyperbolic Bound for the Rate Monotonic Algorithm. In: *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, pp. 59–66. IEEE Computer Society Press, Los Alamitos, CA, USA, 2001.

[18] G. Bosman. *A Survey of Co-Design Ideas and Methodologies*. Master's thesis, Vrije Universiteit Amsterdam, NL, 2003.

[19] B.H.M. Bukkems. *Sheet Feedback Control Design in a Printer Paper Path*. PhD thesis, Eindhoven University of Technology, NL, 2007.

[20] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, Dordrecht, NL, 1997.

[21] G.C. Buttazzo, G. Lipari, and L. Abeni. Elastic Task Model for Adaptive Rate Control. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 286–295. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[22] L. Chen. *Timed Processes: Models, Axioms and Decidability*. PhD thesis, University of Edinburgh, UK, 1993.

[23] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8 (2): pp. 244–263, 1986.

[24] M.B.G. Cloosterman, N. van de Wouw, W.P.M.H. Heemels, and H. Nijmeijer. Robust Stability of Networked Control Systems with Time-Varying Network-Induced Delays. In: *Proceedings of the 45th IEEE conference on decision and control*, pp. 4980–4985. IEEE Control Systems Society Press, 2006.

[25] H. Corporaal. Embedded Systems Design. In: *Progress. White Papers 2006*, pp. 7–28. Utrecht, 2006.

[26] R. de Nicola and F. Vaandrager. Action versus State Based Logics for Transition Systems. In: *Proceedings of the LITP Spring School on Theoretical Computer Science on Semantics of Systems of Concurrent Processes*, pp. 407–419. Springer, Dordrecht, NL, 1990.

[27] M. de Wulf, L. Doyen, and J.F. Raskin. Systematic Implementation of Real-Time Models. In: *Proceedings of Formal Methods*, pp. 139–156. Springer, Dordrecht, NL, 2005.

[28] B.P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[29] E.A. Emerson and E.M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2 (3): pp. 241–266, 1982.

[30] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. *Real-Time Systems*, 4 (4): pp. 331–352, 1992.

[31] J.A. Fisher and P. Faraboschi. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, San Francisco, CA, USA, 2005.

[32] O. Florescu, M. de Hoon, J.P.M. Voeten, and H. Corporaal. Performance Modelling and Analysis Using POOSL for an In-Car Navigation System. In: *Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging*, pp. 37–45, 2006.

[33] O. Florescu, M. de Hoon, J.P.M. Voeten, and H. Corporaal. Probabilistic Modelling and Evaluation of Soft Real-Time Embedded Systems. In: *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 206–215. Springer, Dordrecht, NL, 2006.

[34] O. Florescu, J. Huang, J.P.M. Voeten, and H. Corporaal. Strengthening Property Preservation in Concurrent Real-Time Systems. In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 106–109. IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

[35] O. Florescu, J.P.M. Voeten, and H. Corporaal. Property-Preserving Synthesis for Unified Control- and Data-Oriented Models. In: *Proceedings of the Forum on Specification & Design Languages 2005 (FDL'05)*, pp. 531–542, 2005.

[36] O. Florescu, J.P.M. Voeten, and H. Corporaal. *Applications of Specification and Design Languages for SoCs*, Chapter Property-Preservation Synthesis for Unified Control- and Data-Oriented Models, pp. 247–262. Springer, Dordrecht, NL, 2006.

[37] O. Florescu, J.P.M. Voeten, J. Huang, and H. Corporaal. Error Estimation in Model-Driven Development for Real-Time Software. In: *Proceedings of the Forum on Specification and Design Languages*, pp. 228–239, 2004.

[38] O. Florescu, J.P.M. Voeten, M. Verhoef, and H. Corporaal. Reusing Real-Time Systems Design Experience Through Modelling Patterns. In: *Proceedings of the Forum on Specification and Design Languages 2006*, pp. 375–380, 2006.

[39] O. Florescu, J.P.M. Voeten, M. Verhoef, and H. Corporaal. *Advances in Design and Specification Languages for Embedded Systems*, Chapter Reusing Systems Design Experience Through Modelling Patterns, pp. 329–348. Springer, Dordrecht, NL, 2007.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[41] M.G.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Eindhoven University of Technology, NL, 2002.

[42] S.V. Gheorghita, T. Basten, and H. Corporaal. Application Scenarios in Streaming-Oriented Embedded System Design. In: *Proceedings of the International Symposium on System-on-Chip*, pp. 175–178. IEEE Press, Piscataway, NJ, 2006.

[43] S.V. Gheorghita, S. Stuijk, T. Basten, and H. Corporaal. Automatic Scenario Detection for Improved WCET Estimation. In: *Proceedings of the 42nd Design Automation Conference*, pp. 101–104. ACM Press, New York, NY, USA, 2005.

[44] M. Gries. Methods for Evaluating and Covering the Design Space During Early Design Development. *Integration, the VLSI Journal*, 38 (2): pp. 131–183, 2004.

[45] M. Gries, J. Janneck, and M. Naedele. Reusing Design Experience for Petri Nets Through Patterns. In: *Proceedings of High Performance Computing*, pp. 453–458. Springer, Dordrecht, NL, 1999.

[46] M. Hendriks and M. Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In: *Proceedings of Workshop on Parallel and Distributed Real-Time Systems*, 2006.

[47] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier Science Publishers Ltd, Essex, UK, 2007.

[48] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In: *Proceedings of First International Workshop on Embedded Software*, pp. 166–184. Springer, Dordrecht, NL, 2001.

[49] T.A. Henzinger, Z. Manna, and A. Pnueli. An Interleaving Model for Real Time. In: *Proceedings of the 5th Jerusalem Conference on Information Technology*, pp. 717–730, 1990.

[50] J. Hooman, K. Hillel, I. Ober, A. Votintseva, and Y. Yushtein. Supporting UML-Based Development of Embedded Systems by Formal Techniques. *International Journal of Software and Systems Modeling*. To appear.

[51] J. Huang. *Predictability in Real-Time Software Design*. PhD thesis, Eindhoven University of Technology, NL, 2005.

[52] J. Huang, J.P.M. Voeten, M. Groothuis, J. Broenink, and H. Corporaal. A Model-Driven Design Approach for Mechatronic Systems. In: *Proceedings of the IEEE International Conference on Application of Concurrency to System Design*, pp. 127–136. IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.

[53] J. Huang, J.P.M. Voeten, A. Ventevogel, and L.J. van Bokhoven. Platform-Independent Design for Embedded Real-Time Systems. In: *Proceedings of the Forum on Specification and Design Languages*, 2003.

[54] Open SystemC Initiative. *SystemC 2.1 Language Reference Manual*. 2005.

[55] G. Kahn. The Semantics of Simple Language for Parallel Programming. In: *Proceedings of IFIP Congress*, pp. 471–475, 1974.

[56] H. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California at Los Angeles, USA, 1968.

[57] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In: *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 338–349. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

[58] J. Kim and K.G. Shin. Execution Time Analysis of Communicating Tasks in Distributed Systems. *IEEE Transactions on Computers*, 45 (5): pp. 572–579, 1996.

[59] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2 (4): pp. 255–299, 1990.

[60] S. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16 (1963): pp. 83–94, 1963.

[61] J.P. Lehoczky. Real-Time Queueing Theory. In: *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 186–195. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[62] H.R. Lewis and C.H. Papadimitriou. Elements of the Theory of Computation. *ACM SIGACT News*, 29 (3): pp. 62–78, 1998.

[63] Y.T.S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Springer, Dordrecht, NL, 1999.

[64] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *VLSI Signal Processing Systems*, 29 (3): pp. 197–207, 2001.

[65] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of the Association for Computing Machinery*, 20 (1): pp. 46–61, 1973.

[66] S. Manolache, P. Eles, and Z. Peng. Schedulability Analysis of Applications with Stochastic Task Execution Times. *ACM Transactions on Embedded Computing Systems*, 3 (4): pp. 706–735, 2004.

[67] Mechanical Engineering Department, Eindhoven University of Technology. Dynamics and Control Technology, 2007. `http://www.dct.tue.nl/`.

[68] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs NJ, 1989.

[69] M. Müller-Olm, D.A. Schmidt, and B. Steffen. Model-Checking: A Tutorial Introduction. In: *Proceedings of the 6th International Symposium on Static Analysis*, pp. 330–354. Springer, Dordrecht, NL, 1999.

[70] X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In: *Proceedings of the 3rd Workshop on Computer-Aided Verification*, pp. 376–398. Springer, Dordrecht, NL, 1991.

[71] X. Nicollin and J. Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114 (1): pp. 131–178, 1994.

[72] L. Noonan and C. Flanagan. An Effective Network Processor Design Framework: Using Multi-Objective Evolutionary Algorithms and Object Oriented Techniques to Optimise the Intel IXP1200 Network Processor. In: *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pp. 103–112. ACM Press, New York, NY, USA, 2006.

[73] Océ. VarioPrint 2090, 2007. `http://www.oce.com/nl/Products/Scanners/Office+documents/VP2090/default.htm`.

[74] OMG. *Unified Modeling Language (UML) - Version 1.5*. OMG document formal/2003-03-01, Needham MA, 2003.

[75] OMG. *UML Profile for Schedulability, Performance, and Time Specification - Version 1.1*. OMG document formal/2005-01-02, Needham MA, 2005.

[76] OMG. Systems Modelling Language (SysML) Specification, 2007. `http://www.sysml.org/specs.htm`.

[77] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P. van der Wolf, and E.F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *Computer*, 34 (11): pp. 57–63, 2001.

[78] POOSL, 2007. `http://www.es.ele.tue.nl/poosl`.

[79] A. Prior. *Past, Present and Future*. Oxford University Press, Oxford, UK, 1967.

[80] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1977.

[81] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd, Essex, UK, 1998.

[82] J.H. Sandee. *Event-Driven Control in Theory and Practice. Trade-offs in Software and Control Performance*. PhD thesis, Eindhoven University of Technology, 2006.

[83] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology, USA, 1995.

[84] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., New York NY, 1994.

[85] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design. In: *Proceedings of the 38th Design Automation Conference*, pp. 667–672. ACM Press, New York, NY, USA, 2001.

[86] J.A. Stankovic. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, 28 (4): pp. 751–763, 1996.

[87] Telelogic. Rhapsody, 2007. http://www.ilogix.com/sublevel.aspx?id=53.

[88] Telelogic. TAU Generation 2, 2007. http://www.telelogic.com/products/tau/g2/.

[89] The Mathworks. The Matlab/Simulink Homepage, 2007. http://www.mathworks.com.

[90] B.D. Theelen. *Performance Modelling for System-Level Design*. PhD thesis, Eindhoven University of Technology, 2004.

[91] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P. van der Putten, and J.P.M. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In: *Proceedings of the Fifth International Conference on Formal Methods and Models for Codesign*, pp. 139–148. IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.

[92] R. Thus. *Generation of Models Based on Modelling Patterns*. Master's thesis, Eindhoven University of Technology, NL, 2007.

[93] L.J. van Bokhoven. *Constructive Tool Design for Formal Languages: From Semantics to Executing Models*. PhD thesis, Eindhoven University of Technology, NL, 2002.

[94] L.J. van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Software Synthesis for System Level Design Using Process Execution Trees. In: *Proceedings of the 25th Euromicro Conference*, pp. 463–467, 1999.

[95] R. van de Molengraft, M. Steinbuch, and B. de Kraker. Integrating Experimentation into Control Courses. *IEEE Control Systems Magazine*, 25 (1): pp. 40–44, 2005.

[96] P. van den Bosch, M. Verhoef, G. Muller, and O. Florescu. Modeling of Hardware Software Performance of High-Tech Systems. In: *Proceedings of the Seventeenth International Symposium of the International Council on Systems Engineering*, 2007.

[97] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, NL, 1997.

[98] A.J.C. van Gemund. Symbolic Performance Modeling of Parallel Systems. *IEEE Transactions on Parallel Distributed Systems*, 14 (2): pp. 154–165, 2003.

[99] F.N. van Wijk, J.P.M. Voeten, and A.J.W.M. ten Berg. *System Specification & Design Languages*, Chapter An Abstract Modelling Approach Towards System-Level Design-Space Exploration, pp. 267–282. Springer, Dordrecht NL, 2003.

[100] J.P.M. Voeten. Performance Evaluation with Temporal Rewards. *Journal of Performance Evaluation*, 50 (2/3): pp. 189–218, 2002.

[101] J.P.M. Voeten, M. Geilen, L. van Bokhoven, P. van der Putten, and M. Stevens. A Probabilistic Real-Time Calculus for Performance Evaluation. *Proceedings of the 11th European Simulation Symposium*, pp. 608–617, 1999.

[102] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *Internation Journal on Software Tools for Technology Transfer*, 8 (6): pp. 649–667, 2006.

[103] G. Witvoet. *The Design of an Experimental Paper Path Setup*. Technical Report DCT-2005-141, Eindhoven University of Technology, NL, 2005.

[104] D.H.H. Yoon. A Survey of System Design Methodologies. In: *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems*, pp. 392–396. IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

Samenvatting

Voorspelbaar ontwerpen van tijdkritische systemen

De complexiteit van ingebedde tijdkritische systemen heeft onderzoek naar raamwerken en technieken voor het structureren en automatiseren van hun ontwerpproces gemotiveerd. Zulke raamwerken zijn voornamelijk van belang voor ingebedde en veiligheid-kritieke systemen die moeilijk correct te ontwerpen zijn binnen een beperkte tijd tot introductie op de markt. Ontwerpmethodologiën reduceren het risico van dure ontwerpherhalingen door het construeren van modellen te ondersteunen. Software/Hardware Engineering (SHE) is een algemeen toepasbare systeemniveau ontwerpmethodologie die analyse van zowel functionele correctheidsalsook prestatieëigenschappen mogelijk maakt. Door SHE te gebruiken wordt een ontwerper geholpen in het construeren van modellen en het toepassen van de analysetechnieken op basis van verscheidene richtlijnen en modelleringspatronen. Een belangrijke eigenschap van SHE is dat het gebaseerd is op formele methoden welke garanderen dat de verkregen analyseresultaten ondubbelzinnig zijn. SHE bevat ook richtlijnen en technieken voor automatische synthese van controlesoftware voor tijdkritische systemen. Ook dit is gebaseerd op formele methoden om te zorgen dat eigenschappen van een model (met inbegrip van tijdsgerelateerde eigenschappen) behouden blijven tijdens de softwarerealisatie. Dit proefschrift draagt zowel bij aan de modelleer- en analysefase van het ontwerp alsook aan de correcte synthese naar een efficiënte softwareimplementatie van een systeem.

Om de juistheid en prestatie van ingebedde tijdkritische systemen te kunnen garanderen, is een vroege evaluatie van hun eigenschappen noodzakelijk. Om dit te bereiken, hebben we een aantal taalonafhankelijke modelleringspatronen ontwikkeld die een eenvoudige opbouw van modellen voor ontwerpruimteëxploratie mogelijk maken. De modelleringspatronen bestrijken typische componenten voor tijdkritische systemen zoals deze worden beschouwd in klassieke scheduling theorie zoals periodieke en aperiodieke taken, middelen voor berekeningen en communicatie, generatoren van inputgebeurtenissen en verzamelaars voor outputgebeurtenissen. We hebben tevens een patroongebaseerde systeembeschrijvingstaal (PDL) ontworpen die kan worden gebruikt om een tijdkritisch systeem te beschrijven in termen van de benodigde patronen en de waarden van hun parameters. Deze taal is makkelijk te gebruiken en vereist geen kennis van welke modelleringstaal dan ook die typisch voor de analyse van tijdkritische systemen worden gebruikt. Het belangrijkste voordeel van deze taal is de mogelijkheid tot automatische generatie van modellen in verschillende modelleringstalen. Als voorbeeld presenteren we een implementatie van de modelleringspatronen in de algemeen toepasbare formele modelleringstaal genaamd Parallel Object-Oriented Specification Language

(POOSL), welke aan de SHE methodologie ten grondslag ligt, samen met een gereedschap dat een PDL beschrijving vertaalt in een POOSL model. Als resultaat van de expressiviteit van POOSL zijn de gegenereerde modellen geschikt voor zowel hard als soft tijdkritische systemen aangezien ze analyse van het systeemgedrag zowel in het slechtste geval alsook in een gemiddelde situatie toelaten, wat de juiste dimensionering van het eindproduct garandeert.

Om de implementatie van een parallel tijdkritisch systeem vanuit een model op "correcte" wijze uit te voeren, is het belangrijk om te begrijpen wat de relatie is tussen de eigenschappen van het model en die van z'n bijbehorende implementatie. In dit proefschrift presenteren we een mechanisme, waarvan mathematisch wordt bewezen dat het correct is, om de afwijking in het tijdgedrag tussen een model en z'n overeenkomstige implementatie te bepalen. Op basis daarvan kunnen we de eigenschappen van het gerealiseerde systeem voorspellen. Verder hebben we een metriek gedefiniëerd die het behoud van waarneembare eigenschappen tussen model en realisatie uitdrukt. We stellen een methode voor ter bepaling van een bovengrens voor deze metriek om zo de waarneembare eigenschappen van een realisatie te kunnen voorspellen op basis van het model. Tevens laten we zien dat deze bovengrens verlaagd kan worden door prioriteit te geven aan het uitvoeren van waarneembare acties. Op basis van dit resultaat hebben we een bestaande methode voor modelsynthese uitgebreid met een sterker behoud van eigenschappen.

Door middel van een praktijkvoorbeeld tonen we hoe de bijdragen van dit proefschrift toegepast kunnen worden voor de modelgedreven ontwerpmethode van het tijdkritisch systeem dat de controle van het doorvoeren van papier in een printer bewerkstelligt. Met dit voorbeeld laten we zien hoe een systeem kan worden gemodelleerd door gebruik te maken van de modelleringspatronen, hoe z'n tijdkritische eigenschappen kunnen worden afgeleid en geanalyseerd en hoe voorspellingen over de eigenschappen van de realisatie kunnen worden gedaan. De synthese van het systeem brengt een implementatie tot stand die deze eigenschappen behoudt en een correct lopend systeem garandeert.

Acknowledgements

First of all, I would like to thank Prof. Henk Corporaal for giving me the opportunity of doing my PhD in the Electronic Systems group. Henk is one of the most knowledgeable persons in the field and he provided me with careful guidance along these four years of research. The many brainstorming sessions that we had in this time helped me to advance in my research and to improve the results I obtained.

I would like to give my special thanks to Jeroen Voeten for all his support, advice and suggestions that he gave me in these years. In the beginning of my research, he helped me in finding my research direction. Along these four years of my PhD study, we had many interesting discussions and he has always encouraged me in applying the ideas that I had. Next to being a good supervisor, he was also a pleasant person to talk to and he gave me nice hints regarding travelling around.

I am grateful to Prof. Ralph Otten, the head of the Electronic Systems group, and to Marja and Rian, our group secretaries, for their kindness and help that they have always given me. I would like to thank my former colleagues in the group for the nice moments we shared: Akash, Amir, Andreas, Arno, Bart Theelen, Bart Mesman, Benny, Călin, Dominik, Hamed, Jose, Jurjen, Lech, Marc, Mark, Mathias, Patrick, Peter, Philip, Piet, Sander, Simona, Srinath, Szymon and Twan. I would like to mention Jinfeng particularly and thank him for the very interesting technical discussions that we had.

I would also like to thank my former colleagues in the Boderc project: Adriaan, Anget, Bauke, Björn, Maurice, Erik, Evert, Frans, Gerrit, Heico, Hennie, Jan, Jan-Mathijs, Jozef, Lou, Marcel, Marieke, Peter van den Bosch, Peter Visser and Zhouri. They have been nice colleagues and I enjoyed the time spent with them and the interesting discussions that we had these years. I would particularly like to thank Marieke for the enjoyable girls-talking lunches that we had together. Special thanks go to Björn, Jeroen de Best, René and Gerard for their help with the setup of my case study and for providing me the software package that I needed in order to implement the case study.

The members of the committee are specially appreciated for reading my thesis, giving good comments and participating in my defence session.

I wish to thank my friends here in the Netherlands, especially Ramona, with whom I shared cheerful moments and whose company made life more beautiful. Moreover, instant messaging and VoIP shortened the distance to all my friends from home and around the world who always had a smile for me.

My very warm thanks go to my parents who have supported and encouraged me along the long and difficult path of my studies. They have always believed in me

and encouraged me to pursue my dreams. I owe them this achievement.

The last but not the least, my wholehearted thanks go to my beloved husband Vali who has been by my side in both good and bad times. He both rejoiced with me at the happy moments, as well as encouraged me to go ahead and pass over the difficult times. We gave each other the strength and the support to go through this most challenging period of our life together. Without him, this book would not exist. With all my love, I dedicate this book to Vali.

Oana Florescu

Eindhoven, December 2007

Once you have travelled, the voyage never ends, but is played out over and over again in the quietest chambers, that the mind can never break off from the journey.

Pat Conroy (1945 -)

About the Author

Oana Florescu was born in Constanţa, Romania, on December 28th, 1978. She graduated from the Computer Science and Engineering Department within "Politehnica" University of Bucharest, in July 2002, as the third out of about 200 students. She carried out her graduation project at Motorola DSP R&D Center Romania during a six-month scholarship. The research of her project was on analysis of mapping applications on heterogeneous multi-processor architectures. In July 2003, she graduated from the Advanced Studies program at the same department as the first of her year.

Since September 2002, in parallel with her studies, she has also been working within the compilers team at Motorola DSP R&D Center Romania. In 2003, she has been offered a PhD candidate position at Eindhoven University of Technology, Netherlands. Since September 2003 until September 2007, Oana pursued her PhD degree in the Electronic Systems group at the Electrical Engineering Department. The focus of her research was on predictable design of real-time systems within the Boderc (Beyond the Ordinary: Design of Embedded Real-Time Control) project coordinated by the Embedded Systems Institute. In the summer of 2006, she went for a three-month internship at IBM Research Laboratory in Zürich, Switzerland. In October 2007, she returned to Zürich to join Google Inc.

Oana's personal interests are reading, travelling, dancing, especially Latin dances, and music, in particular singing. Since September 2005 until July 2007, she has been a member of the Eindhoven Students Music Association, performing classical music in the Vokollage choir.

List of Refereed Publications

Book Chapters

- O. Florescu, J.P.M. Voeten, M.H.G. Verhoef and H. Corporaal. *Advances in Design and Specification Languages for Embedded Systems*, Chapter Reusing Real-Time Systems Design Experience Through Modelling Patterns, pp 329-348. Springer, Dordrecht, NL, 2007.

- P. van den Bosch, O. Florescu and M.H.G. Verhoef. *Boderc: Model-Based Design of High-Tech Systems*, Chapter Modeling of Performance, pp 103-116. Embedded Systems Institute, Eindhoven, NL, 2006.

- O. Florescu, J.P.M. Voeten and H. Corporaal. *Boderc: Model-Based Design of High-Tech Systems*, Chapter Model-Driven Design of Real-Time Systems, pp 163-172. Embedded Systems Institute, Eindhoven, NL, 2006.

- O. Florescu, J.P.M. Voeten and H. Corporaal. *Applications of Specification and Design Languages for SoCs*, Chapter Property-Preservation Synthesis for Unified Control- and Data-Oriented Models, pp 247-262. Springer, Dordrecht, NL, 2006.

- J. Huang, J.P.M. Voeten, O. Florescu, P.H.A. van der Putten and H. Corporaal. *Advances in design and specification languages for SoCs*, Chapter Predictability in Real-Time System Development, pp 123-140. Springer, Dordrecht, NL, 2005.

Conferences

- P. van den Bosch, M.H.G. Verhoef, G. Muller and O. Florescu. *Modeling of Hardware-Software Performance of High-Tech Systems*. In: *Proceedings of the Seventeenth International Symposium of the International Council on Systems Engineering*, 2007.

- B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten and J. Voeten. *Software/Hardware Engineering with the Parallel Object-Oriented Specification Language*. In: *Proceedings of the Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pp 139-148, IEEE Computer Society Press, Los Alamitos, CA, USA, 2007.

- O. Florescu, J.P.M. Voeten, M.H.G. Verhoef and H. Corporaal. *Reusing Real-Time Systems Design Experience Through Modelling Patterns*. In: *Proceedings of the Forum on Specification and Design Languages*, pp 375-380, 2006. ***Best Paper Award***.

- O. Florescu, J. Huang, J.P.M. Voeten and H. Corporaal. *Strengthening Property Preservation in Concurrent Real-Time Systems*. In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp 106-109, IEEE Computer Society Press, Los Alamitos, CA, USA, 2006.

- O. Florescu, M. de Hoon, J.P.M. Voeten and H. Corporaal. *Probabilistic Modelling and Evaluation of Soft Real-Time Embedded Systems*. In: *Proceedings of the Embedded Computer Systems: Architectures, Modelling, and Simulation*, pp 206-215, Springer, Dordrecht, NL, 2006.

- O. Florescu, M. de Hoon, J.P.M. Voeten and H. Corporaal. *Performance Modelling and Analysis Using POOSL for an In-Car Navigation System*. In: *Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging*, pp 37-45, 2006.

- O. Florescu, J.P.M. Voeten and H. Corporaal. *Property-Preservation Synthesis for Unified Control- and Data-Oriented Models*. In: *Proceedings of the Forum on Specification and Design Languages*, pp 531-542, 2005.

- O. Florescu, J.P.M. Voeten and H. Corporaal. *A Unified Model for Analysis of Real-Time Properties*. In: *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*, pp 220-227, 2004.
- O. Florescu, J.P.M. Voeten and J. Huang and H. Corporaal. *Error Estimation in Model-Driven Development for Real-Time Software*. In: *Proceedings of the Forum on Specification and Design Languages*, pp 228-239, 2004.
- S.V. Gheorghita, W.F. Wong and O. Florescu. *EPIC − Adaptive EPIC Bridge*. In: *Proceedings of the 14th International Conference on Control Systems and Computer Science*, vol. 2, pp 92-97, 2003.

Theses

- O. Florescu. *Compiler Optimisations for the PowerPC Architecture*. Advanced Studies Graduation Thesis, "Politehnica" University of Bucharest, RO, 2003.
- O. Florescu. *DSP-MCU Bridge*, Graduation Thesis, "Politehnica" University of Bucharest, RO, 2002.