Modeling and Validating Distributed Embedded Real-Time Control Systems

Marcel Verhoef

ISBN 978-90-9023705-3

Typeset with LaTeX 2_ε
Printed by Print Partners Ipskamp, Enschede
Cover design by Silvian de Jager

Abstract

The development of complex embedded control systems can be improved significantly by applying formal techniques from control engineering and software engineering. It is shown how these approaches can be combined to improve the design and analysis of high-tech systems, both in theory and practice. The semantics of the integration of two established rigorous techniques has been defined formally in this work. The strength of this integrated semantics is demonstrated by means of a significant industrial case study: the embedded control of a printer paper path, whereby the full development life-cycle from model to realization is covered. The resulting model-driven design approach fits the current engineering practice in industry and is both flexible and effective.

IPA dissertation series 2009-01

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and under responsibility of the Embedded Systems Institute as part of the Boderc project, which was partially supported by the Dutch Ministry of Economic Affairs under the Senter TS progam.

Modeling and Validating Distributed Embedded Real-Time Control Systems

Een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus, prof. mr. S. C. J. J. Kortmann,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op woensdag 21 januari 2009
om 15.30 uur precies

door

Marcel Henri Gerard Verhoef

geboren op 5 augustus 1968
te Papendrecht

Promotor:

Prof. dr. F. W. Vaandrager

Copromotor:

Dr. J. J. M. Hooman, Radboud Universiteit Nijmegen &
Embedded Systems Institute

Manuscriptcommissie:

Prof. dr. B. P. F. Jacobs

Prof. dr. L. Thiele, Swiss Federal Institute of Technology Zürich (CH)

Prof. dr. ir. P. G. Larsen, Engineering College Aarhus (DK)

Dr. ir. J. P. M. Voeten, Technische Universiteit Eindhoven &
Embedded Systems Institute

Dr. ir. J. F. Broenink, Universiteit Twente

Preface

This dissertation is long overdue. Already in 1993, when I graduated from Delft University of Technology, I had been looking for opportunities for a suitable PhD position in Computer Science. But I did not succeed in finding one for various reasons. So, I ended up in industry and spent ten years of my life building complex computer systems and being reasonable successful at it as well. Every now and then there was still this itch that needed satisfaction, of leaving behind unfinished business in academia. But the pressure of working in industry usually kept me far from doing something sensible about it. Having seen several colleagues trying to complete a PhD along side a full time job did not really help either.

All this changed in November 2002 when an advertisement appeared in "Technisch Weekblad" seeking PhD candidates for an applied research project on multidisciplinary design of real-time embedded control systems at the Embedded Systems Institute. Having struggled with these issues for several years in industry made me decide to seek a position in this project, called BODERC, and the result is now in front of you. It was both a blessing and a challenge returning to academia after so many years. Of course I brought a lot of practical experience to the project, but my academic skills needed polishing, and a lot of it as well. But with the help of old and new friends, I believe we did some interesting work and had a lot of fun doing it.

Of course, papers where produced and conferences were visited. But, perhaps because I was the only BODERC PhD candidate with a priori grey hair, I also got involved in quite a number of interesting "extra-curricular" activities, such as writing a book on VDM++ and organizing a special session at ISOLA'04 which led to the publication of a special issue of the STTT journal which I also co-edited. I have been part of two programme committees of high profile symposia and I visited ETH Zürich, University of Newcastle and the Engineering College at Aarhus several times for joint research work. I was co-organizer of the Industry Day at FM'05 at Newcastle and last but not least I was asked to give two invited lectures at Boston Scientific at Minneapolis, USA, and CSK Systems at Tokyo and Nagoya in Japan on early results of my research work.

So, obviously, completing this thesis before the BODERC project was finished in March 2007 was out of the question. Since my return to Chess, I have been working on this document as any other project in industry: running from one infeasible deadline to another, using floating priority scheduling. But here it is, my magnus opus, and I hope that you enjoy reading it as much as I have enjoyed writing it. Finally, my sincere apologies to those who have tried to provoke and stimulate me to take (and complete) a PhD over the past years and had to wait 15 years on this result!

Voor mijn ouders:

Gerrit Hendrik Verhoef en

Dikkie Adriana van Herk

Contents

Chapter 1

Introduction

Computers are all around us and we use them every day, sometimes even without giving it a second thought. The term "computer" often refers to the personal computer (PC), which is used to send e-mail and browse the Internet or perhaps a video game console that is used for entertainment. But computers are also part of the alarm clock, coffee machine, dishwasher, video recorder, DVD player, photo camera, television set and mobile telephone. This class of systems is often referred to as "embedded systems". Wikipedia defines an embedded system [1] as: *"a special-purpose system in which the computer is completely encapsulated by the device it controls"*. Corporaal observes in a recent white paper [20] that you can easily count up to a hundred embedded devices in an average family household nowadays.

We become more and more dependent on the proper operation of these embedded systems. Not only because they are efficient and convenient to use but also because they potentially affect the quality of life. Sangiovanni-Vincentelli mentioned in his presentation [89] at the 2006 Design Automation and Test in Europe (DATE) conference that a modern, high-end, car contains 80 microprocessors executing several million lines of code. These microprocessors are used to control not only the car radio and air conditioning, but also the air bag, cruise control, fuel injection, brakes and power steering. A failure in any one of those critical embedded systems may have severe consequences. But the general public is typically not aware of this, because these computers are deeply embedded in the system, hidden well out of plain sight. Dependability issues are typically associated with the military, medical or aeronautical domains but not so much with consumer or capital goods. For example, does one ask about the code coverage statistics of the power steering unit (an embedded system that contains a microprocessor which executes possibly several thousands lines of code) when you buy a new car? In 2004, Deutsche Welle reported [2] that the reliability rating of German cars, which used to be unrivalled and universally acclaimed, has been steadily decreasing for several years in succession as compared to their main competitors. Analysts believe that this may very well be due to the increased complexity as outlined by Sangiovanni-Vincentelli.

The impact of embedded systems is likely to grow even far beyond what is possible today. The on-going miniaturization and wireless digital communication has made mobile computing already a reality. We have seen the desktop PC shrink, first to a laptop and then to a personal digital assistant (PDA) in less than a decade, without a

[1] See *http://en.wikipedia.org/wiki/Embedded_systems*

[2] See *http://www.dw-world.de/dw/article/0,2144,1400331,00.html*

significant loss in performance. Personal audio systems, like Apple's iPod, are now common place. In principle, you can reach any one, at any place, at any time. The growth in application areas seems to be limited only by the amount of power such a mobile device requires. Visionaries claim that we will be moving towards *ubiquitous computing* [3], a paradigm whereby the distinction between computers and their environment will eventually disappear completely. Advocates like Aarts [1] also refer to this as "ambient intelligence". Companies such as Philips [4] already demonstrate that this is not just science fiction. They are building actual prototypes of products for the consumer, lighting and medical markets based on these ideas in their ExperienceLab.

The economic relevance of embedded systems is easily demonstrated. For example, take mobile telephony. Market analysts such as Informa Telecoms & Media [5] predicted in 2005 that the number of mobile hand-sets deployed world-wide would reach 1 billion early in 2007 which corresponds to roughly twenty percent of the population on Earth! Moreover, this target was reached in just fifteen years and the market is far from saturated. Growth is expected to continue by at least ten percent per year until 2012. These numbers are just staggering and it is obvious that such a market potential generates an enormous amount of pressure on the companies that build these kinds of products. Production volumes are extremely high, profit margins are typically low which implies that you have to reach the market with a new product before your competitor, in order to be economically successful. This so-called "time-to-market" (TTM) pressure is therefore the beast to beat.

Companies invest huge amounts of money and effort in order to reduce the production time and cost-price of their products. This has created a secondary economy consisting of companies that deliver (half-) products and services to achieve those goals. For example, Gartner [6] reports that the revenues for electronic design automation (EDA) will experience double-digit growth in 2006, reaching 4.5 billion US Dollar. But do all these investments lead to good products? Unfortunately not. It seems that the well-known adage *"Price, Time, Quality - Pick Any Two for Success"* is still a fact of life, as is shown in Figure 1.1.

After the famous CHAOS report from the Standish Group [7] appeared in 1994, there have been numerous published examples of projects failing or products malfunctioning. Despite efforts to improve the quality of computerized systems, it remains difficult to make error-free systems. Most surprisingly, end-users seem to have accepted that as a given fact. People are used to reboot their computer if a problem occurs. If it does not work, you just download the latest software from the web-site. Updates and upgrades have become part of the business model of the product. Even more so, only limited warranties [8] are provided and companies typically do not accept any liability from the use of their products. Would you buy a car if you would have to sign such a legal document? Open source software comes with a so-called *"as-is"* disclaimer, without warranty of any kind. The GNU General Public License [9] actually contains the following sentence: *"The entire risk as to the quality and performance of the program is with you.*". Yet, open source software is often believed to be of higher quality than most commercial software, because it is exposed to public scrutiny.

[3] See *http://en.wikipedia.org/wiki/Ubiquitous_computing*

[4] See *http://www.research.philips.com/*

[5] See *http://www.informatm.com*

[6] See *http://www.gartner.com*, Doc. Id. G00143619

[7] See *http://www.standishgroup.com/sample_research/chaos_1994_1.php*

[8] See for example the End-User Licence Agreement at *http://www.microsoft.com*.

[9] See *http://gplv3.fsf.org*.

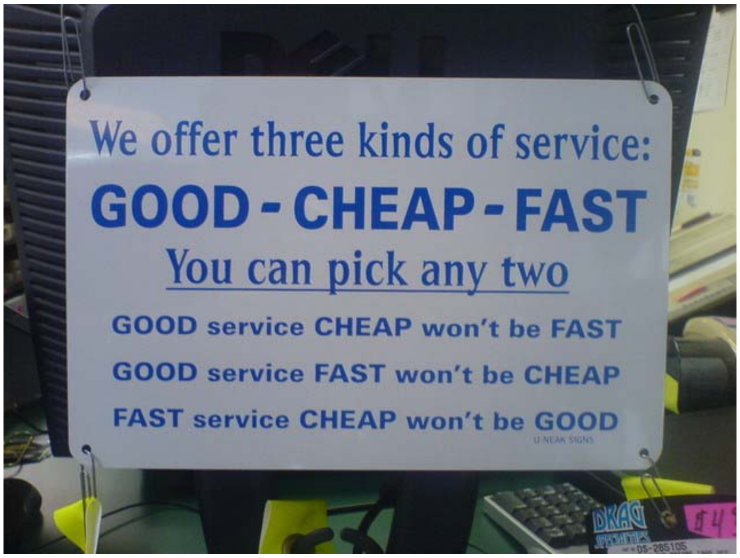

We offer three kinds of service:
GOOD - CHEAP - FAST
You can pick any two
GOOD service CHEAP won't be FAST
GOOD service FAST won't be CHEAP
FAST service CHEAP won't be GOOD
U NEAK SIGNS

Figure 1.1: Decision making at large: how to find the optimum?

Quality is a major issue in embedded systems development, mainly because of the production volumes involved. Intel Corporation was forced to recall a substantial number of their early Pentium processors in 1994 because a problem was found in the floating point unit after the product release. Harrison reported at the 2005 ForTIA Industry Day [72] that Intel wrote off 475 million Dollar because of the Pentium FDIV bug and suffered considerable damage to their reputation. But even in a low-volume market things can go spectacularly wrong with great consequences. On June 4, 1996, the inaugural flight of the Ariane-5 rocket failed. About 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. The Cluster mission, consisting of four identical scientific satellites, was lost during this event. Conservative estimates suggest that this accident costed the European tax payer in the order of 300 million Euro. The Inertial Reference System (abbreviated in French: SRI), which is used to determine the attitude of the launcher, shut down mid flight because an exception occurred in the software calculating the current flight path. Virtually the same system had been used to launch Ariane-4 rockets successfully for many years, but it was used outside its original specification in this particular case. The investigation showed that the system was never tested under flight conditions despite suggestions from the responsible engineers. In fact, the Ariane-5 Accident Report [71] states: *"... it was jointly agreed* not *to include the Ariane-5 trajectory data in the SRI requirements and specification.*". However, the report does *not* state *why* this decision was made. It is commonly believed that the time-to-market pressure, to have this new generation launcher operational as soon as possible, may have contributed to this decision, taking into account the excellent track-record of a similar system on Ariane-4. Johnson reports on similar problems at NASA in [59]. The "Faster, Better, Cheaper" initiative, which was announced in 1998, fostered a culture in which engineers took considerable risks to innovate with new design in order to meet requirements. In hindsight, time-to-market is one of the contributing factors [10] to the loss of the Mars Polar Lander mission in 1999.

1.1 The embedded systems design challenge

One might argue that the examples mentioned above are somewhat dated and do not reflect the current state of practice. But in fact, Johnson has demonstrated in [60] that the average likelihood of projects succeeding has only *marginally improved* over the last decade, despite substantial investments in tools and processes. It is generally believed that the performance in the embedded systems domain is rather worse than better. Why is this the case? Looking at general trends there are a few potential reasons.

The design gap problem. According to Moore's Law [74], the performance of hardware is roughly doubled every eighteen months. But recent advances in networking, packaging and integration technology has enabled the development of heterogeneous embedded computing platforms that show a potential exponential growth in performance and thus complexity [58]. These platforms are commonly referred to as System-On-Chip or Network-On-Chip and usually combine multiple and interconnected radio-frequency, analog and digital components on a single chip. However, the technology we use to design the applications for these new platforms cannot keep up with this tremendous growth in capabilities, primarily because they are currently focused on designing single, monolithic systems. In other words, the complexity of the problem grows much faster than the capabilities of today's leading design tools. This is commonly referred to as the "design gap".

The moving target problem. Rapidly evolving technology and the constant quest for reducing cost-price forces designers of embedded systems to operate on the edge of what is technically feasible. In order to stay competitive they sometimes need to adopt novel technology even while a product is already under development. One of the key problems in embedded systems design is the *validation* of these design decisions. How much effort and time does it take to check that the intent of a design choice works out in practice? Over-dimensioning is the usual approach to accommodate for uncertainty in the design but this is typically not economically viable because it increases the cost price. Sometimes actual prototypes need to be built in order to assess the feasibility of some potential solution. Managing this process is regarded as the key to success and it is often referred to as "shooting at a moving target".

The requirement versus design paradox. Making design decisions in the early phases of the system life-cycle is notoriously difficult. In this stage, requirements are often unclear and under-specified, at best leading to a long list of properties that the system shall eventually satisfy. In the past, emphasis has been put on managing the requirements process, such that sufficient information is available at the time the design decisions are made. However this is often not realistic, in particular in the domain of embedded systems. At the time when requirements are elaborated, the major architectural design decisions also need to be taken, primarily in order to meet the time-to-market target for the product. But how can one make these crucial decisions when there is still so much uncertainty? This is in particular true for performance criteria that the system must meet because they are in general surprisingly hard to quantify and evaluate. It is obvious that elaboration of the requirements is guided by the chosen architecture but in turn the definition of the architecture depends on clear and unambiguous requirements. System architects have to deal with this paradox, for example by applying iterative development processes in order to close the design loop.

Multi-disciplinary design. Systems are traditionally designed in a mono-disciplinary style usually with an organizational structure to reflect this (e.g. mechanical department, electronics department, software department and so on). While in the past systems where developed out-of-phase (mechanical design precedes electrical design which in turn precedes software design) nowadays concurrent engineering is applied in order to save development time. However, system-level requirements that cannot be assigned to a single discipline, such as performance, typically cause great problems during the integration phase because the responsibility to meet the requirement is shared among all disciplines. The root cause of this problem is the lack of cross-discipline design interaction. This problem cannot be solved by improving the internal organization; the way (embedded) software is currently being developed is fundamentally different from, for example, mechanical and electrical design. These engineers basically speak a different language, are concerned about different types of problems and use different techniques to address and solve these problems. This challenge is dominant in the embedded systems domain because the computer and the device it controls both loose their function if they were to be separated. Hence, they cannot be designed in isolation which makes the cross-discipline communication mandatory.

1.2 The BODERC research project

The issues listed in the previous section played an important role in defining the objectives for a new research project at the Embedded Systems Institute (ESI) in the summer of 2002. The central idea was to explore model-based engineering as a methodology for the design and analysis of high-tech systems. It was believed that: "*the product creation time can be reduced significantly by the use of multi-disciplinary models during the early product development phases*" [47]. The project should therefore bring researchers from different engineering disciplines and industrial practitioners together in an "Industry as a Laboratory" setting [82]. Océ Technologies [10], a leading manufacturer of high-volume document printing systems, became the so-called carrying industrial partner or "problem statement owner" in the project. Océ and ESI spearheaded a consortium consisting of the companies Imtech and Chess and researchers from the Technical University of Eindhoven, Radboud University Nijmegen and the University of Twente. The project was partially financially supported by the Netherlands Ministry of Economic Affairs under the Senter TS program.

The difficulty of multi-disciplinary research was already demonstrated during the definition phase of the project. The participants were unable to reach an agreement on the definition of the term "model". Each discipline seemed to have its own definition that was incompatible with what others used. The debate continued until one of the participants observed that Bo Derek, the famous movie actress, is also a model and this point was of course conceded quickly. It actually inspired the name of the project: BODERC, which is an acronym that stands for "Beyond the Ordinary: Design of Embedded Real-time Control". The project was started in September 2002 and was completed in March 2007.

High-tech mechatronic systems, such as high-volume printers, are complex and so is the associated design process. Many implementation choices need to be made and the impact of each decision is difficult to assess due to this inherent complexity. This makes the design process error prone and vulnerable to failure as other downstream design

[10] See *http://www.oce.com*.

choices may be based on it, causing a cascade of potential problems. Moreover, it may take some time to realize that a decision is wrong because it will require feedback in the design process. Usually this happens at system integration and testing or product manufacturing. The repairs required to fix these problems cause significant project delays and cost overruns or sometimes even worse: product cancelation. Three reasons are identified in the BODERC project that seem to be the root cause of this problem and they are listed here for convenience:

1. Reasoning about system-level properties is difficult because a common language is lacking. Each engineering discipline uses its own method, vocabulary and style of reporting. This incompatibility causes confusion often leading to misunderstandings and wrong assumptions being made on the sub-designs of other disciplines. These inconsistencies are hard to spot because there is usually no structured system design reasoning process in place.

2. Many design choices are made implicitly, usually based on previous experience, intuition or even assumptions. System-level reasoning is made difficult if the rationale behind such a decision is not quantified. The reasons are sometimes kept hidden on purpose, for example if strong personal preference or politics plays a role. This may perhaps lead to a local optimum in the system design but only rarely to a global optimum. It is therefore necessary to make design knowledge explicit in order to enable the dialogue at the system level.

3. Dynamic or time dependent aspects of a system are complex to grasp and more over, there are not many methods and tools available to support reasoning about time varying aspects in design, in contrast to static or steady-state aspects.

The effects of the above mentioned points are amplified by the complexity of the product under development (a high-volume printer typically consists of tens of thousands of components and millions lines of code) and the complexity of the design process (number of people involved, organizational structure, out-of-phase or multi-site development, etcetera). The hypothesis of the project is that light-weight models that capture the system-level behavior and a reasoning method that indicates how and when to use them will release the aforementioned tension considerably. A good system engineering methodology shall expose implicit or hidden design choices and replace the usual “hand-waving” by design rationale which is based on objective, quantified and verifiable information.

The goal of the BODERC project is graphically presented in Figure 1.2. The aim is to develop a model-based methodology that supports multi-disciplinary design (space exploration) by predicting system performance. The developed models, methods and techniques shall be applicable in the early design phases and must satisfy industrial application constraints. Hence, the methodology shall be usable in an industrial context with its particular people, organization and constraints such as product and process legacy, time, effort and money.

1.3 The goal of this thesis

The aim of this thesis is to define a method that supports the multi-disciplinary design of embedded systems. This is obviously a broad field, therefore we focus on the scope set by the BODERC project definition: the design of distributed real-time control systems. Furthermore, a number of challenging sub-goals were defined:

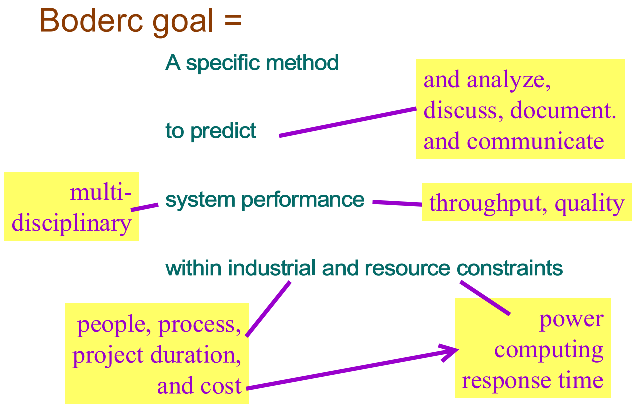

Boderc goal =
A specific method
to predict
and analyze, discuss, document. and communicate
multi-disciplinary
system performance
throughput, quality
within industrial and resource constraints
people, process, project duration, and cost
power computing response time

Figure 1.2: The BODERC research project goal from [47]

1. The method shall be able to address design problems at the system level; these are so-called cross-cutting concerns that usually affect more than one engineering discipline such as performance and dependability.
2. The method shall be able to predict whether or not both the functional and extra-functional properties are satisfied by the proposed system architecture.
3. The method shall provide means of abstraction that are appropriate for modeling the problem at hand; this may require support for different levels of abstraction for different parts of the problem but from within a single framework.
4. The method shall be cost effective; the amount of effort invested in modeling should be balanced with the insight gained from the analysis.
5. The method shall be easy to adopt for the average engineer currently working in the field at acceptable initial investment.

The overall aim is to be able to address industrial size problems, whereby the “grand challenge” is modeling and analysis of the paper path of a high-volume printer. The goal of this thesis is to investigate whether or not a method exists, or can be defined, that satisfies the requirements listed above, whereby its overall effectiveness is studied and demonstrated on this industrial case study.

1.4 Organization of this thesis

The first part of this thesis looks at several state-of-the-art performance evaluation methods and tools. The aim of the exercise is to understand the capabilities and limits of those methods by applying them on the same case study. Modular Performance Analysis (MPA), Symbolic Timing Analysis for Systems (SymTA/S), Parallel Object-Oriented Specification Language (POOSL), Timed Automata and the Vienna Development Method (VDM) are used to model and analyze an in-car radio navigation system. The result of this comparison is presented in Chapter 2 and it is based on the following publications:

[105] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef and Paul Lieverse. *System Architecture Evaluation using Modular Performance Analysis - A Case Study*. Appeared in *International Journal of Software Tools for Technology Transfer (STTT)*, Volume 8, No 6, pages 649-667. Springer, 2006. This is an extended version of the research paper that was published at the First International Symposium On Leveraging Applications of Formal Methods - ISOLA 2004.

[51] Martijn Hendriks and Marcel Verhoef. *Timed Automata Based Analysis of Embedded Systems Architectures*. Appeared in the proceedings of 20th International Parallel and Distributed Processing Symposium IPDPS 2006, *Workshop on Parallel and Distributed Real-Time Systems - WPDRTS*. IEEE, 2006.

[34] Oana Florescu, Jeroen Voeten, Marcel Verhoef and Henk Corporaal. *Reusing Real-Time Systems Design Experience Through Modelling Patterns*. Chapter in *Advances in Design and Specification Languages for Embedded Systems*. Selected papers FDL 2006 (best paper award). pages 329-348. Springer, 2007.

The second part of this thesis addresses the concerns identified during the comparison. Method improvements are proposed and implemented. Small case studies are used to check the upgraded tool support. Timed VDM++ is extended with asynchronous operations and an explicit notion of system architecture in Chapter 3. The semantics of these language extensions are defined and the implemented tool support is again applied to the in-car radio navigation case study. The improved VDM++ notation is coupled to 20-SIM, a dynamic systems modeling and simulation environment in Chapter 4. The semantics of both methods is reconciled and the integrated tools are applied to a case study: a water tank level controller. The results presented are based on the following publications:

[101] Marcel Verhoef, Peter Gorm Larsen and Jozef Hooman. *Modeling and Validating Distributed Embedded Real-Time Systems with VDM++*. Appeared in the proceedings of *FM 2006: Formal Methods*. LNCS 4085, pages 147-162. Springer, 2006.

[102] Marcel Verhoef, Peter Visser, Jozef Hooman and Jan Broenink. *Co-simulation of Real-Time Embedded Control Systems*. Appeared in the proceedings of *Integrated Formal Methods*. LNCS 4591, pages 639-658. Springer, 2007.

[56] Jozef Hooman and Marcel Verhoef. *Formal Semantics of a VDM Extension for Distributed Embedded Systems*. Paper included in festschrift to honor professor Willem-Paul de Roever. In *Correctness, Concurrency and Compositionality*. LNCS Festschrift Series, Springer, 2008 (to appear).

The third part of this thesis puts these results back into the industrial context. Both the embedding of the methods and tools in an industrial design process is considered in Chapter 5 and their application to the "grand challenge" of this thesis in Chapter 6 : the printer paper path. And, last but not least, the result of this research work is discussed and evaluated in Chapter 7. This work is based on the following publications:

[88] Heico Sandee, Maurice Heemels, Gerrit Muller, Peter van den Bosch, Marcel Verhoef. *Threads of Reasoning: A Case Study in Printer Control*. Appeared in the proceedings of the 16th Annual International INCOSE Symposium. International Council on Systems Engineering, 2006.

[32] John Fitzgerald, Peter Gorm Larsen, Simon Tjell, Marcel Verhoef. *Validation Support for Distributed Real-Time Embedded Systems in VDM++*. Appeared in the proceedings of the 10th IEEE High Assurance Systems Engineering Symposium, pages 331-340. IEEE, 2007.

[4] Zoe Andrews, John Fitzgerald, Marcel Verhoef. *Resilience Modeling Through Discrete Event and Continuous Time Co-simulation*. Extended abstract appeared in the proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2007.

Other publications

Several related published works are not included in this thesis:

[96] Peter van den Bosch, Gerrit Muller, Marcel Verhoef and Oana Florescu. *Modeling of Hardware Software Performance in High-Tech Systems*. Appeared in the proceedings of the 17th Annual International INCOSE Symposium. International Council on Systems Engineering, 2007.

[100] Marcel Verhoef and Peter Gorm Larsen. *Interpreting Distributed System Architectures Using VDM++*. Appeared in the proceedings of the 5th Conference on System Engineering Research. Stevens Institute of Technology, 2007.

[30] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat and Marcel Verhoef. *Validated Designs for Object-Oriented Systems*. ISBN 1-85233-881-4. Springer, 2005, 402 pages.

[99] Marcel Verhoef and Jozef Hooman. *Evaluating Embedded Systems Architectures*. Summary of Chapter 2 from this thesis which appeared in the BODERC final report [47]. Pages 151-160. ESI, 2006.

[95] Peter van den Bosch, Oana Florescu, Marcel Verhoef and Gerrit Muller. *Modeling of performance*. Chapter in the BODERC final report [47]. Pages 101-114. ESI, 2006.

[72] Tiziana Margaria, Bernhard Schätz, Marcel Verhoef. *Formal Methods Going Mainstream: Costs, Benefits, Experiences*. Report on the ForTIA Industry Day at FM 2005. Appeared in the BCS FACS Newsletter Issue 2006-2, pages 34-38. British Computer Society, 2006.

[98] Marcel Verhoef. *On the use of VDM++ for Specifying Real-time Systems*. Appeared in the proceedings of the First Overture Workshop at FM 2005, Technical Report CS-TR-969, pages 26-43. Newcastle University, 2006.

Chapter 2

Evaluating Embedded System Architectures

2.1 Introduction

An architectural description of a product is typically made during the initial phases of an industrial product creation process. For example, an *Operational Concept Description* document from the IEEE 12.207 system life cycle standard may be produced. Such a document does not only list functional and extra-functional requirements, boundary conditions and other restrictions for the design, but it also contains high-level Use-Cases. These Use-Cases, or scenarios, describe how the system is typically used and they are the starting point for the design of the embedded system architecture. Although there is no principle limit to the number of scenarios that can be analyzed, it is not uncommon to first concentrate on those Use-Cases that have the highest expected impact on the set of target system-level requirements. It is the system architect who makes this selection, often based on previous experience. Quantitative performance analysis can be used to guide the design process.

However, when a new system is being developed, there is typically little quantitative data available to work with. Therefore, course grain assumptions are used initially. Typically, these values are "guestimates" or extrapolated performance figures obtained from systems developed previously. During the design and development phases, the system architect will constantly try to improve the accuracy of the models by using for example better estimation techniques on details of the design, such as worst-case execution time analysis of existing or new source code, by benchmarking new critical system components on the target hardware or by performing measurements on existing and comparable systems. It is clear that performance analysis is an activity that needs to be performed throughout the system life cycle, in particular because requirements are likely to change over time.

In this chapter, we investigate several techniques that can be used to evaluate performance properties of embedded system architectures such as latency, throughput and resource utilization. We focus on these properties in particular, because they play a significant role in the selection of a suitable embedded architecture. The challenge is to decide, at design time:

- which of the proposed architectures is best suited, or,
- how to distribute functionality on a proposed embedded architecture, or,
- how to select suitable architecture parameters,

such that required performance targets and cost levels are met. This is often a trade-off between competing or even adversary requirements. Performance analysis techniques can be used to expose these design conflicts. This is demonstrated in this chapter by applying five different techniques to a common case study. These techniques are: Modular Performance Analysis (MPA), Symbolic Timing Analysis for Systems (SymTA/S), Timed Automata, the Parallel Object-Oriented Specification Language (POOSL) and the Vienna Development Method (VDM++). The aim of the experiment is to better understand the capabilities and limits of each method and to determine the value of the predictions derived from each model. It is certainly not the intent to determine which method is best. The experiment is too small and it has not been executed under controlled circumstances. The comparison was performed during the ARTIST2 workshop held at the Lorentz Center at Leiden University (November 2005) where experts on the relevant techniques where challenged to attack a common set of problems [1]. The case study described in the next section, which was originally used for an early version of [105], was put forward by the author of this thesis. The scope of the comparison presented here is limited to those techniques that were represented at the workshop.

Some interesting observations can be drawn from the comparison because the five techniques and their associated tools are very different. MPA is based on a deterministic queuing theory and uses Matlab as a front-end to compose and analyze abstract performance models extremely efficiently. SymTA/S combines deterministic queuing theory with classical scheduling theory to build abstract performance models using a nice and intuitive user interface. MPA and SymTA/S both provide hard, but not necessarily tight, results. Timed Automata is a general purpose modeling framework that can be analyzed using the UPPAAL model checker, possibly leading to accurate results. POOSL and VDM++ belong to the class of formal modeling languages that can be subjected to rigorous analysis techniques such as interactive theorem proving and model checking. However, discrete event simulation is used to analyze the POOSL and VDM++ models here. While MPA and SymTA/S abstract away from the actual computation that is performed by the system, Timed Automata, POOSL and VDM++ allow to describe the system functionality in more detail.

The well-known Y-chart, as proposed by Kienhuis et al in [62] and shown in Figure 2.1, is used as a framework for our comparison. The central idea of the Y-chart, which is not specific to any performance analysis technique in particular, is to build an abstract model of the concrete system that bundles all information needed for performance analysis. The following steps are taken to construct a Y-chart model:

1. identify key usage scenarios and system functions and quantify event rates, message sizes and execution times;
2. identify resources and their communication structure and quantify resource and communication capacities;
3. compose a system model, calculate (or simulate) and evaluate.

[1] The problem set can be found at *http://www.tik.ee.ethz.ch/∼leiden05*

The model resulting from steps 1 and 2 unifies essential information about the environment, about the available computation and communication resources, about the application tasks (or dedicated HW/SW components), as well as the system architecture itself. First, the Y-chart is explained in more detail.

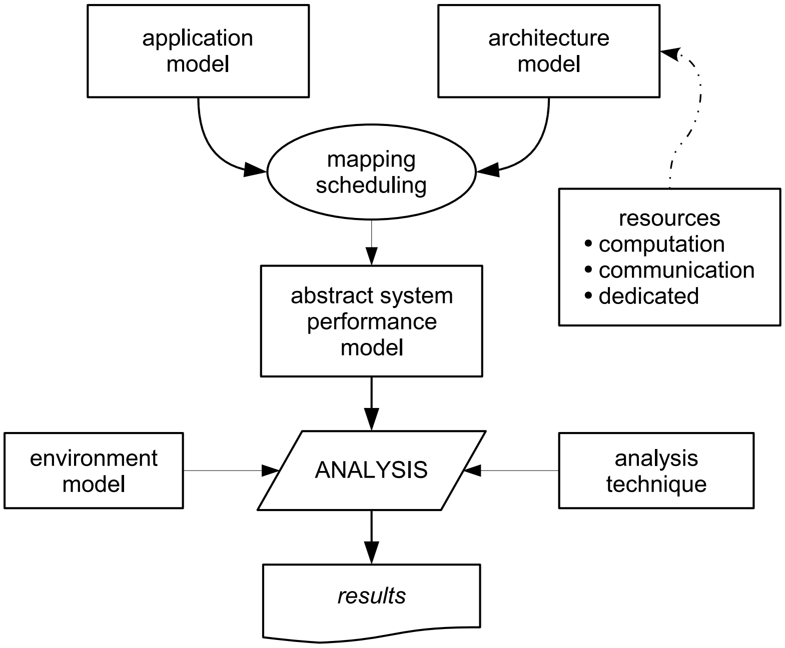


Figure 2.1: The Y-chart method for performance analysis

The application model (top-left in Figure 2.1) provides an abstract definition of the software or application logic that needs to run on the system. Application specific performance requirements are usually specified in this model. We will demonstrate how annotated UML sequence diagrams can be used to represent application models.

The architecture model (top-right in Figure 2.1) defines on which hardware the application(s) shall be deployed. It describes which computation resources (such as microprocessors) are available and how they are interconnected using communication resources (such as buses). The architecture model is typically composed from a library of well-defined standard resources (middle-right in Figure 2.1). These resource models provide information about the properties of the generic computing and communication resources that are available, such as processor speed and communication bus bandwidth. This information is typically found in data sheets or benchmarks, or can be obtained from measurements on existing systems. This library might also contain black-box descriptions of highly specialized components that are used for some dedicated task in the system, e.g. an encryption device.

The abstract system performance model is constructed by describing the deployment of the application model (the software) on the architecture model (the hardware, indicated as the so-called “mapping” in Figure 2.1). Furthermore, architecture parameters, such as the type of scheduling or arbitration used on each resource, are specified in this model. UML deployment diagrams or AADL models may be used to describe the mapping, but in this chapter we use an intuitive informal approach for deployment.

The environment model (bottom-left in Figure 2.1) defines the interface between the system and the surrounding environment. It describes for example how often system functions will be called, how much data is provided as input to the system and so on. Typically, end-to-end system-level requirements are specified in environment models. Environment models can, for example, be derived from measurements or traces.

The suitability of the proposed system architecture can be determined once the abstract system performance model has been evaluated with a certain technique in step 3. This is often difficult due to adversary requirements, such as for example cost price versus performance. Typically, a set of near-optimal solutions exists which requires heuristics for the "final" decision making. However, the process needs to be repeated if one of the requirements is clearly not met. This usually requires changing the application model (e.g. improve the algorithm to reduce processor load) or changing the architecture model (e.g. select a faster processor or bus) or changing the mapping (e.g. reallocate applications to different computation resources). This process is repeated until all requirements are met. This evaluation approach is commonly referred to as system-level performance optimization or, when higher levels of automation are involved, design space exploration. Note that it is not just restricted to performance requirements in the narrow sense used here. Other aspects that influence the choice of architecture, such as power usage, may also be taken into account.

First, the application and architecture models are presented in Section 2.2.1 and the environment model is defined in Section 2.2.2. Then, in sections 2.3.1 - 2.3.5, the modeling techniques are introduced. And finally, the results and lessons learned from the case study are discussed in Section 2.4.

2.2 The In-Car Radio Navigation system case study

The case study presented in this section is inspired by a system architecture definition study for a distributed in-car radio navigation system. Such a system typically executes a number of concurrent applications that share a common platform. Nevertheless, each application has individual performance requirements that need to be met by the platform. During the system definition phase, several candidate platform architectures might be proposed by the engineers and the system architect needs to evaluate each one and decide which one to implement. First, the system sub-model is presented in Section 2.2.1 and the environment sub-model is presented in Section 2.2.2.

2.2.1 Modeling the system

An overview of the system is presented in Figure 2.2. It is composed of three main clusters of functionality:

- The man-machine interface (MMI) which takes care of all interaction with the user, such as handling key inputs and graphical display output.

- The navigation functionality (NAV) which is responsible for destination entry, route planning and turn-by-turn route guidance giving the driver both audible and visual advices. The navigation functionality relies on the availability of a map database, typically stored on a CD or DVD, and positioning information, e.g. speed and GPS. The positioning sensors are not shown and considered here.

- The radio functionality (RAD) which is responsible for basic tuner and volume control as well as handling of traffic information services such as RDS TMC

(Radio Data System / Traffic Message Channel). RDS TMC (or TMC for short) is broadcast along with the audio signal of radio channels.

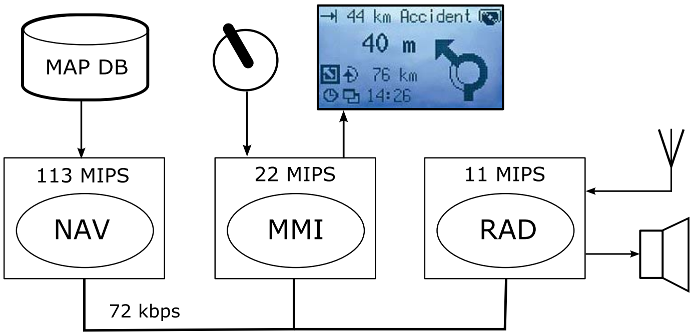


Figure 2.2: High-level overview of a distributed radio navigation system

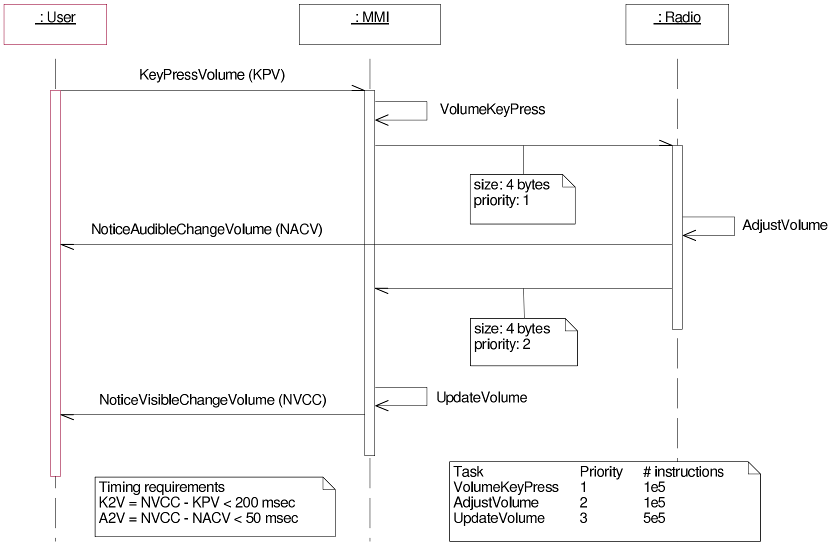
Step 1 of the Y-chart approach - scenario inventory

In our case study, we have selected three distinctive Use-Cases or scenarios:

1. "Change Volume" – The user turns the rotary button and expects near instantaneous audible feedback from the system. Furthermore, the visual feedback (the volume setting on the screen) should be timely and synchronized with the audible feedback. This seemingly trivial Use-Case is actually quite complex because many components are affected. Changing the volume might involve commanding a digital signal processor (DSP) and an amplifier in such a way that the quality of the audio signal is maintained while changing the volume. For example, rapid volume changes need to be damped because it would otherwise cause "clipping" which is disturbing to the user. This scenario is shown in detail in Figure 2.3. Note that three operations are identified, *VolumeKeyPress*, *AdjustVolume* and *UpdateVolume*. *VolumeKeyPress* takes care of the rotary button event handling. *AdjustVolume* interfaces with the DSP subsystem to actually change the volume and finally *UpdateVolume* which changes the volume setting on the display. Execution times and message sizes are estimated and annotated in the Sequence Diagram together with the two principle timing requirements applicable to this scenario. Priorities are defined in descending order (0 implies highest priority).

2. "Address Look-up" – Destination entry is supported by a smart "typewriter" style interface. By turning a knob the user can move from letter to letter; by pressing it the user will select the currently highlighted letter. The map database is searched for each letter that is selected and only those letters in the on-screen alphabet are enabled that are potential next letters in the list. This scenario is shown in detail in Figure 2.4. Note that the *SearchAddress* operation is expensive compared to the other operations and that the size of the output value of the operation is 16 times larger than the input message.

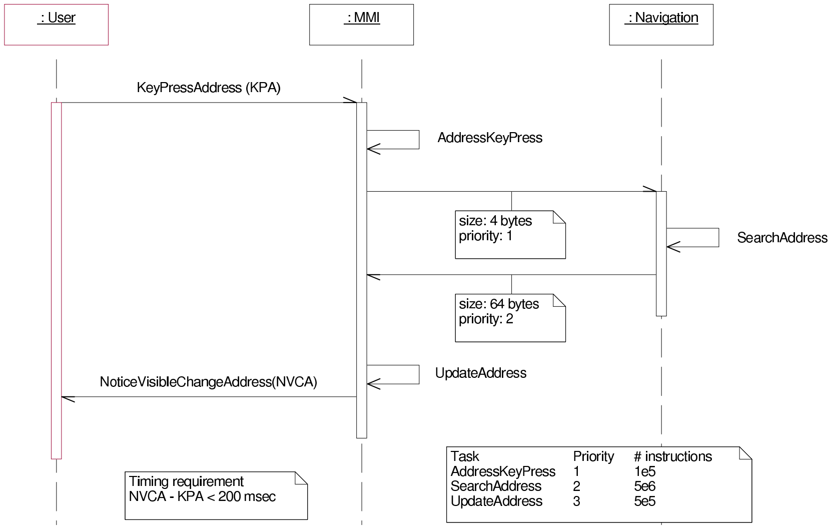
3. "TMC Message Handling" – Digital traffic information is important for in-car radio navigation systems. It enables features such as automatic re-planning of the



Timing requirements
K2V = NVCC - KPV < 200 msec
A2V = NVCC - NACV < 50 msec

Task	Priority	# instructions
VolumeKeyPress	1	1e5
AdjustVolume	2	1e5
UpdateVolume	3	5e5

Figure 2.3: Annotated Sequence Diagram for “Change Volume”



Timing requirement
NVCA - KPA < 200 msec

Task	Priority	# instructions
AddressKeyPress	1	1e5
SearchAddress	2	5e6
UpdateAddress	3	5e5

Figure 2.4: Annotated Sequence Diagram for “Address Look-up”

planned route in case a traffic jam occurs ahead. It is also increasingly important to enhance road safety by warning the driver, for example when a ghost driver is spotted just ahead on the planned route. TMC is such a digital traffic information service. TMC messages are broadcast by radio stations together with stereo audio sound. TMC messages are encoded: only problem location identifiers and message types are transmitted. The map database is accessed to translate these identifiers and to construct human readable text. The TMC message handling scenario is shown in Figure 2.5.

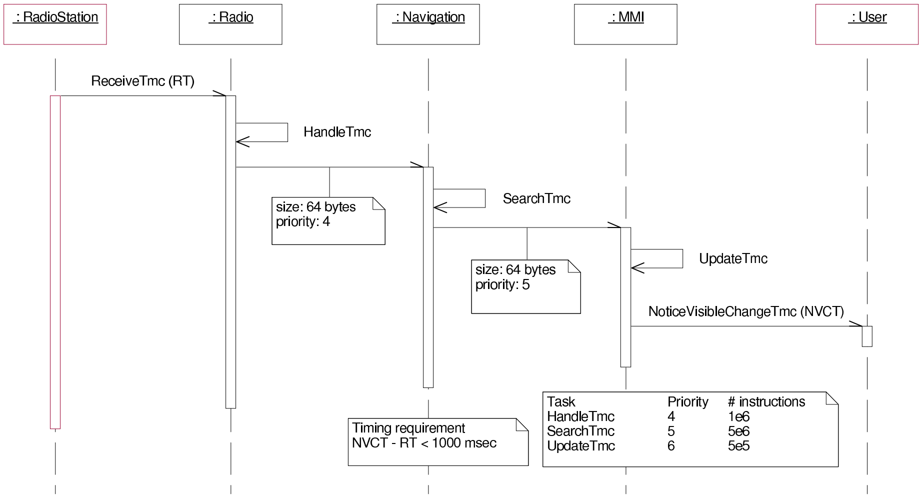


Figure 2.5: Annotated Sequence Diagram for “TMC Message Handling”

Note that the Sequence Diagrams are all annotated in such a way that they are useful for performance analysis. The order of magnitude of the numbers shown in the diagrams is realistic. This completes the first step of the recipe described in Section 2.1.

Step 2 of the Y-chart approach - resource inventory

The scenarios have an interesting property: they can occur in parallel. TMC messages must be processed while the user changes the volume or enters a destination. However “Change Volume” and “Address Look-up” can not occur at the same time because they share a common resource; the rotary button is used for both. The architecture shown in Figure 2.2 suggests to assign the three clusters of functionality each to its own processing unit, whereby the computation resources are interconnected by a single communication bus.

Figure 2.6 shows that there are more potential architectures that might be applicable. Note that the capacity of the resource units and communication infrastructure is quantified, completing step 2 of the recipe described in Section 2.1. The order of magnitude of the numbers shown in the diagram is correct; they are taken from the data sheets of several commercially available automotive CPUs. Observe that architecture (b) can only be evaluated if we introduce an additional operation on the MMI resource that transfers the data from one communication link to another, in the case that NAV wants to communicate to RAD or vice versa. This is the case for the “TMC Message Handling” scenario. The adapted Sequence Diagram for this special case is shown in Figure 2.7.

2.2.2 Modeling the environment

In order to analyze the proposed embedded architecture, we also need to characterize the so-called *workload* that the environment imposes onto the system. In this case study, we simply describe how often each application is invoked. We can abstract away from the complexity of the environment by describing the stimuli as a (p, j, d, o)-tuple.

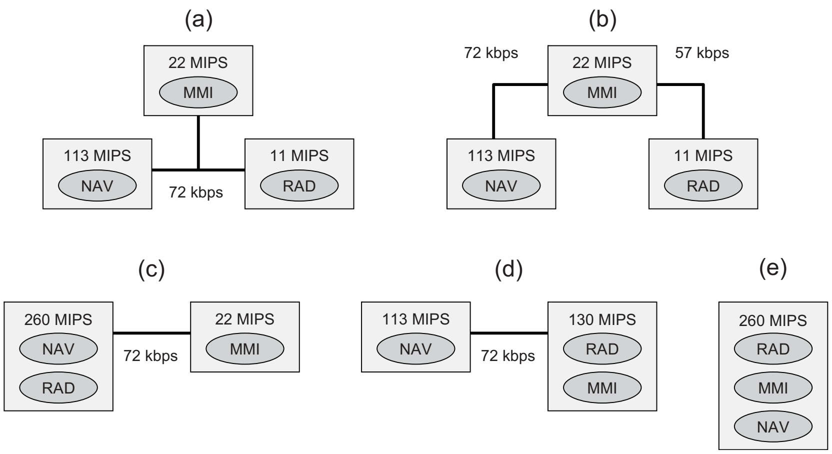
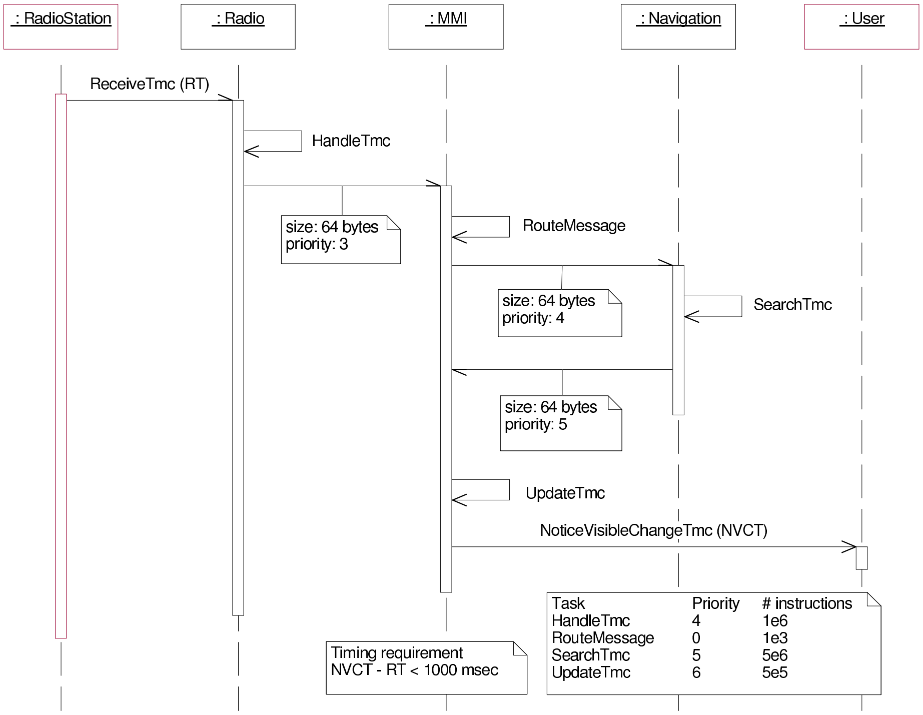


Figure 2.6: Alternative system architectures to explore



Task	Priority	# instructions
HandleTmc	4	1e6
RouteMessage	0	1e3
SearchTmc	5	5e6
UpdateTmc	6	5e5

Timing requirement
NVCT - RT < 1000 msec

Figure 2.7: Sequence Diagram for “TMC Message Handling” on architecture (b)

The p parameter describes the period of the stimulus, j describes the jitter, d the minimal inter arrival time and o the offset for the start of the first period. The most common stimuli arrival patterns can be described or approximated by this approach, including for example burst and sporadic behavior. The relationship between the parameters is graphically depicted in Figure 2.8. The (p, j, d, o)-tuple basically defines the *time in-*

terval in which a stimulus will occur. This model can be enriched with an additional stochastic variable which defines the distribution of the event within that interval. Similarly, the model can be extended to describe how much data is provided to the system at each event. But these extensions are out of scope for this thesis.

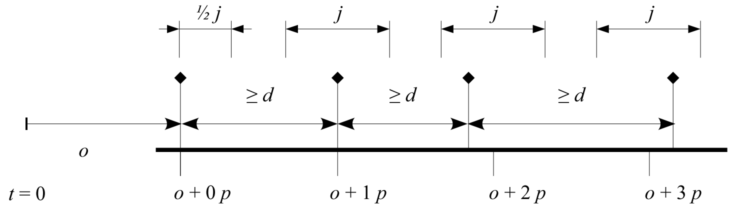


Figure 2.8: Workload definition using the (p, j, d, o)-notation

2.2.3 The modeling and analysis challenge

The generic question that is investigated in this chapter is how to distribute functionality over the available resources, such that all performance requirements are met. More specifically, some typical design-time questions can be formulated:

- **Design question 1**. Does the proposed architecture meet the performance requirements of all applications?
- **Design question 2**. How robust is the chosen architecture with respect to changes in application or architecture parameters?
- **Design question 3**. Is it possible to replace components by cheaper, less powerful, equivalents to save cost while maintaining the required performance targets?

These questions will be addressed in sections 2.3.1 - 2.3.5 when the modeling techniques are introduced. The initial values for environment model used in this case study are shown in Table 2.1. The influence of the jitter, delay and offset parameters will be considered later. Fixed priority scheduling is assumed on all resources.

Event name	**period** p	**jitter** j	**delay** d	**offset** o
KeyPressVolume (KPV)	31.25	0	0	0
KeyPressAddress (KPA)	1000	0	0	0
ReceiveTmc (RT)	3000	0	0	0

Table 2.1: Initial case study workload definitions (values in msec)

2.3 The performance modeling methods

The last step in the procedure proposed in Section 2.1 is to compose and analyze the abstract system performance model using some technique, based on the data collected in steps 1 and 2. We will demonstrate how this is done using Modular Performance Analysis (MPA) in Section 2.3.1, Symbolic Timing Analysis for Systems (SymTA/S) in Section 2.3.2, Timed Automata in Section 2.3.3, Parallel Object-Oriented Specification

Language (POOSL) in Section 2.3.4 and finally using the Vienna Development Method in Section 2.3.5. Each technique is introduced shortly and a flavor of the created system and environment models is given. The caveats of the modeling exercise are discussed and the analysis results obtained from these methods are compared in Section 2.4 in order to draw more general conclusions.

2.3.1 Modular Performance Analysis

Short overview of the technique

Modular Performance Analysis (MPA) was developed by Thiele et al at ETH Zürich [18]. MPA is a compositional modeling technique based on a general event and resource model. A set of basic building blocks, called *abstract components*, is available to build a queuing network that represents the system that we want to analyze. These abstract components are used to describe the handling of incoming events under the, possibly delayed, availability of resources. The events are described by a pair of interval bound functions $\overline{\alpha}$, the so-called lower and upper *arrival curves* $\overline{\alpha}^l$ and $\overline{\alpha}^u$. These curves describe the respective bounds on the number of events that are to be handled by the component for any given interval size. $\overline{\alpha}$ is also called the *event-based* arrival curve. Similarly, resources are described by a pair of interval bound functions β, the so-called lower and upper *service curves* β^l and β^u. These curves describe the bounds on the available resource capacity for any given interval size.

How do these interval bound functions relate to a real system? Consider a task in the system that handles a stream of events (e.g. an interrupt handling routine). A trace of this event stream can be described by a cumulative function $R(t)$, which is defined as the number of events seen on the event stream in the time interval $[0, t\rangle$. Each event is processed by the task which is deployed on a resource. The availability of this resource is described by a cumulative function $C(t)$, which is defined as the total capacity available on the resource in the time interval $[0, t\rangle$. The events are emitted on the output of the task after handling the event, resulting in an output event trace described by $R'(t)$. Similarly, the resource capacity that is left after the event is handled, is described by $C'(t)$. If we assume that processing an event always takes a finite, non-zero and positive amount of time then it is clear that $R(t) \neq R'(t)$. Similarly, if we assume that processing an event always consumes a finite, non-zero and positive amount of resource capacity, we can claim that $C(t) \neq C'(t)$. With these assumptions in mind, we provide a definition for arrival and service curves.

Definition 2.3.1 *Arrival Curves. Let $R(t)$ denote the number of events that arrive on an event stream in the time interval $[0, t\rangle$. Then, R, $\overline{\alpha}^u$ and $\overline{\alpha}^l$ are related to each other by the following inequality:*

$$\overline{\alpha}^l(t - s) \leq R(t) - R(s) \leq \overline{\alpha}^u(t - s), \forall s < t \qquad (2.1)$$

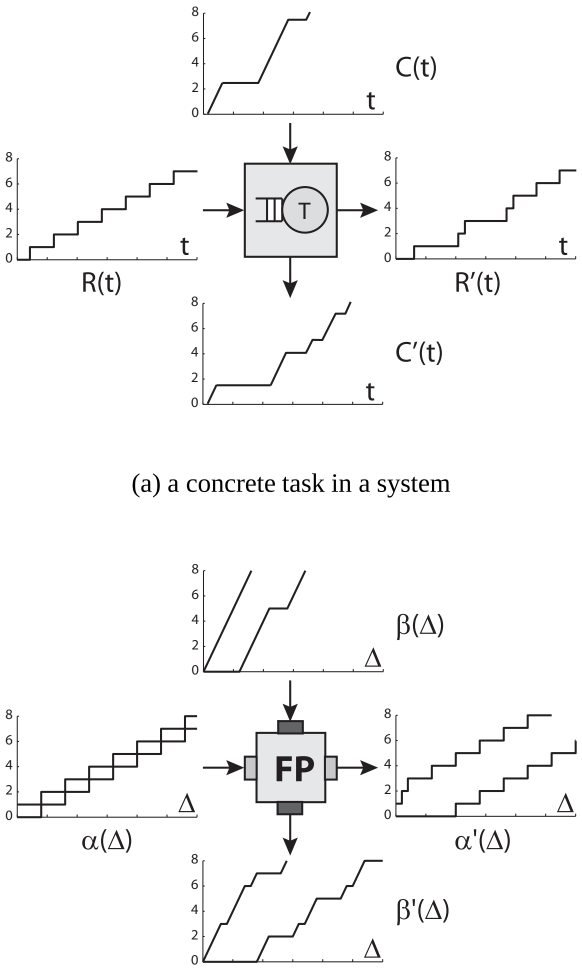
with $\overline{\alpha}^l(0) = \overline{\alpha}^u(0) = 0$.

Definition 2.3.2 *Service Curves. Let $C(t)$ denote the number of processing or communication cycles available from a resource over the time interval $[0, t\rangle$. Then C, β^u and β^l are related by the following inequality:*

$$\beta^l(t - s) \leq C(t) - C(s) \leq \beta^u(t - s), \forall s < t \qquad (2.2)$$

with $\beta^l(0) = \beta^u(0) = 0$.

The relationship between a concrete task running on a real system and an abstract component in MPA is visualized in Figure 2.9. While R and C describe a specific behaviour of the system, α and β represent *all* possible behaviors of the system. This is the key abstraction that is provided by this technique. However, R and C are defined over t while α and β are defined over some time interval Δ. This implies that information about absolute time is lost in this transformation. We do not know any more *when* a certain situation occurs. As shown in Figure 2.9, each abstract component in MPA takes a pair of arrival and service curves as its input but also produces a pair of arrival curves and service curves. These output curves describe respectively the properties of the resulting event stream and the remaining resource capacity *after* the event is handled. They can again be used as inputs to downstream components in the queuing network.



(a) a concrete task in a system

(b) an abstract component in MPA for fixed priority scheduling

Figure 2.9: Cumulative functions versus interval bound functions

The semantics of each abstract component is formally defined by a set of mathematical functions that relate the input arrival and service curves to the output arrival and service curves using Real-Time Calculus (RTC) [92]. This transformation has two effects. First, since event processing takes time, the output arrival curves will exhibit a time delay. Second, since event processing consumes resources, the output service curves will exhibit a drop in resource availability. The amount of time delay and resource usage depends on the cost of handling a single event and the scheduling policy used, because the latter determines when resource capacity is available to process the event. RTC provides formulas for several well-known scheduling policies, such as fixed-priority preemptive (FP), generalized processor sharing (GPS), time division multiple access (TDMA) and earliest deadline first (EDF).

Abstract components in MPA can be used to describe both computation as well as communication resources. As said earlier, in order to calculate the delay per event, we need to specify how expensive handling such an event is. This is achieved by introducing a cost function c that transforms event-based arrival curves $\overline{\alpha}$ into a *resource-based* arrival curves α, i.e. $\alpha(\Delta) = c \circ \overline{\alpha}(\Delta)$. The pair of resource-based arrival curves α^l and α^u describe the bounds on the generated resource demand for any given interval size. In the most basic scenario, which is applicable to our case study, every arriving event generates the same resource demand, i.e. the worst-case execution demand equals the best-case execution demand. Resource based arrival curves can then be obtained by multiplying the event based arrival curves with a constant that represents the resource demand of a single event. Here, we use the number of instructions as specified in the annotated UML sequence diagrams as presented in Section 2.2 for computation and the message sizes for communication resources.

The service curves of a resource can be determined using data sheets, using analytically derived properties, or by measurement. For example, in the simplest case of an unloaded processor, whose capacity we measure in available processing cycles per time unit as shown in Figure 2.6, both the upper and the lower resource curves are equal and are represented by straight lines $\beta^u(\Delta) = \beta^l(\Delta) = f \cdot \Delta$, where f equals the processor speed, i.e. the number of available processing cycles per time unit. With service curves, we may also model communication resources. In the case of an unloaded bus, f equals the available bandwidth. The service curves then represent the minimum and maximum number of transmittable bits, for any given time interval.

An open source implementation of MPA in Java for Matlab/Simulink is available from *http://www.mpa.ethz.ch*. A detailed treatment of MPA and this case study is provided in [105].

Modeling the case study

How can systems be modeled using Modular Performance Analysis? Suppose we want to model Architecture (a) of the case study described in Figure 2.6. We start by declaring four *resource components*. Resource components are abstract components that only produce a pair of service curves that describe the unloaded resource, as proposed in the previous section. We need three resource components for the processors in Architecture (a) and one for the bus that connects the processors. The resource components will form the columns in our queuing network, as shown in Figure 2.10. Resources flow vertically through this model, while events flow horizontally. The rows are used to describe the applications that are deployed on those resources.

Consider the “Change Volume” scenario as presented in Figure 2.3. Each task invocation and each message exchange is represented by an abstract component in the

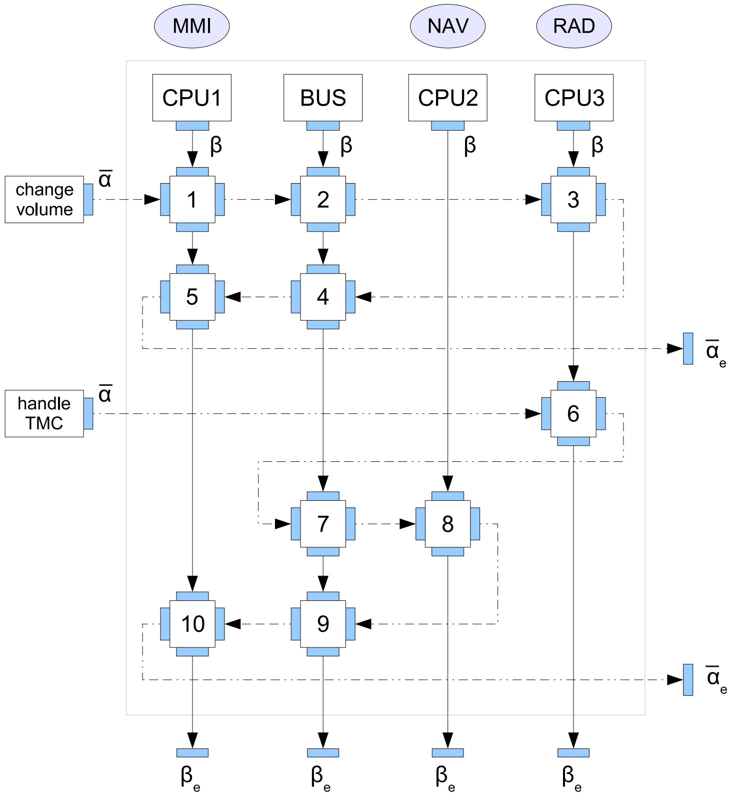


Figure 2.10: Example MPA queuing network for Architecture (a)

queuing network. The task *VolumeKeyPress* is represented by abstract component 1, *AdjustVolume* by component 3 and *UpdateVolume* by component 5. Note that component 5 uses the remaining resources of component 1, which implies that 1 has a higher priority than 5. This conforms to our case study, where we use fixed priority preemptive scheduling on all resources. A similar situation applies to components 2 and 4 which represents the message handling on the bus, whereby the call to *AdjustVolume* has priority over the result coming back from the call.

We can study the effect of multiple, concurrent, applications executing on the same architecture by adding another set of rows. In Figure 2.10, we have also added the "TMC Message Handling" scenario to the queuing network. Component 6 represents the *HandleTmc* task from Figure 2.5, component 8 corresponds to *SearchTmc* and component 10 represents *UpdateTmc*. In the vertical direction, the resource flows are terminated by *resource sinks*. Resource sinks simply take a pair of service curves as their input. These service curves describe the end-to-end behavior of the resource in terms of remaining capacity. In the horizontal direction, the event flows are started by a *load scenario* and they are terminated by an *event sink*. The event sinks simply take a pair of arrival curves as their input. Similarly, these arrival curves describe the end-to-end behavior of the application in terms of timing. Note that both resource usage as well as timing information can be obtained from the same model.

Load scenarios are abstract components that only provide a set of arrival curves. These arrival curves are used to define the workload of the system, based on the

(p, j, d, o)-tuple as presented in Section 2.2, with two exceptions. First, note that the offset parameter o from the (p, j, d, o)-tuple is not meaningful in the time interval domain since that parameter does not affect the event inter arrival time. Second, sporadic input events only have a lower bound on the period, which can be specified using only the delay parameter d, in order to specify the minimal inter arrival time. The resulting arrival curve pair has a strict periodic event stream with period d as its upper bound and $y = 0$ as its lower bound because the arrival of the first event may take forever. In particular this lower bound causes pessimistic results in real-time calculus as will be shown later. Recall that Δ represents an arbitrarily sized time interval. The relationship between the (p, j, d)-triple and arrival curves for $p > 0$ is then defined by:

Definition 2.3.3 *load scenario*

$$\overline{\alpha}^l(\Delta) = \left\lfloor \frac{\Delta - j}{p} \right\rfloor \tag{2.3}$$

$$\overline{\alpha}^u(\Delta) = \begin{cases} \min\left\{ \left\lceil \frac{\Delta + j}{p} \right\rceil, \left\lceil \frac{\Delta}{d} \right\rceil \right\} & \textit{if } d > 0 \\ \left\lceil \frac{\Delta + j}{p} \right\rceil & \textit{if } d = 0 \end{cases} \tag{2.4}$$

whereby $\Delta \geq 0$ *and* $j \geq 0$.

In Figure 2.11, the relation between these parameters and the corresponding arrival curves is graphically depicted. Note that in this particular example the jitter is much greater than the period which is typical for a so-called event streams with bursts. This also explains the steep ascend at the beginning of the upper arrival curve.

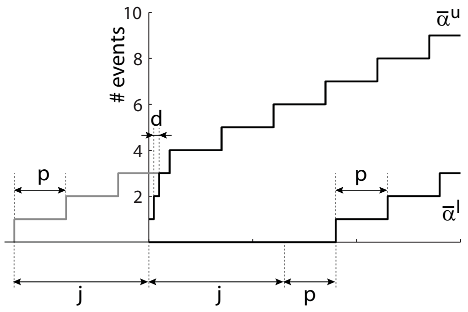


Figure 2.11: The relationship between the (p, j, d)-triple and arrival curves

Figure 2.12 shows some typical examples of arrival curves. The arrival curves in Figure 2.12 (a) model a strictly periodic event stream, while the arrival curves in Figure 2.12 (b) model a periodic event stream with jitter, and the arrival curves in Figure 2.12 (c) model a periodic event stream with bursts. The arrival curves in Figure 2.12 (d) model an event stream with more complex timing behavior. This event stream may have short steep bursts, longer lasting less steep bursts, and the maximum long-term period does not equal the minimum long-term period. An event stream with such complex behavior can not be represented accurately using the (p, j, d, o)-tuple.

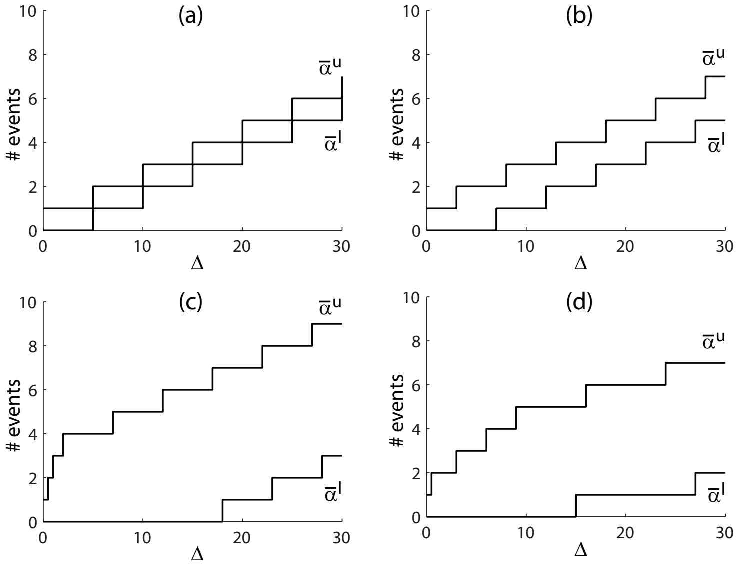


Figure 2.12: Examples of typical arrival curves

Analysis of the model - theory

The abstract system performance models, as described in the previous section, can be analyzed using Real-Time Calculus to complete step 3 from the recipe presented in Section 2.1. Real-Time Calculus belongs to the class of so-called deterministic queuing theories. These models can be solved analytically, without simulation. Analysis of the queuing network provides us with bounds on the propagation delay and resource usage for each component individually as well as end-to-end. In addition, the so-called backlog, the number of outstanding events which corresponds to maximum queue size needed, can be determined for each component.

Real-Time Calculus extends the concepts of the well-known Network Calculus [69] to the domain of real-time systems. It unifies the notions of computation and communication and provides powerful semantic models for abstract components to describe specific scheduling policies. Furthermore, modular performance analysis also allows hierarchical modeling, whereby abstract components can be decomposed into lower-level queuing networks. This feature is not demonstrated in this case study.

Network Calculus is based on min-max calculus. Min-max calculus is the combination of the min-plus and max-plus calculi. Min-plus calculus and max-plus calculus both define a special algebra (the min-plus dioid and max-plus dioid, respectively). An excellent introduction to these calculi is provided in [5], we present a short overview here. Traditionally, we are used to work with the algebraic structure $(\mathbb{R}, +, \times)$, i.e. with the set of reals endowed with the operations of addition and multiplication, that possess a number of properties such as associativity, commutativity, distributivity, etcetera.

In contrast, min-plus calculus works with an algebraic structure $(\mathbb{R} \cup \infty, \vee, +)$. Here, the operation of addition becomes the computation of the infimum (or the minimum), and the operation of multiplication becomes the addition. Most axioms known from conventional algebra still apply to this algebraic structure. In max-plus calculus, the infimum and minimum are replaced by supremum and maximum. In Real-Time

Calculus, we often need to compute convolutions and de-convolutions defined in min-plus and max-plus calculus. These operations are defined as follows [69]:

Definition 2.3.4 *min-max convolution and de-convolution*
The min-plus convolution $\otimes$ and the min-plus de-convolution $\oslash$ of two functions f and g are defined as:

$$
\begin{aligned}
(f \otimes g)(\Delta) &= \inf_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} && (2.5)\\
(f \oslash g)(\Delta) &= \sup_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} && (2.6)
\end{aligned}
$$

The max-plus convolution $\overline{\otimes}$ and the max-plus de-convolution $\overline{\oslash}$ of two functions f and g are defined as:

$$
\begin{aligned}
(f \,\overline{\otimes}\, g)(\Delta) &= \sup_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} && (2.7)\\
(f \,\overline{\oslash}\, g)(\Delta) &= \inf_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} && (2.8)
\end{aligned}
$$

Recall that abstract components are specified by a set of functions, that relate the input arrival and service curves to the output arrival and service curves, more formally:

$$
\begin{aligned}
\alpha' &= f_\alpha(\alpha, \beta) && (2.9)\\
\beta' &= f_\beta(\alpha, \beta) && (2.10)
\end{aligned}
$$

The relations f_α and f_β depend on the processing semantics of the component, and must be determined such that $\alpha'(\Delta)$ correctly models the event stream with event trace $R'(t)$ and that $\beta'(\Delta)$ correctly models the resource availability $C'(t)$. Consider a fully preemptive task that is triggered by an incoming event stream. This task is started at every event arrival to process the incoming event, and active tasks are processed in a greedy fashion in FIFO order, while being restricted by the availability of resources. Such a component can be modeled as an abstract component with following internal relations:

$$
\begin{aligned}
\alpha'^{u}_{FP} &= \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\} && (2.11)\\
\alpha'^{l}_{FP} &= \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\} && (2.12)\\
\beta'^{u}_{FP} &= (\beta^u - \alpha^l) \,\overline{\oslash}\, 0 && (2.13)\\
\beta'^{l}_{FP} &= (\beta^l - \alpha^u) \,\overline{\otimes}\, 0 && (2.14)
\end{aligned}
$$

Components with these processing semantics are common in the area of real-time embedded systems, and we will refer to them as *fixed priority* (FP) components. To model a component with different processing semantics, one has to determine the appropriate internal relations f_α and f_β. The min-max algebra ensures that *hard* bounds are always calculated, since it computes convolutions and de-convolutions over interval bound functions that describe the minima and maxima for any time interval Δ. This is why MPA is suitable to analyze real-time systems, because guarantees can be given about the worst-case.

When an event stream with arrival curves α is processed by an FP component on a resource with service curve β, the *maximum delay* d_{max} experienced by any event on the event stream is bounded by [69, 18]:

$$\begin{aligned} d_{max} &\leq \sup_{\lambda \geq 0} \left\{ \inf \left\{ \tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau) \right\} \right\} \\ &\overset{def}{=} \textit{Del}\,(\alpha^u, \beta^l) \end{aligned} \tag{2.15}$$

When an event stream is processed by a sequence of several components, we could simply add the different maximum delays of each individual component together, to obtain an end-to-end delay guarantee. However, in this case we can exploit the phenomenon known as “Pay Bursts Only Once” [69], and the end-to-end delay guarantee can be tightened to [69]:

$$d_{max} \leq \textit{Del}\,(\alpha^u, \beta^l_1 \otimes \beta^l_2 \otimes \ldots \otimes \beta^l_n) \tag{2.16}$$

Similarly, the maximum buffer space b_{max} that is required to buffer an event stream with arrival curve α in the input queue of an FP component on a resource with service curve β is bounded by [69]:

$$b_{max} \leq \sup_{\lambda \geq 0} \{\alpha^u(\lambda) - \beta^l(\lambda)\} \overset{def}{=} \textit{Buf}\,(\alpha^u, \beta^l) \tag{2.17}$$

When the buffers of several consecutive components use the same shared memory, the total required buffer space can even be tightened to:

$$b_{max} \leq \textit{Buf}\,(\alpha^u, \beta^l_1 \otimes \beta^l_2 \otimes \ldots \otimes \beta^l_n) \tag{2.18}$$

In Figure 2.13, the relations between α, β, d_{max} and b_{max} are depicted graphically. From this figure, we see that d_{max} and b_{max} are bounded by the maximum horizontal and maximum vertical distance between the upper arrival curve and the lower service curve respectively. This corresponds to the intuition, that d_{max} and b_{max} occur when the maximum load arrives at the same time when the minimum resources are available.

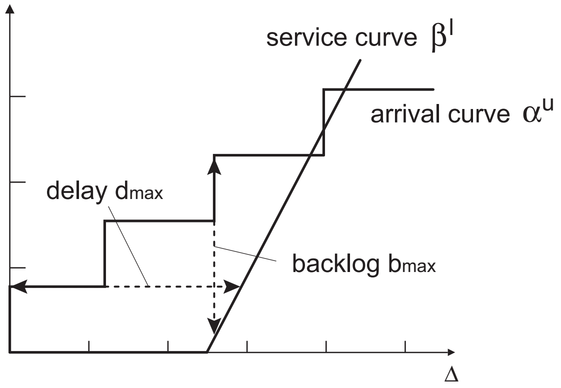


Figure 2.13: Delay and backlog obtained from arrival and service curves

Analysis of the model - practice

Three design challenges where posed in Section 2.2.3 and they are discussed here.

Design question 1. We build the abstract system performance model for the "Change Volume" and "TMC Message Handling" scenarios, as depicted in Figure 2.10, as well as the performance model for the "Address Lookup" and "TMC Message Handling" scenarios. For both models, we compute the upper bounds to the end-to-end delay of every event stream, as described in the last section, and then we merge the results obtained from the two analysis runs to obtain our overall result. For the TMC delay, we take the bigger value of the two runs. This process is repeated for all proposed architectures. From the results presented in Figure 2.14, we see that all architectures fulfill the requirements (as mentioned in Figure 2.3, 2.4 and 2.5) on the different maximum end-to-end delays. Furthermore, the results suggest that architectures (d) and (e) process the input data to the system particularly fast. This may be explained partly by the reduced communication overhead in these architectures, but most probably, these architectures are also over-dimensioned.

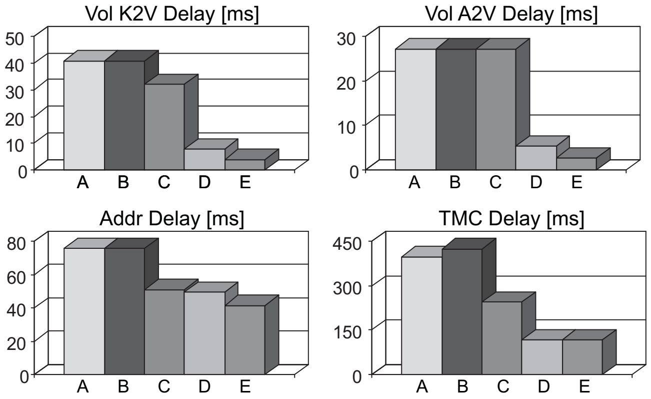


Figure 2.14: Maximum end-to-end delays for each system architecture

Design question 2. To investigate the robustness of architecture (a), we first compute its sensitivity towards changes in the input data rates. These sensitivity results are shown in Figure 2.15. The height of the columns in this figure depict the increase of end-to-end delays relative to the respective specified maximum end-to-end delays, in dependence to increasing input data rates. For example, the tallest column in Figure 2.15 shows us that if we increase the data rate of the "Change Volume" scenario slightly (e.g. by $4\ \%$, to $33.3\ events/s$), the end-to-end delay of the TMC message handling increases by $1.14\ \%$ of its specified maximum end-to-end delay (i.e. $1.14\ \%$ of $1000\ ms$ or $11.4\ ms$).

From the results shown in Figure 2.15, we see that architecture (a) is sensitive towards increasing the input data rate of the "Change Volume" scenario, while increasing the input data rate of the "Address Look-up" and the "TMC Message Handling" scenarios do not really affect the response times. And in fact, further analysis reveals that in order to still guarantee all system requirements, we must not increase the input data rate of the "Change Volume" scenario by more than $7\ \%$, while we could increase the input data rate of the other two scenarios by a factor of more than 20.

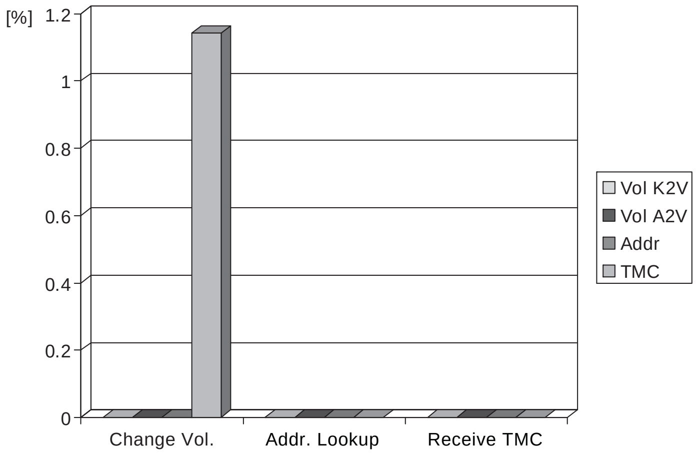


Figure 2.15: Sensitivity towards changes in the input data rates

After investigating the system sensitivity towards changes in the input data rates, we investigate the system sensitivity towards changes in the resource capacities. These sensitivity results are shown in Figure 2.16. The height of the columns in this figure depicts the increase of end-to-end delays relative to the respective specified maximum end-to-end delays, in dependence to decreasing resource capacities. For example, from the tallest column in Figure 2.16 we know that if we decrease capacity of the MMI processor by $1\ \%$ (e.g. to 21.78 MIPS), the end-to-end delay of the TMC message handling increases by $3.22\ \%$ of its specified maximum end-to-end delay (i.e. $3.22\ \%$ of $1000\ ms$ or $32.2\ ms$).

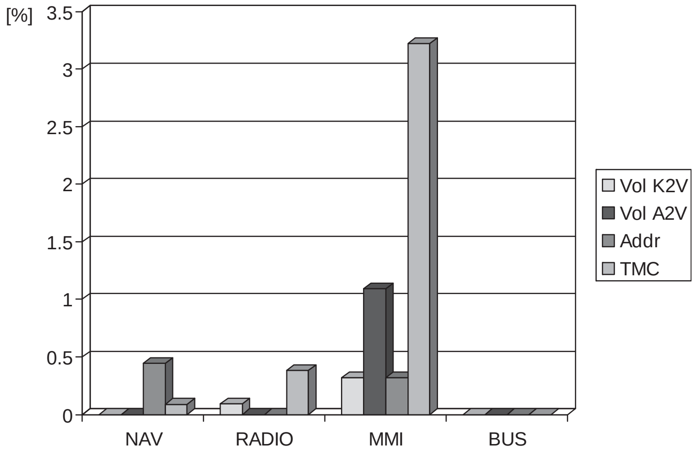


Figure 2.16: Sensitivity towards changes in the resource capacities

From the results shown in Figure 2.16, we see that architecture (a) is most sensitive towards the capacity of the MMI processor. This suggests that the MMI processor is a potential bottleneck of architecture (a). To investigate this further, we compute the end-to-end delay of the TMC message handling for different MMI processor capacities. The results of these computations are shown in Figure 2.17.

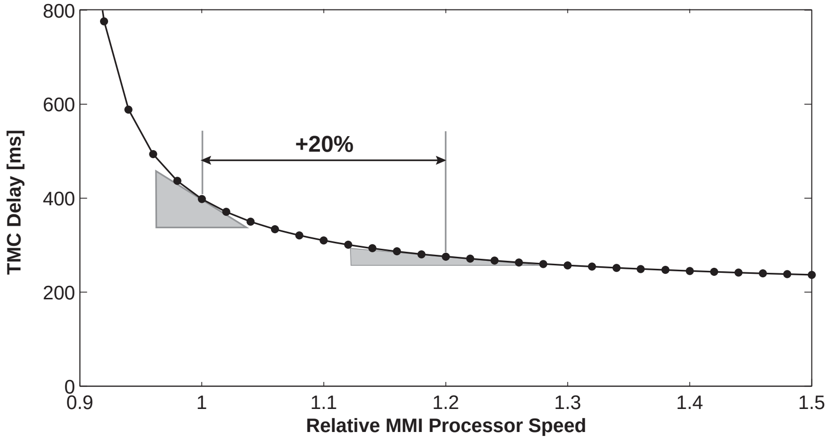


Figure 2.17: TMC delay versus MMI processor speed

From Figure 2.17, we see that indeed at its given operation point, the end-to-end delay of the TMC message handling in architecture (a) is sensitive towards changes of the MMI processor capacity. And the analysis reveals that with a decrease of the MMI processor capacity to $89\ \%$ of its initial capacity, we cannot guarantee finite response times anymore.

To sum up, the above analysis results suggest that increasing the capacity of the MMI processor would make architecture (a) more robust. To support this statement, we individually increase the capacity of each resource by $20\ \%$, and we then analyze how much we can increase the input data rate of the “Change Volume” scenario while still fulfilling the requirements. Remember, with the initial resource capacities, we can increase the data rate of the “Change Volume” scenario by $7\ \%$ and the data rate of the other two scenarios by a factor of more than 20 while still guaranteeing all requirements. From this analysis, we learn that increasing the resource capacities of the RAD processor, the NAV processor and the BUS does not allow to increase the input date rate of the “Change Volume” scenario more than with the initial capacities, while increasing the MMI processor capacity allows us to increase the data rate of the “Change Volume” scenario by $60\ \%$.

Design question 3. We compute the upper bound to the end-to-end delay of every event stream in architecture (d) for different processor capacities. The results are shown in Figure 2.18.

In the plots in Figure 2.18, the NAV processor capacity is varied in steps of $5\ \%$ from $100\ \%$ down to $10\ \%$ of its initial capacity. At the same time, the MMI/RAD processor capacity is varied in steps of $5\ \%$ from $100\ \%$ down to $20\ \%$ of its initial capacity. As we see from the plots, the delays of the “Change Volume” scenario are not much affected by changes of the NAV processor capacity and the delay of the “Address Look-up” scenario is not much affected by changes of the MMI/RAD processor capacity. On the other hand, the delay of the “TMC Message Handling” scenario is affected by the changes of both processor capacities. From the results, we learn that we could

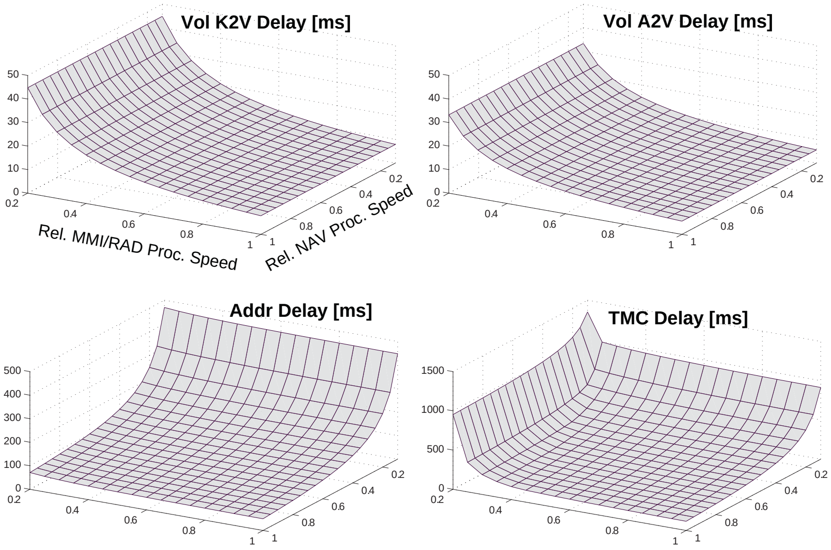


Figure 2.18: Delays versus processor speed in architecture (d)

decrease both the NAV processor capacity as well as the MMI/RAD processor capacity down to $25\ \%$ of their initial capacity (i.e. 29 MIPS and 33 MIPS, respectively) while still guaranteeing the fulfillment of all system requirements.

Observations on the experiment

MPA provides *hard* bounds to all analyzed properties, but these bounds are *not* necessarily tight. The primary cause of this phenomenon is that information is lost when we move from the time to the time interval domain. Consider for example two strictly periodic event streams with the same period p. Now suppose these event streams are related by a constant phase shift, for example, by an offset $o = 0.5\ p$. The arrival curves that describe these event streams are equal, despite the offset, since the distance between two consecutive events on each event stream is identical and does not depend on the offset. When we merge these two event streams together into one, we basically create a new strictly periodic event stream with period $0.5\ p$. However, if we would combine the arrival curves in a similar way using min-max algebra, we would get a more pessimistic result. Calculating the (de-)convolutions of these arrival curves would create a new arrival curve that assumes that both events, in the worst case, would arrive if $\Delta \approx 0$ while the arrival curve of the composed event stream will reach that same worst-case value for $\Delta = 0.5\ p$.

Evaluation of an MPA network is fast (typically a few seconds at most) which supports the interactive nature of the design process. Without any attempts to optimization, analyzing one system architecture in the design space took around $1\ s$ on a Pentium Mobile 1.6 GHz using Matlab 7. Computing the four mesh plots in Figure 2.18 took for example around $5\ min$.

2.3.2 Symbolic Timing Analysis for Systems

Short overview of the technique

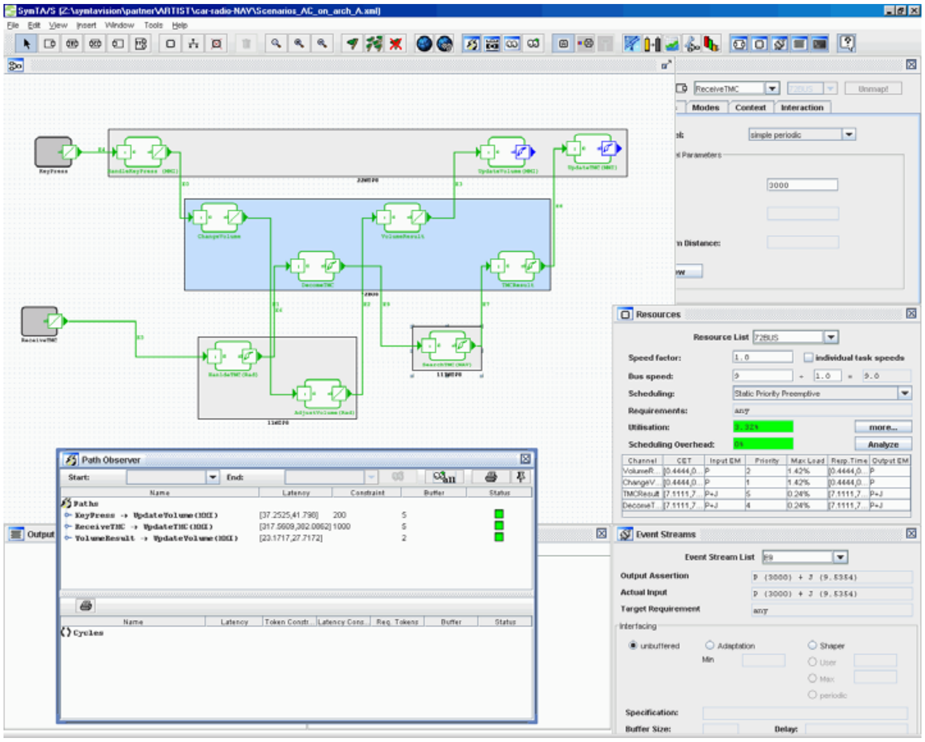
Symbolic Timing Analysis for Systems (SymTA/S) was developed by Ernst and Richter et al at the University of Braunschweig [46]. Tool support is now further developed at the SymtaVision company, a spin-off from the university. SymTA/S is a performance and timing analysis tool based on formal scheduling analysis techniques and symbolic simulation. It supports modeling of heterogeneous architectures, complex task dependencies, context aware analysis and combines optimization algorithms with sensitivity analysis for rapid design space exploration suitable for application in an industrial setting. The input for the comparison was kindly provided by Richter, we did not have access to the tool itself.

SymTA/S uses an approach whereby subsystems in the architecture are seen as entities that interact, or communicate, through event streams. As shown before, event streams can become arbitrarily complex and SymTA/S uses two techniques to tame this complexity. Like Modular Performance Analysis, arrival curves are supported. In addition, so-called event vector systems [42] can be used. Where MPA treats each abstract component identically, SymTA/S uses system-level knowledge to improve the accuracy of the analysis at the abstract component level by using this context information. Event model interfaces (EMIFs) or event model adaptor functions (EAFs) are used so that classical scheduling analysis techniques can be safely applied at the abstract component level. It is claimed that this approach leads to both hard *and* tight results [84].

The use of global context information becomes clear when the model is cyclic, in other words when the output of some task causes new events to appear at the input of an upstream task. Modular performance analysis can only deal with these kind of problems if a fixed-point can be found [105]. SymTA/S iteratively propagates the parameterized event streams through the model automatically until 1) the event stream parameters converge or 2) a task misses its deadline or 3) a specified maximum buffer size is exceeded. This process always terminates because the timing uncertainty, which is defined by the difference between the best- and worst-case event timing interval, grows monotonically with each iteration. EAFs are inserted automatically if the iteration is stopped due to the latter two problems. This breaks the dependency cycle by reducing the timing uncertainty. Similarly, SymTA/S can deal with complex task interdependencies that are hard if not impossible to model in Modular Performance Analysis. An overview of SymTA/S is provided in [84]. The tool is available, as a commercial product, from *http://www.symtavision.com*.

Modeling the case study

The tool provides a convenient graphical user-interface to enter the model. A separate model needs to be constructed for each architecture. Each model consists of a number of resources on which tasks can be deployed. Tasks can be assigned and reassigned to resources by drag-and-drop. Applications are modeled by linking the tasks together into a so-called execution path. The environment is modeled by connecting event generators to the initial tasks. The properties of each entity in the model can be changed interactively by means of pop-up menus, for example to modify the (p, j, d, o)-values of the event generators. A screen dump of the case study being edited in SymTA/S is shown in Figure 2.19.



SymTA/S (Z:\symtavision\partner\ARTIST\car-radio-NAV\Scenarios_AC_on_arch_A.xml)

ReceiveTMC

simple periodic

3000

Resources

Resource List: 72BUS

Speed factor: 1.0 ☐ individual task speeds

Bus speed: 9 ÷ 1.0 = 9.0

Scheduling: Static Priority Preemptive

Requirements: any

Utilisation: 3.32%

Scheduling Overhead: 0%

Channel	CET	Input EM	Priority	Max Load	Resp.Time	Output EM
VolumeR...	[0.4444,0...	P	2	1.42%	[0.4444,0...	P
ChangeV...	[0.4444,0...	P	1	1.42%	[0.4444,0...	P
TMCResult	[7.1111,7...	P+J	5	0.24%	[7.1111,7...	P+J
DecodeT...	[7.1111,7...	P+J	4	0.24%	[7.1111,7...	P+J

Path Observer

Start: End:

Name	Latency	Constraint	Buffer	Status
Paths				
KeyPress → UpdateVolume(HMI)	[37.2525,41.798]	200	5	🟩
ReceiveTMC → UpdateTMC(HMI)	[317.5609,382.0062]	1000	5	🟩
VolumeResult → UpdateVolume(HMI)	[23.1717,27.7172]		2	🟩

Name	Latency	Token Constr...	Latency Cons...	Req. Tokens	Buffer	Status
Cycles						

Event Streams

Event Stream List: E9

Output Assertion: P (3000) + J (9.5354)

Actual Input: P (3000) + J (9.5354)

Target Requirement: any

Interfacing

● unbuffered ○ Adaptation Min ○ Shaper ○ User ○ Max ○ periodic

Specification:

Buffer Size: Delay:

Figure 2.19: SymTA/S analysis of Architecture (a)

Analysis of the model

The end-to-end performance of an application can be shown by looking at the output event stream of the last task in an execution path, similar to event sinks in MPA. Special diagrams can be presented to observe the global and per-application usage of the resource. The path observer window conveniently displays all relevant execution paths, their best-case and worst case execution times and whether or not the associated requirements for each path was met, by color coding. Similarly, incompatible interface connections or local requirements that are not met after analysis are made visible to the user graphically, by changing the color of the entity in the diagram that caused the error. With respect to the three design questions raised in Section 2.2.3, all three can be answered with SymTA/S, but unfortunately only results for question 1 were made available to us for this comparative study.

Observations on the experiment

Composing and evaluating a model is quick, typically in the order of a few seconds to a minute. The tool computes the local optimum per resource using classical formal scheduling analysis techniques like, for example, rate monotonic analysis [16]. This technique corresponds to the fixed-priority preemptive scheduling we specified in the case study. The values obtained for each resource are used to feed a symbolic simulation step where system-level values are derived. Using optimization strategies, this process is repeated automatically until some, user defined, property is reached as described earlier. Like MPA, SymTA/S gives hard but not necessarily tight results, which is primarily caused by the abstractions introduced in the model.

2.3.3 Timed Automata

Short overview of the technique

The timed automata modeling language [2, 7] is a general purpose framework used to describe timed systems. The basic entity in the language is the so-called automaton, which can be represented as a labeled transition system. An automaton consists of *locations* and *transitions*. Locations represent the state of the component and the transitions define the relationships between these states. Time can be modeled by introducing *clocks* as state variables. Clocks are automatic and strictly monotone increasing continuous variables, all with the same rate of change. Clock *invariants* can be added to a location, to denote *when* this state is valid. The transitions define how those locations can be reached starting from some initial location. Transitions can be labeled with *guards* and *actions*. Guards can be used to specify for which clock or state value(s) a transition is enabled. Actions can be used to modify the state, including resetting clock variables. This technique is useful for our purpose mainly because of the expressiveness offered.

The UPPAAL model checker [7] is used to analyze the timed automata model. The tool was developed by Yi and Larsen et al at Uppsala and Aalborg Universities respectively, with help from several other universities including the Radboud University [8]. It provides a graphical user-interface to compose and edit timed automata models. A simulator is available to animate the specification. The model checker is invoked as a batch process from the graphical user-interface. It performs a symbolic exhaustive search over the dynamically generated state space in order to verify some user-defined property. If the property does not hold, a counter example is automatically generated which can be visualized and animated for further analysis. UPPAAL is available for free download from *http://www.uppaal.com*.

Modeling the case study

The principal idea of the model is that system resources are either idle or performing some task, i.e. executing a computation or transferring data. Resource activity is modeled as a location in the timed automata. Transitions are defined from the idle (initial) location to each of the activity locations and vice-versa. The outgoing transitions are guarded by a counter which represents the number of outstanding requests for a particular activity. The counters are used to model the interaction between the different resources and the environment. The transition is enabled when the counter is greater than zero. When such a transition is taken, one is required to stay in the target location for the amount of time that corresponds to the user-defined maximum execution time of that task. This requirement can be relaxed if the best case execution time is also known. Then the time required to stay at the location is at minimum the best case execution time and at maximum the worst case execution time. The actual value taken is determined by a non-deterministic choice from this interval. When the execution time is reached, a transition back to the idle location is taken. Note that these models need to be constructed for each architecture that we want to investigate, because the deployment of software tasks over hardware resources are strongly coupled in this approach. The timed automata models are directly at the level of the abstract system performance model of the Y-chart in Figure 2.1, since separate application and architecture models do not exist. Pre-emption of tasks can also be modeled and template automata are available to describe the environment of the system. We will look at the different timed automata models in more detail in the next sections, whereby we investigate architecture (a) from Figure 2.6.

Modeling the computation resources

Figure 2.20 presents the basic automaton that models the behavior of the radio functionality (RAD). From the two sequence diagrams (Figures 2.3 and 2.5), it can be deduced that this functionality in fact consists of two operations, AdjustVolume and HandleTmc respectively. Each operation is represented as a location in the automaton. The automaton has a local clock x and two local constants, WCET_HT and WCET_AV. These constants represent the execution time of the operation, which is calculated as the worst-case execution time (expressed by the number of instructions to execute, as specified in the applicable sequence diagram) divided by the capacity of the hardware component on which it is deployed (which is expressed in million instructions per second, as specified in Figure 2.6).

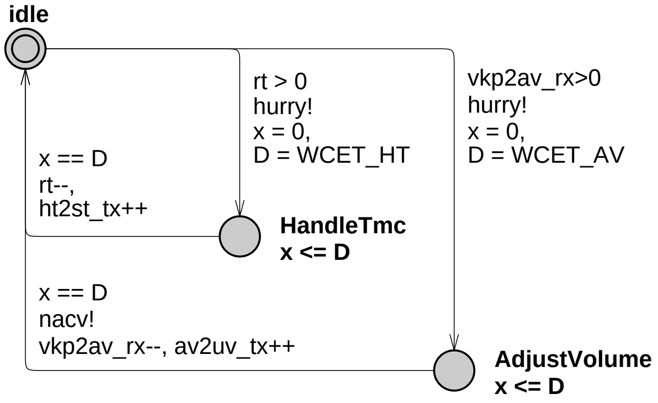


Figure 2.20: The automaton RAD representing the radio sub-system

Note that both outgoing transitions from the idle location in Figure 2.20 offer the hurry! event. This automaton communicates over a so-called *urgent* broadcast channel called hurry. Events offered on urgent channels are immediately processed, it has priority over all other actions in the model. It forces greedy automata behavior in our case, transitions are taken as soon as they are enabled. This modeling trick ensures that the model of the system will keep processing requests as soon as possible (whenever the resource is available). Otherwise the RAD automaton could postpone the handling of events indefinitely, since enabling a transition does not guarantee that it will be taken immediately, which would lead to possibly infinite worst-case response times.

The global variables rt and vkp2av_rx keep track of the number of pending calls to HandleTmc and AdjustVolume respectively. The transition is enabled if the guard evaluates to true, in other words if and only if the associated counter is greater than zero. Thus, in Figure 2.5, if the RAD automaton is in location idle and the ReceiveTmc event arrives, which is modeled by the increment of the rt variable, then the automaton immediately takes the transition to the location HandleTmc, whereby the clock x is reset. With “immediately” we mean that no time elapses between the arrival of the event and the execution of the transition. The automaton stays for WCET_HT time units in location HandleTmc and then returns to the idle location while generating an output event (modeled by incrementing the counter ht2st_tx) and decreasing the input event counter rt. As we will see in the next section, the global variables vkp2av_rx, ht2st_tx and av2uv_tx are the interface of the RAD automaton to the automaton representing the communication link. Similarly, the synchronization navc! is used to signal the completion of the AdjustVolume operation towards the environment, for measuring the end-to-end response time.

Note that the automaton in Figure 2.20 models a non-deterministic non-preemptive scheduler, which is not realistic in most cases. The UPPAAL language allows to model many kinds of schedulers. For instance, the automaton in Figure 2.21 models the radio functionality again, but now with a priority based non-preemptive scheduling strategy, in which the AdjustVolume operation has priority over the HandleTmc operation.

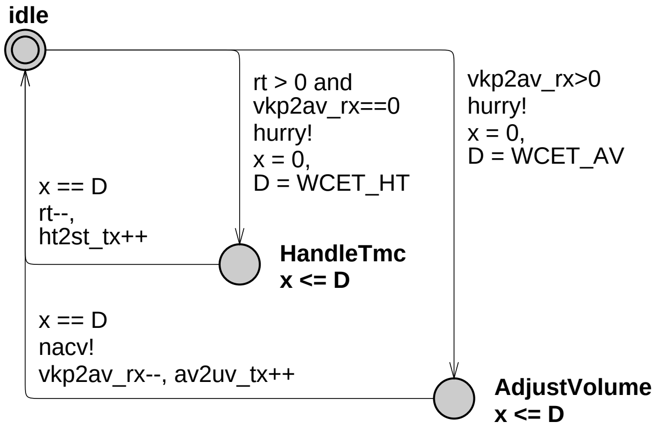
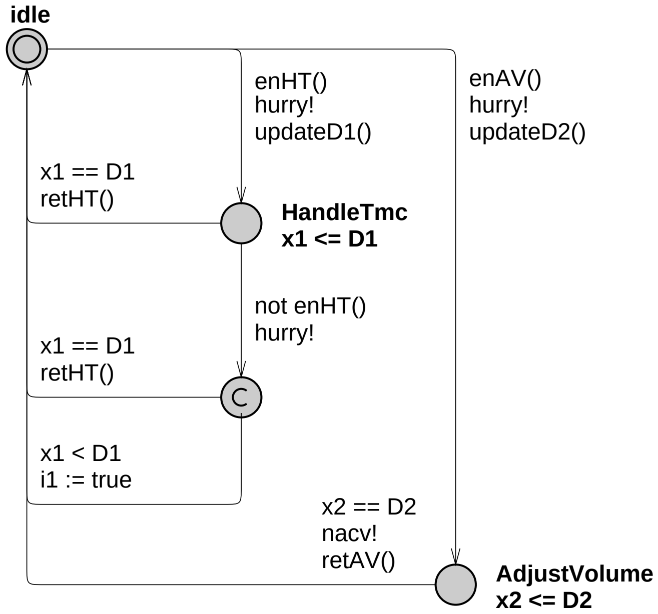


Figure 2.21: Adding scheduling priority to automaton RAD

In Figure 2.21, modeling priority is achieved by the additional expression vkp2av_rx == 0 to the guard of the transition from location idle to location HandleTmc. This means that TMC messages may only be handled if there are no outstanding AdjustVolume requests pending. More changes are required in order to model preemption and these are presented in Figure 2.22.

The guard expressions on the outgoing transitions from the idle location in Figure 2.22 have now been put in simple auxiliary functions enHT and enAV which return a Boolean value. Similarly, the update actions have been put in auxiliary functions updateD1, updateD2, retHT and retAV. This is done to increase readability and to simplify model maintenance. Furthermore, each activity has been given its own clock and a separate deadline variable, such that we can measure progress of each task individually. HandleTmc relates to clock x1 and variable D1 and AdjustVolume relates to clock x2 and variable D2. Now suppose that a low-priority TMC message is being processed, in other words we are at location HandleTmc, and a high-priority AdjustVolume request arrives, since the global variable vkp2av_rx has been incremented by another automaton. The guard expression contained in enHT now returns false since vkp2av_rx is non-zero and the transition towards the committed location is enabled. This transition is immediately taken due to the urgent communication over the hurry broadcast channel. Since no time is allowed to pass in committed locations,



```
// local definitions for the RAD automaton
clock x1, x2;                  int [0,ABOUND] D1 = 0;
bool i1 := false;              int [0,ABOUND] D2 = 0;

// auxiliary functions for the RAD automaton
bool enHT () {                 bool enAV {
  return (rt > 0) and            return (vkp2av_rx > 0);
         (vkp2av_rx == 0);     }
}

void updateD1() {              void updateD2() {
  if (not i1) {                  // reset the clock
    // reset the clock           x2 = 0;
    x1 = 0;                      // update the deadlines
    // set the deadline          if (i1 && (D1<ABOUND-WCET_AV))
    D1 = WCET_HT;                  D1 += WCET_AV;
  }                              D2 = WCET_AV;
}                              }

void retHT () {                void retAV () {
  // update the interface        vkp2av_rx--;
  rt--;                          av2uv_tx++;
  ht2st_tx++;                  }
  // reset the interrupt
  i1 := false;
}
```

Figure 2.22: Adding preemption to automaton RAD

we either return normally to idle if the interrupt occurred exactly at the end of the [0, WCET_HT] time interval (in other words x1 == D1) or we set a Boolean flag i1 to true, to indicate that the execution of HandleTmc has been interrupted. The pending AdjustVolume call is immediately processed since enAV is true. The auxiliary function updateD2 not only sets D2 and resets clock x2, but also increases D1 with WCET_AV, to account for the time lost due to handling this higher priority task. Note that the automaton keeps processing high-priority tasks until all of them are dealt with. Only then will the handling of lower priority tasks resume. Also note that tasks at the same priority level cannot interrupt each other, they simply run to completion. The model can be improved and simplified further by using so-called transition priorities, but this is not shown here.

Thus, preemption can be modeled, but care has to be taken because an integer variable in UPPAAL has a finite domain by definition. Therefore, it must not be the case that a task can be preempted infinitely often, since then D1 can grow to infinity and model checking is not possible anymore. However, the model checker can be used to prove that this is not the case by verifying some finite upper bound for these deadline variables. In this case study the deadline variables were restricted to $\text{ABOUND} = 1 \cdot e^6$, which corresponds to 1 second or roughly ten times the largest worst-case execution time of any task available in the model. This can alternatively be modeled as an explicit error state in the model whereby UPPAAL is then used to prove that this state cannot be reached. The modeling of the other computation components follow the same pattern as described above and are therefore not depicted here.

Modeling the communication resources

Modeling the communication in the system is surprisingly similar to the models we have presented for the computation in the previous subsection. A separate timed automaton is created for each communication resource. The automaton modeling the bus for the communication supporting the “Change Volume” application from Figure 2.3 running in parallel to the “TMC Message Handling” application from Figure 2.5 on architecture (a) is shown in Figure 2.23.

A location is created in the automaton for each message that is exchanged between the computation components that communicate through this link. The location reflects the fact that the message is being sent. The location is occupied for as long as the message transfer takes. We use the constants BYTES4 and BYTES64 in the model (but not shown here) to represent the time to transfer 4 and 64 bytes respectively over the communication link. This constant is again simply calculated as the message length (in bits, which is specified in the augmented sequence diagrams) divided by the bit rate (which is specified in the deployment diagram). Obviously, this formula can be adapted to compensate for expected protocol overhead.

The computation components interface to the communication link using (sharing) the set of global variables that count the number of outstanding messages of a particular type that need to be transferred over the link. If the first message from Figure 2.3 is to be sent from the MMI to RAD, from VolumeKeyPress to AdjustVolume, then the MMI automaton will simply increment the global vkp2av_tx variable to announce the message arrival at the communication resource. If the bus is idle, and all other global variables are zero and assuming that enVKP2AV returns true if vkp2av_tx is non-zero, then the transition towards the location VKP2AV location is immediately taken due to the hurry! synchronization. This location is occupied for BYTES4 time units and then the transition back to idle is taken. This return transition will decrement the vkp2av_tx

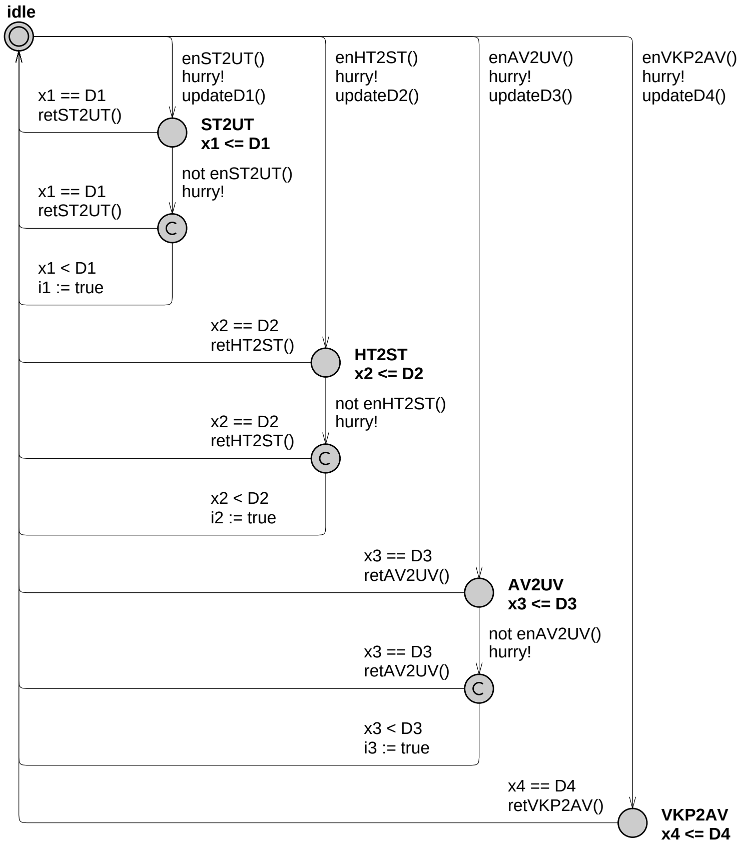


Figure 2.23: The automaton BUS

and increment the vkp2av_rx global variables, to indicate message delivery. In turn, this will enable the corresponding transition in the RAD automaton, as presented in the previous paragraph.

We can use the same strategy for dealing with priorities and scheduling as in computation resources. In fact, the communication resource shown in Figure 2.23 uses fixed-priority preemptive scheduling, as mandated by the case study description, whereby ST2UT has the lowest and VKP2AV has the highest priority. It is fairly easy to represent simple industrial serial bus interfaces such as RS485, priority based protocols such as Controller Area Network (CAN) or complex time-triggered protocols. For example, a solution for a TDMA bus concept is proposed by Perathoner et al in [80]. Less trivial however is the encoding of protocols that break large messages into pieces to prevent bus starvation, such as the well-known TCP/IP protocol stack.

The approach presented here has another interesting characteristic. If the interface (the global variables) remains the same, then it would be simple to replace a certain bus concept by another by merely replacing the bus automata. This would not affect the computation components at all. Therefore, we can easily investigate the impact of different bus protocols for a given deployment of software over hardware, i.e. to perform design space exploration.

Modeling of the environment

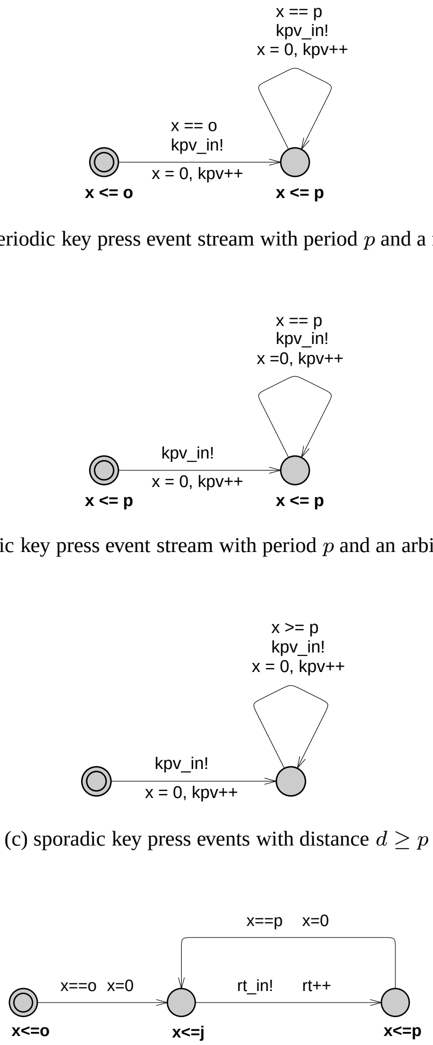
There are two actors that interact with the system from the environment: (i) a *user* who initiates the "Change Volume" or alternatively the "Address Look-up" scenario and (ii) a *radiostation* that initiates the "TMC Message Handling" scenario. There are two timed automata for each actor: a "normal" automaton and a "measuring" automaton. Which one is included in the network of timed automata that models the system depends on the particular worst case execution time property that we want to investigate. If we want to measure the response time of the *"Change Volume"* scenario, we add the "measuring" automaton for the *user* and the "normal" automaton for the *radiostation*. Vice versa, if we want to measure the response time of the *"TMC Message Handling"* scenario, then we add the "measuring" automaton for the *radiostation* and the "normal" automaton for the *user*.

We consider four basic kinds of event arrival models: (i) periodic, (ii) sporadic, (iii) periodic with jitter and (iv) event streams with bursts. For the strict periodic event model, an offset can be specified to force a phase shift in the signal (start of the first period). The periodic and sporadic event models can be expressed by automata as shown in Figure 2.24 (a-c) for key press events, using synchronization kpv_in and event counter kpv. The event model for periodic behavior with jitter (where the jitter is smaller than or equal to the period) can elegantly be expressed by the model proposed by Perathoner et al in [80], which is shown in Figure 2.24 (d) for TMC events, using synchronization rt_in and event counter rt. The behavior of events is called bursty when the jitter becomes larger than the period of the event. This can be modeled, but it is more involved than the previous model of periodic behavior with small jitter. The reason is that the subsequent intervals in which an event can occur now overlap. Figure 2.25 shows the UPPAAL model for a TMC event stream with bursts with offset o, period p, jitter j and a minimal separation time between two consecutive events of d, using synchronization rt_in and event counter rt. Note that we have used explicit channel names in all environment models shown here, mainly to enhance readability. In reality, generalized models are used which are instantiated by passing these explicit channel names as parameters to the timed automata model.

The automaton shown in Figure 2.25 has two local variables, req and hdl. The variable req is incremented every p time units (using clock x) which models that an event may be sent. The clock y is used to keep track of the next deadline for sending an event. An event may be sent if $\mathsf{z} > \mathsf{d}$, where clock z keeps track of the minimum inter-event separation time d, and $\mathsf{req} > 0$. When the event is sent, hdl is incremented. This signals that the deadline for the next event may be incremented by d time units. This is modeled by the reset of clock y after j (for the first event) or after p (for the other events) time units. Note that this automaton leads to a large increase of the state space of the model: it has three local clocks (but if $d = 0$ then clock z can be left out) and two local integer variables that both can count up to $1 + \frac{j}{p}$.

Every event automaton also has a "measuring" companion automaton which is used to record the worst-case response time of a generated system input event. These automata seem complicated at first sight, but they are logical in structure. The general idea is that a randomly chosen event is inserted into the system model by the environment automaton and its end-to-end response time is measured. It is assumed that all queues in the system model are order preserving and that events are never dropped. If the n-th event is inserted at t_1 and the n-th response is seen at t_2 then the observed execution time is $t_2 - t_1$.

The automaton in Figure 2.26 is the measuring companion of the event genera-



(a) strictly periodic key press event stream with period p and a fixed offset o

(b) strictly periodic key press event stream with period p and an arbitrary offset $o \leq p$

(c) sporadic key press events with distance $d \geq p$

(d) periodic TMC event stream with period p, offset o and jitter $j \leq p$

Figure 2.24: Example environment automata

tion automatons shown in Figure 2.24 (a-c). The measurement automaton is triggered whenever the event automaton generates a new system input event (stimulus) by synchronizing over a normal non-urgent channel, in this case kpv_in. The measurement automaton is also triggered whenever the system model generates a system output event (response) by synchronizing over normal non-urgent channels, in this case nacv and nvcv. For each kpv_in signal there should be exactly one corresponding nacv and one nvcv signal. Hence, two counters, cnta and cntv, both initially zero, are used to keep track of the number of stimuli and responses received so far. Both counters are incremented if kpv_in is received. Counter cnta is decremented whenever nacv is seen and cntv is decremented whenever nvcv is received. The self-transitions on the initial loca-

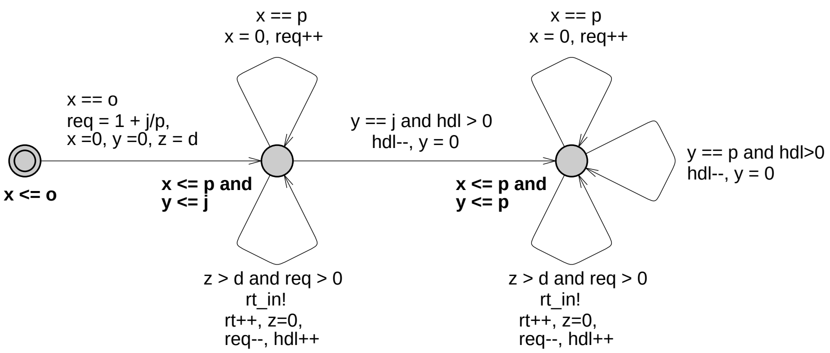


Figure 2.25: environment automaton describing TMC event bursts

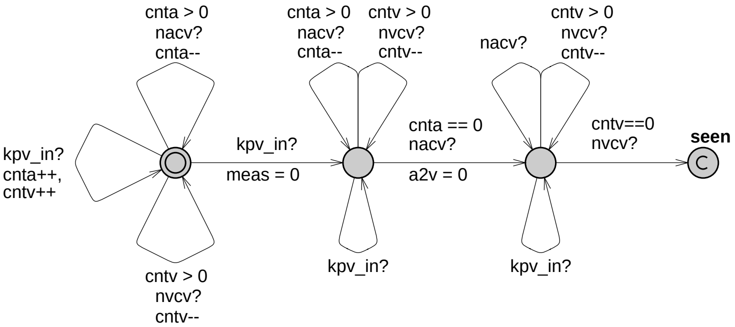


Figure 2.26: “Change Volume” response time measurement automaton *mv*

tion are used to deal with incoming kpv_in, nacv and nvcv events until the automaton chooses a random kpv_in signal to start the measurement. The out-going transition to the next location resets the clock meas, which is used for measuring. From now on, next incoming kpv_in signals are ignored and we wait until the cnta counter reaches zero. This indicates that the nacv response belonging to the kpv_in stimulus has been received. The out-going transition to the third location from the left resets the a2v clock. The next incoming nacv signals are now also ignored and we wait until cntv becomes zero. This indicates that the nvcv response belonging to the kpv_in stimulus has been received and we move to the location seen. The value of the clock meas now contains the *k2v* response time and idem for a2v.

Note that at most one end-to-end measurement can be in progress at any time. Different execution time measurements require therefore different combinations of event stream and measurement automata. Dummy measurement automata, which simply always accept the synchronization actions offered by the event stream and system automata are used for those timing requirements that are currently not under investigation. The modeling of the other measuring companion automata follows the same pattern as explained above and are therefore not depicted here.

Analysis of the model

The system model is constructed by composing a network of timed automata from the resource and environment automata described above. UPPAAL is then asked to verify whether a certain response time is within the set of reachable states of the model. By using a binary search approach manually, the exact best and worst-case response times can be determined. The worst-case response time *k2v* of the "Change Volume" scenario can thus be found using UPPAAL by finding the smallest *C* such that Property 2.19 is satisfied. The operator "**AG**" stands for "always globally" which implies that for all reachable states this property holds.

$$\mathbf{AG}(\mathit{mv.seen} \longrightarrow \mathit{meas} < C) \tag{2.19}$$

The UPPAAL model has been analyzed for 5 different requirements and 5 different sets of environment models. The results have been collected in Table 2.2. UPPAAL version *4.0.6* has been used with default options, unless indicated otherwise. The rows indicate which requirement is measured and in what context (combination of applications) the analysis was performed.

Event model *Requirement*	*po* ($o = 0$)	*pno*	*sp*
HandleTMC (+ ChangeVolume)	336.111	345.269	345.713
HandleTMC (+ AddressLookup)	172.106	239.079	239.079
A2V ChangeVolume (+ HandleTMC)	27.717	27.717	27.717
K2V ChangeVolume (+ HandleTMC)	41.796	41.796	41.796
AddressLookup (+ HandleTMC)	79.075	79.075	79.075

Event model *Requirement*	*pj* ($j \leq p$)	*bur* ($j \leq 2p$, $d = 0$)
HandleTMC (+ ChangeVolume)	> 400.000 (df)	> 500.000 (rdf)
HandleTMC (+ AddressLookup)	239.079	239.079
A2V ChangeVolume (+ HandleTMC)	> 27.715 (bf)	> 27.715 (bf)
K2V ChangeVolume (+ HandleTMC)	> 41.795 (bf)	> 41.795 (bf)
AddressLookup (+ HandleTMC)	79.075	79.075

Table 2.2: UPPAAL worst-case response time analysis results (in milliseconds)
(bf = breadth first, (r)df = (random) depth first)

The first column of the top table shows the results for strictly periodic event streams with a *user-defined* offset o for all events (*po*, $o = 0$), which reflects fully dependent or synchronous environment models. In the second column of the top table, the results are shown for strictly periodic event streams with an *unknown* offset for all events (*pno*, $o \leq p$), which reflects fully independent or asynchronous environment models. The third column of the top table presents the analysis results for sporadic event streams (*sp*, $d \geq p$) where only a lower bound is specified for the event inter arrival time.

In the first column of the second table we show periodic event streams with small jitter (*pj*, $j \leq p$) for the "radio station" environment model and sporadic events for the others. And finally, we use event streams with bursts (*bur*, $j \leq 2p$, $d = 0$) for the "radio station" and sporadic event streams for the others in the second column of the bottom table.

Design question 1. Analysis of the *"TMC Message Handling"* and *"Address Lookup"* scenario combination proved to be no problem. The verification times for *po*, *pno* and *sp* where so small (typically less than a second) that a binary search could easily be performed. The *pj* and *bur* scenarios took a bit more effort, but still a binary search was feasible (verification times typically in the order of a few minutes). The *"TMC Message Handling"* scenario in combination with *"Change Volume"* proved to be a problem for the *pj* and *bur* scenarios. This is due to the large difference in time scales of the event automata: the period of the "radiostation" events is in the order of seconds whereas the period of the "change volume" events is in the order of milliseconds. Such differences are bad for the symbolic representation of clock values that is used by UPPAAL. However, UPPAAL can still be used as a "structured testing" tool with its options for the search order (*df = depth first*, *rdf = random depth first*). The verifier will try to find a *counterexample* of the property. If it finds one, then the constant C in the property is a lower bound on the worst-case response time of the event. This is indicated by the "greater than" symbols in Table 2.2.

Note that the *"Address Look-up"* and *"Change Volume"* worst-case response time values remain constant since (i) they have priority over the "radiostation" related events and (ii) their event model parameters are such that events are never queued for processing. For example, each *"Address Look-up"* event is fully processed on each resource before the next event arrives on the same resource. If we would allow jitter to the *"Address Look-up"* scenario, such that two events might overlap, then the bound of 79.075 would certainly increase. Also note that the results for *po* and *pno* are not identical, which indicates that a phase shift between the environment models *does* matter in this case. We get this result almost for free, neither the modeling nor the analysis effort is much influenced. The approach shown here can treat asynchronous and synchronous environment models by simply removing an invariant from the appropriate environment models.

Design questions 2 and 3. The approach shown in this section is not convenient for studying design questions 2 and 3. This is mainly caused by the fact that we can only study the best- and worst-case execution times, not the resource availability. When searching for these hard and tight worst-case execution time (WCET) results, we explore the state space such that the resource usage is maximized. But that does not imply that all resources are busy all the time. If a WCET value is proven to be in the set of reachable states, we have *no* information as to the level of resource usage. This can be investigated by studying the counter example which is produced if a WCET value is chosen that is slightly lower than the value that was found previously. However, this is a cumbersome and slow process.

Observations on the experiment

UPPAAL is a generic tool. Many model types and scenarios can be expressed (i.e. event models, communication, computation, mix of preemptive and non-preemptive elements) and analyzed. However, manual construction and maintenance of these models is error prone and should be automated to be useful in an industrial setting. Evaluation of the model is in the order of minutes if the state space is tractable; the values then found are hard *and* tight. Tractability however, is mainly determined by the amount of non-determinism in the model. For example, when two event streams have an average period which is orders of magnitude apart, the state space explodes even though the model of the system is small and simple. In this case, the model checker will not be able to find an answer in an acceptable amount of time.

2.3.4 Parallel and Object-Oriented Specification Language

Short overview of the technique

The Parallel Object-Oriented Specification Language (POOSL) is a general purpose formal specification language which lies at the core of the Software/Hardware Engineering (SHE) system-level design method. POOSL was developed by Voeten and Van der Putten at the Technical University Eindhoven [97]. The language contains a set of powerful primitives to formally describe concurrency, distribution, synchronous communication, timing and functional features of a system into a single, high-level, executable model. The language uses a process algebra notation that is strongly influenced by Milner's Calculus of Communicating Systems (CCS) [73].

The formal semantics of POOSL is defined in terms of timed probabilistic labeled transition systems. This mathematical structure enables the unambiguous interpretation of POOSL models, both for validation through simulation as well as verification through model checking and proof. Currently, the SHE method is accompanied by two simulation tools, SHESim and Rotalumis. Verification tools are feasible and planned but not yet available. SHESim is a graphical environment intended for the incremental specification, modification and validation of POOSL models. Rotalumis is a high-speed execution engine which is aimed at batch-oriented simulation.

The simulation algorithm at the core of both tools, first constructs a set of so-called process execution trees (PETs) which are directly derived from the POOSL models. Basically, each POOSL process is translated into a PET whereby the leaves represent the actions that can be taken by the process. Two types of actions are possible: state actions and time actions. The PETs are interpreted using a two-phase execution approach. In the first phase, all PETs greedily execute state actions asynchronously until completion, which implies that they are either a) terminated or b) blocked on a read or write action on a communication channel where no matching data is offered or c) a time step needs to be taken. The execution of the actions performed in phase 1 is considered to have taken zero time. The simulation process can continue if there is at least one process that needs to make a time step, otherwise a deadlock has occurred. In phase 2, time passes for all processes synchronously, by the minimum of all outstanding time steps. As a result, at least one process can continue processing actions. So, after this time step has been taken, the simulation resumes as if it was a new phase 1 step. This simulation algorithm has been proven to correctly implement the formal semantics of POOSL [40]. Both SHESim and Rotalumis have been used to model this case study. The tools are available for free download from *http://www.es.ele.tue.nl/poosl/*.

Modeling the case study

De Hoon constructed a model of our case study in [23] using POOSL. The approach followed in this work is to explicitly model the events that flow through the system instead of using abstractions like arrival curves. A POOSL *data* class, called `EventProperties`, not shown here, is defined to administer the timing properties of an event, such as release time, start time, finish time, relative deadline and so on. Instances of this class are created and given a unique identifier. These event objects are manipulated by the application model of the system. Applications, such as the "Change Volume" scenario, are represented by a directed task graph. A POOSL *process* class, called `ComputationTask`, is defined to provide a generic model of such an application task. This class is shown in Figure 2.27.

≪ process ≫ ComputationTask
≪ instantiation parameters ≫ TaskName : String ComputationLoad : Real Priority : Integer
≪ instance variables ≫ TaskId : Integer

≪ methods ≫
HandleEvent()()
Execute()()

≪ initial method call ≫ HandleEvent()()
≪ messages ≫ in!event out?event task!execute task?executed

```
01 HandleEvent()()
02 | E : EventProperties |
03   in?event(E);
04   par
05     HandleEvent()()
06   and
07     Execute()();
08     out!event(E)
09   rap.

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

Figure 2.27: A POOSL specification of a generic application task

Three parameters are provided whenever an instance of `ComputationTask` is created. The task is given a name, its computation load (in terms of the number of cycles to execute) and a priority. The operation `HandleEvent` is immediately invoked after the initialization of the object. This operation performs a blocking read on the *in*-channel in line 3. The par-and-rap block is executed as soon as an event has arrived. The par-part (line 5) is executed in parallel with the rap-part (line 7 and 8). The par-part is a tail-recursive call to `HandleEvent` which causes the process to be ready to accept new messages on the *in*-channel immediately. The rap-part first calls the `Execute` operation which offers a message on the *task*-channel. This is the request to the resource to allocate a certain number of cycles for this task with a given priority (line 11-12). The `Execute` operation waits until the resource notifies that these cycles have indeed been spent on behalf of this task (line 13-14). A guard is defined in line 14 to check whether it was *this* task that was confirmed to be completed. This check is important because there may be several tasks deployed on the same resource and we need to distinguish each one individually. Finally, the task offers the event on the *out*-channel in line 8 after the operation `Execute` returns in line 7. This triggers the next task in the task graph. Communication over a bus follows a similar pattern, as shown in Figure 2.28. The only notable difference is the name of the channel in the `TransferMsg` operation.

≪ process ≫ CommunicationTask
≪ instantiation parameters ≫ MessageId : Integer MessageSize : Real Priority : Integer
≪ instance variables ≫

≪ methods ≫
HandleEvent()()
TransferMsg()()

≪ initial method call ≫ HandleEvent()()
≪ messages ≫ in!event out?event msg!transfer msg?transferred

```
01 HandleEvent()()
02 | E : EventProperties |
03   in?event(E);
04   par
05     HandleEvent()()
06   and
07     TransferMsg()();
08     out!event(E)
09   rap.

10 TransferMsg()()
11   msg!transfer( MsgId,
12     MessageSize, Priority );
13   msg?transferred( MsgServed
14     | MsgServed = MsgId ).
```

Figure 2.28: A POOSL specification of a generic communication task

The next step is to define the resources of our system model. In Figure 2.29 we present the definition of a fixed-priority preemptive resource as a POOSL *process* class. Here, we show the definition of a computation resource. A communication resource is virtually identical. The `FixedPrioCompResource` waits for scheduling requests on the *task*-channel on line 2. It is the counter action of the message offer on line 11 in Figure 2.27. If such a message arrives, it invokes the `ComputeTask` operation on line 3. This operation executes a `delay` statement on line 10 whereby time is allowed to elapse by $ServLoad/MIPS$ seconds. This corresponds to the worst-case execution time of the task deployed on this resource, since the parameter `ServLoad` is defined as the maximum number of cycles to execute. However, this `delay` statement can be interrupted when another message arrives during the time step (line 11-14). But only messages that offer a priority level that is higher than the current priority are actually accepted in the guard on line 12. The operation `ComputeTask` is called recursively on line 13-14 if this particular situation occurs. The requesting application is informed that the task has been executed on line 15. Finally, the operation `HandleTask` is called again on line 4 to accept the next scheduling request for this resource.

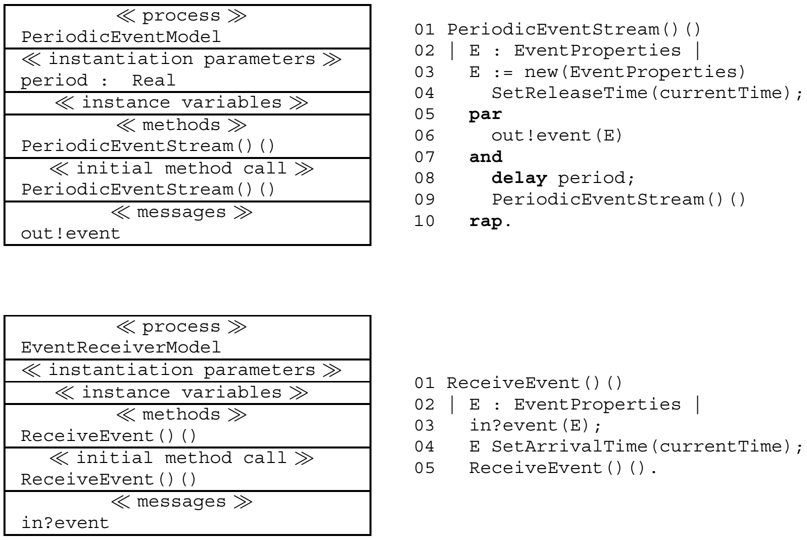
```
≪ process ≫
FixedPrioCompResource
≪ instantiation parameters ≫
MIPS : Real
≪ instance variables ≫
≪ methods ≫
HandleTask()()
ComputeTask()()
≪ initial method call ≫
HandleTask()()
≪ messages ≫
task?execute
task!executed
```

```
01 HandleTask()()
02   task?execute(Task, Load, Prio);
03   ComputeTask(Task, Load, Prio);
04   HandleTask()().

05 ComputeTask(ServTask : Integer,
06   ServLoad : Real,
07   ServPrio : Integer)()
08 | ReqTask, ReqPrio : Integer,
09   ReqLoad : Real |
10   interrupt delay ServLoad / MIPS with
11     ( task?execute( ReqTask, ReqLoad,
12         ReqPrio | ReqPrio > ServPrio );
13       ComputeTask(ReqTask, ReqLoad,
14         ReqPrio)() );
15   task!executed(ServTask).
```

Figure 2.29: A POOSL specification of a fixed priority computation resource

The final components we need in order to compose the abstract performance model is the event generator that reflects the environment and the event sink to register the end-to-end elapse time. In Figure 2.30 we show the `PeriodicEventModel` and the `EventReceiverModel` process classes. The former offers an event on the *out*-channel with a certain period and registers the release time in the event. The latter accepts events on the *in*-channel and registers their arrival time. The application response time is the difference between those two values. Similar components are available for event generators with jitter and bursts. They make use of a random number generator that is available as a standard data class in POOSL.

The final step is to compose the abstract performance model using the template classes that we have presented so far. This can for example be done using the SHESim tool. SHESim provides a graphical user-interface to compose POOSL models. Objects are represented as boxes. The channels are visible as interface points and they can be connected together. The tool verifies whether the interfaces are of the same type when they are composed. Figure 2.31 shows the abstract performance model of architecture (a) in the user-interface of SHESim. On the left, we see the event generators, on the right the event sinks. At the bottom we see the four resources and the remaining components represent the application tasks and message exchanges.



```
≪ process ≫
PeriodicEventModel
≪ instantiation parameters ≫
period : Real
≪ instance variables ≫
≪ methods ≫
PeriodicEventStream()()
≪ initial method call ≫
PeriodicEventStream()()
≪ messages ≫
out!event
```

```
PeriodicEventStream()()
| E : EventProperties |
  E := new(EventProperties)
    SetReleaseTime(currentTime);
  par
    out!event(E)
  and
    delay period;
    PeriodicEventStream()()
  rap.
```

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

```
ReceiveEvent()()
| E : EventProperties |
  in?event(E);
  E SetArrivalTime(currentTime);
  ReceiveEvent()().
```

Figure 2.30: Example event generators and event sinks in POOSL

Analysis of the model

All three of the suggested typical design problems can be studied using this approach. However, hard guarantees cannot be given because there is no proof that we will visit all so-called “corner cases” during simulation. Corner cases are parts of the state space that are only reached under specific conditions, for example a transition in the simulation model with a very low probability attached to it. Exposing these corner cases therefore requires potentially infinitely long simulation runs. Hence, full coverage of the state space can, in general, not be guaranteed.

Design question 1. An abstract performance model must be created using SHESim for each proposed architecture. Finding the worst-case timing of the application requires many simulation runs because the two input event streams are strictly periodic but *not* correlated. This implies that the simulation must be repeated many times whereby random values are chosen for the initial offset of either event stream. Since time is a real valued property in our model, infinitely many values can be selected for this parameter. The maximum response time of the application is the maximum that was found for this value over all simulation iterations. However, the number of iterations must be determined carefully in order to create realistic results. In principle, we only have to simulate until the hyper-period of our event stream is reached because the model of our case study is fully deterministic. Nevertheless the number of simulation iterations can become quite large, for example if the periods of the input event streams and the time delay caused by the fastest task are orders of magnitude apart. Resources usage can be studied by observing the resource models, for example by time stamping the `ComputeTask` operation in Figure 2.28.

Design question 2 and 3. The obvious way to answer these questions is to repeat the experiment as presented in the previous case several times, but each time with different settings for the event input rates or the resources capacities. This is identical to the parameter sweep approach proposed in Section 2.3.1. But, with this model, we can also

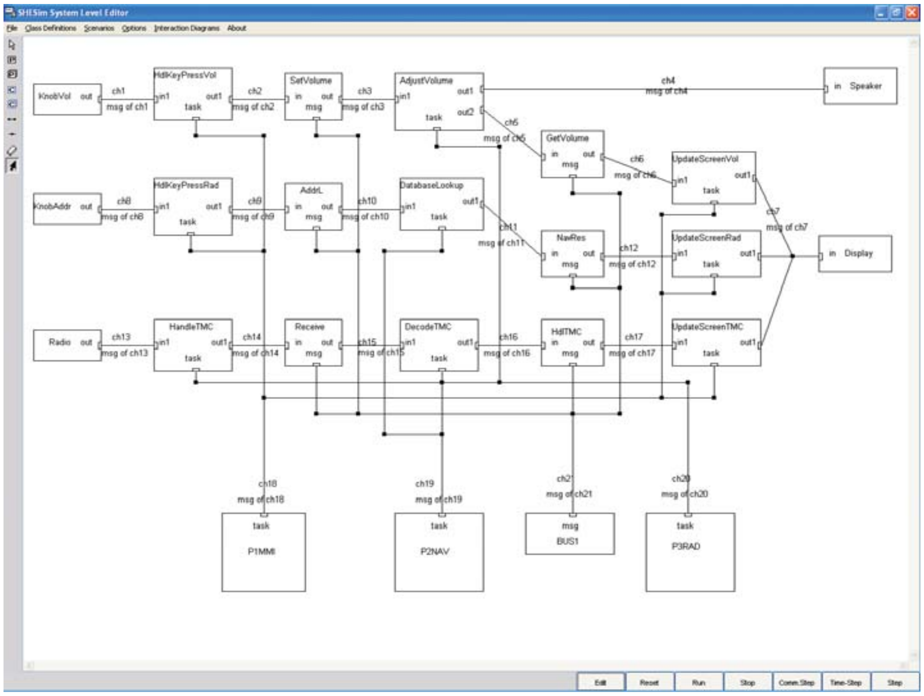


Figure 2.31: The performance model of architecture (a) in SHESim

study the *average* performance of the system, as opposed to the other techniques presented so far. Suppose we know both the best- and the worst-case load of each task in the system, in terms of number of cycles. If we assume some distribution of likelihood in this interval, say a normal distribution, we could vary the cost of executing that function by passing a random value taken from this interval, instead of the worst-case value, whenever we ask the resources to schedule our task (in line 11 in Figure 2.27). This approach was studied by Florescu and De Hoon in [36]. Of course, it requires even more repetitions for the simulations than in the previous case, for example because the hyper-period limitation per simulation run is not longer viable since the model is no longer deterministic. Nevertheless, interesting results can be obtained at reasonable cost that also relate to design question 1. For example, the average end-to-end response time that is reported per application after each simulation run, can be represented as a histogram. This histogram shows us the relationship between the average response time and the required worst-case timing requirement. Is the average close to the limit value or does it only occur in few cases that the worst-case is reached? But we know that this worst-case response time might not have been reached yet because we did not simulate long enough. The approach proposed in [36] is to approximate the histogram by some Gaussian distribution. From this distribution, we can calculate, using standard mathematics, the likelihood that the application would exceed the worst-case timing requirement. This gives us valuable insight into the robustness of the application and for soft real-time applications it may even provide sufficient proof.

Observations on the experiment

Evaluation of this model is in the order of minutes to hours, depending on the property to analyze. Despite the fact that the simulator conforms to the formal semantics of the language, no guarantee can be given that the model is completely covered during simulation. Exact best- and worst case values are not necessarily found during analysis, i.e. the bounds found by the simulator are *not* hard. Exhaustive analysis techniques are available but have not yet been implemented into tools. These exhaustive analysis techniques are subject to the state-space explosion problem, just like in UPPAAL. However, POOSL is able to describe and analyze the nominal (average) behavior of the system and complex system - environment interactions, for example involving timing dependencies between input stimuli. POOSL is well-suited for analysis of soft real-time systems.

2.3.5 Vienna Development Method

VDM++ is an object-oriented and model-based specification language with a formally defined syntax, static and dynamic semantics. It is a superset of the ISO standardized notation Vienna Development Method - Specification Language (VDM-SL) [24]. VDM++ was originally designed in the ESPRIT project AFRODITE and it was subsequently improved and tools were implemented by IFAD [27]. Different VDM dialects are supported by industry strength tools, called VDMTOOLS, which are currently owned and further developed by CSK [2] [31]. A timed extension to VDM++ was delivered as part of the VICE project: “VDM++ In a Constrained Environment” [75].

The dynamic semantics of an executable subset of VDM++ is provided as a constructive operational semantics specified in VDM-SL which is roughly 500 pages including informal explanation [68]. The core of this specification is an abstract state machine which is able to execute a set of formally defined primitive instructions. Special functions are supplied to “compile” each abstract syntax element into such a sequence of instructions. The dynamic semantics specification is executable and can be validated using VDMTOOLS. The test suite contains several thousand test cases which are also used to verify the implementation. The industrial success of VDMTOOLS is, for a large part, due to excellent conformance of the tool to the formally defined operational semantics and the round-trip engineering with UML. We present an overview of the language and the timed extensions in this section. For an in-depth presentation of the language and supporting tools [3] see [30].

Short overview of the technique

In VDM++, a model consists of a collection of class specifications, whereby we distinguish active and passive classes. Active classes represent entities that have their own thread of control and do not need external triggers in order to work. In contrast, passive classes are always manipulated from the thread of control of another active class. We use the term object to denote the instance of a class. More than one instance of a class might exist. An instance is created using the *new* operator, which returns an object reference. A class specification has the following components:

Class header: The header contains the class name declaration and inheritance information. Both single and multiple inheritance are supported.

[2] Free tool support can be obtained from *http://www.vdmtools.jp/en/*.

[3] Many examples can be found at *http://www.vdmbook.com* and *http://www.vdmportal.org*.

Instance variables: The state of an object consists of a set of typed variables, which can be of a simple type such as *bool* or *nat*, to represent Boolean values and natural numbers respectively, or abstract data types such as sets, sequences, maps, tuples, records and object references. The latter are used to specify relations between classes. Instance variables can have invariants and an expression to define the initial state.

Operations: Class methods that may modify the state can be defined implicitly, using pre- and postcondition expressions only, or explicitly, using imperative statements and optional pre- and postcondition expressions.

Functions: Functions are similar to operations except that the body of a function is an expression rather than an imperative statement. Functions are not allowed to refer to instance variables, they are pure and side-effect free.

Synchronization: Operations in VDM++ are re-entrant and their invocation is defined with synchronous (rendez-vous) semantics. It is possible to constrain the execution of an operation by specifying a permission predicate [66]. A permission predicate is a Boolean expression over so-called history counters that acts as a guard for the operation, for example to express mutual exclusion. History counters are maintained per object to count the number of requests, activations and completions per operation.

Thread: A class can be made "active" by specifying a thread. A thread is a sequence of statements which are executed to completion at which point the thread dies. The thread is created whenever the object is created but the thread needs to be started explicitly using the *start* operator. It is possible to specify threads that never terminate.

In the VICE project [75], time was added to VDM++ by assigning a configurable default duration to each basic language construct. Whenever a statement is evaluated by the interpreter, the global notion of time is increased by the specified amount. In this way, it was possible to simulate the timed behavior of a program running on a single processor. In addition, the user can specify the task switch overhead and the scheduling policy used, as simulation parameters. The duration statement was added to the language, with the concrete syntax *duration(d) IS*, which implies that all statements in *IS* are executed instantaneously and then time is increased by *d* time units. The duration statement is used to override the default execution time for *IS*. Furthermore, the periodic statement was introduced, with the concrete syntax *periodic(d)(Op)*. This statement can only be used in the thread clause to denote that operation *Op* is called strict periodically every *d* time units. The *time* keyword can be used to refer to the current value of the so-called simulation "wall-clock".

Modeling the case study

There are many ways in which the VDM++ language can be used to model systems. Very high-level and abstract specifications can be made, which are usually tailored towards rigorous analysis of certain system aspects, for example using deductive proof. This traditional style of formal specification is not used here, we intentionally apply a design oriented style of specification. The model shall reflect the implementation structure of the system as closely as possible, including formal descriptions of the actual computations performed. This style of specification is typically more commonly used

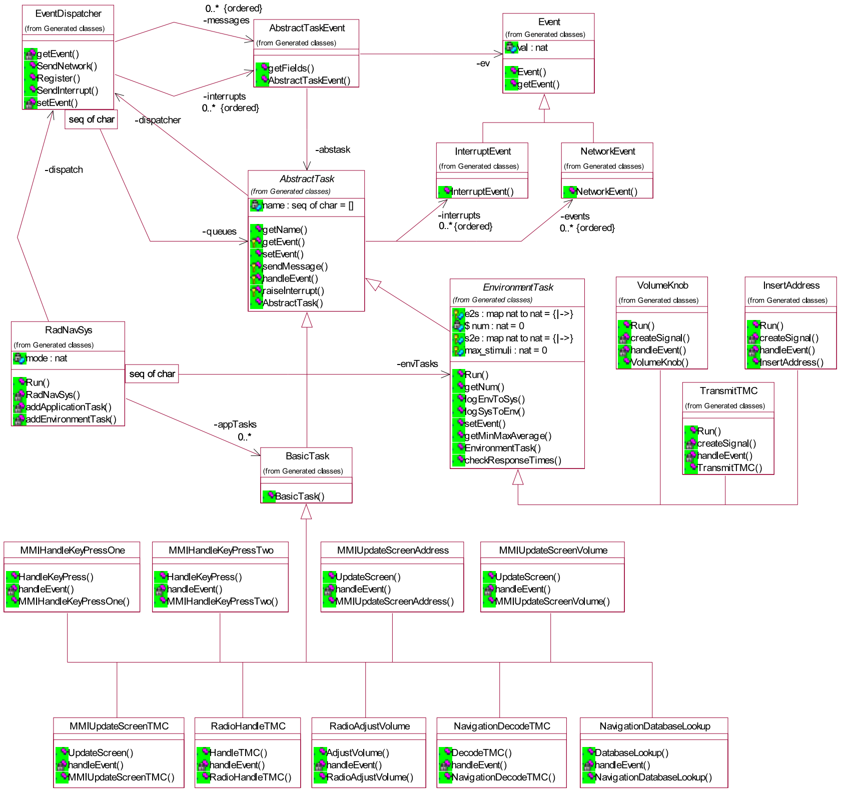


Figure 2.32: UML class diagram of the Timed VDM++ case study model.

in industry and also sets VDM++ apart from the other methods discussed previously. It is a pragmatic approach whereby emphasis is on exploring the design by simulation and testing, not necessarily on finding undisputable formal proof of correctness.

The in-car radio navigation system can be modeled as a set of classes. A UML class diagram of the model is presented in Figure 2.32. All environment and all the application tasks are modeled as separate threads. These threads obviously need to communicate, for example to exchange stimuli and responses between the environment and the system or for inter thread communication within the system. The mechanism available to communicate between objects in VDM++ is to use method invocation. However, the Timed VDM++ language developed in the VICE project provides synchronous operation call semantics only. This implies that an operation is always executed by the thread of the callee. This complicates the specification of embedded systems because they are reactive and therefore asynchronous by nature.

The usual way to break this strong coupling is to use a so-called event loop mechanism using message passing. The receiving thread performs a blocking read on some input queue. The sending thread does not call the required operation directly, but puts a message in the input queue of the receiving thread. The sending thread can continue after message delivery and the receiving thread will unblock and process the incoming message by invoking the appropriate operation. Such an event loop mechanism is at the basis of the VDM++ model presented here and this basically mimics the functionality offered by a contemporary real-time operating system.

A small framework was developed for modeling reactive systems. Assume given a class called `Event` that only supports some basic identity functions. We define two specialized classes called `InterruptEvent` and `NetworkEvent` by sub-classing from `Event`. These derived classes are used to distinguish communication between the environment threads and system threads from inter system thread communications. Furthermore, `InterruptEvents` shall be treated instantaneously while a time penalty is associated with `NetworkEvents`, whereby the former also have priority over the latter.

The `AbstractTask` class presented in Figure 2.33 provides the basic functionality to handle events. Two separate input queues are maintained, one for interrupt and one for network events. There will be a single `EventDispatcher` class instance that will process the routing of all messages between all `AbstractTasks` in the model. The `EventDispatcher` can inject messages by calling the public `setEvent` operation of the applicable `AbstractTask` instance. Note that the operation `getEvent` gives priority to interrupt over network events. The `AbstractTask` can send messages to other tasks on the system by calling `sendMessage` or back to the environment by calling `raiseInterrupt`. The `EventDispatcher` will handle the message in both cases without blocking the `AbstractTask`.

Modeling the system

Notice that the class `AbstractTask` is passive, since there is no thread specification. The derived class `BasicTask`, as presented in Figure 2.34, actually implements the event loop mechanism. It has now become an active object that is constantly processing incoming events. Note that the thread will block if there are no messages available due to the permission predicate defined on the `getEvent` operation declared in the `sync` block of the base class `AbstractTask`.

The class `BasicTask` acts as a generic base class for all system threads in our model. Consider for example the class `RadioAdjustVolume`, shown in Figure 2.35. Here, we see an implementation of the abstract operation `handleEvent`, which calls the operation `AdjustVolume` synchronously and then sends a message to the next task called `UpdateScreenVolume` asynchronously. Note that we have abstracted away entirely from the complexity of whatever the operation `AdjustVolume` really does, by using the `skip` statement, since we are just concerned about the time penalty specified by the duration statement. But the `skip` statement could of course be replaced by the algorithm that specifies its behavior appropriately.

The specification of the other system tasks follows the same strategy as the class `RadioAdjustVolume` and they are therefore not discussed in detail here.

Modeling the environment

The `EnvironmentTask` class presented in Figure 2.36 provides the basic functionality to generate and administer events. Each event can be given a unique number by calling the operation `getNum`. The operation `logEnvToSys` is used to register the time when an input stimulus has been generated and the operation `logSysToEnv` is used to register the time at which the response was observed. The worst-case response time of each stimulus / response pair can be checked by calling the `checkResponseTimes` function. The operation `getMinMaxAverage` is used to compute the minimum,

```
class AbstractTask

instance variables
  name : seq of char := [];

  events : seq of NetworkEvent := [];
  interrupts : seq of InterruptEvent := [];

  dispatcher : EventDispatcher

operations
  public AbstractTask: seq of char * EventDispatcher ==> AbstractTask
  AbstractTask (pnm, ped) == atomic ( name := pnm; dispatcher := ped; );

  public getName: () ==> seq of char
  getName () == return name;

  public setEvent: Event ==> ()
  setEvent (pe) ==
    if isofclass(NetworkEvent,pe)
    then events := events ^ [pe]
    else interrupts := interrupts ^ [pe];

  protected getEvent: () ==> Event
  getEvent () ==
    if len interrupts > 0
    then ( dcl res : Event := hd interrupts;
           interrupts := tl interrupts; return res )
    else ( dcl res : Event := hd events;
           events := tl events; return res );

  protected handleEvent: Event ==> ()
  handleEvent (-) == is subclass responsibility;

  protected sendMessage: seq of char * nat ==> ()
  sendMessage (pnm, pid) == dispatcher.SendNetwork(name, pnm, pid);

  protected raiseInterrupt: seq of char * nat ==> ()
  raiseInterrupt (pnm, pid) == dispatcher.SendInterrupt(name, pnm, pid)

sync
  -- setEvent and getEvent are mutually exclusive
  mutex (setEvent, getEvent);

  -- getEvent is blocked until at least one message is available
  per getEvent => len events > 0 or len interrupts > 0

end AbstractTask
```

Figure 2.33: The `AbstractTask` base class.

```
class BasicTask is subclass of AbstractTask

operations
  public BasicTask: seq of char * EventDispatcher ==> BasicTask
  BasicTask (pnm, ped) == AbstractTask(pnm, ped);

thread
  while (true) do
    handleEvent(getEvent())

end BasicTask
```

Figure 2.34: The derived class `BasicTask`.

maximum and average response time observed. Note that this class is passive since no

```
class RadioAdjustVolume is subclass of BasicTask

operations
  public RadioAdjustVolume: EventDispatcher ==> RadioAdjustVolume
  RadioAdjustVolume (pde) == BasicTask("AdjustVolume",pde);

  public AdjustVolume: () ==> ()
  AdjustVolume () == duration (100) skip;

  handleEvent: Event ==> ()
  handleEvent (pe) ==
    ( AdjustVolume();
      sendMessage("UpdateScreenVolume", pe.getEvent()) )

end RadioAdjustVolume
```

Figure 2.35: The `RadioAdjustVolume` class.

thread has been declared, neither here nor in the base classes.

The derived class `VolumeKnob`, shown in Figure 2.37 defines the behavior of the environment thread responsible for inserting key press events into the system for the "Change Volume" scenario. The `createSignal` operation is declared as a periodic thread with a period of 1000 time units. It creates a new event identifier, registers the current time and injects the event into the system model by calling the operation `raiseInterrupt`. The operation `handleEvent` is called as soon as the response returns to the environment. The current time is logged and its consistency is checked by the post condition, which states that the response time of all events received so far shall be less than 200 time units.

The specification of the other environment tasks follows the same strategy as the class `VolumeKnob` and they are therefore not discussed in detail here. Finally, the top-level system model `RadNavSys`, as presented in Figure 2.38, can be constructed from the building blocks presented.

The VICE interpreter needs to be set up appropriately before the model can be executed. We use the interpreter in preemptive scheduling mode with all integrity checking options enabled. This includes dynamic type, invariant, pre- and post condition checking. Furthermore, task priorities are specified in a simple external text file, whereby each class name is related to an associated priority level. Once an instance of this class is created by the interpreter, it will be assigned this priority level. This thread priority is immutable during execution and remains valid until the thread dies. The model is started by running the "`new RadNavSys().Run()`" command. The simulation returns with the collected execution time statistics in case the timing requirements where met. The simulation terminates prematurely if a worst-case execution time limit was exceeded, for example in a post condition of the `handleEvent` operation of an environment task. The VDMTOOLS debugger can be used to capture such an event and to analyze its cause.

Analysis of the model

We have not been able to evaluate architecture (a) from Figure 2.6 using the Timed VDM++ notation that was developed in the VICE project, using VDMTOOLS version 7.2. The main reason is that all threads in the model are implicitly bound to the same physical resource, in other words a single CPU. Therefore, only evaluation of architecture (e) could possibly lead to useful results here. Since only the active thread can

```
class EnvironmentTask is subclass of AbstractTask

instance variables
  static private num : nat := 0;
  protected max_stimuli : nat := 0;

  -- e2s is used for all out-going stimuli (environment to system)
  protected e2s : map nat to nat := {|->};
  -- s2e is used for all received responses (system to environment)
  protected s2e : map nat to nat := {|->}

functions
  public checkResponseTimes: map nat to nat * map nat to nat * nat -> bool
  checkResponseTimes (pe2s, ps2e, plim) ==
    forall idx in set dom ps2e &
      ps2e(idx) - pe2s(idx) <= plim
  pre dom ps2e inter dom pe2s = dom ps2e

operations
  public EnvironmentTask: seq of char * EventDispatcher * nat ==> EnvironmentTask
  EnvironmentTask (tnm, disp, pno) ==
    ( max_stimuli := pno; AbstractTask(tnm, disp) );

  public getNum: () ==> nat
  getNum () == ( dcl res : nat := num; num := num + 1; return res );

  public setEvent: Event ==> ()
  setEvent (pe) == handleEvent(pe);

  public Run: () ==> ()
  Run () == is subclass responsibility;

  public logEnvToSys: nat ==> ()
  logEnvToSys (pev) == e2s := e2s munion {pev |-> time};

  public logSysToEnv: nat ==> ()
  logSysToEnv (pev) == s2e := s2e munion {pev |-> time};

  public getMinMaxAverage: () ==> nat * nat * real
  getMinMaxAverage () ==
    ( dcl min : [nat] := nil, max : [nat] := nil, diff : nat := 0;
      for all cnt in set dom s2e do
        let dt = s2e(cnt) - e2s(cnt) in
          ( if min = nil then min := dt
            else (if min > dt then min := dt);
            if max = nil then max := dt
            else (if max < dt then max := dt);
            diff := diff + dt );
      return mk_(min, max, diff / card dom s2e) )

sync
  -- getNum is mutually exclusive to ensure unique values
  mutex (getNum);

  -- getMinMaxAverage is blocked until all responses have been received
  per getMinMaxAverage => card dom s2e = max_stimuli

end EnvironmentTask
```

Figure 2.36: The class `EnvironmentTask`

move time forward in Timed VDM++, it is hard if not impossible to specify distributed systems in which truly concurrent behavior can occur. It is obvious that environment and system models should not influence each other, except for the exchange of stimuli and responses on their interface. But this is not the case here, since the threads used to model the environment are running in the same execution context as the model. The impact of the environment threads on the progress of time in the system model can be

```
class VolumeKnob is subclass of EnvironmentTask

operations
  public VolumeKnob: EventDispatcher * nat ==> VolumeKnob
  VolumeKnob (ped, pno) == EnvironmentTask("VolumeKnob", ped, pno);

  handleEvent: Event ==> ()
  handleEvent (pev) == duration (0) logSysToEnv(pev.getEvent())
  post checkResponseTimes(e2s,s2e,200);

  createSignal: () ==> ()
  createSignal () ==
    duration (0)
      if (card dom e2s < max_stimuli) then
        ( dcl num : nat := getNum();
          logEnvToSys(num);
          raiseInterrupt("HandleKeyPress", num) );

  public Run: () ==> ()
  Run () == start(self)

thread
  periodic (1000) (createSignal)

end VolumeKnob
```

Figure 2.37: The `VolumeKnob` class

reduced by using the `duration(0)` construct. But this restricts the expressiveness to pure periodic behavior because we need a pseudo-random delay, modeled using a duration, to specify jitter. Furthermore, there is no guarantee that the environment threads are executed on time, since the VDM++ simulator has limited options for priority based scheduling. For efficiency reasons, the simulator favors a dominant run-to-completion semantics, which may temporarily postpone higher priority tasks. This situation is reported to the user when it occurs, but it seriously complicates the interpretation of the simulation results. There is no guarantee that the input stimuli are representative and therefore the simulation results obtained cannot be trusted a priori.

Observations on the experiment

It is clear from our results that the Timed VDM++ notation from the VICE project has only limited usefulness for performance analysis of distributed real-time systems. This experiment confirms earlier findings reported in [98]. The notation is not sufficiently expressive, the supporting tools have significant limitations and moreover the interpretation of simulation data is cumbersome. VDMTOOLS produces a textual trace file which needs to be interpreted by hand. We developed some special purpose tool support, called “ShowVICE” to aid in this activity. Figure 2.39 presents the user-interface after parsing a trace file. A relevant subset of the trace can be selected for further investigation, which is visualized as a time annotated message sequence chart in Figure 2.40. Tools such as this are essential in order to raise the user productivity.

2.4 Comparing the models

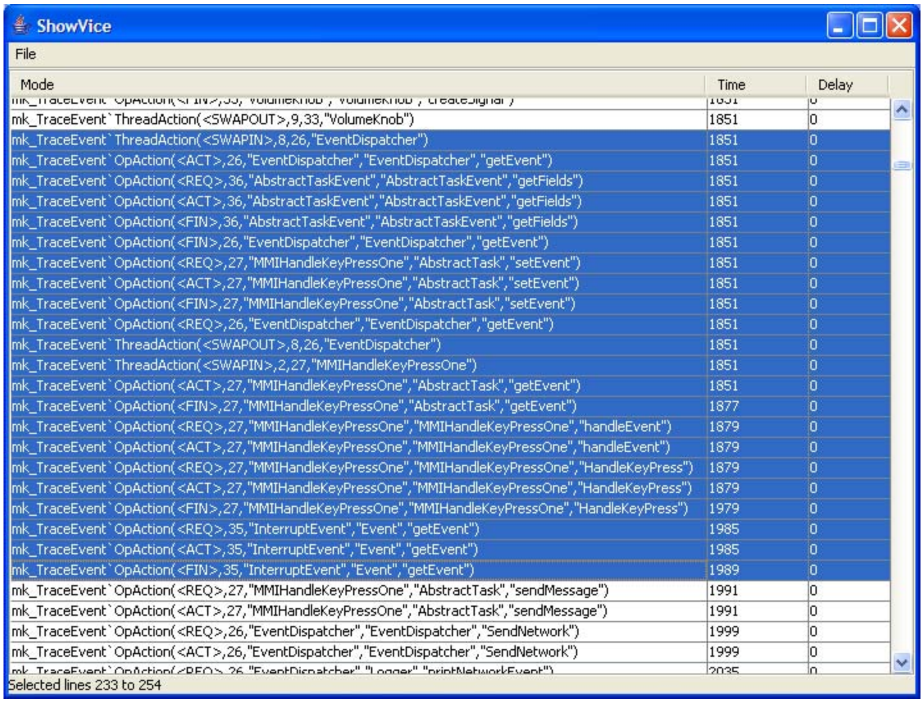
The case study was modeled using several techniques but the question is: How do the answers found during analysis relate? Consider for example the system-level response time for each of the applications in the case study. Based on the properties of the

```
class RadNavSys

types
  public perfdata = nat * nat * real

instance variables
  dispatch : EventDispatcher := new EventDispatcher();
  appTasks : set of BasicTask := {};
  envTasks : map seq of char to EnvironmentTask := {|->}

operations
  RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( addApplicationTask(new MMIHandleKeyPressOne(dispatch));
      addApplicationTask(new RadioAdjustVolume(dispatch));
      addApplicationTask(new MMIUpdateScreenVolume(dispatch));
      addApplicationTask(new RadioHandleTMC(dispatch));
      addApplicationTask(new NavigationDecodeTMC(dispatch));
      addApplicationTask(new MMIUpdateScreenTMC(dispatch));
      startlist(appTasks); start(dispatch) );

  addApplicationTask: BasicTask ==> ()
  addApplicationTask (pbt) ==
    ( appTasks := appTasks union {pbt};
      dispatch.Register(pbt) );

  addEnvironmentTask: EnvironmentTask ==> ()
  addEnvironmentTask (pet) ==
    ( envTasks := envTasks  munion {pet.getName() |-> pet};
      dispatch.Register(pet);
      pet.Run() );

  public Run: () ==> map seq of char to perfdata
  Run () ==
    ( addEnvironmentTask(new VolumeKnob(dispatch,10));
      addEnvironmentTask(new TransmitTMC(dispatch,10));
      return { name |-> envTasks(name).getMinMaxAverage() |
               name in set dom envTasks } )

end RadNavSys
```

Figure 2.38: The top-level specification RadNavSys

techniques themselves, we would expect to find results as depicted in Figure 2.41. MPA and SymTA/S provide hard but not necessarily tight bounds for these values. The approximations inherent to these methods may yield conservative results. Simulation based techniques, such as POOSL and VDM++, do not provide tight bounds because the model is not guaranteed to be fully covered, which may lead to results that are too optimistic. Timed automata can find hard and exact bounds within a user-defined accuracy, but only if the state space remains tractable. Tractability is, however, not guaranteed a priori. Although it is fairly easy to inspect timing aspects using timed automata, it is hard to analyze the load per resource. The only guarantee we have is that none of the resources is over-allocated. The other methods in comparison do provide detailed resource usage information.

Modeling comes at a price and there is a clear trade-off between abstraction and accuracy. We list a number of relevant questions. How much effort is required in order to get a result on time and within a certain error margin? Can this error margin be determined at all a priori? Both MPA and SymTA/S are methods that are clearly tailored to support early life-cycle decision making. Models are easy to construct and evaluate. Suitable levels of automation are available for design exploration and sensitivity analysis. As we learned from additional experiments at Océ, not reported in detail here, their



Mode	Time	Delay
mk_TraceEvent`OpAction(<FIN>,33,"VolumeKnob","VolumeKnob","createSignal")	1851	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,9,33,"VolumeKnob")	1851	0
mk_TraceEvent`ThreadAction(<SWAPIN>,8,26,"EventDispatcher")	1851	0
mk_TraceEvent`OpAction(<ACT>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<ACT>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<FIN>,36,"AbstractTaskEvent","AbstractTaskEvent","getFields")	1851	0
mk_TraceEvent`OpAction(<FIN>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","AbstractTask","setEvent")	1851	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","EventDispatcher","getEvent")	1851	0
mk_TraceEvent`ThreadAction(<SWAPOUT>,8,26,"EventDispatcher")	1851	0
mk_TraceEvent`ThreadAction(<SWAPIN>,2,27,"MMIHandleKeyPressOne")	1851	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","getEvent")	1851	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","AbstractTask","getEvent")	1877	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","handleEvent")	1879	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","handleEvent")	1879	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1879	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1879	0
mk_TraceEvent`OpAction(<FIN>,27,"MMIHandleKeyPressOne","MMIHandleKeyPressOne","HandleKeyPress")	1979	0
mk_TraceEvent`OpAction(<REQ>,35,"InterruptEvent","Event","getEvent")	1985	0
mk_TraceEvent`OpAction(<ACT>,35,"InterruptEvent","Event","getEvent")	1985	0
mk_TraceEvent`OpAction(<FIN>,35,"InterruptEvent","Event","getEvent")	1989	0
mk_TraceEvent`OpAction(<REQ>,27,"MMIHandleKeyPressOne","AbstractTask","sendMessage")	1991	0
mk_TraceEvent`OpAction(<ACT>,27,"MMIHandleKeyPressOne","AbstractTask","sendMessage")	1991	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","EventDispatcher","SendNetwork")	1999	0
mk_TraceEvent`OpAction(<ACT>,26,"EventDispatcher","EventDispatcher","SendNetwork")	1999	0
mk_TraceEvent`OpAction(<REQ>,26,"EventDispatcher","Logger","printNetworkEvent")	2035	0

Selected lines 233 to 254

Figure 2.39: The main user-interface of "ShowVice", showing a parsed log file.

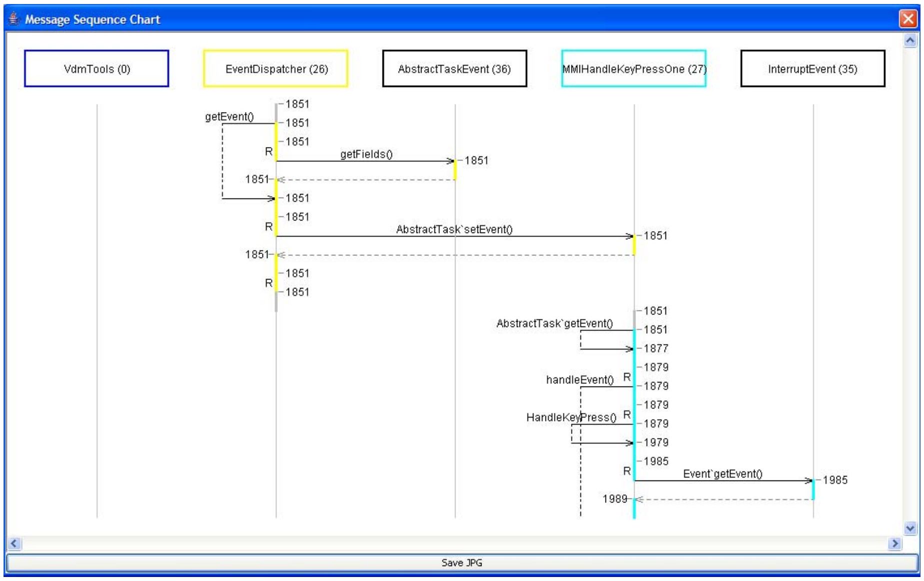


Figure 2.40: Time annotated message sequence diagram of the trace subset.

weakness is lack of support for time dependent input stimuli, which typically leads to analysis results that have little value in practice.

Building timed automata, VDM++ and POOSL models involves significantly more

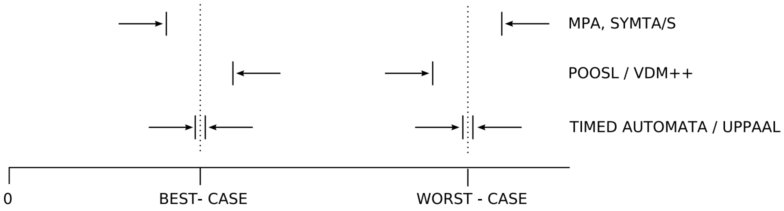


Figure 2.41: A general comparison of results found

work than MPA and SymTA/S, although modeling templates were developed for POOSL to overcome this problem in part, see [35, 37]. Furthermore, analysis takes more time and is in general not guaranteed to lead to deterministic results. In the case of timed automata, expert knowledge may be required to modify the model such that tractability is achieved. However, the models built with these techniques can be described in greater detail if needed, for example to deal with time dependent input stimuli, but obviously at the cost of model analysis efficiency. POOSL can provide feedback on the nominal (average) system behavior while timed automata, MPA and SymTA/S can only investigate the performance bounds. Currently, Timed VDM++ can only realistically analyze the timing behavior of single CPU multi-threaded systems.

Table 2.3 provides an overview of the analysis results for the worst-case response time of all applications deployed on architecture (a) in Figure 2.6. The abbreviations K2V and A2V mentioned in the table refer to the two system-level performance requirements of the *"Change Volume"* scenario shown in Figure 2.3. The models were evaluated against pure periodic environment stimuli with an unknown offset between the two event streams. With the exception of VDM++, this situation can be suitably analyzed by all techniques and therefore enables a fair comparison of the results.

Tool *Requirement*	*Uppaal*	*POOSL*	*SymTA/S*	*MPA*
HandleTMC (+ ChangeVolume)	345.27	366.94	382.09	390.09
HandleTMC (+ AddressLookup)	239.08	234.26	253.30	265.85
A2V ChangeVolume (+ HandleTMC)	27.72	27.71	27.72	28.16
K2V ChangeVolume (+ HandleTMC)	41.80	41.78	41.80	42.24
AddressLookup (+ HandleTMC)	79.08	78.90	79.08	84.07

Table 2.3: Worst-case response time results (in ms)

The parameters in the case study were chosen such that the *ChangeVolume* application always has the highest priority on all resources, as can be seen in Figure 2.3. Because fixed priority pre-emptive scheduling is used on all resources, this application gets access to the resource as soon as it is required. This is why the A2V / K2V rows in the table contains almost identical results for each method. But even this seemingly trivial exercise leads to the insight that the "Change Volume" scenario is self-interrupting. The execution of `UpdateVolume` due to the *n*-th event is interrupted by the execution of `VolumeKeyPress` of the *n+1*-th event, because this task has a higher priority. In fact, we can manually calculate this result, as shown in Equations 2.20 and 2.21, and we find exactly the same values, which demonstrates the validity of the results found by the tools.

$$\textsc{a2v} \quad = \quad \frac{4 \cdot 8}{72 \cdot 10^3} + \frac{5 \cdot 10^5}{22 \cdot 10^6} + \frac{1 \cdot 10^5}{22 \cdot 10^6} = 27.717 \;\; \mathtt{ms} \tag{2.20}$$

$$\textsc{k2v} \quad = \quad \frac{1 \cdot 10^5}{22 \cdot 10^6} + \frac{4 \cdot 8}{72 \cdot 10^3} + \frac{1 \cdot 10^5}{11 \cdot 10^6} + \textsc{a2v} \; = 41.797 \;\; \mathtt{ms} \tag{2.21}$$

When we compare the first two columns in Table 2.3, we see that the POOSL results are indeed slightly more optimistic than the values found by UPPAAL. This is due to the fact that there are infinitely many possible values for the offset. In this particular case, UPPAAL was able to handle this property symbolically. It is also clear that both SymTA/S (third column) and MPA (fourth column) are slightly more conservative than UPPAAL, as expected.

Comparing the analysis results showed us that each technique introduces hidden assumptions and approximations of its own. To our surprise, the initial results did *not* conform to the expectation illustrated in Figure 2.41. A discussion was started on the meaning of the results, to gain more insight. Apart from a better problem understanding, this also included improving the case study specification, discovering subtle modeling errors in almost all models produced and even bugs and hidden limitations in the (prototype) analysis tools. Table 2.3 is in fact the result of *several iterations* due to this debate.

2.5 Discussion and conclusions

Our performance models of software applications are based on an estimate of the number of instructions that the resource shall execute. This value is obviously a rough approximation that in general might not be accurate. For example, the peak capacity of modern CPU architectures can only be achieved for specific types of algorithms; digital signal processors are optimized for repeated multiply-addition operations that occur frequently in fast Fourier transformations (FFT) for example. Moving an application from one resource to another might not only rely on the number of instructions to execute. Furthermore, issues such as caching, have not been considered here either. But despite these approximation, it suffices for our purpose since we are interested in high-level analysis of early design models. The feedback gained from the analysis of these models provides valuable insights that can be used to guide the design process, for example to reduce development risk by early detection of potential performance bottlenecks. If we want to make more accurate predictions, it is always possible to lower the abstraction level by adding more detail, or to benchmark the operation on the target architecture and use the measured value instead of the estimate.

More results based on the in-car radio navigation case study are available, for example using the DeSiX methodology [12] and the work of Florentz [33]. But these results are on par with those reported here. Besides, our aim was not to be exhaustive in our survey. Many other existing techniques have not been considered either and only a single case study was used for comparison. Neither did we attempt to determine what the “best” method is, since this is context dependent. Many qualities influence failure or success.

Five state-of-the-art techniques for performance analysis have been investigated and compared. We showed how these techniques relate by means of an experiment. To the best of our knowledge, it is the first time that such a quantitative comparison was performed on a single case study at this scale, although there exist several surveys that attempt to make a qualitative comparison. The exception being the recent work of Perathoner et al reported in [79, 81]. They investigated the influence of system abstractions on the performance analysis of distributed real-time systems, applying several different techniques to a set of well-known benchmark problems and concluded that the accuracy of the various approaches may differ significantly and that none of the methods performs best in all cases.

In conclusion, we do argue that in-depth knowledge of the application domain, the method used and awareness of the limitations of the tools are *equally important* critical success factors. This seems obvious, but in practice it is hardly ever the case that all three aspects are covered to the same extent. The small experiment has clearly demonstrated that it does pay off to use more than one method. Weaknesses in the models will be exposed by comparing the models and the analysis results. The models, the tools and the analysis results should not be taken for granted.

Chapter 3

Extending VDM++ for Distributed Real-Time Systems

3.1 Introduction

The complexity of embedded systems is rapidly increasing; they are becoming distributed almost by default, for example due to the System-on-Chip design philosophy which is often used nowadays. Safety-critical applications have traditionally been federated, meaning that each “function” has its own CPU with minimal interconnections to other functions in the system. This approach is expensive and for some application areas, such as the automobile industry, it is no longer economically viable to do so. The current trend is rather to combine functions together on the same processing unit and then distribute their operation between a number of networked fault-tolerant processors in order to reduce cost. It is not hard to imagine that finding the “right” deployment of functionality over such a distributed architecture, that meets all the imposed system-level requirements, is quite a challenging problem.

It is natural to advocate the use of formal techniques in this application area in order to cope with this complexity and indeed a large body of knowledge exists on their use. Most formal techniques however, are not able to deal with the combination of complex behavior, timing, concurrency and in particular distribution in a flexible and intuitive way. Tool support often does not scale very well to the size of problems faced by industry. System development lead times remain substantial, even if formal methods can be usefully applied.

The Vienna Development Method (VDM) has been used in several large-scale industrial projects [94, 57, 30, 65]. Their success was very much due to the solid formal basis of the notation and the availability of robust and commercial grade tools. However, not much is known about the application of VDM in the area of distributed real-time embedded systems. In earlier work [98], we reported that it is very hard to describe such systems in VDM and this was confirmed by our findings in the previous chapter. The language is not sufficiently expressive and important tool features are missing to analyze such models.

The aim of this chapter is to make VDM++ better suited for describing distributed embedded real-time systems and to enable the design space exploration as mentioned before. In Section 2.3.5, an overview of the notation and the existing timed extension was presented. The limitations experienced in our earlier work are summarized in Section 3.2 and we introduce the main proposed adaptations in Section 3.3 : the addition of deployment and asynchronous communication. The in-car radio navigation case study is revisited in Section 3.4 that demonstrates the impact of the proposed changes. In Section 3.5, we define an abstract formal semantics of the extended language and discuss how the semantics has been validated. Finally, in Section 3.6 we present related work and we discuss the results achieved.

3.2 The limitations of timed VDM++

In Chapter 2 and in previous work [98], we assessed the suitability of timed VDM++ for distributed real-time embedded systems. We list the most important problems here.

1. Operations in VDM++ are synchronous; calls are either blocked on a permission predicate (guard) or executed in the context of the thread of control of the caller. The caller has to wait until the operation is completed before it can resume. This is very cumbersome when embedded systems are modeled. These systems are typically reactive by nature and asynchronous. An event loop can be specified to describe this, but the complexity of the model is increased and analysis of the model becomes harder.

2. Timed VDM++ supports a uni-processor multi-threading model of computation which means that at most one thread can claim the processor and only this active thread can push time in the model forward. This is insufficient for describing embedded systems because 1) they are often implemented on a distributed architecture and 2) these systems need to be described in combination with their environment. The subsystems and the environment are independent and therefore need their own notion of time which requires a multi-processor multi-threading model of computation.

3. The duration statement in timed VDM++ denotes a time penalty that is independent of the resource that executes the statement. When deployment is considered, it is essential to also be able to express time penalties that are relative to the capacity of the computation resource. Furthermore, there should be an additional time penalty that reflects the message handling between two computation resources whenever a remote operation call is performed.

4. Timed VDM++ allows for the specification of periodic threads. But two restrictions hamper the effective use of this language construct. First of all, only strictly periodic threads can be specified and in practice a more flexible solution is required, for example to specify jitter. And secondly, it is assumed that the periodic thread has run to completion before the next period is due. In other words, the thread creation and activation are strongly coupled in Timed VDM++. Hence, it is not possible to simulate a burst of periodic thread releases or periodic threads that may have overlapping release intervals.

3.3 Proposed changes

Our aim is to minimize the impact on the existing language as much as possible. Ideally, we want to remain backwards compatible in order to reuse existing models and tools. Therefore, we have not considered to merge VDM++ with other techniques. Informally, we propose the following changes:

1. The semantics of timed VDM++ is based on the assumption that at most one thread can push time forward in the model. We propose a richer semantics in which this limitation is removed. Any thread that is running on a computation resource or any message that is in transit on a communication resource can cause time to elapse. Models that contain only one computation resource are compatible to models in timed VDM++.

2. The suggestion is to introduce the *async* keyword in the signature of an operation to denote that an operation is asynchronous. The caller shall no longer be blocked, it can immediately resume its own thread of control after the call is initiated. A new thread is created and started immediately to execute the body of the asynchronous operation.

3. A collection of special predefined classes, *BUS* and *CPU*, are made available to the specifier to construct the distributed architecture in his model. The *system* class is used to contain such an architecture model. User-defined classes can be instantiated and deployed on a specific *CPU* in the model. The communication topology between the computation resources in the model can be described using the *BUS* class.

4. The *duration* statement is kept intact to specify time delays that are independent of the system architecture. In addition, we introduce the *cycles* statement, with a similar concrete syntax, to denote a time delay that is relative to the capacity of the resource. The time delay incurred by the message transfer over the *BUS* can be made dependent of the size of the message being transferred, which is a function of the parameter values passed to the operation call.

5. We adopt the more general notation for specifying periodic threads as previously introduced in Section 2.2.2 using the (p, j, d, o)-tuple, with concrete syntax *periodic (p,j,d,o) (Op)*, for enhanced modeling flexibility. Furthermore, we decouple the task release moment from the task activation in the operational semantics to allow for potentially overlapping task release intervals.

We will demonstrate the impact of these changes in Section 3.4 using a small case study and in Section 3.5 we present the semantics of the main extensions.

3.4 Modeling the in-car radio navigation system

In Chapter 2 we have studied the design of an in-car radio navigation system. Such an infotainment system typically executes several concurrent software applications that share a common, and often distributed, hardware platform. Each application has individual requirements that need to be met and the question is whether all requirements can be satisfied when a particular architecture is chosen. In this section, we present a VDM++ model of the distributed in-car radio navigation system using the suggested language improvements. We have focused on modeling the non-functional performance aspects because these will highlight the impact of the language changes most prominently. This aims to demonstrate that it is easy to describe distributed architectures and the associated deployment of functionality onto it. The model presented here reflects one of the proposals that was considered during the design, consisting of three processing units connected through an internal communication bus. An overview of the case study is presented in Figure 3.1.

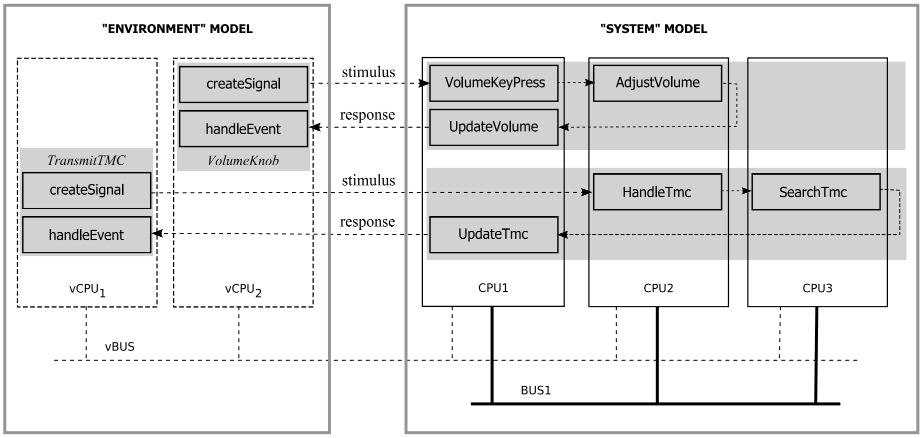


Figure 3.1: Informal overview of the case study

Two application scenarios are running on the system: "*Change Volume*" and "*TMC Message Handling*". Each application consists of three individual tasks. The "*Change Volume*" application, represented by the top right gray box, controls the volume of the radio. The task VolumeKeyPress takes care of all user interface input handling, AdjustVolume modifies the volume accordingly and UpdateVolume displays the new volume setting on the screen. The "*TMC Message Handling*" application, indicated by the bottom right gray box in Figure 3.1, handles all Traffic Message Channel (TMC) messages. TMC messages arrive at the HandleTmc task where they are checked and forwarded to the SearchTmc task to be translated into human readable text which is displayed on the screen by the UpdateTmc task.

Two additional applications represent the environment of the system: *VolumeKnob* and *TransmitTMC*. The former is used to simulate the behavior of a user turning the volume knob at a certain rate and the latter is used to simulate the behavior of a radio station that transmits TMC messages. Both applications inject stimuli into the system, using the task createSignal and observe the system response using the task handleEvent.

In the remainder of this section, we will present how applications and tasks from the informal case study description relate to classes, operations and threads in VDM++ using the proposed language extensions. Furthermore, we will show how distributed architectures are described and how objects are deployed. We present the environment model in more detail in Section 3.4.1 and the system model in Section 3.4.2.

3.4.1 The environment model

There are two environment applications in our case study. Each application is represented by a class and the tasks are represented by asynchronous operations in that class. An instance of the class is automatically deployed on an implicit computation resource, denoted by the dashed boxes in Figure 3.1. Environment applications operate in parallel to the system and independent of each other. Execution of an environment application does *not* affect the notion of time in other environment or system applications. Environment applications communicate with the system applications over an implicit communication resource, as shown by vBUS in Figure 3.1. This so-called virtual bus connects all computation resources in the model. Communication over the virtual bus is instantaneous, no time is lost to propagate messages over this implicit communication resource.

Typical system-level temporal and timing properties can be specified over the stimulus / response interface between the environment and the system model. Informal examples of these requirements are: *"The order of the VolumeKnob stimuli is preserved by the output response sequence of the system."* and *"The maximum allowed response time shall be less than 1000 time units for each individual stimulus arriving at HandleTmc."*. These requirements can be modeled using standard VDM++ constructs. For example, the latter end-to-end response time requirement is formulated as a post condition to the operation `handleEvent` in the `TransmitTMC` class, which is presented in Figure 3.2.

```
class TransmitTMC

instance variables
  static private id : nat := 0;
  protected e2s : map nat to nat := {|->};
  protected s2e : map nat to nat := {|->}

operations
  getNum: () ==> nat
  getNum () == ( dcl res : nat := id; id := id + 1; return res );

  async public handleEvent: nat ==> ()
  handleEvent (pev) == s2e := s2e munion {pev |-> time}
  post forall idx in set dom s2e & s2e(idx) - e2s(idx) <= 1000;

  async createSignal: () ==> ()
  createSignal () ==
    ( dcl num : nat := getNum();
      e2s := e2s munion {num |-> time};
      RadNavSys`radio.HandleTmc(num) )

thread
  periodic (3000, 6000, 1000, 0) (createSignal)

sync
  mutex(getNum)

end TransmitTMC
```

Figure 3.2: The *TransmitTMC* class

Two instance variables are maintained to log the stimuli (*e2s*) and the responses (*s2e*). These variables are mappings from a unique natural number provided by the operation `getNum`, to identify each stimulus, to another natural number that represents the time at which the event was recorded. Uniqueness is guaranteed by the `sync` predicate which specifies that calling the operation `getNum` is mutual exclusive. At most one invocation of this operation can be active at any point in time. The `time` keyword in VDM++ provides access to the "wall clock" of the interpreter whenever the model is executed. The periodic thread `createSignal` is executed every 3000 time units, with a jitter of 6000 time units and a minimal event separation time of 1000 time units. This operation injects TMC events into the system by calling the asynchronous operation HandleTmc of an instance of the *Radio* class shown in Figure 3.3. The operation handleEvent is called by the system at the end of the UpdateTMC operation (not shown here), indicating that the event was completely processed by the "*TMC Message Handling*" application. The worst-case response time requirement is encoded as a post condition to the handleEvent operation. The post conditions are checked at runtime when the model is simulated. The interpreter will stop automatically whenever an integrity constraint evaluates to false and the state of the model can be inspected to determine the cause of the problem. Other timeliness requirements can be specified in a similar way.

3.4.2 The system model

There are two independent applications that consist of three tasks each in the system model of our example. Tasks can either be triggered by external stimuli directly or by receiving messages from other tasks indirectly. A task can also actively acquire or provide information by periodically checking for available data on an input source or delivering new data to an output source. All three notions of task activation are supported by our approach. Note that task activation by external stimuli can be used to model *interrupt handling*. The HandleKeyPress and HandleTmc tasks in Figure 3.1 belong to this category. All other tasks in our system model are message triggered, because operation invocation implies message exchange over the communication resource BUS1, shown in Figure 3.1. Note that we already demonstrated the use of periodic task activation in the environment model (createSignal).

```
class Radio

operations
  async public AdjustVolume: nat ==> ()
  AdjustVolume (pno) ==
    ( duration (150) skip;
      RadNavSys`mmi.UpdateVolume(pno) );

  async public HandleTmc: nat ==> ()
  HandleTmc (pno) ==
    ( cycles (1E5) skip;
      RadNavSys`navigation.SearchTmc(pno) )

end Radio
```

Figure 3.3: The *Radio* class

Application tasks are modeled by asynchronous operations in our VDM++ extension. Figure 3.3 presents the definition of AdjustVolume and HandleTmc tasks from Figure 3.1, which are grouped together in the *Radio* class for convenience. We use the skip statement for illustration purposes here. It can be replaced with an arbitrary complex statement to describe the actual system function that is performed, for example changing the amplifier volume set point. Note that the operation AdjustVolume uses the duration statement to denote that a certain amount of time expires independent of the resource on which it is deployed. This duration statement states that changing the volume set point always takes 150 time units. For illustration purposes, the operation HandleTmc uses the cycles statement instead, to denote that a certain amount of time expires *relatively* to the capacity of the computation resource on which it is deployed. If this operation is deployed on a resource that can deliver 1000 cycles per unit of time, then the delay (duration) would be 100 time units. A suitable unit of time can be selected by the modeler.

A special built-in class called *CPU* is provided to create computation resources in the system model. Each computation resource is characterized by its processing capacity, specified by the number of available cycles per unit of time, the scheduling policy that is used to determine the task execution order and a factor to denote the overhead incurred per task switch. For this case study, fixed priority preemptive scheduling with zero overhead is used, although our approach is not restricted to any scheduling policy in particular.

```
system RadNavSys

instance variables
  -- create the application tasks
  static public mmi := new MMI();
  static public radio := new Radio();
  static public navigation := new Navigation();

  -- create CPU (policy, capacity, task switch overhead)
  CPU1 : CPU := new CPU(<FP>, 22E6, 0);
  CPU2 : CPU := new CPU(<FP>, 11E6, 0);
  CPU3 : CPU := new CPU(<FP>, 113E6, 0);

  -- create BUS (policy, capacity, message overhead, topology)
  BUS1 : BUS := new BUS(<FCFS>, 72E3, 0, {CPU1, CPU2, CPU3})

operations
  -- the constructor of the system model
  public RadNavSys: () ==> RadNavSys
  RadNavSys () ==
    ( -- deploy MMI on CPU1
      CPU1.deploy(mmi);
      -- deploy Radio on CPU2
      CPU2.deploy(radio);
      -- deploy Navigation on CPU3
      CPU3.deploy(navigation) )

end RadNavSys
```

Figure 3.4: The top-level system model for the case study

A special built-in class *BUS* is provided to create communication resources in the system model. A communication resource is characterized by its throughput, specified by the number of messages that can be handled per unit of time, the scheduling policy that is used to determine the order of the messages being exchanged and a factor to denote the protocol overhead per message. The granularity of a message can be determined by the user. For example, it can represent a single byte or a complete Ethernet frame, whatever is most appropriate for the problem under study. Here, we use “first come, first served” scheduling with zero overhead, but again the approach is not restricted to any scheduling policy in particular. An overview of the top-level VDM++ system model is presented in Figure 3.4.

3.5 Abstract Operational Semantics

In this section we formalize the semantics of the proposed changes to VDM++, as described in Section 3.3. To highlight the main changes and modifications, an abstract basic language which includes the new constructs is defined in Section 3.5.1. We describe the intended meaning and discuss the most important issues that had to be addressed when formalizing this. In Section 3.5.2, a formal operational semantics is defined. Validation of this semantics is discussed in Section 3.5.3.

3.5.1 Syntax and informal semantics

To be able to highlight the formal semantics of the extensions proposed in the previous section, we define a syntax which abstracts away from many aspects and constructs in VDM++. For example, our syntax does not contain class definitions with explicit definitions of synchronous and asynchronous operations. Instead, we assume given a set *Operations* of operations, with typical element *op* and predicate *syn?*$($*op*$)$ which is true if and only if the operation is synchronous. We also assume that the body of each operation is compiled into a sequence of basic instructions. We abstract from most local, atomic instructions and consider only the **skip** instruction here.

Our time domain is the nonnegative real numbers; *Time* $= \{t \in \mathbb{R} \mid t \geq 0\}$. We use d to denote a time value and **duration** (d) as an abbreviation of **duration**(d) **skip**. Assume that, for an instruction sequence *IS*, the statement **duration**(d) *IS* is translated into *IS* ˆ **duration**(d), where internal durations inside *IS* have been removed and the "ˆ" operator concatenates the duration instruction to the end of a sequence. The concatenation operation is also used to concatenate sequences and to add an instruction to the front of the sequence. Functions *head* and *tail* yield the first element and the rest of the sequence, respectively, and $\langle\rangle$ denotes the empty sequence. The **cycles** statement has been omitted here since it is equivalent to the duration statement, given a certain deployment. The **periodic** statement has been generalized to allow the periodic execution of an instruction sequence instead of an operation call only. Let *ObjectId* be the set of object identities, with typical element *oid*. The syntax of the instructions is given in Table 3.1.

Instr.	I ::=	**skip** $\mid$ **call**$($*oid*$,$ *op*$)$ $\mid$ **duration**(d) $\mid$ **periodic**(d) *IS*
Instr. Seq.	*IS* ::=	$\langle\rangle$ $\mid$ Iˆ*IS*

Table 3.1: Abstract syntax of basic instructions

These basic instructions have the following informal meaning:

- **skip** represents a local statement which does not consume any time.
- **call**$($*oid*$,$ *op*$)$ denotes a call to an operation *op* of object *oid*. Depending on the *syn?* predicate, the operation can be synchronous (i.e., the caller has to wait until the execution of the operation body has terminated) or asynchronous (the caller may continue with the next instruction and the operation body is executed independently). There are no restrictions on re-entrance here, but in general this can be restricted by permission predicates as discussed in Section 2.3.5. These are not considered here and also parameters are ignored.
- **duration**(d) represents a time progress of d time units. When d time units have elapsed the next statement can be executed. As shown in Section 3.4.2, $cycles(d)$ can be expressed as a duration statement.
- **periodic**(d) *IS* leads to the execution of instruction sequence *IS* each period of d time units. We have generalized the (p, j, d, o)-tuple into a single parameter

here, since there exists a simple deterministic algorithm that computes the delay to the next activation based on those four parameters.

The distributed architecture of an embedded control program can be represented by so-called nodes. Let *Node* be the set of node identities. Nodes are used to represent computation resources such as processors. On each node a number of concurrent threads are executed in an interleaved way. The function *node* : *Thread* → *Node* denotes on which node each thread is executing. Each thread executes a sequential program, that is, a statement (an instruction sequence) expressed in the language of Table 3.1. Furthermore, assume given a set of links, defined as a relation between nodes, i.e., $\mathit{Link} = \mathit{Node} \times \mathit{Node}$, to express that messages can be transmitted from one node to another via a link. In the semantics described here, we assume for simplicity that a direct link exists between each pair of communicating nodes. Note that the built-in classes *CPU* and *BUS*, as used in the radio navigation case study, are concrete examples of a node and a link.

3.5.2 Formal Operational Semantics

The formalization of the precise meaning of the language described above raises a number of questions that have to answered and on which a decision has to be taken. We list the main points:

- How to deal with the combination of synchronous and asynchronous operations, e.g. does one has priority over the other, how are incoming call request recorded, is there a queue at the level of the node or for each object separately? We decided for an equal treatment of both concepts; each object has a single FIFO queue which contains both types of incoming call requests.

- How to deal with synchronous operation calls; are the call and its acceptance combined into a single step and does it make a difference if caller and callee are on different nodes? In our semantics, we distinguish between a call within a single node and a call to an operation of an object on another node.

  For a call between different nodes, a call message is transferred via a link to the queue of the callee; when this call request is dequeued at the callee, the operation body is executed in a separate thread and, upon completion, a return message is transmitted via the link to the node of the caller.

  For a call within a single node, we have made the choice to avoid a context switch and execute the operation body directly in the thread of the caller. Instead, we could have placed the call request in the queue of the callee.

- Similar questions hold for asynchronous operations. On a single node, the call request is put in the queue of the callee, whereas for different nodes the call is transferred via a link. However, no return message is needed and the caller may continue immediately after issuing the call.

- How are messages between nodes transferred by the links? In principle, many different communication mechanisms could be modeled. As a simple example, we model a link by a set of messages which include a lower and an upper bound on message delivery. For a link l, let $\delta_{min}(l)$ and $\delta_{max}(l)$ be the minimum and maximum transmission time. It is easy to extend this and make the transmission time dependent of, e.g. message size and link traffic.

- How to deal with time, how is the progress of time modeled? In our semantics, there is only one global step which models progress of time on all nodes. All other steps do not change time; all assumptions on the duration of statements, context switches and communications have to be modeled explicitly by means of duration statements.

- What is the precise meaning of **periodic**(d) *IS* if the execution of *IS* takes more than d time units? We decided that after each d time units a new thread is started to ensure that every d time units the *IS* sequence can be executed. Of course, this might potentially lead to resource problems for particular applications, but this will become explicit during analysis.

The operational semantics presented in this section defines the execution of the language given in Table 3.1 formally. To focus on the essential aspects, we assume that the set of objects is fixed and need not be recorded in the configuration. However, object creation can be added easily, see e.g. [55]. Threads can be created dynamically, e.g., to deal with asynchronous operation calls. Let *Thread* be a set of thread identities; each thread i is related to one object, denoted by o_i. This also leads to the deployment of threads using the $node$ function defined earlier: $\mathit{node}(i) = \mathit{node}(o_i)$. Finally, we extend the set of instructions *Instruction* with an auxiliary statement **return**(i). This statement will be added during the executing at the end of the instruction sequence of a synchronous operation which has been called by thread i.

To capture the state of affairs at a certain point during the execution, we introduce a *configuration* (Definition 3.5.1). Next we define the possible steps from one configuration to another, denoted by $C \longrightarrow C'$ where C and C' are configurations (Definition 3.5.3). This finally leads to a set of runs of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$ (Definition 3.5.9).

Definition 3.5.1 (Configuration) A *configuration* C contains the following fields:

- *instr* : *Thread* → *Instr. Seq.* which is a function which assigns a sequence of instructions, as defined in Table 3.1, to each thread.

- *curthr* : *Node* → *Thread* yields for each node the currently executing thread.

- *status* : *Thread* → $\{\mathit{dormant}, \mathit{alive}, \mathit{waiting}\}$ to denote the status of threads.

- q : *ObjectId* → $\mathit{queue}[\mathit{Thread} \times \mathit{Operations}]$ records for each object a FIFO queue of incoming calls, together with the calling thread (needed for synchronous operations only).

- *linkset* : *Link* → $\mathit{set}[\mathit{Message} \times \mathit{Time} \times \mathit{Time}]$ records the set of the incoming messages for each link, together with lower and upper bound on delivery. A message may denote a call of an operation (including calling thread and called object) or a return to a thread.

- *now* : *Time* to denote the current time.

For a FIFO queue, functions *head* and *tail* yield the head of the queue and the rest, respectively; *insert* is used to insert an element and $\langle\rangle$ denotes the empty queue. For sets we use *add* and *remove* to insert and remove elements. For a configuration C, we use:

- $C(f)$ to obtain its field f. For example, $C(\mathit{instr})(i)$ yields the instruction sequence of thread i in Configuration C.
- $\mathit{exec}(C, i)$ as an abbreviation for $C(\mathit{curthr})(\mathit{node}(i)) = i$, which expresses that thread i is executing on its node.
- $\mathit{fresh}(C, \mathit{oid})$ to yield a fresh, not yet used, thread identity (so with status *dormant*) corresponding to object *oid*.

To express modifications of a configuration, we define the notion of a variant.

Definition 3.5.2 (Variant) The *variant* of a configuration C with respect to a field f and value v, denoted by $C[\ f \mapsto v\ ]$, is defined as

$$(C[\ f \mapsto v\ ])(f') = \begin{cases} v & \text{if } f' = f \\ C(f') & \text{if } f' \neq f \end{cases}$$

Similarly for parts of the fields, such as $\mathit{instr}(i)$.

Steps have been grouped into several definitions, leading to the following overall definition of a step.

Definition 3.5.3 (Step) $C \longrightarrow C'$ is a *step* if and only if it corresponds to the execution of an instruction (Definition 3.5.4), a time step (Definition 3.5.5), a context switch (Definition 3.5.6), the delivery of a message by a link (Definition 3.5.7), or the processing of a message from a queue (Definition 3.5.8).

Definition 3.5.4 (Execute Instruction) A step $C \longrightarrow C'$ corresponds to the execution of an instruction if and only if there exists a thread i such that $\mathit{exec}(C, i)$ and $head(C(\mathit{instr})(i))$ is one of the following (underlined) instructions:

- $\underline{\textbf{skip}}$:
  Then the new configuration equals the old one, except that the skip instruction is removed from the instruction sequence of i, that is,
  $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i))\ ]$
- $\underline{\textbf{call}(\mathit{oid}, \mathit{op})}$:
  Let *IS* be the explicit definition of operation *op* of object *oid*. We consider four cases:
  - Caller and callee are on the same node, i.e. $node(i) = node(\mathit{oid})$.
    - If $\mathit{syn?}(\mathit{op})$ then *IS* is executed directly in the thread of the caller:
      $C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS}\hat{\ }\mathit{tail}(C(\mathit{instr})(i))\ ]$
    - If not $\mathit{syn?}(\mathit{op})$, we add the pair (i, op) to the queue of *oid*:
      $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),$
      $\qquad q(\mathit{oid}) \mapsto \mathit{insert}((i, \mathit{op}), C(q)(\mathit{oid}))\ ]$
  - Caller and callee are on different nodes, i.e. $node(i) \neq node(\mathit{oid})$. Suppose link l connects the nodes, then the call is transmitted via l, so $m = (\textbf{call}(i, \mathit{oid}, \mathit{op}), C(\mathit{now}) + \delta_{min}(l), C(\mathit{now}) + \delta_{max}(l))$ is added to the linkset of l.
    - If $\mathit{syn?}(\mathit{op})$, thread i becomes *waiting*:
      $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),\ \mathit{status}(i) \mapsto \mathit{waiting},$
      $\qquad \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

  - ∗ Similarly for asynchronous operations, when not *syn?*(op), except that then the status of i is not changed:
    $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),$
    $\quad \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

- **duration**(d):
  A duration statement leads to global progress of time. This time step will be defined in Definition 3.5.5.

- **periodic**(d) *IS*:
  In this case, *IS* is added to the instruction sequence of thread i and a new thread $j = \mathit{fresh}(C, o_i)$ is started which repeats the periodic instruction after a duration of d time units, i.e.
  $C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS},\ \mathit{instr}(j) \mapsto \textbf{duration}(d)\,\hat{}\,\textbf{periodic}(d)\ \mathit{IS},$
  $\quad \mathit{status}(j) \mapsto \mathit{alive}\ ]$

- **return**(j):
  In this case, we have $node(i) \neq node(j)$ and let l be the link which connects these nodes. Then $m = (\textbf{return}(j), C(\mathit{now}) + \delta_{min}(l), C(\mathit{now}) + \delta_{max}(l))$ is transmitted via l, i.e.
  $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),\ \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

Definition 3.5.5 (Time Step) A step $C \longrightarrow C'$ is called a *time step* only if all current threads are ready to execute a duration instruction or have terminated. More formally, for all i with $\mathit{exec}(C, i)$, $C(\mathit{instr})(i)$ is $\langle\rangle$ or of the form $\textbf{duration}(d)\,\hat{}\,\mathit{IS}$. Time may progress with t time units if

- t is smaller or equal than all durations that are at the head of an instruction sequence of an executing thread, and

- $C(\mathit{now}) + t$ is smaller or equal than all upper bounds of messages in link sets.

Define the maximal length of the time step t_m as the largest t satisfying these conditions. Durations in instruction sequences are modified by the following definition which yields a new function from threads to instruction sequences, for any thread i,
$\mathit{NewDuration}(C, t_m)(i) =$

$$\begin{cases} \textbf{duration}(d_i - t_m)\,\hat{}\,\mathit{tail}(C(\mathit{instr})(i)) & \text{if } \mathit{head}(C(\mathit{instr})(i)) = \textbf{duration}(d_i) \\ C(\mathit{instr})(i) & \text{otherwise} \end{cases}$$

Let $C' = C[\ \mathit{instr} \mapsto \mathit{NewDuration}(C, t_m)\ ]$

Definition 3.5.6 (Context Switch) A step $C \longrightarrow C'$ corresponds to a context switch if and only if there exists a thread i which is alive, not running, and has a non-empty program which does not start with a duration, i.e., $\neg\mathit{exec}(C, i)$, $C(\mathit{status})(i) = \mathit{alive}$, $C(\mathit{instr})(i) \neq \emptyset$, and $head(C(\mathit{instr})(i)) \neq \textbf{duration}(d)$ for any d. Then i becomes the current thread and a duration of δ_{cs} time units is added to represent the context switching time:
$C' = C[\ \mathit{instr}(i) \mapsto \textbf{duration}(\delta_{cs})\,\hat{}\,C(\mathit{instr})(i), \mathit{curthr}(\mathit{node}(i)) \mapsto i\ ]$

Note that more than one thread may be eligible as the current thread on a node at a certain point in time. In that case, a thread is chosen nondeterministically in our operational semantics. Fairness constraints or a scheduling strategy may be added to reduce the set of possible execution sequences and to enforce a particular type of node behavior, such as round robin or priority-based pre-emptive scheduling.

Definition 3.5.7 (Deliver Link Message) A step $C \longrightarrow C'$ corresponds to the message delivery by a link if and only if there exists a link l and a triple (m, lb, ub) in $C(\mathit{linkset})(l)$ with $lb \leq C(\mathit{now}) \leq ub$. There are two possibilities for message m:

- $\textbf{call}(i, \mathit{oid}, \mathit{op})$: Insert the call in the queue of object *oid*:
  $C' = C[\ q(\mathit{oid}) \mapsto \mathit{insert}((i, \mathit{op}), C(q)(\mathit{oid})),$
  $\qquad\quad \mathit{linkset}(l) \mapsto remove((m, lb, ub), C(\mathit{linkset})(l))\ ]$

- $\textbf{return}(i)$: Wake-up the caller, i.e.
  $C' = C[\ status(i) \mapsto \mathit{alive},\ \mathit{linkset}(l) \mapsto remove((m, lb, ub), C(\mathit{linkset})(l))\ ]$

Definition 3.5.8 (Process Queue Message) A step $C \longrightarrow C'$ corresponds to the processing of a message from a queue if and only if there exists an object *oid* with $\mathit{head}(C(q)(\mathit{oid})) = (j, \mathit{op})$. Let $j = \mathit{fresh}(C, \mathit{oid})$ be a fresh thread and *IS* be the explicit definition of *op*. If the operation is synchronous, i.e. *syn?*(op), then we start a new thread with *IS* followed by a return to the caller:
$C' = C[\ instr(j) \mapsto \mathit{IS}\hat{}\,\textbf{return}(j),\ status(j) \mapsto \mathit{alive},\ q(\mathit{oid}) \mapsto \mathit{tail}(C(q)(\mathit{oid}))\ ]$
Similarly for an asynchronous call, where no return instruction is added:
$C' = C[\ instr(j) \mapsto \mathit{IS},\ status(j) \mapsto \mathit{alive},\ q(\mathit{oid}) \mapsto \mathit{tail}(C(q)(\mathit{oid}))\ ]$

Definition 3.5.9 (Operational Semantics) The operational semantics of a specification in the language of Table 3.1 is a set of execution sequences of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$, where each pair $C_i \longrightarrow C_{i+1}$ is a step (Definition 3.5.3) and the initial configuration C_0 satisfies a number of constraints:

- no thread has status *waiting*;
- on each node, the currently executing thread is *alive*;
- a thread is dormant if and only if it has an empty execution sequence;
- all queues and link sets are empty, and
- the auxiliary instruction **return** does not occur in any instruction sequence.

To avoid Zeno behavior, we require that for any point of time t there exists a configuration C_i in the sequence with $C_i(now) > t$.

3.5.3 Validation

The formal operational semantics has been validated by formulating it in the typed higher-order logic of the verification system PVS [1] and verifying properties about it using the interactive theorem prover of PVS.

In fact, the formal operational semantics presented in this chapter is based on a much larger constructive (and therefore executable) operational semantics of the extended language, which has been specified in VDM++ itself. This "bootstrapping" approach [67] allows us to interpret models written in the modified language by symbolic execution of its abstract syntax in the constructive operational semantics model using the existing and unmodified VDMTOOLS.

A large collection of test cases has been created to observe the behavior of each new language construct and we are fairly confident that the proposed language changes

[1] The PVS files and all VDM++ models are available on-line at *http://www.marcelverhoef.nl*. The PVS interactive theorem prover is freely available from *http://pvs.csl.sri.com/*.

are consistent. The constructive operational semantics is currently approximately 100 pages including the test suite. It can be used as a specification to implement the proposed language changes in VDMTOOLS.

3.6 Related work and concluding remarks

One might argue that VDM and therefore this work, is not very relevant for distributed real-time embedded systems at all. Of course, we believe that this is not true. The Japanese company CSK, which owns the intellectual property rights to VDMTOOLS, is targeting this market in particular and they have already expressed interest in our ideas and results. For example, we were granted access to the company confidential dynamic semantics specification of the interpreter in order to perform our research.

Related to our formal semantics is work in the context of UML about the precise meaning of active objects, with communication via signals and synchronous operations, and threads of control. In [83] a labeled transition system has been defined using the algebraic specification language CASL, whereas [55] uses the specification language of the theorem prover PVS to formulate the semantics. Note that UML 2.0 adopts the run-to-completion semantics, which means that new signals or operation calls can only be accepted by an object if it cannot do any other local action, i.e., it can only proceed by accepting a signal or call. In our VDM++ semantics there are much less restrictions imposed on threads. In addition, none of these works deal with deployments. Related to that aspect is the UML Profile for Schedulability, Performance and Time, and research on performance analysis based on this profile [9].

In summary, we propose an extension of VDM++ to enable the modeling of distributed real-time embedded systems. These language extensions allows us to experiment with different deployment strategies at a very early stage in the design. On the syntactic level, the changes seem minor but they make a big difference. The model of the in-car navigation system presented in this chapter is significantly smaller than the model that was created earlier with Timed VDM++ in Section 2.3.5. Moreover, the new model covers a much larger part of the problem domain. We believe that important system properties can be validated in a very cost-effective way if these features are implemented in VDMTOOLS.

A constructive operational semantics was defined for a language subset to prototype and validate the required improvements in the semantics. The changes are substantial but they still fit the general framework of the full VDM++ dynamic semantics. Furthermore, a generalized abstract operational semantics, that is not specific to the VDM family of languages, is presented in this Chapter.

Chapter 4

Co-simulation of Distributed Embedded Real-Time Control Systems

4.1 Introduction

Computing systems that are intimately coupled to the environment which they monitor and control are commonly referred to as embedded systems. We focus on the class of embedded systems that control a physical process in the real world. We refer to these systems as embedded control systems. Examples are the control unit of a washing machine and the fuel injection system in a private car. Embedded control systems execute an algorithm that ensures the correct behavior of the system as a whole. The common element of all these systems is that timeliness is of concern. Control actions have to be taken *on time* to keep the physical process in the required state. Hence, embedded control systems are real-time systems.

This is in particular true for the class of high-tech systems such as for instance wafer steppers and high-volume printers and copiers. The productivity of these machines, which is often their most important selling point, depends on the performance of the embedded control system. Typically, these complex machines are composed of several subsystems that need to work together to get the job done, which may require multi-layer and distributed control. For example, each subsystem may have its own embedded control system to perform its specific function while another, dedicated, subsystem coordinates the system as a whole by telling the other subsystems what to do and when. It is not hard to imagine that the design of the control strategy for these systems is challenging.

This is complicated by the fact that systems are often developed out-of-phase. Typically, mechanical design precedes electronics design which precedes software design. Although there is a trend towards concurrent engineering to reduce development time, the lead times for mechanical design and engineering typically still exceed those of electronics and software. System level design considerations are validated during the test and integration phase, which may cause significant delays in the project in case an important issue was overlooked. Software is often the only part of the system that can be changed at this late stage. These late changes can cause a significant increase in the complexity of the software, especially when a carefully designed software architecture is violated to compensate for some unforeseen problems in the hardware. Hence, it is important to get as much feedback as possible in the earliest stages of the system design life-cycle, to prevent this situation.

Model-based design addresses this challenge. Reasoning about system-level properties is enabled by creating abstract, high-level and multidisciplinary models of the system under construction. Mono-disciplinary models typically allow optimization of single aspects of the design, while multidisciplinary models allow reasoning about fitness for purpose across multiple system aspects. Suppose, for instance, that the position of a sheet of paper in the paper path of a printer is measured with a sensor that generates an interrupt when the edge of the sheet is observed. High interrupt loads can occur on the embedded control system if these sensors are placed physically close together, because they are triggered right after one another. A very powerful processor may be required in order to deal with this sudden peak load, in particular when a short response time must be guaranteed for each event. There is a clear trade-off between spatial layout and performance in this example. Analysis of multidisciplinary models provides valuable insight into the design such that these trade-offs can be made in a structured way, earlier, and with more confidence.

This approach was studied in the BODERC project [47] in which the author participated. We observed that creating multidisciplinary models is far from trivial. The notations and the engineering and analysis approaches that are advocated by the involved disciplines are different and the resulting models are typically not at the same level of abstraction. Henzinger and Sifakis [53] even claim that these are fundamental problems and that a new mathematical foundation is required to reason about these integrated multidisciplinary models. The approach taken in this chapter is different. We would like to be able to combine the state of the art in each discipline in a useful and consistent way. In other words, we want to construct multidisciplinary models from mono-disciplinary models. We are certainly not the first to propose this idea but we believe that our solution to this problem is novel.

Contribution of this chapter. We have reconciled the semantics of two existing formal notations such that system models, which are composed of sub-models written in either language, can be conveniently studied in combination. We also demonstrate how this is achieved in practice by tool coupling. The result is a light-weight modeling approach that enables construction of multidisciplinary models that can be simulated, in addition to the analysis techniques already available for each sub-model individually. Moreover, the reconciled semantics ensures reliable simulation results which can be obtained with little effort.

Structure of this chapter. An overview of the current state of practice is presented in Section 4.2. Modeling and analysis of embedded control systems is discussed by introducing a motivating case study in Section 4.3. The results of the simulation using the tool coupling are shown in Section 4.4. The semantic integration is presented from a formal perspective in Section 4.5. Finally, we look at related and future work and we draw conclusions in Section 4.6.

4.2 Current state of practice in academia and industry

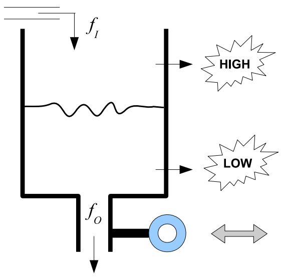
The importance of model-based design is widely recognized and we observe that many contenders, typically originating from a specific discipline, are extending their techniques to cater for this wider audience. Matlab/Simulink is an example of this trend.

In combination with their Stateflow and Real-time Workshop add-on products, they provide a tool chain for embedded systems design and engineering. It is particularly well-suited for fine grained controller design. This is not surprising because the roots of the tools are firmly based in systems theory. Stateflow can be used to model the control software using finite state machines. However, this technique is not very convenient for specifying complex algorithms. One has to write so-called S-functions or provide a piece of C-code in order to execute the Stateflow model. Timing is idealized by the assumption that all transitions take a fixed number of timer ticks. Scheduling and deployment of software on a distributed system cannot easily be described and analyzed. Henriksson [52] designed and implemented the TrueTime toolkit on top of Simulink which provides a solution for describing scheduling and deployment, but the software models remain at a low abstraction level. We believe that these tools are *not* acceptable to embedded software engineering at large, because insufficient support is provided for modern software engineering approaches to design and implement complex real-time software.

A similar situation arises from IBM Rational Technical Developer (formerly known as Rational Rose Real-time) and Telelogic Rhapsody (now also an IBM company). These software development environments are increasingly used in real-time embedded systems development [26]. They provide modeling capabilities based on the Unified Modeling Language (UML) and the System Modeling Language (SysML) and are supported by mature development processes (RUP and Harmony respectively). Both tools aim to develop executable models that are deployed on the target system as soon as possible to close the design loop. This requires the model to evolve to a low level of abstraction early in the design process in order to achieve that goal. Actions are coded directly in the target (programming) language and timing can be specified by using so-called timer objects provided by the modeling framework. However, their resolution and accuracy is determined by the services of the operating system running on the target platform; they are not part of the modeling language. Moving code from one platform to another might lead to completely different timing behavior. Similarly, task priorities and scheduling are implementation specific. We believe that these tools are *not* acceptable to the control engineer at large, because no support is provided to design and analyze the control laws that the system should implement.

Is it possible to support control and software engineers using a single method or tool? Several attempts have been made to unify both worlds. For example, Hooman, Mulyar and Posta [54] have co-simulated Rose Real-time software models with control laws specified in Matlab/Simulink. They removed the platform dependent notion of time in Rose Real-time by providing a platform neutral notion of time instead. This is achieved by development of an interface that sits in between Rose Real-time and Simulink, which exposes the software simulator of Rose Real-time to the Simulink internal clock. While this is a step forward, it also shows that Rose Real-time is not very suitable for the co-simulation of control systems, because it lacks a suitable notion of simulation time and the run-to-completion semantics does not allow interrupts due to relevant events of the physical system under control. I-Logix has recently announced integration of Rhapsody with Simulink but the technical details have not yet been unveiled.

Lee et al [22] propose a component based, actor oriented approach. They define a framework in which all components are concurrent and interact by sending messages according to some communication protocol. The communication protocol and the concurrency policies together are called the model of computation. Ptolemy-II [22] is a system-level design environment that supports heterogeneous modeling and design us-



$$\frac{d\mathrm{V}}{dt} = f_{\mathrm{I}} - f_{\mathrm{O}} \tag{4.1}$$

$$f_{\mathrm{O}} = \begin{cases} \frac{\rho \cdot g}{\mathrm{A} \cdot \mathrm{R}} \cdot \mathrm{V} & \text{if valve = open} \\ 0 & \text{if valve = closed} \end{cases} \tag{4.2}$$

Figure 4.1: The water tank level control case study

ing this approach. It supports several domains, each of which is based on a particular model of computation, such as for example discrete event, synchronous data flow, process networks, finite state machines and communicating sequential processes. They can be combined at liberty to describe the system under investigation. This approach seems to be a major step forward for model based design of real-time embedded systems, but paradoxically, it does neither appeal to control engineers nor to software engineers. Perhaps the approach proposed by Ptolemy-II upsets the current way of working so much that it is considered too high a risk to use in an industrial environment, as was our own experience. Currently, only simulation is offered as a means of model validation and synthesis is under development for some domains. Verification of Ptolemy-II models is not yet possible because the semantics of actors has not been formally defined.

4.3 Modeling and analysis of embedded control systems

The complexity of embedded control design and analysis is probably best explained by means of a motivating example. We use the level control of a water tank in this chapter. This example is small and simple, but it contains all the basic elements of an embedded control system. These elements are presented in detail in this section. An overview of the case study is presented in Figure 4.1. The case study concerns a water tank that is filled by a constant input flow f_{I} and can be emptied by opening a valve resulting in an output flow f_{O}. The volume change is described by equations (4.1) and (4.2), where A is the surface area of the tank bottom, V is the volume, g is the gravitation constant, ρ is the density of the liquid and R is the resistance of the valve exit.

From the system theoretic point of view, we distinguish the *plant* and the *controller* of an embedded control system, as shown in Figure 4.2. The plant is the physical entity in the real world that is observed and actuated by the controller. More accurately, we study feedback control in this chapter. Feedback controllers compute and generate a control action that keeps the difference between the observed plant state and its desired value, the so-called set-point, within a certain allowed margin of error at all times. The plant is a dynamic system that is usually represented by differential equations if it is described in the continuous time (CT) domain or by difference equations if it is described in the discrete time (DT) domain.

The water tank case study is an example of a continuous time system, described by differential equation (4.1). Controllers observe some property of the plant and they change the state of the plant by performing a control action, according to some control

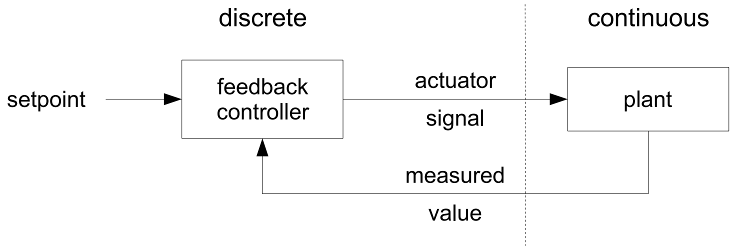


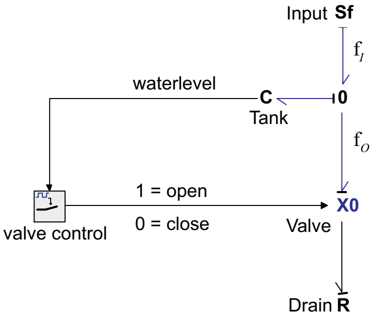
Figure 4.2: System theoretic view of a control system

law. This control law keeps the system as a whole in some desired state. In our case study, the water level is observed by three sensors: a pressure sensor at the bottom of the tank which measures the current water level continuously and two discrete sensors that raise an alarm if a certain situation occurs. The top sensor informs us when the water level exceeds the high water mark and the bottom sensor fires if the water level drops below the low water mark. The aim of the controller is to keep the water level between the low and high watermark. The controller can influence the water level by opening or closing a valve at the bottom of the tank. We assume that the valve is either fully open or fully closed. Plant modeling and controller descriptions are discussed in more detail in the following sections.

4.3.1 Plant modeling

To model the plant of the embedded control system, we use so-called bond graphs [61, 13] in this chapter. Bond graphs are directed graphs, showing the relevant dynamic behavior of the system. Vertices are the sub-models and the edges, which are called *bonds*, denote the ideal (or idealized) exchange of energy. Entry points of the sub-models are the so-called *ports*. The exchange of energy through a port (p) is always described by two implicit variables, effort ($p.e$) and flow ($p.f$). The product of these variables is the amount of energy that passes through the port. For each physical domain, such a pair of variables can be specified, for example: voltage and current, force and velocity. The half arrow on the vertex at the bonds shows the positive direction of the flow of energy, and the perpendicular stroke indicates the computational direction of the two variables involved. They connect the energy flows to the two variables of the bond. The equations that define the relationship between the variables are specified as real equalities, not as assignments. Port variables obtain a computational direction (one as input, the other as output) by means of computational causal analysis on the graph. This efficient algorithm ensures that the underlying set of differential equations can be solved deterministically by rewriting the equations as assignment statements such that a consistent evaluation order is enforced whenever a solution is calculated. Bond graphs are physical-domain independent, due to analogies between the different domains on the level of physics. Mechanical, electrical, hydraulic and other system parts can all be modeled with bond graphs. Bond graphs may be mixed with block diagrams in a natural way to cover the information domain. Control laws are usually specified with block diagrams and the plant is specified with bond graphs to model a controlled mechatronic system. Figure 4.3 shows the bond graph plant model of the water tank case study. The S_f element is the input flow f_I. The C element describes the water tank. The equations of the tank are next to the figure. The R element describes the drain. The $\mathsf{X0}$ element is a so-called switching junction which describes the valve.

When the valve is opened, a flow f_O will be drained from C. There is no flow from C when the valve is closed.



```
variables
  real volume, level;
parameters
  real area = 1.0;
  real gravity = 9.81;
  real density = 1.0;
equations
  // p.e = pressure, p.f = flow rate
  // integrate flow to obtain volume
  volume = int(p.f);
  level = volume / area;
  p.e = gravity * level * density;
```

Figure 4.3: The bond graph plant model of the water tank case study

Differential equations are the general format for representing dynamic systems mathematically. For specifying a plant model many continuous-time representations exist, e.g., bond graph models, ideal physical models, block and flow diagrams and so on. A common property is that all these model types are directly related to a set of differential equations. For the subset of linear time-invariant plant models, alternative description techniques exist, such as the s-plane, frequency response and state-space formats [70].

System theory has provided many analysis techniques for time-invariant linear models and design techniques for their associated controllers, for which certain properties can be proven to hold. However, real world systems often tend to be nonlinear and time varying. The task of the control engineer is to find a suitable linearization such that system theory can still be applied to design a controller. Alternatively, simulation can be used if the dynamic system can be described by a collection of so-called ordinary differential equations. This includes the linear time-invariant models mentioned earlier, as well as non-linear and time varying differential equations. Partial differential equations can be approximated by lumped parameter models in ordinary differential equations and also non-deterministic (or stochastic) models can be simulated. Although simulation can never provide hard answers, it is often used because it can address a much larger class of problems than linear analysis. For example, it can be used to determine whether a linearized model is a good abstraction of the original non-linear model, since both models can be simulated.

The basic method used in simulation is to solve a differential equation numerically instead of analytically. Approximations of the solution are computed by means of integration of the differential equations. These numerical integration techniques are commonly referred to as “solvers” and they exist in many flavors. Examples of well-known solvers are Euler, Runge-Kutta and Adams-Bashforth [44, 45]. These solvers belong to the class of fixed step size integration algorithms. Also many variable step size algorithms exist. Selection of the right solver is non-trivial and requires a good understanding of the model itself. For example, variable step size solvers are typically required when the dynamic system is described by (combined CT and) DT models. In addition, since an approximation of the solution is computed, an integration error is introduced. This error might lead to instability if the solver, and its parameters, are not carefully selected.

4.3.2 Controller description

According to Cassandras and Lafortune [17], a system belongs to the class of discrete event systems if the state can be described by a set of discrete values and state transitions are observed at discrete points in time. We adopt this definition here. Discrete event models can be used to describe the behavior of digital computers, which implement certain control laws. Computers execute instructions based on a discrete clock. The result of an instruction becomes available after a certain number of clock ticks has elapsed. Sensor input samples and actuator output values are seen as discrete events in this model of computation.

In order to bridge the gap between continuous time and discrete event simulation, we obviously need to introduce the notion of events in the continuous time solver. Here, we distinguish two different event types: a) state events and b) time events. State events occur when the solution of a differential equation reaches some value p. Time events occur when the solver has reached some time t. Consider a solver that produces a sequence of time steps $time$ and a sequence of solutions $state$ for variable x then we can declare events as follows

$$\textsc{ree}\,(x,p) \overset{def}{=} \mathit{state}\,(x,n-1)-p<0 \;\wedge\; \mathit{state}\,(x,n)-p\geq 0 \tag{4.3}$$

$$\textsc{fee}\,(x,p) \overset{def}{=} \mathit{state}\,(x,n-1)-p>0 \;\wedge\; \mathit{state}\,(x,n)-p\leq 0 \tag{4.4}$$

$$\textsc{te}\,(t) \overset{def}{=} \mathit{time}\,(n-1)<t \;\wedge\; \mathit{time}\,(n)=t \tag{4.5}$$

whereby n is the index used in both sequences. The event REE is the so-called rising edge zero crossing and FEE is the falling edge zero crossing. The zero crossing functions of the solver ensure that $time(n)$ is an accurate approximation within user-defined bounds. The time event TE is generated as soon as the solver has exactly reached time t, whereby the solver ensures that the solution x in $state(x,n)$ at $time(n) = t$ is an accurate approximation. For our case study, we define two edge triggered events: REE (*level*, 3.0) and FEE (*level*, 2.0), whereby *level* is a shared continuous time variable that represents the height of the water level in the tank. This variable is declared on line 2 of Figure 4.3 and line 4 of Figures. 4.5 and 4.6. An event is declared as a normal equation in 20-SIM [19] as shown in Figure 4.4. In this example, we increment a simple event counter `eue` and inform the CT solver that the DE model needs to be updated, by setting the variable `fireDES`.

```
// check for the upper water level limit
if (eventup(level - 3.0)) then
    eue = eue + 1;
    fireDES = true;
end;
```

Figure 4.4: The REE (*level*, 3.0) event in 20-SIM

We use VDM++ [30] in this chapter to describe the controller. We extend the notation reported in earlier work [101], which is also presented in the previous chapter, such that the behavior of this discrete event controller can be analyzed by means of co-simulation with the continuous time plant model. For simplicity, we assume a single processor system `cpu1` that executes the controller in our example.

We demonstrate that two styles of control can be used: event driven control, shown in Figure 4.5, and time triggered control, presented in Figure 4.6. Both models have shared continuous sensor and actuator variables *level* and *valve*, which are declared on Line 4 and 5. Whenever the VDM++ instance variable *level* is read, it will contain the actual value of the *level* variable of the continuous time model as shown on line 11 of Figure 4.3. Similarly, whenever instance variable *valve* is assigned a value in VDM++, it will immediately change the state of X0 in Figure 4.3.

For event driven control, as shown in Figure 4.5, two asynchronous operations, *open* and *close* are defined in lines 8 and 11 respectively. The former will be the handler for the REE (*level*, 3.0) event and the latter is the handler for the FEE (*level*, 2.0) event. In other words, these two asynchronous operations will be called automatically by the simulation framework whenever the corresponding event fires. This will cause the creation of a new thread. This thread will die as soon as the operation is completed. In VDM++, all statements have a default duration, which can be redefined using the `duration` and `cycles` statements. The duration statement on line 9 states that opening the valve in this case takes $50\,msec$. The cycles statement on line 12 denotes that closing the valve takes 1000 cycles. Assuming this class is deployed on a processor with a capacity of 100000 cycles per second, then executing `valve := false` will take $10\,msec$. Note that the result of the assignment is available *after* this time has passed. The `sync` clause on line 14-17 states that the two operations are declared mutually exclusive. This implies that only one operation call can be active at any time and they cannot be interrupted by each other. All threads that do not meet this requirement are blocked until the currently executing thread terminates or yields.

```
01  class EventDrivenControl
02
03  instance variables
04    static public level : real;
05    static public valve : bool := false  -- default is closed
06
07  operations
08    async static public open: () ==> ()
09    open () == duration(0.05) valve := true;
10
11    async static public close: () ==> ()
12    close () == cycles(1000) valve := false;
13
14  sync
15    mutex(open, close);
16    mutex(open);
17    mutex(close)
18
19  end EventDrivenControl
```

Figure 4.5: Event driven control of the water tank in VDM++

Time triggered control, as presented in Figure 4.6, is provided by the *loop* operation in line 8-15. The `periodic` clause in line 18 states that the operation *loop* is called periodically, once per second, starting at $t = 1\,sec$. Again we use the `duration` and `cycles` constructs here to specify the time required to open and close the valve.

4.4 Tool support

We implemented a discrete event simulator to execute VDM++ models as described in the previous section, as a proof of concept. We coupled this tool to the 20-SIM [19] continuous time simulator for dynamic systems. This tool has the ability to make

```
class TimeTriggeredControl

instance variables
  static public level : real;
  static public valve : bool := false   -- default is closed

operations
  loop: () ==> ()
  loop () ==
    -- first check high water mark
    if level >= 3
    then duration(0.05) valve := true
    -- then check low water mark
    else if level <= 2
         then cycles(1000) valve := false;

threads
  periodic(1.0, 0.0, 0.0, 1.0)(loop)

end TimeTriggeredControl
```

Figure 4.6: Time triggered control of the water tank in VDM++

calls to user-defined libraries from within the simulation. We implemented a simple DLL in C++ to exchange arbitrary sequences of double precision reals over a TCP/IP connection. The same library is used in the VDM++ simulator to set-up a connection. The progress of time in the simulators on either end of the connection is synchronized by exchanging the current time, time steps, actuator and sensor values and events, whereby the current time is always strictly monotonically increasing. In this section we will focus on the construction and use of the interface. In the next section we will look at the semantics in more detail.

The behavior of the interface is shown in the UML sequence diagram in Figure 4.8. We use an XML configuration file to describe the information that is exchanged over the link. The interface is completely model independent. For brevity, we use an informal description as presented in Figure 4.7. The keywords `sensor` and `actuator` are defined as perceived from the perspective of the discrete event simulator. Basically, we define a `sensor[]` array, an `actuator[]` array and an `event[]` array. These arrays provide the bindings for all variables and events. The `abort` keyword is used to stop the simulation, in addition to other tool specific stop criteria that may be defined, and gives control back to the user, for example to inspect the state of the model.

```
sensor[1] = cpu1.Controller`level
actuator[1] = cpu1.Controller`valve
event[1] = REE(level,3.0) -> cpu1.Controller`open
event[2] = FEE(level,2.0) -> cpu1.Controller`close
event[3] = TE(15.0) -> abort
```

Figure 4.7: The interface configuration file

The XML configuration file is read by both simulations when the interface is started, indicated by `initialize` in Figure 4.8. When a message is sent from VDM++ to 20-SIM, indicated as `updateCT` in Figure 4.8, the message contains the current time T, the target time step t_s, and the value of each defined actuator variable at T from `actuator[]`. So, for our case study only three values are exchanged in this direction for every step. Upon arrival, the operation `updateCTmodel` calls the continuous time solver and tries to perform the time step t_s. Either this time was reached or the solver stopped due to an event that occurred at t_r. When a message is sent from 20-SIM to VDM++, indicated as `updateDE` in Figure 4.8, the message contains the current time T, the realized time step $t_r \leq t_s$, the value of each defined sensor variable at $T + t_r$ from `sensor[]`, followed by a monotone increasing counter for each declared `event[]`. This counter is incremented when the event occurred at $T + t_r$. This allows us to monitor the integrity of the interface. Several events can be detected at the same time, but an event can only occur once per iteration. Six values are offered when a message is sent from 20-SIM to VDM++ in the water tank model. Upon arrival, the operation `updateDEmodel` processes all events, updates the shared continuous variables and performs a simulation step on the discrete event model, after which we iterate.

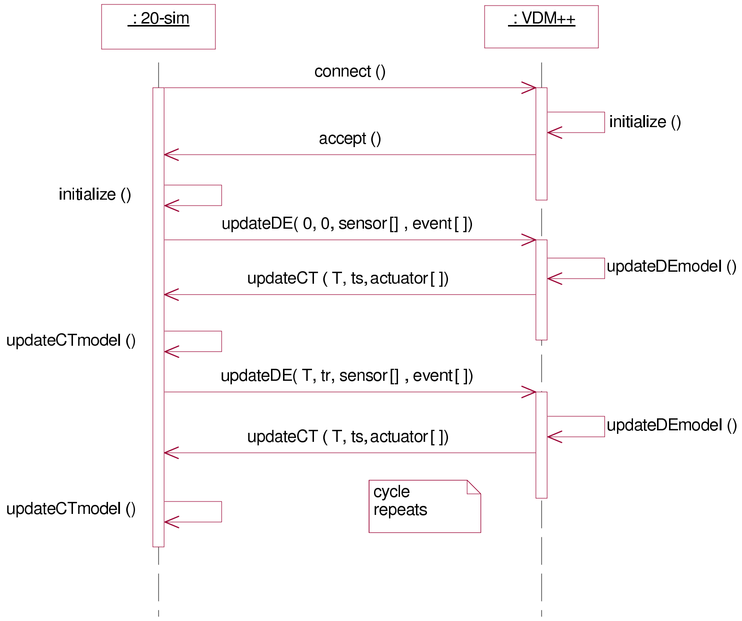


Figure 4.8: Tool interface behavior as a UML sequence diagram

Figure 4.9 presents a co-simulation run for our case study using event driven control. In other words, we are studying the behavior of the two asynchronous operations *open* and *close*, as shown on lines 8-12 in Figure 4.5. The top screen shows the evolution of the *level* sensor variable. The middle screen shows the evolution of the *valve* actuator variable. The bottom screen shows when the controller has been activated. This is monitored by means of a simple counter that is increased whenever the VDM++ model executes either of these asynchronous operations. Note that these operations are only executed when either of the sensors is tripped.

Figure 4.10 presents a co-simulation run for our case study using time triggered control. In other words, we are studying the behavior of the periodic *loop* operation, as shown on lines 8-18 in Figure 4.6. The top screen shows the evolution of the *level* sensor variable. The middle screen shows the evolution of the *valve* actuator variable. The bottom screen shows when the controller has been activated. This is monitored by means of a simple counter that is increased whenever the VDM++ model executes the *loop* operation. Notice that the discrete controller is indeed invoked every second.

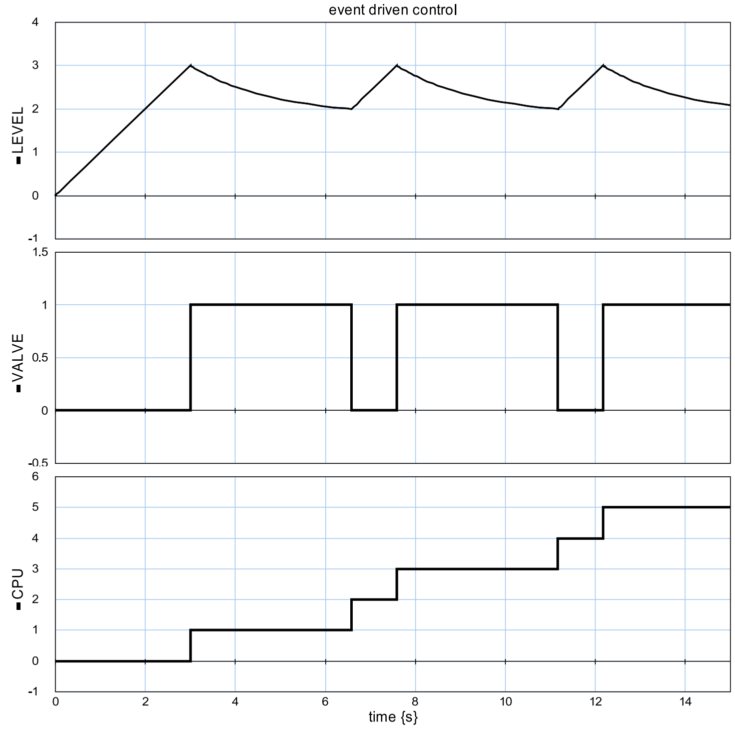


Figure 4.9: Co-simulation of the water tank case study using event driven control

Moreover, observe that the valve was not opened at $t = 8\,sec$ because *level* was 2.96 at that time. The overshoot would have been substantially smaller if event based control was used or a smaller period was chosen.

We can change many system parameters in the discrete event simulator and observe their impact, such as the processor speed, task switch overheads, and the scheduling policy, without modifying the models shown in Figures 4.5 and 4.6. Similarly, we can change parameters in 20-SIM, such as the input flow rate, the liquid density, the resistance of the valve exit, etcetera.

4.5 Reconciled operational semantics

We extend the abstract and formal operational semantics for distributed embedded real-time systems of Chapter 3 in this section. Recall that a two-phase elaboration is used. In the first phase, all active threads perform atomic actions asynchronously until they need to perform a time step. In the second phase, this time step is performed synchronously for all threads and all pending messages. One of the key features of the work presented here is that state modifications computed in phase one are only made visible after the time step in phase two has been completed, in order to guarantee consistency in the presence of shared continuous variables and arbitrary interleaving of multiple, concurrent, threads.

The main aim of the extended operational semantics presented here is to formalize the interaction between the discrete event simulator, which executes a control program,

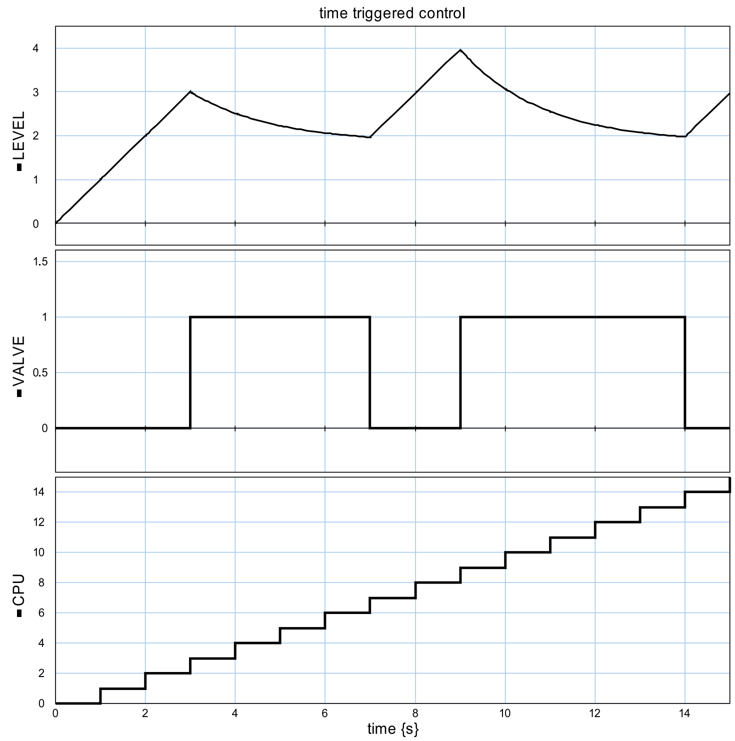


Figure 4.10: Co-simulation of the water tank case study using time triggered control

and a solver for a continuous time plant model. We have omitted many details that have already been formalized in Chapter 3, such as the links between nodes, message transfer along these links, the definition of operations, guards and the concept to define periodic threads. In this section, we concentrate on the interaction between discrete event and continuous time models by means of sharing state variables and exchanging events. In Section 4.5.1 we define the syntax of a simple imperative language which serves as an illustration of the basic concepts, without trying to be complete. The operational semantics of this language is defined in Section 4.5.2. The tool support described in the previous section conforms to this formal operational semantics.

4.5.1 Syntax and informal semantics revisited

The distributed architecture of an embedded control program can be represented by so-called nodes. Nodes are used to represent computation resources such as processors. On each node a number of concurrent threads are executed in an interleaved way. Each thread performs a sequential program, that is, a statement (instruction sequence) expressed in the language of Table 3.1. In fact, we need to extend the syntax of the language in order to demonstrate the extensions to the operational semantics proposed here. Let *Value* be a domain of values, such as the reals and let *Var* be a set of variables. The syntax of our enhanced sequential programming language is given in Table 4.1, with $c \in$ *Value*, $x \in$ *Var*, and $d \in$ *Time*.

Value Expr.	$e ::=$	$c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Bool Expr.	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Instr.	$I ::=$	**skip** $\mid x := e \mid$ **call**(oid, op) $\mid$ **duration**(d) $\mid$
		periodic(d) *IS* $\mid$ **if** b **then** IS_1 **else** IS_2 **fi** $\mid$
		while b **do** *IS* **od**
Instr. Seq.	$\mathit{IS} ::=$	$\langle\rangle \mid I\,\hat{}\,\mathit{IS}$

Table 4.1: Abstract syntax of basic instructions - revisited

These basic instructions have the following informal meaning:

- **skip** represents a local statement which does not consume any time.
- $x := e$ assigns the value of expression e to x.
- **call**(oid, op) denotes a call to an operation *op* of object *oid*. Depending on the *syn?* predicate, the operation can be synchronous (i.e., the caller has to wait until the execution of the operation body has terminated) or asynchronous (the caller may continue with the next instruction and the operation body is executed independently). There are no restrictions on re-entrance here, but in general this can be restricted in VDM by so-called permission predicates. These are not considered here and also parameters are ignored.
- **duration**(d) represents a time progress of d time units. When d time units have elapsed the next statement can be executed.
- **periodic**(d) *IS* leads to the execution of instruction sequence *IS* each period of d time units.
- **if** b **then** IS_1 **else** IS_2 **fi** executes instruction sequence IS_1 if b evaluates to true and IS_2 otherwise.
- **while** b **do** *IS* **od** repeatedly executes instruction sequence *IS* as long as b evaluates to true.

New in the approach taken here is that the execution of threads is interleaved with steps of the continuous time solver. The interface between the discrete event and continuous time models consist of shared variables and events. First, we define the shared variables.

Assume given a set of variables $\mathit{Var} = \mathit{InVar} \cup \mathit{OutVar} \cup \mathit{LVar}$ where *InVar* is the set of input / sensor variables, *OutVar* is the set of output / actuator variables, and *LVar* a set of local variables. The input and output variables (also called I / O-variables) are global and shared between all threads and the continuous model. Hence, they can also be accessed by the solver of a continuous model, which may read the actuator variables and write the sensor variables. Let $\mathit{IOVar} = \mathit{InVar} \cup \mathit{OutVar}$.

Next, we define a notion of events. The continuous time solver may send events to the discrete event control program. Let *Event* be a set of events, which can be defined by using the primitives REE (x, c), FEE (x, c), and TE (d), as proposed in Equations 4.3-4.5. Assume that an event handler has been defined for each event, i.e., an instruction sequence and a node on which this statement has to be executed as a new thread, denoted by the function $\mathit{evhdlr} : \mathit{Event} \rightarrow \mathit{Instr.\ Seq.} \times \mathit{Node}$.

4.5.2 Formal Operational Semantics

A new issue arises with respect to the formal definition of the extended operational semantics, in addition to the questions already raised in Section 3.5.2 :

- What is the effect of the interleaved execution of assignments to shared variables in different threads? Recall that the execution of basic statements such as skip and assignment takes zero time. Hence, in our semantics any sequence of statements between two successive duration statements is executed atomically (in zero time). For instance, if we execute the instruction sequence **duration**$(1)\hat{}x := 1\,\hat{}\,x := x+1\,\hat{}\,$**duration**$(1)$ in parallel with the sequence **duration**$(1)\,\hat{}\,x := 5\,\hat{}\,y := x\,\hat{}\,$**duration**$(1)$ then there are two possible results; we might get $x = 5 \wedge y = 5$ or $x = 2 \wedge y = 5$. This in contrast with **duration**$(1)\,\hat{}\,x := 1\,\hat{}\,$**duration**$(1)\,\hat{}\,x := x+1\,\hat{}\,$**duration**$(1)$ in parallel with **duration**$(1)\,\hat{}\,x := 5\,\hat{}\,$**duration**$(1)\,\hat{}\,y := x\,\hat{}\,$**duration**$(1)$, where additionally $x = 2 \wedge y = 1$, $x = 2 \wedge y = 2$, $x = 6 \wedge y = 5$, and $x = 6 \wedge y = 6$ are possible.

Hence, the execution of an instruction sequence might be interleaved with statements of other threads or a step of the continuous time solver. Concerning the shared I / O-variables in *IOVar*, this means that we have to ensure atomicity explicitly. Hence, we introduce a kind of *transaction* mechanism to guarantee consistency in the presence of arbitrary interleaving of steps. Thread i is only allowed to modify I / O-variable x if there is no transaction in progress by any other thread. The transaction is committed immediately after the thread performs a time step. This will be explained in detail in Defs. 4.5.2, 4.5.4 and 4.5.5.

To capture the state of affairs at a certain point during the execution, we introduce a *configuration* (Definition 4.5.1). Next we define the possible steps from one configuration to another, denoted by $C \longrightarrow C'$ where C and C' are configurations (Definition 4.5.3). This finally leads to a set of runs of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$ (Definition 4.5.9).

Definition 4.5.1 (Configuration) A *configuration* C contains the following fields:

- *instr* : *Thread* → *Instr. Seq.*
  which is a function which assigns a sequence of instructions to each thread.

- *curthr* : *Node* → *Thread*
  yields for each node the currently executing thread.

- *status* : *Thread* → $\{$*dormant*, *alive*, *waiting*$\}$
  denotes the status of threads.

- *lval* : *LVar* × *Thread* → *Value*
  denotes the value of each local variable for each thread.

- *ioval* : *IOVar* → *Value*
  denotes the committed value of each sensor and actuator variable.

- *modif* : *IOVar* × *Thread* → *Value* $\cup \{\perp\}$
  denotes the values of sensor and actuator variables that have been modified by a thread and for which the transaction has not yet been committed (by executing a duration statement). The symbol $\perp$ denotes that the value is undefined, i.e., the thread did not modify the variable in a non-committed transaction.

- q : *ObjectId* → *queue*[*Thread* × *Operations*]
  records for each object a FIFO queue of incoming calls, together with the calling thread (needed for synchronous operations only).

- *linkset* : *Link* → *set*[*Message* × *Time* × *Time*]
  records the set of the incoming messages for each link, together with lower and upper bound on delivery. A message may denote a call of an operation (including the calling thread and called object) or a return to a thread.

- *now* : *Time*
  denotes the current time.

For a FIFO queue, functions *head* and *tail* yield the head of the queue and the rest, respectively; *insert* is used to insert an element and $\langle\rangle$ denotes the empty queue. For sets we use *add* and *remove* to insert and remove elements. For a configuration C we use:

- $C(f)$ to obtain its field f. For example, $C(\mathit{instr})(i)$ yields the instruction sequence of thread i in configuration C.

- $\mathit{exec}(C, i)$ as an abbreviation for $C(\mathit{curthr})(\mathit{node}(i)) = i$, which expresses that thread i is executing on its node.

- $\mathit{fresh}(C, \mathit{oid})$ to yield a fresh, not yet used, thread identity (so with status *dormant*) corresponding to object *oid*.

To express modifications of a configuration, we define the notion of a variant.

Definition 4.5.2 (Variant) The *variant* of a configuration C with respect to a field f and value v, denoted by $C[\, f \mapsto v\,]$, is defined as

$$(C[\, f \mapsto v\,])(f') = \begin{cases} v & \text{if } f' = f \\ C(f') & \text{if } f' \neq f \end{cases}$$

Similarly for parts of the fields, such as $\mathit{instr}(i)$.

We define the value of an expression e in a configuration C which is evaluated in the context of a thread i, denoted by $[\![\, e\,]\!](C, i)$. The main point is the evaluation of a variable, where for an I / O-variable we use the *modif* field if there is an uncommitted change:

$$[\![\, x\,]\!](C, i) = \begin{cases} C(\mathit{modif})(x, i) & \text{if } x \in \mathit{IOVar}, C(\mathit{modif})(x, i) \neq \bot \\ C(\mathit{ioval})(x) & \text{if } x \in \mathit{IOVar}, C(\mathit{modif})(x, i) = \bot \\ C(\mathit{lval})(x, i) & \text{if } x \in \mathit{LVar} \end{cases}$$

The other cases are trivial, e.g., $[\![\, e_1 \times e_2\,]\!](C, i) = [\![\, e_1\,]\!](C, i) \times [\![\, e_2\,]\!](C, i)$ and $[\![\, c\,]\!](C, i) = c$. It is also straightforward to define when a Boolean expression b holds in the context of thread i in configuration C, denoted by $[\![\, b\,]\!](C, i)$. For instance, $[\![\, e_1 < e_2\,]\!](C, i)$ if and only if $[\![\, e_1\,]\!](C, i) < [\![\, e_2\,]\!](C, i)$, and $[\![\, \neg b\,]\!](C, i)$ if and only if not $[\![\, b\,]\!](C, i)$.

Definition 4.5.3 (Step) $C \longrightarrow C'$ is a *step* if and only if it corresponds to the execution of an instruction (Definition 4.5.4), a time step (Definition 4.5.5), a context switch (Definition 4.5.6), the delivery of a message by a link (Definition 4.5.7), or the processing of a message from a queue (Definition 4.5.8).

Definition 4.5.4 (Execute Instruction) A step $C \longrightarrow C'$ corresponds to the execution of an instruction if and only if there exists a thread i such that *exec*(C, i) and $head(C(\textit{instr})(i))$ is one of the following (underlined) instructions:

$\underline{\textbf{skip}}$:
Then the new configuration equals the old one, except that the skip instruction is removed from the instruction sequence of i, that is,
$C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i))\ ]$

$\underline{x := e}$:
We distinguish two cases, depending on the type of variable x.

- If $x \in \textit{IOVar}$ we require that there is no transaction in progress by any other thread, that is, for all i' with $i' \neq i$ we have $C(\textit{modif})(x, i') = \bot$. Then the value of e is recorded in the modified field of i:
  $C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i)), \textit{modif}(x, i) \mapsto [\![\, e\,]\!](C, i)\ ]$

  As we will see later, all values belonging to thread i in C(*modif*) are removed and bound to the variables in $C(\textit{ioval})$ as soon as thread i completes a time step (Definition 4.5.5). This corresponds to the intuition that the result of a computation is available only at the end of the time step that reflects the execution of a piece of code.

- If $x \in \textit{LVar}$ then we change the value of x in the current thread:
  $C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i)), \textit{lval}(x, i) \mapsto [\![\, e\,]\!](C, i)\ ]$

$\underline{\textbf{call}(\textit{oid}, \textit{op})}$:
Let *IS* be the explicit definition of operation *op* of object *oid*. We consider four cases:

- Caller and callee are on the same node, i.e. $node(i) = node(\textit{oid})$.
  - If *syn?*(*op*) then *IS* is executed directly in the thread of the caller:
    $C' = C[\ \textit{instr}(i) \mapsto \textit{IS}\hat{\ }\textit{tail}(C(\textit{instr})(i))\ ]$
  - If not *syn?*(*op*), we add the pair $(i, \textit{op})$ to the queue of *oid*:
    $C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i)),$
    $q(\textit{oid}) \mapsto \textit{insert}((i, \textit{op}), C(q)(\textit{oid}))\ ]$

- Caller and callee are on different nodes, i.e. $node(i) \neq node(\textit{oid})$. Suppose link l connects these nodes. Then the call is transmitted via link l, which is represented by adding message $m = (\textit{call}(i, \textit{oid}, \textit{op}), C(\textit{now}) + \delta_{min}(l), C(\textit{now}) + \delta_{max}(l))$ to the linkset of l.
  - If *syn?*(*op*), thread i becomes *waiting*:
    $C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i)),\ \textit{status}(i) \mapsto \textit{waiting},$
    $\textit{linkset}(l) \mapsto \textit{insert}(m, C(\textit{linkset})(l))\ ]$
  - Similarly for asynchronous operations, when not *syn?*(*op*), except that then the status of i is not changed:
    $C' = C[\ \textit{instr}(i) \mapsto \textit{tail}(C(\textit{instr})(i)),$
    $\textit{linkset}(l) \mapsto \textit{insert}(m, C(\textit{linkset})(l))\ ]$

$\underline{\textbf{duration}(d)}$:
A duration statement leads to global progress of time, including a time step in the

solver of the continuous model of the environment. This time step will be defined in Definition 4.5.5.

periodic(d) *IS*:
In this case, *IS* is added to the instruction sequence of thread i and a new thread $j =$ *fresh*(C, o_i) is started which repeats the periodic instruction after a duration of d time units, i.e.
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS},\ \mathit{instr}(j) \mapsto \mathbf{duration}(d)\hat{\ }\mathbf{periodic}(d)\ \mathit{IS},\ \mathit{status}(j) \mapsto \mathit{alive}\ ]$

if b **then** IS_1 **else** IS_2 **fi**

- If $[\![\ b\ ]\!](C, i)$ then $C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS}_1\hat{\ }\mathit{tail}(C(\mathit{instr})(i))\ ]$
- Otherwise, $C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS}_2\hat{\ }\mathit{tail}(C(\mathit{instr})(i))\ ]$

while b **do** *IS* **od**:

- If $[\![\ b\ ]\!](C, i)$ then
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{IS}\hat{\ }\mathbf{while}\ b\ \mathbf{do}\ \mathit{IS}\ \mathbf{od}\hat{\ }\mathit{tail}(C(\mathit{instr})(i))\ ]$
- Otherwise, $C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i))\ ]$

return(j):
In this case we have $node(i) \neq node(j)$. Let l be the link which connects these nodes. Then $m = (\mathit{return}(j), C(\mathit{now}) + \delta_{min}(l), C(\mathit{now}) + \delta_{max}(l))$ is transmitted via l:
$C' = C[\ \mathit{instr}(i) \mapsto \mathit{tail}(C(\mathit{instr})(i)),\ \mathit{linkset}(l) \mapsto \mathit{insert}(m, C(\mathit{linkset})(l))\ ]$

Definition 4.5.5 (Time Step) A step $C \longrightarrow C'$ is called a *time step* only if all current threads are ready to execute a duration instruction or have terminated. More formally, for all i with *exec*(C, i), $C(\mathit{instr})(i)$ is $\langle\rangle$ or of the form **duration**$(d)\ \hat{}\ $*IS*. Then the definition of a time step consists of three parts: (1) the definition of the maximal duration of the time step as allowed by the VDM model, (2) the execution of a time step by the solver, leading to intermediate configuration C_s (3) updating all durations of all current threads, committing all variables of the current threads, and dealing with events generated by the solver.

1. Time may progress with t time units if

   - t is smaller or equal than all durations that are at the head of an instruction sequence of an executing thread, and
   - $C(\mathit{now}) + t$ is smaller or equal than all upper bounds of messages in link sets.

   Define the maximal length of the time step t_m as the largest t satisfying these conditions.

2. If $t_m > 0$ the solver tries to execute a time step of length t_m in configuration C. Concerning the variables, the solver will only use the *ioval* field, ignoring the *lval* and *modif* fields. It will only read the actuator variables in *OutVar* and it may write the sensor variables in *InVar* in field *ioval*. As soon as the solver generates one or more events, its execution is stopped. This leads to a new configuration C_s and a set of generated events *EventSet*. Since the solver takes a positive time step, we have $C(\mathit{now}) < C_s(\mathit{now}) \leq C(\mathit{now}) + t_m$. If

$C_s(\mathit{now}) < C(\mathit{now}) + t_m$ then *EventSet* $\neq \emptyset$. Moreover, $C_s(f) = C(f)$ for field $f \in \{\mathit{instr}, \mathit{curthr}, \mathit{status}, \mathit{lval}, \mathit{modif}\}$.

If $t_m = 0$ then the solver is not executed and $C_s = C$ and *EventSet* $= \emptyset$. This case is possible because we allow **duration**(0) to commit variable changes, as shown in the next point.

3. Starting from configuration C_s and *EventSet*, next (a) the durations are decreased with the actual time step performed, leading to configuration C_d (b) transactions are committed for threads with zero durations, leading to configuration C_m, and (c) new threads are created for the event handlers, leading to final configuration C'.

   Let $t_s = C_s(\mathit{now}) - C(\mathit{now})$ be the time step realized by the solver.

   (a) Durations in instruction sequences are modified by the following definition which yields a new function from threads to instruction sequences, for any thread i,
   $\mathit{NewDuration}(C, t_s)(i) =$
   $$\begin{cases} \textbf{duration}(d_i - t_s)\hat{\ }\mathit{tail}(C(\mathit{instr})(i)) & \text{if } \mathit{head}(C(\mathit{instr})(i)) = \textbf{duration}(d_i) \\ C(\mathit{instr})(i) & \text{otherwise} \end{cases}$$
   Let $C_d = C_s[\ \mathit{instr} \mapsto \mathit{NewDuration}(C, t_s)\ ]$

   (b) Let $\mathit{ThrDurZero}(C) = \{i | \mathit{exec}(C, i) \text{ and } head(C(\mathit{instr})(i)) = \textbf{duration}(0)\}$ be the set of threads with a zero duration. For these threads the transactions are committed and the values of the modified variables are finalized. This is defined by two auxiliary functions:
   $\mathit{NewIoval}(C)(x) =$
   $$\begin{cases} v & \text{if } \exists\, i \in \mathit{ThrDurZero}(C) \text{ and } C(\mathit{modif})(x, i) = v \neq \bot \\ C(\mathit{ioval})(x) & \text{otherwise} \end{cases}$$
   Note that at any point in time at most one thread may modify the same global variable in a transaction. Hence, there exists at most one thread satisfying the first condition of the definition above, for a given variable x.

   The next function resets the modified field, for any x and i,
   $$\mathit{NewModif}(C)(x, i) = \begin{cases} \bot & \text{if } i \in \mathit{ThrDurZero}(C) \\ C(\mathit{modif})(x, i) & \text{otherwise} \end{cases}$$
   Then $C_m = C_d[\ \mathit{ioval} \mapsto \mathit{NewIoval}(C), \mathit{modif} \mapsto \mathit{NewModif}(C)\ ]$

   (c) For each event $e \in EventSet$ with $\mathit{evhdlr}(e) = (\mathit{IS}_e, n_e)$, let i_e be a fresh - not yet used - thread identity with status *dormant* and $\mathit{node}(i_e) = n_e$. Then we define an auxiliary function $\mathit{EventInstr}(C)$: *Thread* → *Instr. Seq.* which installs event handlers. For any thread i,
   $$\mathit{EventInstr}(C)(i) = \begin{cases} \mathit{IS}_e & \text{if } i = i_e \text{ for some } e \in \mathit{EventSet} \\ C(\mathit{instr})(i) & \text{otherwise} \end{cases}$$
   In addition, we awake the threads of the event handlers by changing their status. Define, for any i,

$$\mathit{NewStatus}(C)(i) = \begin{cases} \mathit{alive} & \text{if } i = i_e \text{ for some } e \in \mathit{EventSet} \\ C(\mathit{status})(i) & \text{otherwise} \end{cases}$$

Then $C' = C_m[\, \mathit{instr} \mapsto \mathit{EventInstr}(C_m), \mathit{status} \mapsto \mathit{NewStatus}(C_m)\,]$

Observe that $C'(now) = C_s(now) = C(now) + t_s$ with $t_s \leq t_m$.

Definition 4.5.6 (Context Switch) A step $C \longrightarrow C'$ corresponds to a context switch if and only if there exists a thread i which is alive, not running, and has a non-empty program which does not start with a duration, i.e., $\neg \mathit{exec}(C, i)$, $C(\mathit{status})(i) = \mathit{alive}$, $C(\mathit{instr})(i) \neq \text{Ø}$, and $head(C(\mathit{instr})(i)) \neq \textbf{duration}(d)$ for any d. Then i becomes the current thread and a duration of δ_{cs} time units is added to represent the context switching time:
$C' = C[\, \mathit{instr}(i) \mapsto \textbf{duration}(\delta_{cs})\hat{\ } C(\mathit{instr})(i), \mathit{curthr}(\mathit{node}(i)) \mapsto i\,]$

Note that more than one thread may be eligible as the current thread on a node at a certain point in time. In that case, a thread is chosen nondeterministically in our operational semantics. Fairness constraints or a scheduling strategy may be added to reduce the set of possible execution sequences and to enforce a particular type of node behavior, such as round robin or priority-based pre-emptive scheduling.

Definition 4.5.7 (Deliver Link Message) A step $C \longrightarrow C'$ corresponds to the message delivery by a link if and only if there exists a link l and a triple (m, lb, ub) in $C(\mathit{linkset})(l)$ with $lb \leq C(\mathit{now}) \leq ub$. There are two possibilities for message m:

- $\mathit{call}(i, \mathit{oid}, \mathit{op})$: Insert the call in the queue of object *oid*:
  $C' = C[\ q(\mathit{oid}) \mapsto \mathit{insert}((i, \mathit{op}), C(q)(\mathit{oid})),$
  $\quad \mathit{linkset}(l) \mapsto remove((m, lb, ub), C(\mathit{linkset})(l))\,]$
- $\mathit{return}(i)$: Wake-up the caller, i.e.
  $C' = C[\, status(i) \mapsto \mathit{alive},\ \mathit{linkset}(l) \mapsto remove((m, lb, ub), C(\mathit{linkset})(l))\,]$

Definition 4.5.8 (Process Queue Message) A step $C \longrightarrow C'$ corresponds to the processing of a message from a queue if and only if there exists an object *oid* with $\mathit{head}(C(q)(\mathit{oid})) = (i, \mathit{op})$. Let $j = \mathit{fresh}(C, \mathit{oid})$ be a fresh thread and *IS* be the explicit definition of *op*. If the operation is synchronous, i.e. *syn?*(*op*), then we start a new thread with *IS* followed by a return to the caller:
$C' = C[\, instr(j) \mapsto \mathit{IS}\hat{\ }\textbf{return}(i),\ status(j) \mapsto \mathit{alive},\ q(\mathit{oid}) \mapsto \mathit{tail}(C(q)(\mathit{oid}))\,]$
Similarly for an asynchronous call, where no return instruction is added:
$C' = C[\, instr(j) \mapsto \mathit{IS},\ status(j) \mapsto \mathit{alive},\ q(\mathit{oid}) \mapsto \mathit{tail}(C(q)(\mathit{oid}))\,]$

Definition 4.5.9 (Operational Semantics) The operational semantics of a specification in the language of Table 4.1 is a set of execution sequences of the form $C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow \ldots$, where each pair $C_i \longrightarrow C_{i+1}$ is a step (Definition 4.5.3) and the initial configuration C_0 satisfies a number of constraints:

- no thread has status *waiting*;
- on each node, the currently executing thread is *alive*;
- a thread is dormant if and only if it has an empty execution sequence;
- the *modif* field is $\perp$ everywhere;

- all queues and link sets are empty, and
- the auxiliary instruction **return** does not occur in any instruction sequence.

To avoid Zeno behaviour, we require that for any point of time t there exists a configuration C_i in the sequence with $C_i(now) > t$.

4.6 Concluding remarks

A multidisciplinary modeling approach shall provide sufficient means of abstraction to support all mono-disciplinary views in order to be industrially applicable. A solid semantic foundation of the combination of these views is required to support meaningful and reliable analysis of the heterogenous model. We believe that this can be achieved by taking a "best of both worlds" approach whereby the software discipline uses a formal specification technique. Firstly, because it provides abstraction mechanisms that allow high-level specification and secondly because its well-defined semantics provides a platform independent description of the model behavior that can be analyzed properly. Software models as advocated by IBM Rational Technical Developer and I-Logix Rhapsody are, in our opinion, not suited for this purpose in particular because they lack a suitable notion of abstraction, time and deployment. We showed how tool integration can be achieved based on the formal semantics proposed in this chapter, which we applied to a case study. Note however that the approach taken here is not specific to any tool in particular.

Nicolescu et al [77] propose a software architecture for the design of continuous time / discrete event co-simulation tools for which they provide an operational semantics in [41]. Our work is in fact an instantiation of that architecture, however, with a difference. Their approach is aimed at connecting multiple simulators on a so-called simulation bus, whereas we connect two simulators using a point-to-point connection. They use Simulink and SystemC whereas we use 20-SIM and VDM++ to demonstrate the concept. The type of information exchanged over the interfaces is identical (the state of continuous variables and events). They have used formal techniques to model properties of the interface, whereas we have integrated the continuous time interface into the operational semantics of a discrete event system. We believe that our approach is stronger because a weak semantics for the discrete event model may still yield unexpected simulation results even though the interface is proven to work consistently. An in-depth comparison of both approaches is subject for further study.

The interface between the continuous time and discrete event models seems to be convenient when resilience of a system is studied. Early experiments performed in collaboration with Zoe Andrews at the Centre for Software Reliability at Newcastle University have shown that it is possible to use this interface for fault injection [4]. Values and events exchanged over this interface can be dropped, inserted, modified, delayed and so on to represent the failure mode of a sensor or actuator, such as for example "stuck at x". The advantage of this approach is that the failure model can remain orthogonal to the continuous time and the discrete event models. These system models need no longer be obscured by explicit failure mode modeling in either plant or controller, which usually clobbers the specification. We certainly plan to explore this further.

Chapter 5

A Development Process for Embedded Control Systems

The market success of novel high-tech systems depends more and more on the ability to address system-level, multi-disciplinary, design issues. The BODERC project suggests to follow a model-driven design philosophy to address this challenge [47]. The proposition is that reasoning about system-level properties is enabled by creating abstract, high-level and multi-disciplinary models. Analysis of these models will provide valuable insight into the system under construction. This closes the design feedback loop in the early stages of the life-cycle instead of postponing it to the system integration phase. It also prevents late changes in the design with obvious positive effects on time, cost and quality.

But how do we come up with these high-level multi-disciplinary models? What steps need to be taken to create and evaluate them? When do we stop modeling and how do the resulting artifacts relate to the design of the system? It is clear that a suitable process needs to be put in place in order to address these issues. Industrial development of high-tech systems simply requires a step-wise approach because of the inherent scale of the effort involved. Usually, project teams consist of tens of engineers which are possibly working on multiple locations. A development process is a “golden reference” which provides focus to a team, because it suggests a specific order in the activities to perform and it defines the expected output of each activity. Progress can be monitored by observing when specific outputs are delivered while quality can be assessed by inspecting these intermediate artifacts. In other words, a development process is essential for the effective and efficient use of tools, techniques and human resources within a project.

The aim of this chapter is to put the results presented in Chapters 3 and 4 into the context of an industrially viable development process. First, the system-level reasoning approach proposed in the BODERC project is presented in Section 5.1. It is argued how these abstract and high-level models can be used for design oriented activities in sections 5.2 and 5.3. The former section concentrates on the control engineering design process, the latter on the software engineering design process. And finally, the relationship between these processes is assessed and discussed in Section 5.4.

5.1 System-level reasoning

The complexity of the products being developed by industry today is increasing at an astonishing rate. This is partly caused by the advances in the implementation technology used but also by the sheer volume of design related information and its average rate of change. The design process has primarily become an exercise in information management: acquire, evaluate, categorize, prioritize, report and distribute. The efficiency of this activity has a strong influence on the ability to innovate and to meet time-to-market targets. The basis of the system-level reasoning approach suggested in the BODERC project is the CAFCR methodology proposed by Muller in [76]. This methodology is shown in Figure 5.1 and uses a system architectural description decomposed into five related views to structure design information:

"C" the *Customer objectives* view - identifying *what* the customer wants to achieve.

"A" the *Application* view - identifying *how* the customer objectives are to be achieved.

"F" the *Functionality* view - identifying *what* functional and extra-functional properties much be accommodated.

"C" the *Conceptual* view - identifying *how* the functional and extra-functional properties are satisfied.

"R" the *Realization* view - identifying *how* the system concept is implemented.

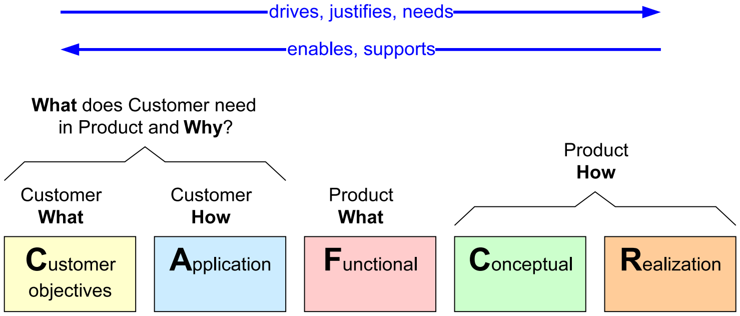


Figure 5.1: Overview of the CAFCR framework from [76]

The framework is focused on capturing the relationship between the customer and the product. The job of the system architect is to safeguard the consistency of these views in order to create a valuable, usable and above all feasible product. However, this work is usually performed within tight industrial constraints, such as project duration and available man power. It is therefore essential to dedicate a significant portion of the available time and effort to the most critical issues. The design methodology for high-tech systems proposed in the BODERC project [50] addresses this particular concern. An iterative approach is proposed which consists of the following three steps per iteration:

1. **Preparation**. The purpose of this step is to gather existing knowledge in order to obtain a good understanding of the product to be developed. It involves making core domain knowledge explicit and identifying dominant realization concerns and system requirements. This provides a level playing field for all the stakeholders involved.

2. **Selection**. The purpose of this step is to select the critical design aspects. It involves the identification of tensions or trade-offs between individual design issues and a qualitative assessment of their associated risk and priority. These can be based on expected customer value, inherent difficulty of the problem or its expected impact on the system as a whole. This approach focuses the attention on the most essential design decisions and prevents spending effort on less relevant issues.

3. **Evaluation**. The purpose of this step is to gain quantitative insight into the identified high priority tensions from the previous step. These results can be obtained either by model-based analysis or by performing measurements on a prototype or comparable system. The aim is to use simple, light-weight, models that can be built and evaluated within hours or a few days at most. This requires finding the appropriate level of abstraction that provides the level of accuracy required. Initially, it may suffice to perform a back-of-the-envelope calculation while more detailed models are required in later stages.

The knowledge gained in the evaluation step is consolidated and evaluated by the design team. Usually, interpretation of the results leads to the identification of new requirements or concerns. Alternatively, the results may give rise to changing the risk or priorities of the critical design aspects. It is clear that iteration is required until the risks associated with each critical design aspect has reached an acceptable low threshold. Note that a design record is built during the iterations which captures the decisions made along the way. These design choices can be rechecked whenever requirements change over time, which is very likely to occur. This is a major benefit over contemporary approaches where usually only the end result of the decision is kept up-to-date, causing significant amounts of rework when requirements change. The industrial strength of the BODERC methodology is based on the embedded process awareness to maintain focus on the important design issues by evaluating risks and assigning priorities. Furthermore, the modeling, measurement and analysis activities are limited in scope to what is essentially required. This time and effort boxed approach provides optimum support for the inherently iterative nature of the design process. The core support techniques used in the BODERC methodology described previously are:

1. the key driver method [48, 49]

2. threads of reasoning [88, 87]

3. budget-based design [38, 39]

These techniques will be discussed in the following sub-sections.

5.1.1 The key driver method

The key driver method is a technique that helps to structure information obtained during requirements elaboration. It presents the relationship between the essential customer objectives and the dominant requirements of the system from the viewpoint of a specific stakeholder as a structured graph. Usually only a few customer objectives, or key drivers, are mapped onto tens of requirements. In other words, it provides a visual presentation of the requirements justification because each requirement can be traced back to the customer objective from which it originates. It is useful for the designers to understand why a requirement is important and how it relates to the other system requirements. The graphical presentation is more convenient for gaining overview, while an exhaustive textual description can be used to capture the rationale. The key driver method covers the CAF-views of the CAFCR framework. The key drivers represent the main customer objectives from the "C"-view and they are placed on the left of the diagram. The system requirements from the "F"-view are presented on the right of the diagram and the application drivers from the "A"-view are placed in the middle of the diagram. Arrows are used to express the relationships, whereby the line width is used to indicate the importance of the relationship. Rows can be used to indicate priority, i.e. top row has highest priority, bottom row has lowest priority and so on. Drawing conventions, such as line color, can be used to increase readability. A square box around a requirements indicates a "must-have" property while a rounded box indicates a unique selling point of the product. An example key driver model of a copier is shown in Figure 5.2.

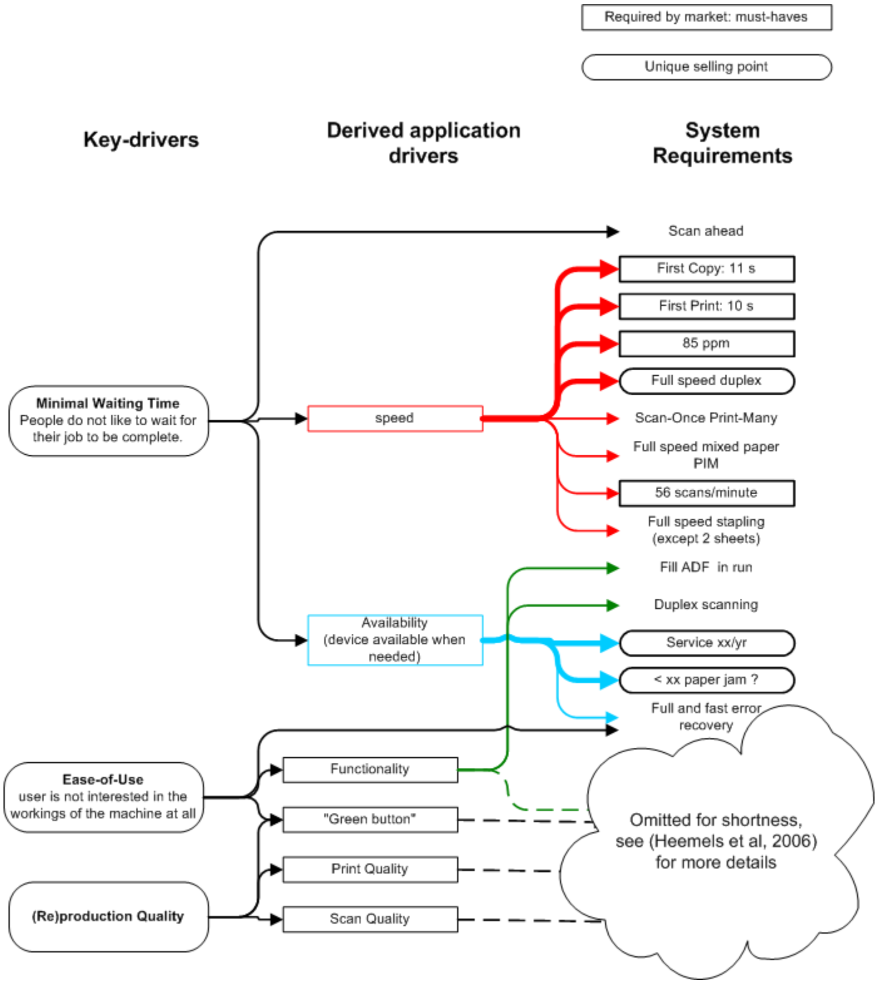

Required by market: must-haves
Unique selling point
Key-drivers
Derived application drivers
System Requirements
Scan ahead
First Copy: 11 s
First Print: 10 s
85 ppm
Full speed duplex
Minimal Waiting Time
People do not like to wait for their job to be complete.
speed
Scan-Once Print-Many
Full speed mixed paper PIM
56 scans/minute
Full speed stapling (except 2 sheets)
Fill ADF in run
Duplex scanning
Availability (device available when needed)
Service xx/yr
< xx paper jam ?
Full and fast error recovery
Ease-of-Use
user is not interested in the workings of the machine at all
Functionality
"Green button"
Print Quality
(Re)production Quality
Scan Quality
Omitted for shortness, see (Heemels et al, 2006) for more details

Figure 5.2: An example key driver model of a copier from [50]

5.1.2 Threads of reasoning

Engineers are typically confronted with many possible realization options at design time. Each potential solution, or design driver, has advantages and disadvantages and usually there is also a significant amount of uncertainty involved. This leads to conflicts in the design. A conflict is defined as the situation where a specific design choice influences one or more design drivers positively, while influencing others negatively. The threads of reasoning technique aims at composing a graphical overview of the conflicts that relate to a number of selected design drivers. These design drivers, the design choices and their consequences are referred to as threads. The method is called threads of reasoning because it reveals the rationale behind the decision making. In other words, the decision making is made explicit and traceable which enables and supports the objective dialogue on complex trade-off issues during the design. Threads of reasoning can use information from all CAFCR-views. An iterative and step-wise approach is used to define and elaborate the threads:

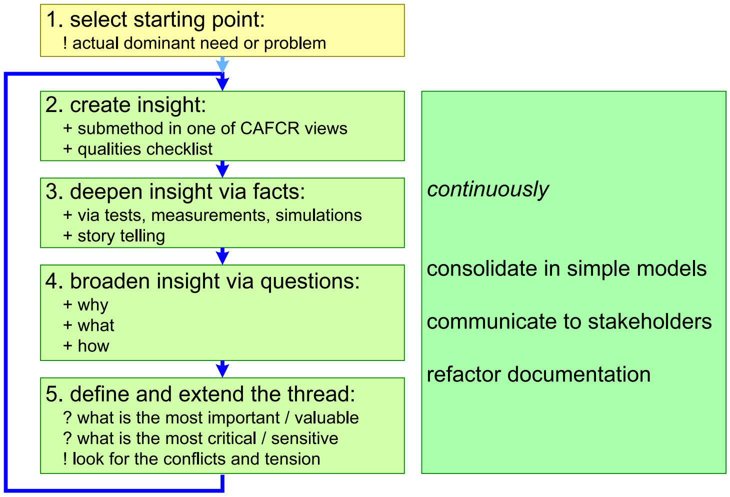

1. select starting point:
! actual dominant need or problem
2. create insight:
+ submethod in one of CAFCR views
+ qualities checklist
3. deepen insight via facts:
+ via tests, measurements, simulations
+ story telling
4. broaden insight via questions:
+ why
+ what
+ how
5. define and extend the thread:
? what is the most important / valuable
? what is the most critical / sensitive
! look for the conflicts and tension
continuously
consolidate in simple models
communicate to stakeholders
refactor documentation

Figure 5.3: Overview of the threads of reasoning approach

1. **Select a starting point**. Typically the most critical design driver is used as a starting point. Whether or not it really is the most critical one is not important because the elaboration process will quickly reveal the true relevance of the design driver. Another design driver will be selected at the next iteration if the importance of the current design driver was initially overrated.

2. **Create insight**. The second step is used to produce an inventory of known facts and questions about the chosen design driver. This information is gathered by means of informal analysis, for example using the story telling technique [76]. The inventory is consolidated and communicated to all stakeholders.

3. **Deepen insight**. The third step is aimed at answering the questions identified in the second step. This can be done by model-based analysis or by test and measurements on existing systems. Back-of-the-envelope calculations or rules of thumb may suffice in the first iteration but usually in-depth modeling and

analysis is required in order to take well-founded design decisions with greater accuracy at later stages.

4. **Broaden insight**. Design drivers are usually closely related and can therefore typically not be studied in isolation. The fourth step is used to establish the volatility of other design drivers with respect to changes in the design driver under study. This may for example influence the risk and priority properties associated with those design drivers.

5. **Define and extend the thread**. The previous steps have generated a wealth of new information and has most likely also uncovered new potential problems as well. Step five aims to organize and prioritize this information and present it concisely to all stakeholders as a thread of reasoning. This requires filtering of the information whereby emphasis is put on decisions that are the most conflicting or have the highest impact for the customer. The objectivity is not lost because the underlying information is still available for inspection.

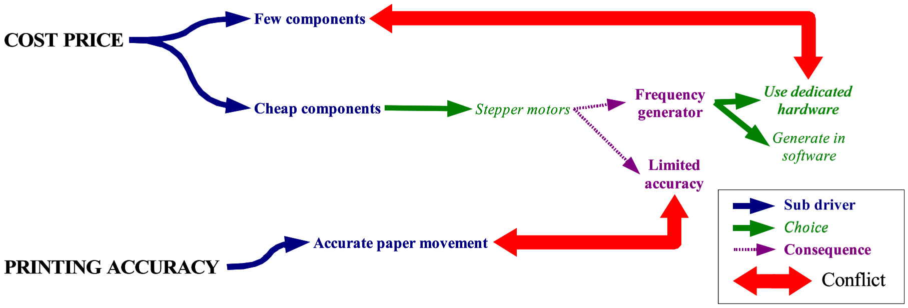


Figure 5.4: An example thread of reasoning in the design of a printer paper path

5.1.3 Budget-based design

Designing distributed embedded real-time systems is difficult because the system has to operate within well-defined resource constraints in order to guarantee deterministic behavior at all times. For example, the performance of the CPU, the bandwidth of the network or the amount of available memory is usually limited. But also power usage or heat dissipation may become an issue. Design complexity is increased significantly if these scarce resources are shared among multiple subsystems or when the resource consumption depends on the dynamic behavior of the system as a whole. This usually leads to conflicts at design time because designers will either try to claim the critical resource or they ignore the embedded restriction entirely in order to meet their local design goal. The BODERC methodology addresses this problem with the budget-based design technique. The general principle is simple. The amount of available resource is the so-called budget. The budget is split into parts and each part is assigned to a specific product function or subsystem. It is the responsibility of the designer of that function or subsystem to allocate the assigned budget to its components, and so on. Making such an explicit budget raises the awareness among engineers that resources are indeed limited available. This causes a healthy design dialogue across all disciplines early in the life cycle whereby a budget request is typically scrutinized by the other users of the critical resource. This may result in exploring other design alternatives, for example if the budget is deemed insufficient. The technique involves four steps:

1. **Scope of the budget**. It is important to define the scope of the budget explicitly, for example to define which parts of the design are included. Add-on modules may be regarded as part of the entire product and thus part of the budget but others may not. Similarly, the applicable operating modes of the product need to be defined appropriately.

2. **Select a decomposition**. The budget needs to be decomposed into parts in order to manage it successfully. There is no universal recipe for defining such a decomposition, but usually physical or functional decomposition is used. Each budget item is allocated to a specific subsystem or function and a responsible designer. This person collects the required performance data and is responsible for meeting the budget.

3. **Find quantitative data**. Finding suitable performance numbers for a subsystem or function can be very difficult because they are in general hard to measure or estimate. Sometimes measurements from existing systems can help but in many cases guestimates or simulations are required in order to obtain quantitative data. The budget-based design approach forces the engineer to think about management of the resources in advance.

4. **Provide a clear overview**. A clear overview must reveal the essence of the budget to the system architect and his stakeholders, preferably in the glimpse of an eye. A good budget has at most tens of quantities listed which are graphically presented. A budget is an evolutionary design artifact which needs to be checked and updated on a regular basis.

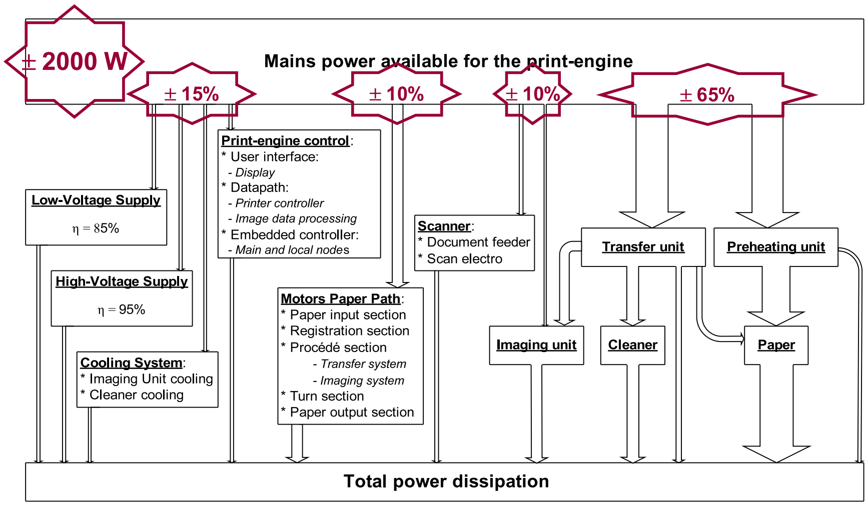


Figure 5.5: Graphical representation of the power budget of a copier

5.1.4 From analysis towards design

The BODERC methodology presented in the previous sections in aimed at supporting the high-level reasoning process in the very early stages of the life-cycle. Model-based design is used to support this activity and the complexity of those models will grow over time. Firstly because the demands for the required accuracy of the models is usually increasing and secondly because the scope of the models typically grows. The purpose of the models evolves from analysis towards design, whereby we enter the realm of the domain specific methods and tools more and more. These methods and tools are presented in the next sections. A typical control engineering development process is presented in Section 5.2 and a software engineering approach is discussed in Section 5.3.

5.2 Control engineering process

Visser and Broenink have taken a control engineering viewpoint and suggest a systematic design trajectory for embedded control systems in [104, 103]. This design trajectory is based on a workflow for the development of embedded control systems which was proposed earlier by Broenink and Hilderink in [14] and is shown in Figure 5.6. First, the dynamic behavior of the system is described in the *physical systems modeling* activity. The models from this phase can be investigated by means of simulation or analyzed mathematically. Second, the *control law design* takes place. The control laws can be studied using simulation or mathematically analyzed, for example with respect to convergence and stability. Third, the *embedded control software design* is created. Usually, some semi-formal notation is used to describe the software architecture and these design artifacts may be demonstrated using simulation or checked mechanically using some static analysis tool. And finally, the *embedded control software realization* takes place. After coding the software, which may be (partly) automated, the implementation is integrated and tested on the real hardware. The embedded control system is validated by exposing it to a significant number of test cases. Iteration is required in each phase if irregularities are exposed during analysis. Larger errors may require moving back to a previous step.

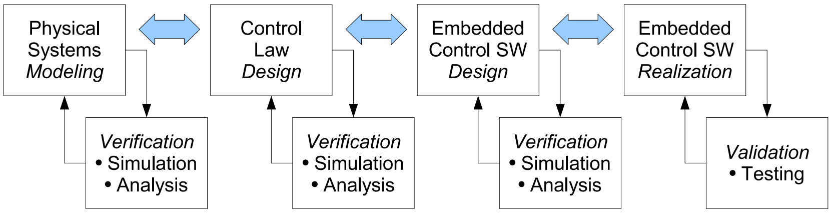

Physical Systems Modeling
Control Law Design
Embedded Control SW Design
Embedded Control SW Realization
Verification • Simulation • Analysis
Verification • Simulation • Analysis
Verification • Simulation • Analysis
Validation • Testing

Figure 5.6: Embedded control system development workflow

The problem with this workflow is that the block arrows in Figure 5.6 between the different phases usually cause a paradigm shift. Modeling and analysis approaches on either side of the arrow are not compatible and there exists no smooth transition from one phase to the next or vice versa. Partial, automated, solutions exist for some of these issues but in general it involves human intervention causing substantial amounts of work. This is expensive because it needs to be repeated whenever an iteration is performed and moreover it is very error prone. In our opinion, this is a blocking factor for industrial application, a show stopper in fact for truly iterative and concurrent design, that must be addressed in order to achieve a significant process improvement. Visser and Broenink propose an alternative in [104]: a step-wise refinement approach that provides a near-seamless transition from model to realization.

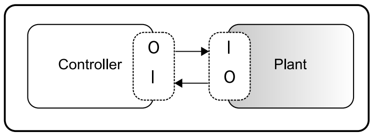


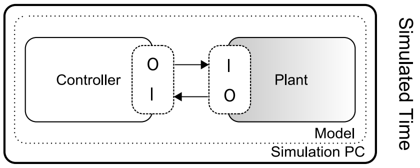
Figure 5.7: Basic embedded control system decomposition proposed in [104]

The usual way to describe embedded control systems is to distinguish the so-called *controller* and the *plant*. The former is the computer system implementing the control strategy, the latter is the physical process that is being controlled. Visser and Broenink propose to add an explicit interface notion to this paradigm, the so-called I/O model, as shown in Figure 5.7. The absence of this particular model in contemporary design trajectories, such as Matlab/Simulink, is one of the reasons for the aforementioned paradigm shifts. Interface requirements are considered too late in these approaches, which usually causes substantial rework when models are elaborated. Maintenance of these models then becomes very costly, for example when requirements change, because at least four models (one controller-plant pair with and one pair without I/O model) need to be updated. Furthermore, the introduction of an explicit interface model introduces a higher level of abstraction at the model level: we move from mere wire connections to strongly typed ports which exchange flows or signals. This allows for seamless modeling in port-based design methods, such as bond graphs [13].

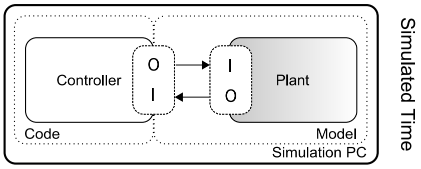
The design trajectory based on this enhanced system decomposition strategy provides the basis for a step-wise, model driven, approach to move from model towards implementation, whereby both the controller *and* the plant will evolve. Various so-called "*x*-in-the-loop" simulation techniques are used to check whether each elaboration step is still compliant to the high-level system-level requirements. This provides a *gradual* introduction of increased system complexity instead of the "direct-to-target" approaches advocated by the leading tool vendors. Furthermore, the design trajectory is easier to manage because explicit decisions are made when, where and how complexity is added. In each step, model abstractions are replaced by their elaborated counterparts, which can then be subjected to analysis. This leads to less errors because attention of the designer is very focused whereby development time can be time-boxed in order to ensure overall productivity. The development trajectory consists of six stages, which are summarized in Table 5.1.

The purpose of this design trajectory is not to create the optimal controller for the given plant but to optimize the design of the complete system. As mentioned before, the explicit structure of this process ensures a consistent view of the I/O throughout the design trajectory. The trajectory is roughly split into two parts, the *simulated time* and the *real-time* part. In the former (stages 1-3), we have total control over the notion of time on either side of the interface. Progress of time at the system-level can be managed explicitly. This enables for example faster than real-time simulations required to obtain design feedback quickly in the early stages of the life-cycle. In the latter case (stages 4-6), the so-called wall-clock, which is the locally perceived notion of time, is equal to a physical clock in the real world on at least one side of the interface. The synchronicity to this physical clock dictates the system-level notion of time.

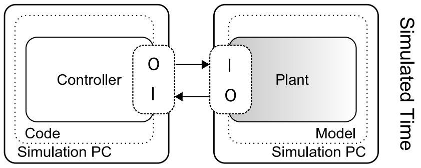
Stage 1 – Embedded Control System specification using state-of-the-art dynamic systems modeling tools, whereby the interface layer is explicitly modeled.



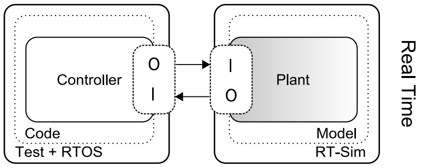
Stage 2 – Software artifacts are created for the controller, whereby "Software-in-the-loop" co-simulation is used to check the control code against the (unmodified) plant model.



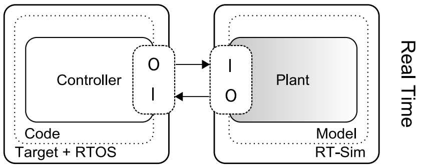
Stage 3 – The controller and plant models are executed on separate computers, which enables CPU usage estimation for the control code and validation of the interfaces.



Stage 4 – The controller code is moved to the real-time *test* platform and the plant model is executed on a real-time simulator.



Stage 5 – The controller code is moved to the real-time *target* platform and the plant model is executed on a real-time simulator.



Stage 6 – The controller code, running on the target platform is connected to the physical plant. Test scenarios are executed to validate the total system.

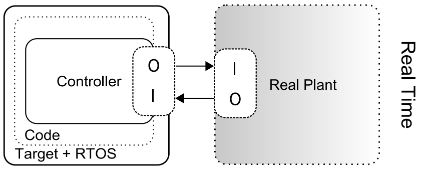


Table 5.1: Control system design trajectory proposed by Visser and Broenink in [104]

The transition from stage 3 to 4 should therefore be addressed with care. In stage 4 we run a simulation of the plant in real-time. The time required to perform the computation to update the state of the plant model is very important. It is firstly bounded by the dynamics of the continuous time model and secondly by the imposed controller constraints. The first point implies that the reciproke of the maximum computation time should be less than half of the highest possible frequency that may occur in the continuous time model. The second point is determined by the style of control. In the case of time-driven (synchronous) control, the computation time should not exceed the smallest possible sample time interval length. In the case of event-driven (asynchronous) control, the computation time should not exceed the required maximum observation delay for an event. The strongest of these bounds should be taken, otherwise we possibly introduce unwanted system behaviors. The usual approach to limit these effects and to circumvent the associated problems is to assume a fixed time delay between action and observable reaction. This can be modeled as a simple time delay in stages 4-5, as shown in Figure 5.8. In addition, if time-driven control is applied then the use of fixed step-size solvers is advocated for both controller and plant, whereby this time delay corresponds to the step size of the continuous time solver. The time delay is removed in stage 6 since the real plant reacts infinitely fast.

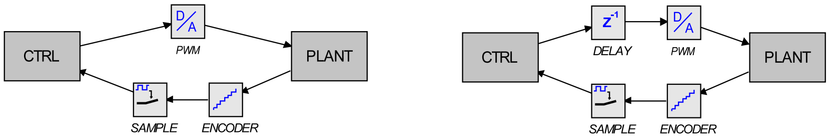


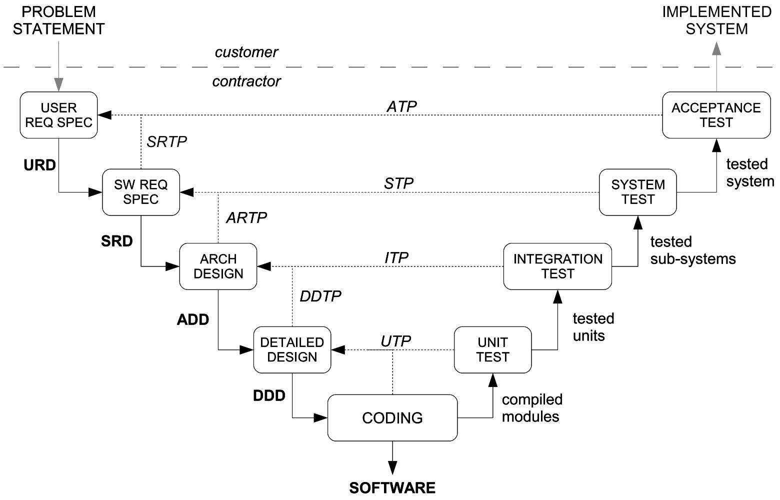
Figure 5.8: Standard (left) and elaborated (right) I/O model in stages 4-5

Although the design trajectory proposed by Visser and Broenink is a significant step forward from the control engineering perspective, it does not necessarily address the needs of the engineer that produces the software artifacts in stage 2. The implicit assumption in the approach of Visser and Broenink is that the control code is automatically generated from the dynamic system models and this is in practice quite problematic. The primary cause for this problem is that the amount of code for the controller is usually only a very small portion of the total application software, in particular in the area of high-tech systems. Furthermore, the techniques used to design these other parts of the application software, are not well suited for modeling control applications, and vice versa. In order to understand this mismatch, we need to look at the software development processes in more detail.

5.3 Software engineering process

The basis for most industrial software engineering processes is the well-known "V"-model as shown in Figure 5.9. We use the ESA PSS-05 definitions here for convenience, but this standard is comparable to other industry standards such as ARP 4754 [43] and DO-178B [90]. The "V"-model provides a step-wise approach for the development of software. Originally, it was regarded as a single-shot process whereby the deliverables from one phase where the input to the next phase. This process is usually referred to as the "waterfall" approach which was suggested by Royce in [86]. In many cases, this cascade of intermediate design artifacts, which increase in number, size and complexity as the project evolves, typically leads to severe productivity losses since more effort is required to create and maintain them as time progresses. Brooks [15] and Johnson [60] have shown that the key problems with this approach are: lack of project and product overview and lack of problem and client interaction. Alternate approaches are suggested to overcome this situation. For example, Boehm proposed the so-called "spiral development" model in [11]. He suggests to deliver the total system using increments whereby the scope of the system is enlarged after each iteration. Selection of the extended scope is driven by explicit risk analysis, whereby he proposes to attack the highest risk components first. Within each iteration, the waterfall process is being used but this is far less problematic due to the restricted scope per iteration.

Most modern software development processes such as the Rational Unified Process (RUP) are inspired by this approach [64]. The related Dynamic Systems Development



	DESIGN ARTIFACTS		*V&V ARTIFACTS*
URD	User Requirements Document	SRTP	Software Requirements Test Plan
SRD	Software Requirements Document	ARTP	ARchitecture Test Plan
ADD	Architecture Design Document	DDTP	Detailed Design Test Plan
DDD	Detailed Design Document	UTP	Unit Test Plan
		ITP	Integration Test Plan
		STP	System Test Plan
		ATP	Acceptance Test Plan

Figure 5.9: The “V”-model as described in ESA PSS-05-10 [28]

Method[1] (DSDM) has been notably successful applying this iterative development process, in particular in the area of database enabled client-server applications. Explicit priority-based requirements management, time-boxed system development and tailored tool support are the key success factors for this approach. Many of today’s web applications have been developed using these techniques and this has been a major contributor to the interest in so-called agile system development. Iterative design is also acknowledged in the field of high-tech systems as can be observed in the ECSS-E-40[2] standard, developed jointly by the European Space Agency and industry. However, these light-weight management processes have not yet demonstrated their added value in practice. We believe that this is mainly due to the fact that the equivalent of the relational database paradigm, which has been the driving force behind the success of the rapid application development techniques described earlier, is *missing* in this particular application field. There is no commonly accepted way to characterize the class of distributed real-time embedded control systems. The Unified Modeling Language [3] (UML) seems to gain industrial acceptance because of its flexibility and versatility and the notation has become more mature with the recent release of version 2.0 of the standard. Several domain specific modeling extensions have been proposed, the so-called *profiles*, but their industry acceptance is still rather poor, mainly because tool support

[1] See *http://www.dsdm.org*.

[2] See *http://www.ecss.nl*.

[3] See *http://www.uml.org*.

is lacking. The Model Driven Architecture [4] (MDA) philosophy that is proposed by the Object Management Group (OMG) to address this issue is still far from mature.

We propose to use formal description techniques to address this challenge. This class of languages, with a well-defined syntax and semantics, are used to create models which can be subjected to rigorous formal analysis. The results from the analysis drives the, usually informal, reasoning process about the system and its properties, which in turn drives the elaboration of the formal model. In other words, formal methods are inherently model-driven. This approach usually exposes potential problems in early phases of the design and moreover, explicitly describes the way these issues are addressed. This contributes positively to the increased quality and reduced cost of the system development. We have demonstrated in the previous chapter that this is technically feasible for the development of distributed embedded real-time control systems, even in the early stages of design, using a combination of VDM++ and bond-graphs. VDM++ is a general purpose model oriented formal specification language that is well-suited for the description of large-scale industrial systems. Furthermore, the industrial grade tool support enables round-trip engineering with UML, which provides a bridge towards the currently accepted mainstream software engineering practice in industry. But so far, the impact on the development processes have not yet been considered.

Mukherjee, Larsen and Verhoef and others have proposed a set of guidelines for the specification, analysis and development of real-time systems using VDM++ in [21]. These guidelines are inspired by [85] and extend the more traditional specification process described in [30]. This guideline document provides a step-wise plan comparable to the approach taken by Visser and Broenink for control applications. Each step elaborates the previous step by focusing on certain aspects of the system. Complexity increases over time because the system scope grows due to this refinement. This process remains manageable due to the abstraction mechanisms offered by the formalism and the powerful tool support for analysis. The steps are summarized here for convenience.

- **step 1** : *Requirement capture* (URD / SRD in Figure 5.9).
  Requirements elicitation is performed using a mix of formal and informal techniques. Typically, Use-Case analysis is performed whereby critical parts of the system are formally modeled, in our case using VDM. The resulting specifications are at a high-level of abstraction, focusing on the key properties of the system. Specifications do not necessarily reflect a particular system structure, in order to prevent design bias at the requirements level. These formal specifications can be subjected to rigorous analysis using powerful tools to ensure their internal consistency and integrity. The same techniques can also be used to validate the requirements, for example by testing, model checking or formal proof. Conjectures about the system are formulated and the analysis shows whether or not, or under which circumstances, these properties hold. Usually it takes several iterations to obtain a consistent view of the system. The level of formality that is applied follows from the type of application that is being developed. In our vision, and to stimulate industrial uptake, a light-weight approach should be followed whenever possible. In this thesis we use prototyping, simulation and testing as our means to validate the formal models. The resulting test suite forms the basis of the system acceptance test, which closes the “V”-cycle already from day one.

- **step 2** : *Software architecture - static structure* (ADD in Figure 5.9).

[4] See *http://www.omg.org/mda*.

The formalized requirements from step 1 are transformed into a potential system architecture. Again a mix of formal and informal techniques are used to specify the system, whereby we focus on its static structure. This approach follows the usual object-oriented design paradigm. What sub-systems can we identify and what are the interfaces between them? How do the interfaces relate, what services do they offer and what kind of data is exchanged between them? The result of this step is a description of the design in UML, whereby critical parts of the model are elaborated in VDM++, in particular for the sequential behavior of the identified operations in the model. Round-trip engineering tool support is available to enforce consistency between the UML and VDM++ models and to generate consistent documentation directly from these models. The VDM++ models can again be subjected to rigorous formal analysis, whereby the validation suite from step 1 drives the validation of this step. Note that an explicit environment model is required to assess the system model, in particular for embedded systems. The rigorous analysis and the different view-points taken in this step usually highlight additional requirements that will force iteration back through step 1. The test suite has evolved as well. It will reflect the chosen system structure defining tests to be performed at the sub-system level for example. Note that we move down on *both* sides of the "V"-cycle at the same time.

- **step 3** : *Software architecture - dynamic structure* (ADD / DDD in Figure 5.9). The architecture which was designed in step 2 will be adapted to reflect the dynamic behavior of the system. Concurrency considerations drive the selection of so-called active and passive classes in the design which may require restructuring parts of the static architecture. The latter forces iteration through steps 1 and 2 but the impact is usually rather limited because the complex sequential behavior remains intact in most cases. The power of rigorous formal analysis is perhaps best demonstrated in this phase because absence of e.g. dead-lock is notoriously hard, if not impossible, to demonstrate using informal means.

- **step 4** : *Hardware architecture and software deployment* (DDD in Figure 5.9). Real-time embedded systems usually operate in a constrained environment which limits the implementation options. The purpose of this step is to deploy the software architecture from step 3 onto some abstract representation of the possibly distributed hardware architecture and to enrich these models with performance data. The combined hardware, software and environment models can then be used for performance analysis, such that system fitness for purpose can be assessed prior to implementation. This step enables investigation of many extra-functional system properties such as timeliness and throughput. The designer may wish to modify either the software or the hardware design if the expected performance is not within some safe margin of error of the required value. The former forces iteration through steps 1 to 3 if it cannot be solved by means of changing the hardware architecture or the deployment.

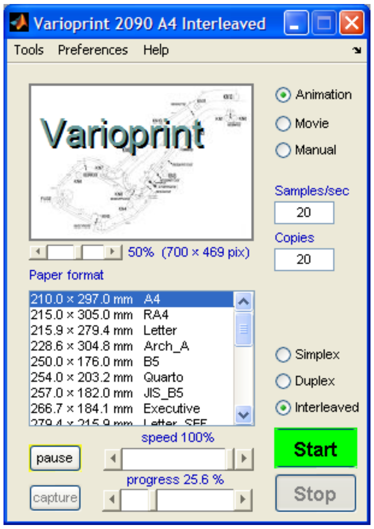
- **step 5** : *Implementation* (software in Figure 5.9). The system design that results from step 4 provides the baseline for the distributed implementation. The level of abstraction of the specification is lowered by repetitive refinement until a fully explicit model is available. This model is translated into the target implementation language, either by manual coding or automated code generation. Each refinement step is checked by means of the validation suite already developed, component by component. It is equally important to check the extra-functional properties of the implemented component on the target platform after the functional correctness has been demonstrated. This may force iteration through the previous steps, if the measured performance does not conform to the assumptions made in step 4.

5.4 Discussion and conclusion

A system-level reasoning approach was already presented in Chapter 2 of this thesis. The so-called "Y-chart", as shown in Figure 2.1, is used for the evaluation of performance properties of embedded system architectures. The approach taken in the BODERC project is related but it is not specific to performance modeling, it has a substantially wider scope. The BODERC methodology [50] covers requirement capture towards design while the Y-chart assumes the system requirements to be known a priori and focuses on design space exploration as the primary activity. The aim of the BODERC methodology is to gain focus on the most critical system aspects as soon as possible, using simple models that create insight into a design issue within a reasonable amount of time.

Perhaps best proof of the success of this approach is provided by Beckers, Heemels, Bukkems and Muller in [6]. They have developed a high-level simulation model of the Varioprint paper path using Matlab/Simulink. A topological view of the paper path can be abstracted directly from the the original mechanics drawing in just a few hours by means of a purpose built support tool. This two dimensional topological view, the so-called track, is then enriched with the position information of the pinches, sensors, switches and the connection between motors and pinches. A timing table is provided for each motor detailing its predicted angular speed at each moment in time. Simple logical rules can be added to influence the timing table contents. Together with job and sheet information, such as number of pages to print, inter-sheet distance and the page size per sheet, this provides sufficient information for a coarse grain simulation. The result is a set of position-time and velocity-time diagrams. These can be used to check whether sheets are, for example, hitting each other or overlapping. Moreover, the movement of sheets can be animated, superimposed over the original CAD drawing, as shown in Figure 5.10. This provides valuable input during the design of the paper path layout. Trade-offs between spatial layout, positioning of sensors and actuators and sheet scheduling can be investigated. The light-weight modeling approach enables a turn-around time of just a few hours which supports the interactive nature of the design process in these early stages. Furthermore, the model can be extended easily to address other concerns.

Orbons claims in [47] that this approach has already led to a significant cost and time reduction at Océ. A complete engineering prototype and development iteration could be skipped, saving many man-years of effort. The success of this modeling approach was largely due to the conscious simplification of the model, where only the desired behavior of a sheet and ideal movement of parts is considered. All disturbances and variations of actual hardware performance are ignored. However, the most critical effects are taken into account, such as software and actuation delays and maximum acceleration and deceleration rates. This provides valuable input to downstream engineering activities that can follow a budget-based design approach to meet these identified margins of error. However, this so-called "Happy Flow" kinematic model does *not* provide guidance on *how* to design the embedded control system software that controls the paper path. Neither does it assist in deciding what to do if the dynamic effects of



Varioprint 2090 A4 Interleaved

○ Animation ● selected

- ● Animation
- ○ Movie
- ○ Manual

Samples/sec: 20

Copies: 20

50% (700 × 469 pix)

Paper format

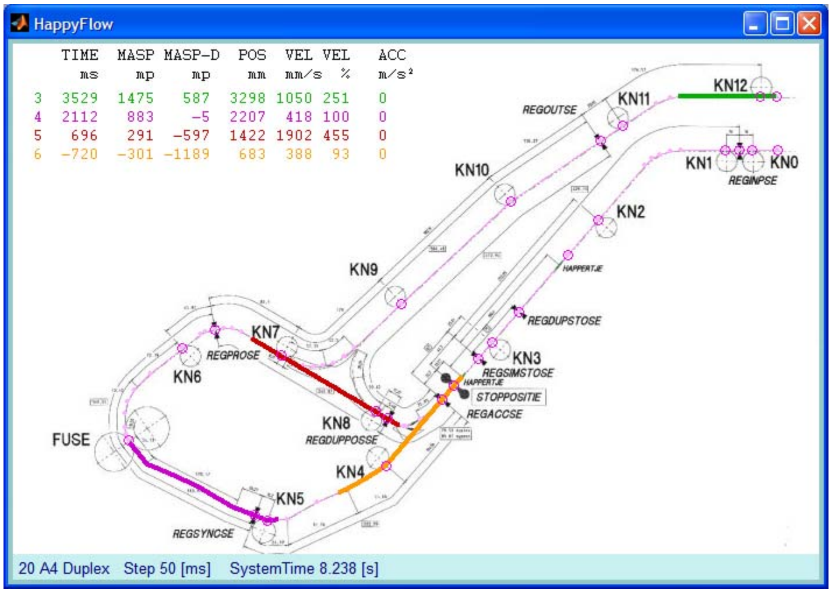
210.0 × 297.0 mm	A4
215.0 × 305.0 mm	RA4
215.9 × 279.4 mm	Letter
228.6 × 304.8 mm	Arch_A
250.0 × 176.0 mm	B5
254.0 × 203.2 mm	Quarto
257.0 × 182.0 mm	JIS_B5
266.7 × 184.1 mm	Executive
279.4 × 215.9 mm	Letter_SEF

- ○ Simplex
- ○ Duplex
- ● Interleaved

speed 100%

progress 25.6 %

(a) user-interface of the simulator



HappyFlow

```
     TIME  MASP MASP-D   POS   VEL VEL    ACC
       ms    mp     mp    mm  mm/s   %   m/s²
  3  3529  1475    587  3298  1050 251    0
  4  2112   883     -5  2207   418 100    0
  5   696   291   -597  1422  1902 455    0
  6  -720  -301  -1189   683   388  93    0
```

20 A4 Duplex Step 50 [ms] SystemTime 8.238 [s]

(b) animation of the kinematic model

Figure 5.10: The “Happy Flow” model of a printer paper path from [6]

some sub-system exceed the allocated design margins from the kinematic model. The solution is to build a truly dynamic system model, but with virtually the same level of abstraction, flexibility and ease of use as the “Happy Flow” kinematic model.

Chapter 6

Embedded Control of a Printer Paper Path - a Case Study

6.1 Introduction

We introduced the extension of the VDM++ language with an explicit notion of system architecture and deployment in Chapter 3 and we looked at co-simulation of discrete event VDM++ models with continuous time Bond graph models in Chapter 4. Furthermore, the development process of embedded control systems was considered in Chapter 5. We will now apply these partial results to a case study: the paper path of an office printer, as presented in Figure 6.1. The purpose of this case study is to demonstrate a typical design flow for such a system, using the method and techniques proposed in the previous chapters. We focus on modeling and analysis of an important subsystem of a printer: the control system of the so-called paper path. This subsystem is responsible for the internal logistics, it is in charge of managing the flow of sheets through the printer. Therefore, it directly influences the overall productivity and quality of the system as a whole. In nowadays office printers, the paper path can be very complex because of the physical layout, the operating modes (simplex, duplex or mixed mode printing) and different paper sizes that need to be supported, even within a single print job. Throughput and time-to-first-print are important design drivers, whereby 50-100 pages per minute and 5-10 seconds are realistic values for current products in the mid-range market. It is not uncommon that the paper path control system design needs to cater for a whole range of products or even a product family.

One of the most important design criteria however, is the so-called sheet-to-image synchronization. We use the term *sheet* to denote the physical piece of paper that is transported through the printer and we use the term *image* to denote the bitmap that is ultimately printed on the sheet. The former implies the process of managing moving objects and the latter implies the process of managing large volumes of data. At some point in time, these processes need to synchronize, such that the first line of the image is exactly printed near the leading edge of the sheet. The accuracy of this synchronization is determined by the tolerances of all subsystems. It is the primary task of the paper path control system to ensure that this tolerance is dynamically kept within a predefined range, typically expressed in terms of several micrometers.

Printing is the process whereby dry ink particles, the so-called toner, is transferred to the paper and fused. In the system we study here, fusing is performed by applying

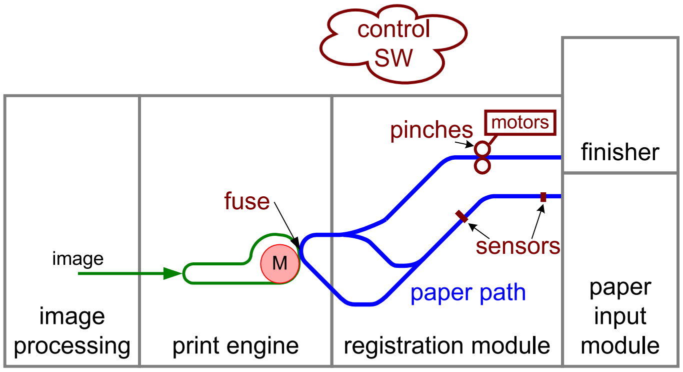


Figure 6.1: Schematic overview of the Océ Varioprint 2090 printer from [47]

both heat and pressure, whereby the toner melts and bonds to the paper. This process is complicated by the fact that transferring the toner to the paper is in fact a two-stage process, as presented in Figure 6.2. The first stage is the so-called *cold process*. The printer contains an endless (but not seamless) optical photoconductive belt, or OPC, that is continuously rotating and is uniformly electro-magnetically charged. The belt passes a light-emitting diode (LED) array that covers the width of the belt. The digital bitmap image is transferred onto the photoconductor belt, line by line. Each pixel in the image line causes a LED to turn on. The emission of light from the LED removes the charge from the photoconductive belt locally. And vice versa, the photoconductive belt remains charged if the LED was not turned on when passing the LED array. The belt is exposed to the toner and the toner will stick to the places where charge is still left on the photoconductor. The toner image is then transferred to the second stage, the so-called *warm process* and finally the photoconductor is cleaned and recharged, ready for the next round.

The warm process consists of a seamless and endless rubber belt, often referred to as the toner transfer belt, or TTF, that is rotating with the same speed as the photoconductor belt. The TTF is heated and since these two belts are touching at some point, toner will stick to it temporarily. Hence, the toner image is transferred from the OPC to the TTF. The rubber belt is then forced through a so-called warm fuse pinch, where it meets a preheated sheet of paper. The distance between the two rolls of the fuse pinch is kept marginally smaller than the thickness of the rubber belt. The rubber belt and the paper are forced through the slit. The elasticity of the belt causes the force necessary to fuse the toner image on the sheet and finally the TTF is cleaned, ready for the next round.

The resulting quality of the image on paper is determined by the properties of the total fusing process, which is an intricate interplay between the mechanical, electrical, physical and chemical sub-processes as described above. Stability of the fusing process is important to guarantee consistent printing quality. It is therefore the dominant driver in the design of the overall control architecture. The photoconductor belt and TTF run at a constant speed to avoid fluctuations as much as possible. The LED array is synchronized on the measured speed of the belt to ensure that all image lines are equidistant and to avoid the seam on the photoconductor. The image processing equipment follows this as a slave process, it has to ensure that the bitmap data for each line is supplied in time. Similarly, the paper path sub-system has to ensure that the sheet

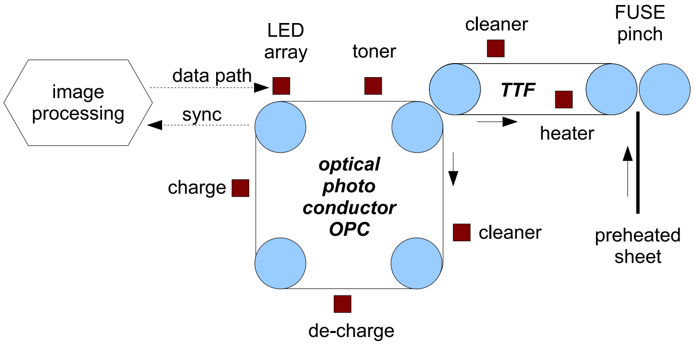


Figure 6.2: Schematic overview of the two-phase printing process

of paper is available at the fuse pinch *on time* and with the *right speed*. Otherwise, the image would be misaligned and the fuse pinch would be polluted. This high-level control approach is called *sheet-follows-image*.

The case study described in this chapter abstracts away from the complexity of the image transfer process entirely. We assume it works nominally here, it runs at some preset, constant, speed. Instead, we focus on the derived requirement imposed on the paper path sub-system to deliver sheets at the right speed at the right moment in time at a particular location. The paper path sub-system is a so-called *mechatronic* system that is composed of many parts. It consists of a number of so-called *pinches* that are used to transport the paper. Each pinch consists of a set of rubber rolls on a metal axis each, whereby the rubber rolls touch each other. The paper moves in between those rolls due to friction. The pinches are set along the paper track, whereby the distance between two pinches is chosen to be less than the smallest paper size dimension that the printer needs to support. This ensures that each sheet is always in control of at least one pinch. Special care has to be taken if a sheet is in multiple pinches at the same time. The paper would either tare or blouse if these pinches would rotate at significantly different speeds.

The pinches are driven by electric motors, typically brushless direct current (BLDC) or stepper motors. The motors are connected to the pinches using a toothed rubber belt over cogged wheels. A drive ratio can be realized by using different diameter cogged wheels. Uni-directional bearings, which allow free rotation in one direction, can be used to circumvent the multiple pinch problem described previously. Sometimes, a single motor and rubber belt drives multiple pinches, in order to save cost. Deciding which pinches to combine is an important task during the design, as is the placement of the paper sensors. Usually, paper sensors are optical sensors that are able to detect the leading and trailing edge of each sheet as it moves through the paper path. Their position is usually a trade-off between mechanical and control requirements. In office equipment, space is usually a scarce resource and sensors need to be mounted such that they can be serviced if needed. However, the position of each sensor determines its control effectiveness and computational demands. Sensors located close to some critical control objective might require potent computers in order to live up to the short response times and sensors located far away may impede the required control accuracy. In many cases, this leads to complex multi-disciplinary design tradeoffs.

Organization of this chapter. The purpose of this chapter is to demonstrate how these design challenges can be supported using the methods and techniques presented in the previous chapters. First the experimental set-up that is used in our experiments is presented in Section 6.2. The modeling approach and the models themselves are discussed in Section 6.3. Last but not least, we present the results from our analysis in Section 6.4 and we draw some conclusions from our work in Section 6.5.

6.2 The paper path experimental set-up

The paper path we study in this chapter is inspired on the Océ Varioprint 2090 mid-range black and white office printer, which is presented in more detail in Figure 6.3.

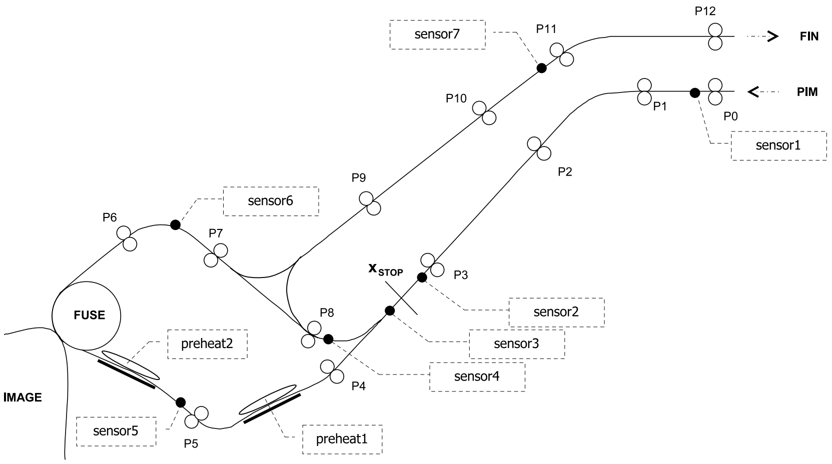


Figure 6.3: Schematic overview of the Océ Varioprint 2090 paper path

The paper path consists of twelve normal pinches, marked P0 to P12 and seven sensors, marked as closed black dots. In addition, two special pinches, so-called pre-heaters, are available. They are positioned close to the fuse pinch. Sheets are inserted into the paper path by the paper input module, or PIM. These are typically stand-alone units that have a simple interface to request each single sheet. This sheet will be delivered with a certain margin of error and the paper path has to compensate for the timing tolerances imposed by this unit. The time difference can be measured by SENSOR1. When the sheet has reached SENSOR2, the sheet will be stopped by pinch P3 and aligned. Alignment is required to compensate for skewed sheets and the alignment unit will ensure that the four corners of the sheet are equidistant to the heart line of the paper path. The sheet is then accelerated towards P4 and passes the first pre-heater. Pinch P5 is the last pinch that can compensate for tolerances in speed and position of the sheet. SENSOR5 is used to detect the final margin of error of the leading edge of the sheet. The compensation has to be completed before the leading edge of the sheet enters the second pre-heater, because it is physically coupled to the motor that drives the fuse pinch and therefore always runs at the same speed. After the fuse pinch, the sheet is either ejected to the finisher (FIN) via P9-P12 or it is re-inserted into the paper path via the duplex loop at P8. Pinch P3 will catch, stop and align the sheet before moving it down again towards P4 for the second run. Note that the actuator at P7 to force simplex or duplex printing is not shown in Figure 6.3. Observe that double sided printing needs to be carefully planned ahead of time, to prevent hitting sheets coming from the paper input module. It is obvious that designing the control algorithm for managing this subsystem productively is challenging.

Selection of the appropriate embedded hardware and software architecture to support such a system is equally difficult and important because it has a potential high impact on both cost-price and performance. It is clear that a mixture of hard- and soft real-time task needs to be accommodated by the system, whereby the physical decomposition into subsystems need to be taken into account. A distributed architecture, consisting of several networked computers, is typically suggested as a way to achieve *separation of concerns*. However, the economic drive towards low-cost price suggests maximum integration instead, to reduce the number of components as much as possible. The tension between these two extremes is complicated even more when development and life-cycle requirements are taken into account. Separate computers for each task would allow for maximum decoupling of development teams which would enable concurrent development to improve the time-to-market whereas a single common platform could complicate this. Capital goods, such as printers, are usually designed as a product family, whereby different product configurations cater for different market segments but are typically based on a common design and architecture. A low-cost product would perhaps require a highly integrated platform while a high-end product would need a much more potent, and possibly distributed, architecture. Last but not least, design of these complex systems can take several years, therefore spanning several technology generations. Adoption of new embedded control solutions mid-project is no exception in order to keep competitive. A flexible way to investigate and accommodate these kind of changes is of course most helpful.

Unfortunately, support for these kinds of system-level design trade-offs is very limited in practice. Design tools are often targeted towards a single technology, architecture or discipline. In the BODERC project [47], we proposed to create abstract, high-level and multidisciplinary models of the system under construction. This model driven design approach allows for early impact analysis of proposed solutions. An experimental set-up was created that allows research on the design and analysis of the paper path sub-system. The experimental set-up was build by the Control Engineering group of the department of Electrical Engineering, Mathematics and Computer Science at the University of Twente [3]. The set-up is in fact a simplification of the Varioprint 2090 paper path, but it exhibits the same basic control challenges. We will now provide a description of the experimental set-up, as presented in Figure 6.4.

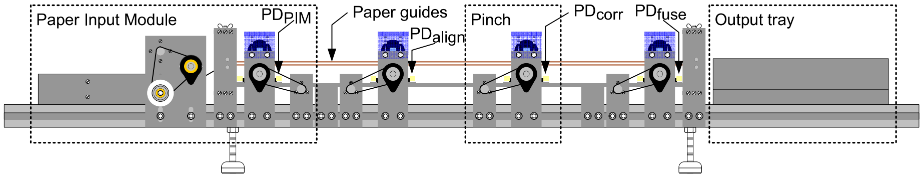


Figure 6.4: Schematic overview of the experimental set-up from [3]

The system can simulate single-sided printing and it is composed of a simple paper input module, which includes a motor-pinch subsystem with a uni-directional bearing and a paper path composed of four pinches. Each pinch on the paper path is driven by a separate electric brushless DC motor. This motor is connected to the pinch by means of a toothed rubber belt over two cogged wheels. Each motor has a built-in rotation sensor, a so-called quadrature encoder which measures the angular position. This sensor delivers a pulse train whereby the frequency corresponds to the rotation speed of the motor axis. The sign of the signal encodes the direction of rotation, clockwise or anti-clockwise. Sheets are delivered into a passive paper tray at the end of the paper path. Four optical sensors, referred to as so-called paper detectors, are mounted in the experimental set-up to observe the leading and trailing edges of the sheets. Each pinch is assigned a specific task:

- The first and second pinch act as the paper input module. Their task is to supply single sheets with a constant speed. They corresponds to the pinch P0 in Figure 6.3. A paper detector is mounted right after the second pinch to simulate SENSOR1 and we will refer to it as PD_{PIM}.

- The third pinch acts as the alignment pinch and it corresponds to pinch P3 in Figure 6.3. Its task is to align the sheet. This process is simulated by temporarily stopping each sheet for a specified amount of time. A paper detector is mounted right after the third pinch to simulate SENSOR2 and we will refer to it as PD_{ALIGN}.

- The fourth pinch acts as the correction pinch and it corresponds to pinch P5 in Figure 6.3. Its task is to ensure that each sheet is delivered on time and with the right speed at the fuse pinch. A paper detector is mounted right after the fourth pinch to simulate SENSOR5 and we will refer to it as PD_{CORR}.

- The fifth pinch acts as the fuse pinch from Figure 6.3. This pinch rotates at some preset and constant speed. A paper detector is mounted right after the fifth pinch and we refer to it as PD_{FUSE}.

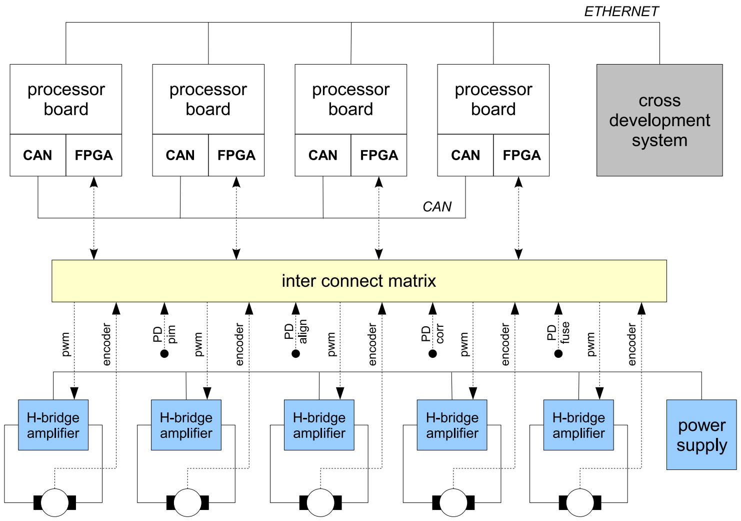


Figure 6.5: Overview of the embedded control system architecture

The experimental set-up is controlled by a distributed embedded control system as shown in Figure 6.5. It consists of four nodes. Each control node consists of a processor board, a Controller Area Network (CAN) field bus interface and a field programmable gate array (FPGA) board, all using the PC104 form factor. The processor board contains an Intel x86 compatible processor with a built-in VGA graphics adapter, flash and DRAM memory and an Ethernet network interface. The processor board can run any operating system, in the case study presented here RTAI real-time Linux is used. TCP/IP over Ethernet is used for on-target debugging, remote file system access and software download. The development station is connected to all nodes over this interface. The CAN and FPGA interfaces are directly connected to the PC104 ISA interface of the processor board. The CAN field bus is used to communicate between the nodes when executing the embedded control application. The FPGA board is used to implement low-level input and output interfaces, for example to handle the quadrature encoder signal and to generate the pulse-width modulation (PWM) signal for each motor. Furthermore, the FPGA I/O board enables high speed and high accuracy measurements that are completely independent from the software running on the x86 CPUs. The distributed embedded control system is designed such that any potential system configuration can be created, which provides maximal flexibility when performing experiments. For example, the H-bridges and amplifiers that drive each motor can be connected to any of the nodes by changing a simple patch cable. All motors can be controlled from a single node, or each motor can be controlled from a separate node. Similarly, software can be deployed on the distributed embedded control system in any configuration. A picture of the experimental set-up is shown in Figure 6.6. The paper input tray and the pinches are on the top of the set-up. The four embedded controller nodes are visible below the paper path. The cross-development system, which is an ordinary personal computer, is not shown in the photograph.

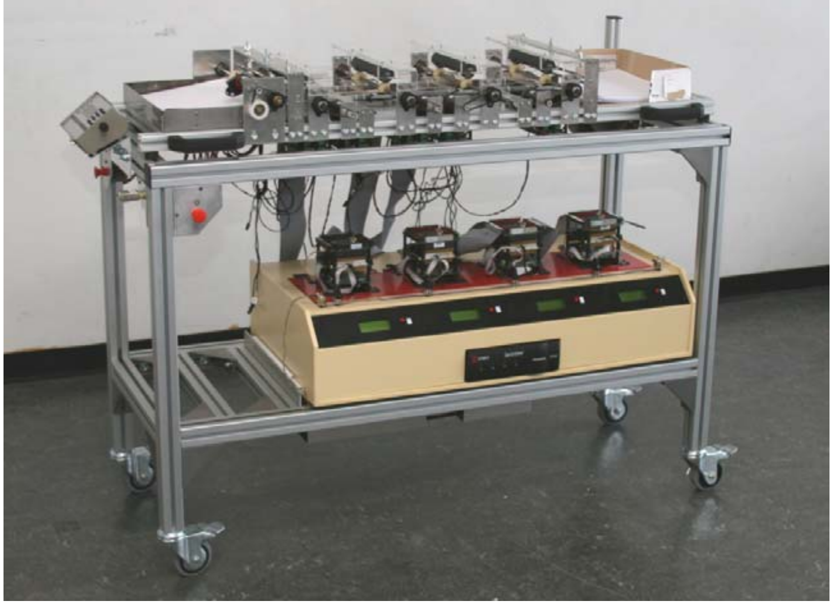


Figure 6.6: The experimental set-up at University of Twente, photo by P.M. Visser

6.3 Modeling the experimental set-up

The experimental set-up described in the previous section has been extensively studied using formal techniques in the BODERC project. We have followed a three level development approach to address the complete system development life-cycle of this system, which is graphically represented in Figure 6.7.

- **First level**. In the first level, the emphasis is on system analysis by modeling, simulation and analysis using formal techniques. It does not necessarily require the use of the experimental set-up. A model of the dynamic behavior of the system (the plant) was built using Bond graphs. This model is presented in detail in Section 6.3.1 and was studied using 20-SIM. A model of the controller software was built using VDM++. This model is presented in detail in Section 6.3.3 and it was studied using VDMTOOLS. The semantic extensions presented in Chapter 4 are used to investigate the interaction between these models to support the multi-disciplinary design dialogue in this phase.

- **Second level**. In the second level, the emphasis is on elaborating the software model of the control application. It does not necessarily require the use of the experimental set-up. The level of detail in the VDM++ model is incrementally increased until source code can be constructed from it straightforwardly. We use automatic code generation, directly from the VDM++ models. The generated code is compiled using a standard C++ compiler running on the simulator host, in our case a normal personal computer running on the Windows platform. The resulting dynamic link library (DLL) can be used for so-called software-in-the-loop simulations against the unmodified model of the plant in 20-sim.

- **Third level**. In the third level, the unmodified C++ code generated from the VDM++ models developed in the second level is compiled for the target platform. The resulting application can be uploaded to the embedded controllers of the experimental set-up for testing. Measurements are taken during the execution of the experiment.

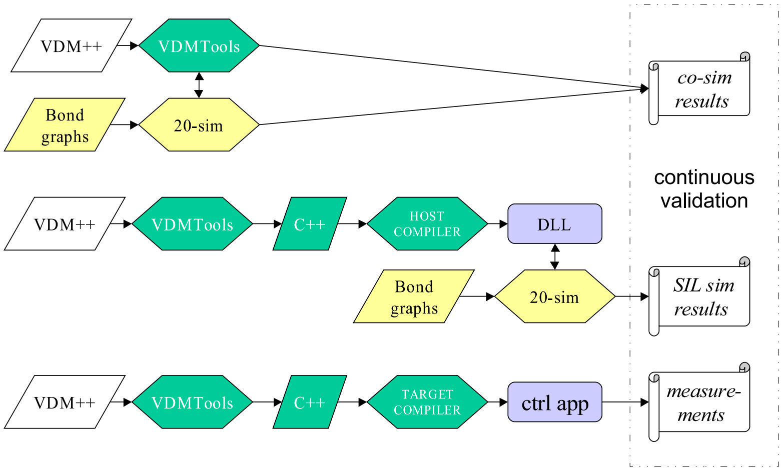


Figure 6.7: The overall development strategy - a three level approach

The philosophy behind this approach is to obtain a design trajectory which enables continuous validation of the system under development for both controller and plant. In the first level, light-weight system models are developed that enable the investigation of multi-disciplinary design trade-offs due to the co-simulation capability. In addition, the VDM++ models developed here can be studied using domain specific analysis tools, e.g. to verify internal consistency of the model using static analysis, testing or even proof. Similarly, the Bond graph models can be scrutinized with the usual control engineering specific analysis approaches for example to investigate control law stability and convergence. System identification techniques can be used to ensure that the dynamic system model is an accurate representation of the physical plant. Of course, this last activity can only take place if the system (or part of the system) is available to perform measurements upon.

In the second level, the software engineer will elaborate the high-level VDM++ models into an implementable software architecture which is also specified in VDM++. Design decisions are made that may affect the behavior of the system as a whole. Major architectural decisions such as deployment of the application on a distributed hardware architecture and the type of scheduling used on and the available performance of each computation and communication element are decided upon. The designer has the ability to assess the impact of these design choices by comparing the simulation results from the second level to those of the first level. This may give rise to partial redesigns if significant differences are reported. This process is iteratively repeated until eventually the source code level is reached. Domain specific analysis techniques can be used after each iteration to verify these refinement steps, e.g. by testing. But in addition, it is also possible to continuously validate the behavior of the system as a whole due to the software-in-the-loop capability.

In the third level, the controller application software is moved from the simulation host to the embedded target. This implies at least recompilation at the source code level and integration of the application with the real-time operating system running on each processor. This step can be verified by comparing the measurements from the experiment performed on the embedded target to the simulations from the first two levels. Any discrepancy which exceeds predefined design margins must either be related to a problem in the target compiler, the operating system and device drivers or it is an incorrect assumption in either of the controller or plant models.

In the next two subsections we will present the model of the printer paper path. The aim is to give the reader some insight into the scope and level of detail that was achieved. We do not show *how* these models were obtained through refinement, we merely present the end result of that iterative design and analysis activity at the end of the second level. However, we will explain the major decisions that were taken during the elaboration process.

6.3.1 Modeling the plant

A dynamic system model of the experimental set-up was developed by Ambrosius and Visser and is presented in detail in [3]. They developed a model for both plant and controller using Bond graphs. In this work, we will only reuse the plant sub-model and we will replace the controller model by a model written in VDM++ in Section 6.3.3. The important elements from the plant model are summarized here for convenience. As suggested in Chapter 5, a very important decision that drives the modeling work is the choice of the appropriate I/O interface between the plant and controller. Ambrosius and Visser have put this interface at the level of the interconnect matrix shown in Figure 6.5

and is presented in more detail in Figure 6.8. The primary reason for the choice of this particular interface is the high likelihood that this interface remains stable during the entire life cycle of the system, since it is commonly used in these kinds of applications. It is composed of the following items:

- The output of the discrete controller model is a real value in the interval $\langle -1, 1 \rangle$ per motor which represents the so-called *duty cycle* of the pulse-width modulation (PWM) signal that drives the power amplifier. Assume a square wave signal with a fixed frequency. The duty cycle is then defined as the ratio between the pulse duration and the period. The sign determines the direction of rotation (clock-wise or anti clockwise).

- The input of the discrete controller is a signed integer value per motor which represents the rotational direction and the number of counted pulses of the rotary encoder [1] that is connected to each motor axis.

- The paper detectors (input to the discrete controller) are modeled as Integer values whereby 0 represents the absence of paper and 1 the presence of paper at the sensor position.

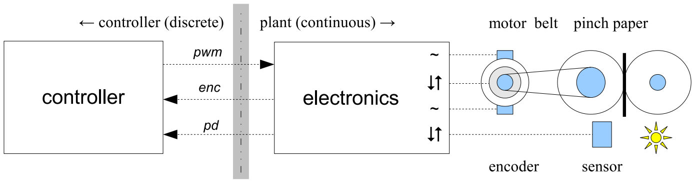


Figure 6.8: Overview of the controller - plant I/O interface

The top-level bond graph model of the plant is shown in Figure 6.9. At the bottom of the figure, we see the interface towards the controller. There is a pair of PWM and encoder signals connected to each motor-belt-pinch icon. These icons represent lower level bond graph models, or sub-models, which we will present later in more detail. The plant model has four motor-belt-pinch sub-models while our experimental set-up has five. The first motor in the set-up is only used to inject new sheets into the paper path. Since its operation is only of minor importance to the total system behavior it is only abstractly represented in the plant model by means of the `FeedSheet` signal. The behavior of the individual sheets is represented by the photographic icon. This sub-model maintains the state of each sheet in the paper path, such as for example its current speed and position. The state of the `PaperDetectors` signal is automatically derived from this information. If the position of a paper detector is in between the leading and trailing edge of at least one sheet then it will yield 1 else 0. Similarly, the sheet is in control of a pinch if the position of the pinch is in between the leading and trailing edge of the current sheet position. The animation icon is used as a monitor which allows us to visualize the simulation graphically.

The pinches drive the sheets and this transfer of energy is influenced by friction. In kinematic models the friction is assumed to be zero but this is usually not very realistic. The friction force which is imposed on each sheet is a function of the mass of the sheet and the speed difference between the sheet and pinch. Of course, the friction force is

[1] See *http://en.wikipedia.org/wiki/Rotary_ encoder*

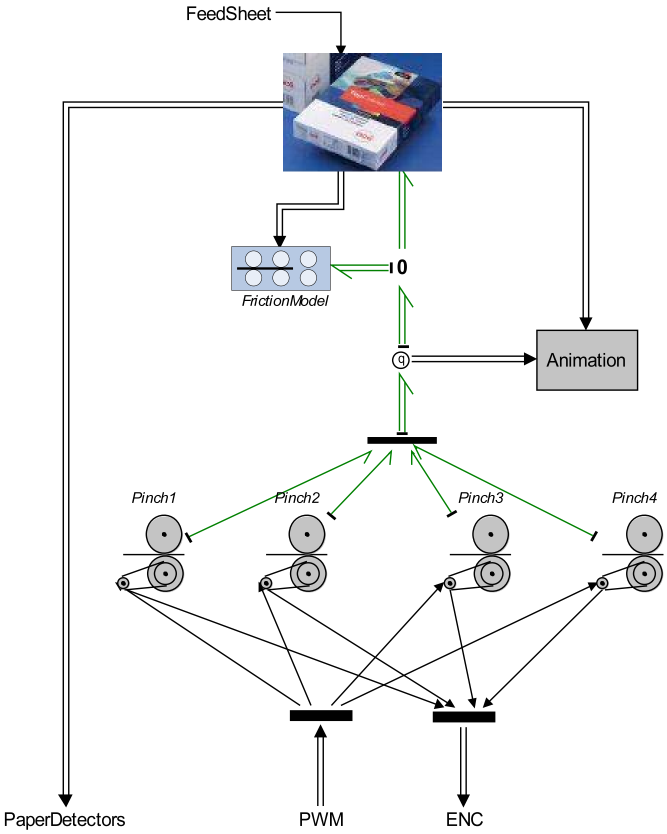


Figure 6.9: Top-level bond graph of the plant model

imposed if and only if the sheet is in control of a pinch. What happens if a sheet is in control of two pinches? In our model we assume that the pinch near the leading edge of the sheet dominates the pinch near the trailing edge. The assumption is that the force imposed by the leading edge pinch will cause the sheet to slip in the trailing edge pinch. This abstraction can be used if and only if the speed of the leading edge pinch is equal to or slightly higher than the speed of the trailing edge pinch. This condition can be checked at simulation time. Of course, only a very trivial friction model is used here, but it can simply be replaced by more complex hybrid friction models if the need arises, without affecting the plant model architecture demonstrated here.

The Bond graph sub-model for the motor, belt and pinch is presented in Figure 6.10. This iconized diagram demonstrates at a very high level of abstraction how the control signal relates to the movement of the sheet. For example, observe that the behavior of the power electronics, the so-called H-bridge, has been modeled as a simple multiplication (or gain) factor. In other words: the amount of power provided to the motor is linear proportional to the duty cycle of the pulse-width modulated input signal obtained from the controller. The motor converts this electrical energy into torque. The torque causes the belt to rotate and the belt in turn drives the pinch, whereby the “Belt and Gear” sub-model simply multiplies the rotational speed of the motor with the gear ratio. And finally, the pinch transfers its energy towards the sheet of paper as described previously. The angular velocity is measured at the motor axis and this value is multiplied by 2π to obtain the number of rotations per second. We will see later how this value is converted into encoder values in the detailed I/O interface model.

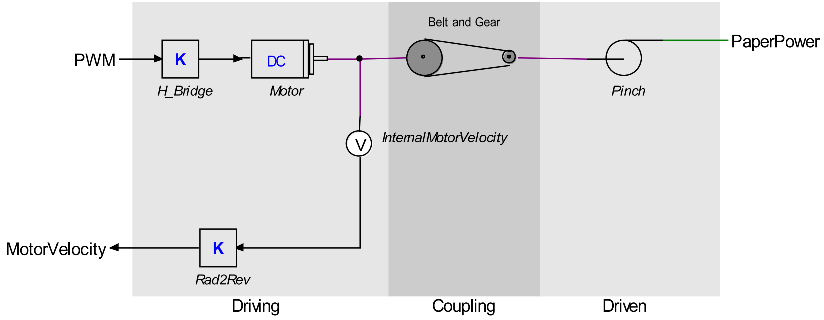


Figure 6.10: Bond graph of the motor, belt and pinch subsystem

The "DC motor" icon in Figure 6.10 is itself a Bond graph sub-model as shown in Figure 6.11, which again demonstrates the explicit hierarchy in the model. The inductance L, internal resistance R, motor torque constant, rotor inertia and Coulomb friction parameters required by this sub-model can usually be found in the supplier data sheet. In our case we use the Maxon RE25 motor [2].

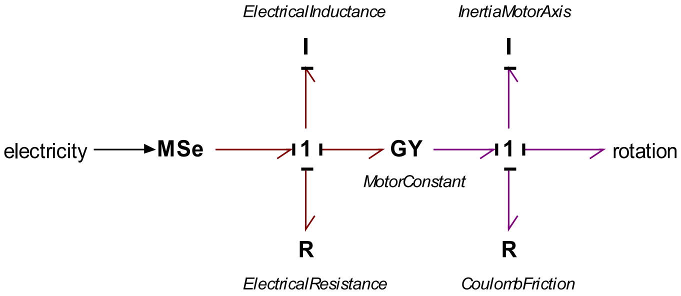


Figure 6.11: Bond graph of the DC motor from Figure 6.10

Finally, we revisit the I/O interface model. The I/O model acts as a mediator between the discrete time controller model and the continuous time plant model, as shown earlier in Figure 6.8. In other words, discrete values need to be converted into continuous signals and vice versa and we have to ensure that this conversion process does not affect the overall analysis at the system level. A detailed overview of the I/O model is shown in Figure 6.12. The top row demonstrates from left to right how the discrete PWM values are converted into their continuous counterpart, in this case an analog voltage between $\langle -1, 1 \rangle$ Volt. This value is delayed by one integration time step in order to keep the real-time properties of the model consistent, as described in Chapter 5 and Figure 5.8. The middle row shows, from right to left, how the rotation of the motor axis is translated into a discrete encoder value. First, the motor speed is integrated into its position. This value is multiplied by $n/2\pi$, whereby n is the number of encoder steps that can be measured per revolution. This value is rounded to the nearest integer and finally sampled. At the bottom row similar steps are taken to discretize the paper detector sensor into an Integer value.

[2] The data sheet can be found at *http://www.maxonmotor.com*.

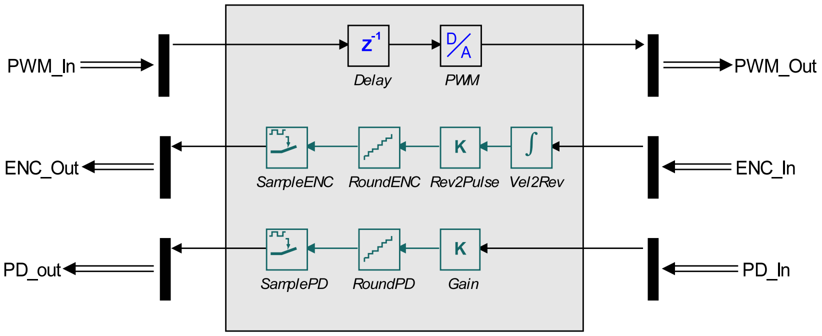


Figure 6.12: Detailed overview of the I/O interface model

6.3.2 Validating the plant model

The core of the plant model is now available, but how do we know if it is fit for purpose and whether or not it accurately describes the system? The usual approach is to validate parts of the model by system identification: tuning the model by performing measurements on (parts of) the system. Consider the Bond graph sub-model for a pinch as shown in Figure 6.13. The model demonstrates how rotation is transformed into translation of the sheet and which disturbances play a role in this process. The major contributors are the inertia and friction properties of the pinch. These values can be estimated, measured or calculated. The impact of the friction parameters was known to be low from previous experience. Their order of magnitude was estimated and this was checked by simple measurements using open-loop control. The inertia of the metal axis is by far the most dominant part of the pinch, therefore its value was calculated based on its mass and radius.

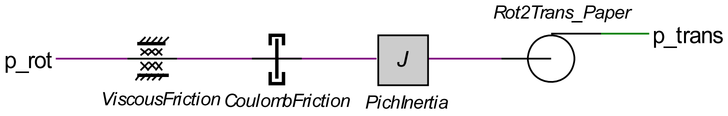


Figure 6.13: Bond graph of a pinch from Figure 6.10

An interesting observation was made when a single motor-belt-pinch sub-system was put on the test bench. A low frequency sinoid oscillation was detected during speed measurements, when the motor was controlled in open loop with a preset and constant duty cycle. The frequency of this disturbance was linear related to the angular velocity of the motor and the length of the rubber belt. The cause of this phenomenon was that the belt is not homogeneous in width and thickness, a common problem occurring with off-the-shelf components which was demonstrated by the fact that other rubber belts appeared to have exactly the same problem. Ambrosius and Visser decided to update the plant model to address this issue. A squared sine wave with a small amplitude (as measured on the test bench) was added to the motor angular velocity. A low-pass filter was added to ensure that this disturbance is only related to the mean velocity and has no effect at higher speeds, as was demonstrated in the measurements. This also required a change at the higher level Bond graph model since the angular velocity is now needed as an input to the disturbance model. An overview of these improvements is presented in Figure 6.14.

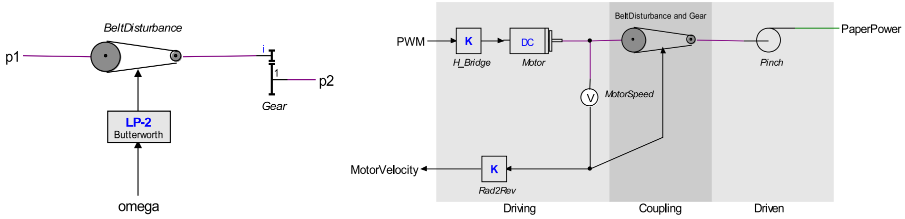


Figure 6.14: Bond graphs of the improved motor, belt and pinch subsystem

Also the behavior of the paper detectors was checked. Light sensitive optical sensors are used to detect the edges of each sheet. However, the light receptacle of the sensors used have a rather wide opening angle which becomes a problem when the sensor cannot be mounted physically close to the sheet. At low speeds, it may actually take a few samples before the sensor is completely covered which would make sheet detection very inaccurate. This problem could be solved in the plant model, for example by adding an hysteresis filter. But instead, Ambrosius and Visser decided to modify the sensor receptacle such that only a narrow beam of light can be detected. The interested reader is referred to [3] for more details on the design of the plant model.

6.3.3 Modeling the controller

An overview of the paper path and the experimental set-up has been presented in Section 6.2. We have fully abstracted away from the image processing part of the Océ Varioprint 2090 in the experimental set-up. We assume that images are delivered at a user defined constant rate, which is simulated in the experimental set-up by the constant angular velocity of the fuse pinch. This velocity ω is determined by the required system throughput performance as described in the following equations:

$$V_{fuse} = (pagesize + isd) \cdot tp / 60 \quad \text{(mm / sec)} \tag{6.1}$$

$$\omega = V_{fuse} / 2\pi \cdot r_{pinch} \quad \text{(rad / sec)} \tag{6.2}$$

whereby *pagesize* represents the size of a sheet (in mm), *isd* represents the inter-sheet distance (in mm), *tp* represents the throughput (in pages per minute) and finally r_{pinch} represents the radius of the fuse pinch (in mm). The inter-sheet distance is defined as the distance between the trailing and leading edge of two consecutive sheets. The primary task of the paper path sub-system is to deliver each sheet on time and at the right speed at the fuse pinch. This requirement has two implications:

1. With respect to **“on time”**. The inter-sheet distance shall be maintained in order to meet the required system performance and to ensure the correct alignment of the image on the sheet. A maximum deviation of 0.5 mm is allowed exactly at the fuse pinch but the inter-sheet distance may vary elsewhere as long as two consecutive sheets do not collide or overlap.

2. With respect to **“right speed”**. The leading edge of each sheet shall have the nominal speed V_{fuse} just before it is in control of the fuse pinch. Printing quality will deteriorate if the speed is too low, because pulling the sheet from the penultimate pinch may cause slip in the fuse pinch. Alternatively, the sheet may

blouse causing folds in the sheet or even paper jams can occur if the sheet is delivered too fast. A maximum deviation of V_{fuse} of 2 % is allowed.

Now the main control goal has been identified, we can look at the secondary tasks to perform by the control application. Considering the pinches in the experimental set-up, we have the following additional requirements:

1. The first pinch is part of the paper input module and it is used to retrieve sheets from the tray. The challenge is to ensure that single sheets are separated. The solution is to control this motor belt pinch sub-system in open loop. Basically the motor is told to accelerate as fast as possible for a very short period of time and then immediately decelerate. The friction force between the pinch and the top sheet is larger than the friction force between the top two sheets, which will cause clear separation of a single sheet.

2. The purpose of the second pinch is to get the sheet under control by moving it down the paper path at the nominal speed V_{fuse}.

3. The purpose of the third pinch is to decelerate, stop and accelerate the sheet. This will simulate the alignment process of the sheet in the Océ Varioprint 2090. The length of the stop period is user defined.

4. The purpose of the fourth pinch is to ensure that the sheet is delivered with the correct inter-sheet distance and speed to the fuse pinch. It will have to compensate for the time lost during alignment of the sheet at the previous pinch.

5. The fifth pinch simulates the fuse pinch. But since it is also the last pinch in the experimental set-up, it also acts as the finisher pinch. As soon as the pinch is in control of a sheet and the leading edge has been detected by the fuse pinch paper detector, it will briefly accelerate to ensure proper delivery to the finisher. The fuse pinch needs to return to V_{fuse} and stabilize before the scheduled arrival of the next sheet.

The control application will have to satisfy all these sub-goals simultaneously. The behavior of a sheet, in terms of its speed through the paper path, is graphically presented in Figure 6.15. The numbers 2 to 5 correspond to the pinch that is in control of the sheet at a given point in time. The grey areas indicate where the paper is in control of two pinches simultaneously. The control application will need to ensure that the angular velocity of the pinch with the higher number is equal or marginally greater than the pinch with the lower number in order to prevent hybrid control phenomena as described in Section 6.3.1.

We will take a step-by-step look at the lifetime of a sheet during its travel through the pinches in order to get a feeling for the control complexity involved. The events mentioned are also shown in Figure 6.15.

1. A new print job arrives and the image processing starts. Meanwhile pinches 2 to 5 are booted up until they run at V_{fuse} and then the first sheet is requested from the paper input tray. The sheet is separated by pinch 1 and it is inserted into the paper path. It will hit pinch 2 with some force and at the wrong speed since we use a rather brute force separation method.

2. Pinch 2 accepts the first sheet and will try to stabilize its speed to V_{fuse}. The timing tolerance caused by the brute force separation is known when the leading edge of the sheet is detected by PD$_{pim}$, as shown by the $\uparrow$-arrow.

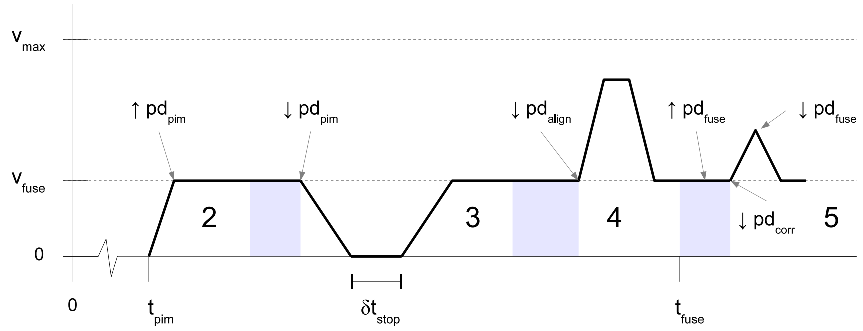


Figure 6.15: Overview of a typical sheet velocity profile

3. The sheet continues to move downstream and gets into joint control of pinches 2 and 3 (the first grey area in the figure). The control application knows that the sheet has left the control of pinch 2 when the trailing edge of the sheet is detected by PD$_{pim}$, as shown by the ↓-arrow. The alignment phase of the sheet can now start because pinch 3 is in full control. The deceleration needs to be quick enough to ensure that the leading edge of the sheet does not reach pinch 4.

4. The sheet is accelerated to V_{fuse} after the user-defined alignment time δt$_{stop}$ has expired. The acceleration must be performed quickly to ensure that the nominal speed has been reached before the leading edge of the sheet hits pinch 4.

5. The sheet continues to move downstream and gets into joint control of pinches 3 and 4 (the second grey area in the figure). The control application knows that the sheet has left control of pinch 3 when the trailing edge of the sheet is detected by PD$_{align}$, as shown by the ↓-arrow. The correction phase of the sheet can now start because pinch 4 is in full control. The sheet is accelerated to compensate for the time lost in the alignment phase and the tolerances caused by the brute force sheet separation. The sheet needs to be decelerated and stabilized to V_{fuse} before the leading edge arrives at pinch 5 (the fuse pinch) with the correct inter-sheet distance.

6. The sheet continues to move downstream and gets into joint control of pinches 4 and 5 (the third grey area in the figure) and the image is printed on the sheet. The alignment accuracy is verified by checking the arrival time of the leading edge of each sheet at PD$_{fuse}$, indicated by the ↑-arrow in the figure. The control application knows that the fuse pinch is in full control of the sheet when the trailing edge of the paper is detected by PD$_{align}$. The sheet is finally accelerated to ensure proper delivery to the finisher.

7. The sheet is in full control of the finisher when the trailing edge of the sheet is detected by PD$_{fuse}$. The speed of the fuse pinch is quickly brought back and stabilized to V_{fuse} in time for the arrival of the next sheet.

The informal description of the requirements for the control application listed above gives us some inspiration for the control application architecture that is required to address these challenges. From the engineering point of view, it is usually a good idea to apply the "separation of concerns" principle, for example to divide the time critical parts from the less time critical parts of the application. For example, the timing requirements for pinches 2 to 5 are very tight. Therefore, the motor-belt-pinch subsystems will get a real-time controller. In fact, we will provide each motor-belt-pinch subsystem its own controller in our architecture because the requirements differ and they may be deployed on different hardware in the final implementation. In contrast, the timing requirements for the high-level sheet flow control, as presented in the sheet life-cycle are far less demanding and a single application may suffice for this purpose. The linking pin between this high-level supervisory control layer and the real-time controllers is a set of so-called sequence controllers, one per real-time controller. These sequence controllers generate so-called set point profiles ahead of time, based on the planning information received from the supervisor. An informal overview of this well-known three tier control application architecture is shown in Figure 6.16.

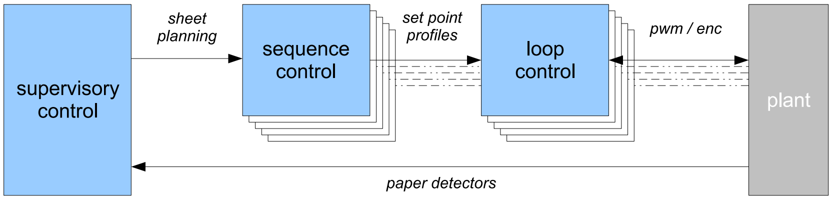


Figure 6.16: An informal overview of the three tier control application architecture

An overview of VDM++ models for the controller application architecture is presented in a bottom-up fashion. We start at the plant model interface and the real-time loop controller and work our way up towards the supervisory control. Each motor-belt-pinch sub-system has an interface consisting of a pulse width modulation input (PWM) and encoder output (ENC). This interface is well suited for feedback control. A standard PID control strategy [3] will be used for pinches 2 to 5. The real-time controller will periodically sample the encoder value. This value is a measure for the distance covered and it is compared to the so-called set point, which represents the intended value. The difference between the two is called the error and with the PID algorithm we calculate a new PWM value to compensate for this measured error. The four PID loop controllers will operate at 1 kHz in our controller models. Open loop control is used for pinch 1. Basically, the set point is forced upon the motor-belt-pinch system by writing the correct PWM value but the encoder value is ignored. For convenience, the loop controller for pinch 1 will also run at 1 kHz.

The loop controller

Consider the VDM++ model for the loop controller shown below. The constructor of the active class *LoopController* takes two arguments. The first argument, *ptp*, determines whether the calculated output value is send to the plant model at the start of the next iteration or immediately. The second argument, *pfb* is used to distinguish the control strategy: closed loop or open loop.

```
class LoopController

instance variables
  -- time-triggered (true) or immediate output (false)
  private hold : bool := true;
```

[3] See *http://en.wikipedia.org/wiki/PID_controller.*

```
  -- closed loop (true) or open loop (false)
  private feedback : bool := true

operations
  public LoopController: bool * bool ==> LoopController
  LoopController (ptp, pfb) == ( hold := ptp; feedback := pfb )
```

It is possible to describe a system at several levels of abstraction in VDM++. Implicit operation definitions will be used in this section for the sake of brevity. Some of these operations are loosely specified on purpose, for example post conditions may look trivial. The reader needs to know about their existence but the actual detailed specification is not relevant to understand the structure of the model. Two operations are defined to access the plant model. The operation *getEnc* will read the current encoder value and *setPwm* will write the current pulse width modulation value. The auxiliary operation *limit* is used to truncate the calculated PWM value.

```
operations
  private getEnc () enc : int
  post true;

  private setPwm (pwm : real)
  pre pwm < 1 and pwm > -1
  post true;

  private limit (ival : real) oval : real
  post oval < 1 and oval > -1
```

The instance variable *output* is used to temporarily store the calculated pulse width modulation value. The operation *CtrlLoop* implements the real-time control strategy and is periodically executed. The operation *calcPID* executes the PID algorithm.

```
instance variables
  private output : real := 0;
  private ltime : real := 0

operations
  public calcPID (enc : real) pwm : real
  ext rd ltime : real
  post true;

  public CtrlLoop: () ==> ()
  CtrlLoop () ==
    -- first retrieve the current encoder value
    ( dcl enc : real := getEnc();
      -- update the old output if time-triggered
      if hold then setPwm(output);
      -- update the local notion of time
      ltime := ltime + 0.001;
      -- calculate the new PWM value
      output := if feedback
                then limit(calcPID(enc))
                else limit(getSetpoint());
      -- update the output if not time-triggered
      if not hold then setPwm(output) )

thread
  -- execute the controller at 1 kHz
  periodic (0.001, 0, 0 , 0.001)(CtrlLoop)
```

The operation *getSetpoint* used inside *CtrlLoop* retrieves the set point from the passive *SetpointProfile* object for the current local time *ltime*. The set point profile can be updated by the sequence controller by calling the asynchronous *addProfileElement* operation. The operations *getSetpoint* and *addProfileElement* are declared mutual exclusive to prevent data corruption by simultaneous access to the *profile* instance variable.

```
instance variables
  private profile : SetpointProfile := new SetpointProfile()

operations
  private getSetpoint: () ==> real
  getSetpoint () == profile.getSetpoint(ltime);

  async public addProfileElement: real * real * real ==> ()
  addProfileElement (px, py, pdt) ==
    profile.addElement(px, py, pdt)

sync
  -- access to the profile is mutual exclusive
  mutex (addProfileElement, getSetpoint);
  mutex (addProfileElement)

end LoopController
```

The set point profile

The passive class *SetpointProfile* is used as a container to collect all knowledge on manipulating so-called set point profiles. A set point profile is an ordered collection (a sequence) of left-closed, right-opened, line elements which together define the evolution of the set point over time. Each line element, or *ProfileElement*, is defined by three real numbers. The first number defines the domain: the starting time t at which this element is valid. The second and third number define the range: the current value at time t and the direction coefficient that is valid from this point in time onwards respectively. Set point profiles are defined from some point in time to infinity, since the last element in the profile is right-opened. The invariant of the *profile* instance variables ensures that the domain is strictly monotonically increasing but it does allow discontinuities in the range. The operation *addElement* can be used to extend the current set point profile.

```
class SetpointProfile

types
  private ProfileElement = seq of real
  inv pe == len pe = 3

instance variables
  profile : seq of ProfileElement := [];
  inv forall i, j in set inds profile &
        i < j => profile(i)(1) < profile(j)(1)

operations
  public addElement: real * real * real ==> ()
  addElement (t,v,a) ==
    profile := profile ^ [[t,v,a]]
    pre len profile > 0 => profile(len profile)(1) < t
```

The operation *getSetpoint* is used to compute the actual set point at some specific point in time based on the abstract continuous time description maintained in the *profile* instance variable.

```
operations
  public getSetpoint: real ==> real
  getSetpoint (t) ==
    if len profile = 0
    then return 0
    else ( dcl prev_pe : ProfileElement := hd profile;
           for curr_pe in tl profile do
             if curr_pe(1) > t
             then return calcSetpoint(t, prev_pe)
             else prev_pe := curr_pe;
           return calcSetpoint(t, prev_pe) )
    pre t >= 0 and len profile > 0 => t > profile(1)(1)

functions
  private calcSetpoint: real * ProfileElement -> real
  calcSetpoint(t, [px, py, pdydx]) == py + pdydx * (t - px)
    pre t >= px

end SetpointProfile
```

The sequence controller

The active class *SequenceController* contains the knowledge to translate high-level paper path planning commands into set point profiles that are used by the loop controllers. Each sequence controller is associated with exactly one loop controller *loopctrl*.

```
class SequenceController

instance variables
  public loopctrl : [LoopController] := nil
```

The operation *initNominal* is used to power-up the pinches until they reach the nominal paper path speed v$_{nom}$. The motors are not started at full throttle immediately, but they are ramped up gradually. The user can influence the power-up time by setting the acceleration parameter a$_{nom}$.

```
operations
  async public initNominal: real * real ==> ()
  initNominal (v_nom, a_nom) ==
    ( -- ramp up the motor to the nominal paper speed
      loopctrl.addProfileElement(0, 0, a_nom);
      -- and maintain a constant speed indefinitely
      loopctrl.addProfileElement(v_nom / a_nom, v_nom, 0) )
    pre v_nom > 0 and a_nom > 0 and loopctrl <> nil;

  async public initPeak: real ==> ()
  initPeak (tpeak) ==
    -- give the sheet a good kick for 60 msec
    ( loopctrl.addProfileElement(tpeak, -40, 0);
      loopctrl.addProfileElement(tpeak+0.060,0,0) )
    pre loopctrl <> nil
```

The operation *setStopProfile* is used to bring the sheet in the paper path to a complete stand still for *dstop* seconds. The procedure will start at t_1 with speed v_1 mm/sec and the sheet will accelerate and decelerate with *acc* mm/sec^2.

```
operations
  async public setStopProfile: real * real * real * real ==> ()
  setStopProfile (t1, v1, acc, dstop) ==
    def dt = v1 / acc in
      ( loopctrl.addProfileElement(t1, v1, -acc);
        loopctrl.addProfileElement(t1+dt, 0, 0);
        loopctrl.addProfileElement(t1+dt+dstop, 0, acc);
        loopctrl.addProfileElement(t1+dt+dstop+dt, v1,0) )
    pre acc <> 0 and loopctrl <> nil

end SequenceController
```

The supervisory controller

The active class *Supervisor* represents the supervisory control in our architecture. It has five instance variables of type *SequenceController*. The links to these objects are created at model instantiation time. Each sequence controller takes care of one motor-belt-pinch subsystem. The mapping *shts* keeps track of the time when each sheet is requested. We will use this information to check whether or not our control goal has been achieved. The operation *startPrintJob* is invoked to start the printing process. The operation *init* is used to ramp up all the pinches to the nominal speed.

```
class Supervisor

instance variables
  public ejectSeqCtrl : [SequenceController] := nil;
  public pimSeqCtrl   : [SequenceController] := nil;
  public alignSeqCtrl : [SequenceController] := nil;
  public corrSeqCtrl  : [SequenceController] := nil;
  public fuseSeqCtrl  : [SequenceController] := nil;

  -- keep track of the time the sheet was requested
  private shts : map nat to real := {|->}

operations
  async public startPrintJob : real * real * real ==> ()
  startPrintJob (ppm, pagesize, isd) ==
    def v_nom = (pagesize + isd) * 60 / ppm in
      ( -- start-up the paper path
        init (v_nom, v_nom * 10);
        -- simulate printing ten sheets after 1 second
        def now = time + 1.0 in
          for idx = 0 to 9 do
            def tstart = now + idx * ppm / 60 in
              ( -- tell the sequence controller
                ejectSeqCtrl.initPeak(tstart);
                -- remember when it is requested
                shts := shts munion {idx+1 |-> tstart} ) )
    pre ejectSeqCtrl <> nil and ppm > 0 and
        pagesize > 0 and isd > 0;

  public init : real * real ==> ()
  init (v_nom, a_nom) ==
    ( pimSeqCtrl.initNominal(v_nom, a_nom);
      alignSeqCtrl.initNominal(v_nom, a_nom);
      corrSeqCtrl.initNominal(v_nom, a_nom);
      fuseSeqCtrl.initNominal(v_nom, a_nom) )
    pre pimSeqCtrl <> nil and alignSeqCtrl <> nil and
        corrSeqCtrl <> nil and fuseSeqCtrl <> nil
```

The core functionality of the supervisory control application is captured in the operations that respond to the paper detectors. For example, the operation *pimDownEvent* will be called whenever a trailing edge of a sheet has been detected by paper detector PD_{pim}. This event signals the start of the alignment process which will bring the sheet to a complete stand still, in our case for 100 msec.

```
operations
  -- operation to initiate the alignment procedure
  async public pimDownEvent: () ==> ()
  pimDownEvent () ==
    -- start decelerating in 10 msec from now
    def dectime = time + 0.01 in
      alignSeqCtrl.setStopProfile(dectime, 50, 500, 0.1)
    pre alignSeqCtrl <> nil
```

The operation *fuseUpEvent* is called whenever the leading edge of a sheet has been detected by PD$_{fuse}$. The arrival time of this event is registered in the *fues* mapping. The operation *corrDownEvent* is called whenever the trailing edge of a sheet has been detected by PD$_{corr}$. The arrival time of this event is registered in the *cdes* mapping.

```
instance variables
  private fue_cnt : nat := 1;
  private fues : map nat to real := {|->}

operations
  async public fuseUpEvent: () ==> ()
  fuseUpEvent () ==
    ( -- measure the arrival time of the event
      fues := fues munion {fue_cnt |-> time};
      -- update the counter
      fue_cnt := fue_cnt + 1 )

instance variables
  private cde_cnt : nat := 1;
  private cdes : map nat to real := {|->}

operations
  async public corrDownEvent: () ==> ()
  corrDownEvent () ==
    ( -- measure the arrival time of the event
      cdes := cdes munion {cde_cnt |-> time};
      -- update the counter
      cde_cnt := cde_cnt + 1 )
```

The information collected in the mappings *shts*, *fues* and *cdes* can be used to check whether or not the control goals where in fact achieved. We can define predicates to check these measured values since we know the position of the pinches and the paper detectors a priori. With respect to the “on time” requirement, we can check whether the measured arrival time at the fuse pinch, maintained in *fues*, corresponds to the expected arrival time of the sheet.

Another approach has to be taken with respect to the “right speed” requirement because there is no sensor to measure the speed of the sheet at the fuse pinch. In stead we take the difference between the leading edge of the sheet reaching PD$_{fuse}$ and the trailing edge of the sheet reaching PD$_{corr}$. These events will always take place in this order because the width of a sheet is wider than the distance between the two sensors. From this information we can approximate the speed of the sheet at the fuse pinch. The operation *evalPrintJob* verifies both control goals and will return `true` if all sheets where printed within the design margins set.

```
values
  -- position of the pinches in the paper path (in mm)
  pinches : seq of real = [0, 145, 320, 495];

  -- position of the paper detectors (in mm)
  sensors : seq of real = [12, 186, 361, 537]

operations
  public evalPrintJob : real * real * real ==> bool
  evalPrintJob (ppm, pagesize, isd) ==
    ( -- calculate the nominal printing speed (mm/sec)
      def v_nom = (pagesize + isd) * 60 / ppm in
      -- calculate the time to reach PD fuse at this speed
      def t_fuse = sensors(4) * v_nom in
        -- iterate over the results
        for idx = 1 to 10 do
          -- calculate the speed of the sheet
          def ddiff = pagesize + sensors(3) - sensors(4) in
          def tdiff = cdes(idx) - fues(idx) in
          def vsheet = ddiff / tdiff in
          -- calculate the expected arrival time for this sheet
          def t_exp = shts(idx) + t_fuse in
          -- calculate the time difference
          def dt = abs(t_exp - fues(idx)) in
          -- calculate the delivery distance
          def dx = dt * vsheet in
            -- check the control goal for this sheet
            if ( dx > 0.25 ) or
               ( vsheet < 0.99 * v_nom ) or ( vsheet > 1.01 * v_nom )
            then return false;
    -- all sheets pass with flying colors
    return true )
    pre card dom shts = 10 and card dom fues = 10 and
        card dom cdes = 10 and sensors(4) - sensors(3) < pagesize and
        forall i in set {1,...,10} & cdes(i) > fues(i)
sync
  -- block evalPrintJob until all ten pages have been processed
  per evalPrintJob => card dom fues > 9 and card dom cdes > 9

end Supervisor
```

The paper path control system class

The final step in constructing the controller model is to describe the embedded system architecture on which the application is deployed. For simplicity we will consider the situation where all software is deployed on a single CPU. The so-called system class *PaperPathController* is constructed for that particular purpose. Five loop controller and sequence controller instances are created as individual instance variables, as shown below.

```
system PaperPathController

instance variables
  -- create the first tier: five loop controllers
  lp1 : LoopController := new LoopController(true,false);
  lp2 : LoopController := new LoopController(true,true);
  lp3 : LoopController := new LoopController(true,true);
  lp4 : LoopController := new LoopController(true,true);
  lp5 : LoopController := new LoopController(true,true);

  -- create the second tier: five sequence controllers
  sc1 : SequenceController := new SequenceController();
  sc2 : SequenceController := new SequenceController();
  sc3 : SequenceController := new SequenceController();
  sc4 : SequenceController := new SequenceController();
  sc5 : SequenceController := new SequenceController();

  -- create the third tier: the supervisory controller
  supervisor : Supervisor := new Supervisor();
```

The use of separate instance variables provides the possibility to deploy specific objects on specific computation resources. Note that in the design of the loop and sequence controllers we have already taken the issue of deployment into account by declaring most of the operations that play a role at run-time to be asynchronous. Furthermore, information flows from the supervisor tier down to the loop controllers but not vice versa. Asynchronous operation calls, as presented in chapter 3, do not block the thread of control of the caller. The asynchronous call will be handled as a seperate thread of control. The operation call will cause traffic on a communication resource if the two objects are deployed on different computation resources. The non-blocking property of asynchronous operations is essential for the design of embedded real-time systems since these systems always need to be able to respond to their environment. Here we assume the availability of a single 20 MIPS processor that uses fixed priority scheduling. All the created objects are explicitly deployed onto this resource and the relationships between these objects is established in the constructor of the system class. The periodic threads inside the loop controllers will start immediately after the constructor of the *PaperPathController* has finished. A print job can be started by calling the *run* operation. Note that this operation will wait automatically until the print job is finished because of the synchronization predicate which has been defined on the *evalPrintJob* operation inside the *Supervisor* class.

```
instance variables
  -- create the CPU on which we will deploy the system
  cpu : CPU := new CPU(<FP>, 20E6)

operations
  public PaperPathController : () ==> PaperPathController
  PaperPathController () ==
    ( cpu.deploy(lp1); cpu.deploy(lp2);
      cpu.deploy(lp3); cpu.deploy(lp4);
      cpu.deploy(lp5);
      cpu.deploy(sc1); sc1.loopctrl := lp1;
      cpu.deploy(sc2); sc2.loopctrl := lp2;
      cpu.deploy(sc3); sc3.loopctrl := lp3;
      cpu.deploy(sc4); sc4.loopctrl := lp4;
      cpu.deploy(sc5); sc5.loopctrl := lp5;
      cpu.deploy(supervisor);
      supervisor.ejectSeqCtrl := sc1;
      supervisor.pimSeqCtrl   := sc2;
      supervisor.alignSeqCtrl := sc3;
      supervisor.corrSeqCtrl  := sc4;
      supervisor.fuseSeqCtrl  := sc5 );

  public run: () ==> bool
  run () ==
    let ppm = 50, papersize = 210, isd = 100 in
      ( supervisor.startPrintJob(ppm, papersize, isd);
        return supervisor.evalPrintJob(ppm, papersize, isd) )

end PaperPathController
```

An abstract overview of the control application architecture and its specification in VDM++ was presented in this section. The structure of the detailed model is identical to the abstract models presented in this section, as can be seen from the UML class diagram in Figure 6.17.

6.3.4 Validating the controller model

The main reason for choosing the PID control strategy is because it is well-known for its excellent performance versus computation ratio. It is a relatively simple feedback control algorithm that requires a low number of computations. However, the parameters of the PID algorithm need to be tuned in order to obtain stability and convergence of the control loop. The Ziegler-Nichols method [106] was used to obtain the appropriate values of these parameters. Some experiments were performed by automatically

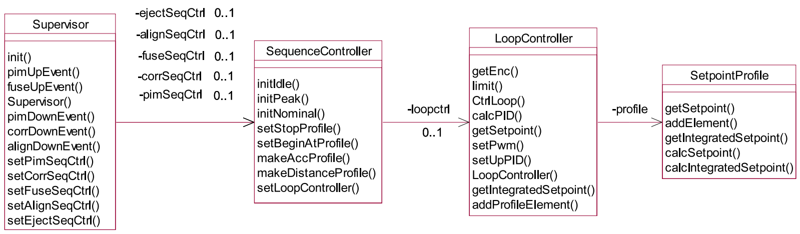


Figure 6.17: UML class diagram of the detailed controller application

synthesizing the trivial PID controller shown in Figure 6.18 towards the target on the experimental set-up using 20-SIM and observing its behavior.

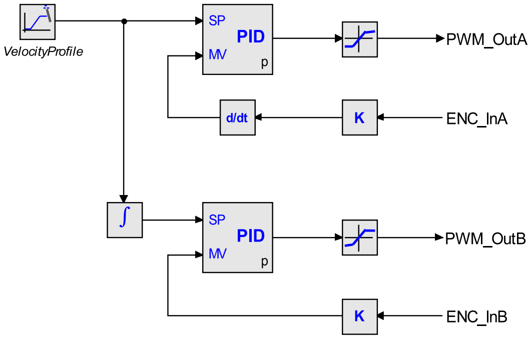


Figure 6.18: 20-SIM models used in Ziegler-Nichols tuning experiments

Surprisingly, the experiments performed by Otto and Ambrosius [3] clearly showed that controlling the velocity of the pinches using PID control was not feasible. However, this was *not* due to the control strategy itself but due the physical properties of the system. Consider a required paper path throughput of 50 pages per minute using A4 paper printed side-ways, an inter sheet distance of 50 mm and a pinch radius of 13.9 mm. This would result in a nominal sheet velocity of approximately 217 mm/sec and an angular pinch speed of 2.48 rad/sec. This corresponds to an angular speed of the motor of 5.1 rad/sec, due to the cogged wheels and belt with a 18 : 37 motor to pinch drive ratio. The encoder on the motor shaft delivers 2000 pulses per revolution or 318 pulses/rad. A sheet moving at the constant nominal speed would therefore cause 1623 encoder pulses per second. Note that the encoder measures distance rather than speed so the derivative of this signal is required for control, as shown at the top of Figure 6.18. However, the encoder value only changes by one or two encoder pulses per PID control loop iteration at 1 khz. It is obvious that the dynamic range of this derived value is insufficient for our control purpose since it is discrete and scales linearly with the paper speed. Lowering the control loop frequency would increase the dynamic range of this value but it would influence the overall performance negatively, such as the ability to prevent over- and undershoot.

Instead, the suggestion was made to move from velocity to position based control. In that case, the encoder value can be used directly as the input to the controller, as shown at the bottom of Figure 6.18, since its dynamic range is now sufficient. This conceptual change was validated on the experimental set-up using the 20-SIM approach described above and the appropriate PID parameters were finally obtained from the Ziegler-Nichols tuning procedure. However, this approach also impacts the controller model, since it was originally based on velocity profiles describing the sheet behavior. Two solutions were suggested to address this issue. The first solution involved providing position set-point profiles. This solution was rejected because it would involve using second order line elements in the *SetpointProfile* class which would increase its complexity substantially. The second solution proposed to calculate the position by integrating the velocity based set point profiles on the fly. This solution was accepted because the iteration required to calculate the integral value of the velocity set point is already naturally available: the control loop itself. The changes required to implement this solution were easy to identify and they did not affect the architecture of the controller application. The improved *calcPID* procedure is shown below.

```
class LoopController

values
  -- the PID parameters (Ziegler/Nichols tuning)
  K    : real = 4.0;
  taud : real = 2.68e-3;
  taui : real = 1.073e-2;
  ts   : real = 0.001;
  N    : real = 10

instance variables
  -- placeholders for the intermediate PID results
  curr_pos  : real := 0.0;
  prev_setp : real := 0.0;
  prev_err  : real := 0.0;
  uP : real := 0.0;
  uI : real := 0.0;
  uD : real := 0.0

operations
  private calcPID: real ==> real
  calcPID (enc) ==
    ( dcl curr_setp : real := getSetpoint(ctime),
          curr_err : real := 0;
      -- calculate the current position by numeric integration
      curr_pos := curr_pos + (prev_setp + curr_setp) / 2 * ts;
      -- calculate the error
      curr_err := curr_pos - 5e-4 * enc;
      -- calculate the proportional part
      uP := K * curr_err;
      -- calculate the integral part
      uI := uI + K * ts * curr_err / taui;
      -- calculate the differential part
      uD := taud / (ts + taud / N) * uD / N + K * (curr_err - prev_err);
      -- update the state
      prev_setp := curr_setp;
      prev_err := curr_err;
      -- return the PID result
      return uP + uI + uD )

end LoopController
```

The verification and validation activity exposed several design issues in the controller application specified in VDM++. Some of them were clear modeling errors that were easy to expose with the help of the powerful static analysis capabilities of VDMTOOLS. In particular the built-in integrity checker pin-pointed at potential weaknesses in the specification. Typically these issues are related to implicit assumptions on the run-time behavior of the model. The tool forces the engineer to make these hidden assumptions explicit for example by specifying invariants or pre- and post conditions, even before an attempt was made to execute the model. These assumptions are not necessarily errors because they may all be satisfied in practice, but this is only very rarely the case. The complexity of the application is usually large so it is virtually impossible for the engineer to take all of the possible behaviors into account when writing the model. Furthermore, these explicit assumptions capture design knowledge which is particularly useful when the model needs to be maintained over a longer period of time, is to be reused or when the model is likely to change.

Perhaps somewhat surprisingly, most of the non-trivial problems were found in the normal sequential parts of the model. For example, the functionality to calculate the set point profile for the correction pinch was notoriously difficult and error prone. The complexity does not arise from the mathematics involved but from the many additional restrictions that need to be taken into account simultaneously while performing the calculations. For example the acceleration and velocity should not exceed predefined minimum and maximum bounds. Furthermore, the acceleration and deceleration rates should be as moderate as possible since high values usually cause overshoot with significant counter control actions at the level of the loop controller, in particular when the inertia of the driven system is high. Moreover, wear and tear and power consumption usually increases with the amount of force applied.

Two approaches were used to check these complex operations at the VDM++ level. First of all, the VDMTOOLS interpreter allows the user to make an instantiation of the model and then execute parts of it, even if it is only partly complete. The algorithms can therefore be prototyped interactively whereby the user gets immediate feedback on the design decisions made. This way of working is comparable to what general purpose scientific modeling tools like Mathematica or Matlab offer. But the VDMTOOLS interpreter does not only execute the model, it also keeps track of its internal consistency by performing dynamic type checking, by enforcing state, type invariants, pre- and post conditions at run-time. The second method to ensure model consistency is to use a test framework, such as VDMUNIT as proposed in [30]. This framework is extremely well-suited for building, maintaining and operating large test suites that can be used for performance and regression testing. Usually the engineer starts by adding the simple test cases that were used in the interactive development phase. The test suite is then augmented with additional and more complex test cases to check the potential weaknesses identified by the integrity checker. VDMTOOLS maintains line-by-line test coverage information when the VDMUNIT test suite is executed. This information can be used to design specific test cases that will increase the coverage. This testing approach does not provide absolute proof that the model is correct but it has shown to be very efficient and remarkably effective in exposing problems.

6.4 Analysis of the simulation results

The aim of the modeling effort described in the previous sections is to analyze the behavior of the system as a whole. Co-simulation of the VDM++ controller model and the 20-SIM plant models is used in this thesis. These results can be investigated after the models have been carefully analyzed with respect to their consistency, as presented in the previous chapters. The results from three different phases in the system engineering life cycle are presented here. First, the results of the co-simulation will be shown in Section 6.4.1 and the results from the software-in-the-loop co-simulation are presented in Section 6.4.2. And finally, the measurements obtained from executing the control application on the experimental set-up are presented in Section 6.4.3.

6.4.1 Co-simulation of the system model

In Section 6.3.2 it was demonstrated how the plant model can be checked in isolation and in Section 6.3.4 this was shown for the controller model. However, the ability to use co-simulation or software-in-the-loop simulation of the combined set of models enables another set of consistency checks to perform. In our experience, it exposes another class of problems that go beyond what can be statically checked automatically. This is not surprising since the obvious problems have already been addressed in the domain specific analysis phases completed previously. In the case of the paper path models, we encountered two problems that were only exposed because they were tested in combination.

The first problem was a simple mistake with potentially severe consequences. The VDM++ controller model used mm/sec as the unit of measure in the set point profiles, but the loop controller measured the distance covered in radians. Hence, the integrated set point values provided to the PID controller were incorrect, causing the wrong output values to be calculated because the error was off the chart every iteration, leading to the constant spin-up of the motor at maximum speed. The root cause of this problem was easily identified since it is very simple to monitor model parameters during simulation. It would have taken substantially more time and effort if the cause of the problem had to be investigated on the embedded target.

The second problem was slightly more complex but is also due to a misinterpretation of the informal requirements. The designers of the plant model assumed that the time earmarked for the alignment of the sheet also included the time to decelerate the paper. However, the designers of the controller model followed a strict interpretation of the requirement: the time needed to decelerate is not included in the alignment time. The designers of the plant model performed a simulation using a simplified controller model and claimed that an inter-sheet distance of 50 mm was feasible at a productivity rate of 50 pages per minute and an alignment time of 200 msec. However, when the experiment was performed on the experimental set-up it turned out that their controller model was incorrect and they circumvented the problem by increasing the inter-sheet distance to 100 mm, reduced the alignment time to 100 msec and operated the alignment motor with maximum acceleration and deceleration values.

As in real life sometimes happens, these lessons learnt were not properly documented and communicated. Therefore, the designers of the controller model based their design on the wrong data. This issue became very clear when the controller model was tested in combination with the plant model. Both the different assumption on the same requirement as well as the lack of communication on the insight gained from the plant experiments were easily identified using the visualization capabilities of 20-SIM. The plant designers developed a three dimensional model of the paper path as a plug-in to their plant model. This interface is shown in Figure 6.9. The visualization runs in parallel with the co-simulation, whereby the virtual prototype is fully synchronized with the simulation state. It also provides the ability to stop, rewind and replay the visualization such that the system behavior can be inspected in detail. Using this facility it was demonstrated convincingly that two consecutive sheets would always collide if the original parameters were used. An example of the visualization in shown in Figure 6.19.

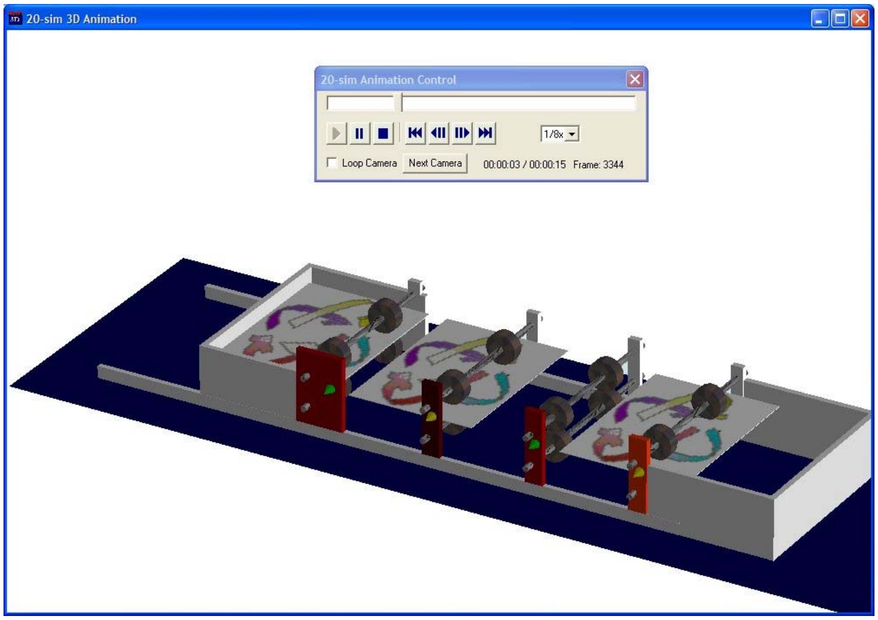


Figure 6.19: 3D visualization of the paper path co-simulation in 20-SIM

6.4.2 Software-in-the-loop co-simulation

Detailed design of the system can start as soon as the high-level VDM++ and Bond graphs models are validated. This activity usually involves lowering the abstraction level of the model, whereby technology specific implementation choices are made such as for example the selection of an operating system and implementation language. These choices may have a significant impact on the behavior of the system. It is therefore important to check for these consequences as soon as possible. This is in particular true for the controller part of the system model since the plant will ultimately be replaced by the physical system or parts thereof. The approach taken in this thesis is to lower the abstraction level of the model in VDM++ using refinement and use automatic code generation to C++ for the final implementation of the system. This approach prevents the usual paradigm shift that occurs when models are implemented. The main advantage is that the analysis tools can be used as long as possible providing maximum support during the design elaboration. Furthermore, the co-simulation interface between the interpreters can be used to continuously validate the model elaborations.

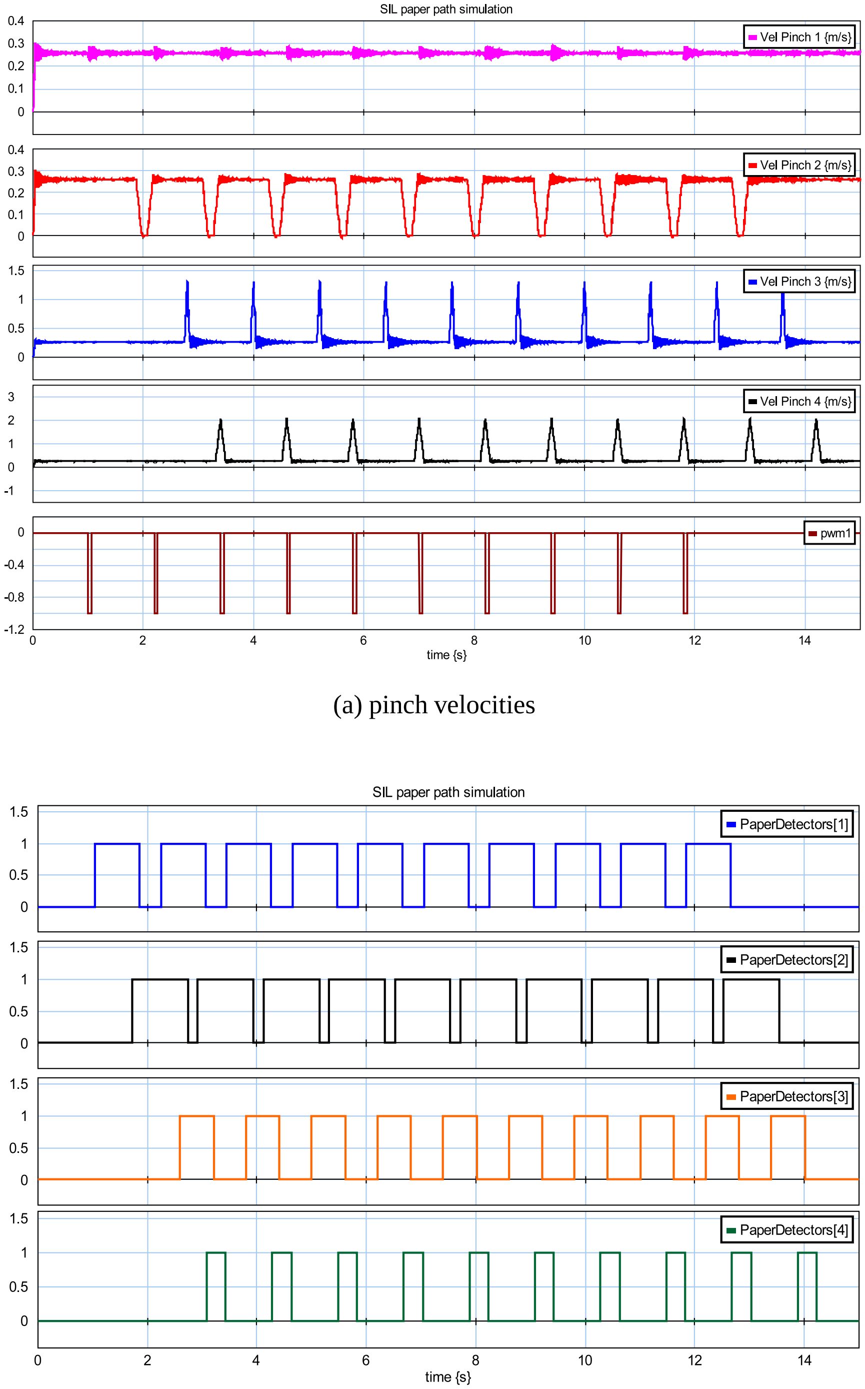
Perhaps somewhat surprising, but the use of automatic code generation is not common practice in industry. The main reason is that the code generators themselves are not trusted and it is believed that the resulting code is difficult to read and does not perform well. In general, these issues are bogus and are not based on facts. For example, code generators are generally considered suspect but the compilers used to build the application are trusted without second thought. Why would the quality of a compiler be necessarily better than the quality of the code generator? The issue of code readability usually occurs when development tools are used for source-level debugging. Not because there are problems in the generated code but rather in the hand-written code to which it interfaces. Current state-of-the-art code generators have an excellent correspondence between model and generated source code, in some cases even with the ability for reverse engineering. Last but not least, there is the issue of performance. It is important to distinguish two dimensions: size and speed. It is true that the amount of source-code generated from the model is usually 1 to 3 times larger than hand-written code. In addition, run-time libraries are usually required which adds to the source line count and object size. VDM++ is no exception to this rule. But as with compilers, it is only a matter of time before code generators are as efficient as hand-written code, but then without the usual coding errors of course. The issue of speed is typically overrated and exaggerated: timing critical parts of an application are a minority and embedded control applications are no exception. In the paper path case study it is only $10\,\%$ of the total code size and this is not uncommon in practice. If performance really is an issue then hand-coding the timing critical parts may be an option but usually it is an indicator for bad design. In contrast, the time gained by automating this activity is significant. It does not only reduce cost and increase quality but it also improves the design cycle because changes are relatively easy to accommodate.

The approach taken in this thesis is to validate the C++ code which is generated from the elaborated VDM++ model by software-in-the-loop simulation. This requires compiling the generated source-code using a compiler that is available on the simulation platform. Visual C++ was used to produce a Windows dynamic link library. This so-called DLL is a drop-in replacement for the co-simulator interface used in the previous step but retains all the analysis capabilities in the dynamic systems modeling side, including the 3D visualization shown before. The compiled C++ code is executed inside the plant simulation loop and its behavior can be observed, as is shown in Figure 6.20.

Figure 6.20 (a) clearly demonstrates that dynamic effects are taken into account as opposed to the "Happy Flow" kinematic simulations shown earlier in Chapter 5. The bottom row in this diagram indicates the set point value that is enforced on motor 1. This motor is controlled in open loop and it is used to separate sheets in the paper input module. The set point is negative because the motor is equipped with a special gearbox that reverses the direction. The top row presents the velocity of pinch 2 and it shows that the pinch has to put some effort into stabilizing the speed of the inserted sheet. The second row presents the velocity of pinch 3 which brings the sheet to a complete stand still. Note that the deceleration starts at the falling edge of the signal shown in the top row of Figure 6.20 (b) which represents PD$_{pim}$. The velocity of pinch 4 is shown in the third row in Figure 6.20 (a). Note that the acceleration is significant because the pinch needs to compensate for the alignment time. It also causes substantial amounts of overshoot after deceleration. But these oscillations have reduced to acceptable levels just before the fuse pinch is reached, as can be seen from the leading edge of the PD $_{fuse}$ signal which is shown on the fourth row in Figure 6.20 (b).

6.4.3 Measurements on the experimental set-up

The final step in the approach presented in this thesis is to run the control application code on the embedded platform, because "the proof of the pudding is in the eating". It was suggested to run the controller on the embedded target connected to a real-time simulation of the plant in the previous chapter. However, this step was skipped in our experiment because the complexity of the case study is moderate and the experimental set-up is equipped with sufficient non-intrusive measurement and debugging facilities. In other words, we believed that the remaining risks did neither require nor justify the investment of this extra intermediate step. And in hindsight, it was the right decision to do so. The unmodified C++ code used in the software-in-the-loop simulation was com-



(a) pinch velocities

(b) paper detectors

Figure 6.20: Software-in-the-loop simulation results

piled using the GNU C++ compiler and linked with the VDMTOOLS run-time library and the RTAI Linux operating system libraries. The VDM++ run-time library provides an implementation for abstract data types such as sets, sequences and mappings. It does not contain operating system specific code. The application was loaded onto the embedded platform and executed. Measurements were taken independent from the running application and this allows an objective comparison of the results. First, an overview is provided of the simulation results and measurements. The leading edge of the motor 1 set point curve and the leading and trailing edges of all paper detectors are used as a base line for this comparison. The requirements are evaluated in Table 6.1.

sheet	*co-simulation*		*software-in-the-loop*		*measurements*	
	Δ *isd*	V$_{fuse}$	Δ *isd*	V$_{fuse}$	Δ *isd*	V$_{fuse}$
01	0.3 (P)	258.87 (P)	0.3 (P)	258.87 (P)	1.8 (P)	262.59 (P)
02	0.3 (P)	258.87 (P)	0.3 (P)	258.87 (P)	0.3 (P)	258.87 (P)
03	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	2.4 (F)	251.72 (F)
04	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	0.4 (P)	257.04 (P)
05	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	1.8 (P)	253.47 (P)
06	0.3 (P)	258.87 (P)	0.3 (P)	258.87 (P)	1.1 (P)	260.71 (P)
07	0.3 (P)	258.87 (P)	0.3 (P)	258.87 (P)	2.5 (F)	264.49 (F)
08	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	2.4 (F)	251.72 (F)
09	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	3.3 (F)	266.42 (F)
10	1.1 (P)	260.71 (P)	1.1 (P)	260.71 (P)	1.8 (P)	262.59 (P)

Table 6.1: Evaluation of the control goals and requirements *(P = pass, F = fail)*

Assume a required printing productivity of 50 pages per minute, printing A4 sheet side-ways (210 mm) with an inter-sheet distance of 100 mm. This requires a nominal sheet velocity of 258.33 mm / sec. Paper detector PD$_{pinch}$ is located 537 mm down the paper path. The arrival time of the leading edge of the first sheet is therefore 2.079 seconds after the first start-of-page signal. The start-of-page signal is simulated by the falling edge of the motor 1 set point profile. The leading edge of the next nine sheets shall arrive at $2.079 + n \cdot (210\,+\,100)\,/\,258.33$ seconds. The required accuracy is 0.5 mm which corresponds to a maximum Δ *isd* of 2 msec. If the required accuracy is achieved then the *isd* column lists “P” for pass or “F” for fail otherwise.

The paper detector PD$_{corr}$ is located 361 mm down the paper path. The distance between the last two paper detectors is therefore 176 mm. If the leading edge of the pinch hits PD$_{pinch}$ then the sheet still has to travel 34 mm before the trailing edge reaches PD$_{corr}$. The measured fuse pinch speed can be determined by dividing this distance by the time difference between the two measured events. The required accuracy is $2\,\%$ of V$_{fuse}$ which corresponds to 5.17 mm / sec. If the required accuracy is achieved then the Δ V$_{fuse}$ column lists “P” for pass or “F” for fail otherwise.

The results from Table 6.1 clearly demonstrate that the control goal has been met in each phase of the development trajectory. Of course, this is not proof of correctness and neither of robustness since only a small print job was used in this case study. But it does demonstrate that the development trajectory enables structured development of complex embedded control systems. Design complexity is tackled by step-by-step elaboration of models towards implementation. Design bias is introduced consciously in each step which focuses the attention of the engineer on the associated risks. These risks are addressed and reduced by continuous validation.

6.5 Discussion and conclusions

A step-wise approach for the development of real-time embedded and distributed control systems was proposed in this chapter and it was put to the test on a non-trivial case study inspired from industrial practice. Continuous time plant models and discrete event controller models where developed using Bond graphs and VDM++ respectively. It was shown how the domain specific analysis tools and techniques can be used to improve the quality of the specifications in isolation, here we used 20-SIM and VDMTOOLS. It was also demonstrated how the combined models can be inspected using the notational extensions and reconciled semantics as presented in chapters 3 and 4 respectively. The problems exposed by this enhanced analysis capability clearly contributed to the cross-discipline design dialogue which is usually lacking in the early phases of the system design. Engineers are forced to investigate the results together in order to find the root cause of the problem. This typically leads to “what-if” questions that can usually be answered by changing some model parameters and rerunning the simulation. This dialogue is usually very constructive because it is relatively easy to change the models. In contrast, the “blame game” is usually played if problems are found during system integration because the number of changes required at the code level are likely to be very significant.

Similarly, the impact of more complex multi-disciplinary design questions, such as optimal position of the sensors versus the computational load and control performance can also be addressed, although it is not explicitly demonstrated in this chapter. We showed how the step-wise development approach fits into the system engineering life cycle and how the path towards the system implementation can be kept under control. An iterative refinement approach was proposed whereby continuous validation is attempted after each step. The impact of this approach was demonstrated by comparing the simulation results of abstract and high-level models to the measurements obtained from the experimental set-up. It convincingly showed the feasibility of the proposed approach, since these results were virtually identical while meeting the overall control objectives. The upfront investment in the modeling effort and the continuous validation approach is in our opinion and industrial experience significantly less than the amount of time required to fix problems at integration time, although there is no hard evidence provided in this chapter to support this claim.

Chapter 7

Conclusions and Outlook

The area of embedded systems brings together computer science, control, electrical and mechanical engineering. Contributions from all these areas of expertise need to work, both in isolation and collectively, in order to achieve the overall system objectives. Multi-disciplinary design of embedded control systems therefore really requires to go beyond the ordinary in order to be or become successful. The usual barriers that exist between these disciplines, both in academia and industry, need to be resolved in order to build embedded systems reliably and predictively, as Henzinger and Sifakis pointed out in their key-note address at Formal Methods 2006 [53]. They suggest to create a new scientific foundation for this class of problems and perhaps this is indeed the way forward. Of course, such an endeavor is beyond the scope of a single PhD thesis and even of a large-scale collaborative research project such as BODERC.

This thesis builds upon the common scientific foundation which is already readily available: mathematics, logic and physics. The focus of this work has been on the integration of existing well-founded modeling and analysis techniques from different engineering disciplines, both in theory and practice. The ability to support cross-discipline design dialogue with appropriate tools, which are also embedded in an engineering method, will remove one of the most dominant obstacles observed in industrial system engineering to date. This has been the main motivation for the chosen research focus. The purpose of this chapter is to assess whether or not this has been achieved. A summary of the research contributions is presented in Section 7.1 and the objectives of thesis are evaluated in Section 7.2 and we close this chapter with a look at the future.

7.1 Summary of research contribution

A number of state-of-the-art performance evaluation methods and tools were put to the test on a simple case study that has been inspired by industrial practice. The aim of the exercise was to determine the capabilities and restrictions of these methods in the context of a few typical design trade-off issues between functional and extra-functional properties that the system should possess. The real value of the study is in the dialogue caused by comparing the numbers obtained from the analysis. The conclusion is that these numbers should always be considered to be suspect because they are derived from a model which is an abstraction of reality. Implicit assumptions made while modeling, or hidden limitations of the techniques used, are usually exposed by comparison to results obtained from different techniques. This rather obvious insight is often forgotten and taken for granted, usually because getting a quantitative result is already considered a victory and a major step forward at design time. It is therefore considered to be good engineering practise to use a multi-method modeling approach to expose potential problems and misconceptions as early as possible in the design process. This increases the confidence in the models and the analysis results but does require commitment and endurance.

The method comparison has lead to a number of scientific publications, most notably [105], [51], [99] and inspired three related MSc projects [78], [23] and [79]. Furthermore, other researchers have looked at different aspects of the case study, such as [35], [36], [12] or use different approaches to construct quantitative performance models, such as [34, 37] and [33]. Last but not least, the comparison was continued with a significantly larger scope involving more case studies and additional tools in [81]. The authors of this paper also exposed a problem in the timed automata models presented earlier in [51] which has been corrected in this thesis. Again, it underlines the importance of the observation made earlier, especially since several peer reviews had not exposed the problem.

The choice for VDM++ in this thesis was mainly subjective and inspired by the previous experience of the author. The notation is well-established in both academia and industry and there exists robust and industrial strength tool support, including a round-trip engineering capability to UML and the availability of code generators. Furthermore, at the start of the BODERC project, there was keen interest from the community at large to extend the notation for use in the embedded systems domain, which provided a stable basis for the research efforts described in this thesis.

The first step towards the final goal is presented in Chapter 3. Timed VDM++ was extended with an explicit notion of system architecture, which enables the creation of context-aware software models at a very high level of abstraction. These language extensions were given an explicit formal semantics and prototype tools were developed to demonstrate the improvements on the in-car radio navigation case study from the earlier comparison work, providing on par results. This work has lead to a number of scientific publications, in particular [101], [30], [100] and [98] and was later implemented in VDMTOOLS.

The second step towards the final goal is presented in Chapter 4. The extended semantics of the improved VDM++ notation developed in Chapter 3 was reconciled with the semantics of continuous time simulations, for which Bond graphs are used in this thesis. The choice for this particular technique was twofold. First of all, this notation is particularly well-suited to describe and analyze dynamic systems and it is targeted explicitly towards multi-disciplinary design challenges. For example, it is possible to describe electronics, hydraulics, pneumatics and mechanics from within a single mathematical framework. Second of all, industry grade tool support is available with access to their main researchers through partners in the BODERC project. Prototype tools were developed to demonstrate the tool coupling using a simple and intuitive example of a water tank level controller. This work was published in [102].

Development projects in industry usually consist of a significant number of people with different backgrounds and experience, which are involved over a long period of time, sometimes even working on several locations simultaneously. Managing these complex projects requires a suitable development process that provides each stakeholder with the overview necessary to perform his or her job. Introducing a novel technique into industrial practice requires embedding into such a development process and this issue is investigated in Chapter 5. Contemporary design trajectories for embedded control applications and formal software models were identified and compared to a classic industrial development process. It was shown how these approaches can be usefully combined.

Finally, the results mentioned above were applied to a significant case study: the design and analysis of the embedded control of a printer paper path. An informal description of the case study, the models and the results obtained are presented in Chapter 6. The exercise has clearly and convincingly demonstrated the added value of the notational enhancements from Chapter 3 and the tool integration from Chapter 4.

7.2 Evaluating the objectives of this thesis

Challenging research goals were set in Section 1.3 and they will be discussed here.

Addressing system-level design. The case study in Chapter 6 has demonstrated that the research results from this work can indeed be used to address multi-disciplinary system-level design. The ability to create high-level and abstract models of both the software and the hardware architecture enables for example the discussion on distribution and deployment, as was shown in Chapter 3 and [100]. This technique can be used to replace the typical oversimplified notion of software and hardware which is used in most contemporary dynamic system modeling approaches, as was presented in Chapter 4. The end result is an integrated multi-method modeling and analysis approach that can improve the cross-discipline system-level design dialogue significantly.

Prediction of functional and extra-functional properties. The prediction of functional properties of the system is of course intrinsically provided by the methods used: VDM++ and Bond graphs. There exist many types of extra-functional system-level properties, such as for example quality, dependability, maintainability and adaptability. The main focus of the work presented in this thesis has been on performance, in particular on the timeliness of distributed embedded real-time control systems. A solution is provided by means of the context-aware software models presented in Chapter 3. However, it is not possible to claim that all types of extra functional properties can be suitably addressed. Neither is it possible to claim that hard guarantees on worst-case timeliness properties can be provided. Simulation has known limitations with respect to its ability to cover the state space exhaustively and this is also true for the work presented here. But this is not necessarily a show stopper in practice. The insight gained by early system life-cycle modeling and analysis, as advocated in this thesis, should be sufficiently accurate such that it can replace hand-waving. The case studies presented in this thesis have demonstrated that this is well within reach. Sensitivity analysis, as shown in Section 2.3.1, can pin-point potential bottlenecks in the design even though the technique itself is known to provide pessimistic results.

Heterogeneous levels of abstraction. Early system life-cycle modeling requires the ability to construct a system model out of sub-system models that are not necessarily at the same level of maturity. For example, one sub-system model may be specified abstractly while another is already more detailed, but neither should restrict the analysis capability at the system level. This ability is basically provided by the methods used. Both VDM++ and Bond graphs have explicit support for multiple levels of abstraction. For example, implicit and explicit operations can be used in VDM++ while decomposition in Bond graphs is strongly developed. The work presented here has not affected that capability negatively. Since the interface between the two models is defined in terms of sensor and actuator signals, either model can be replaced without affecting the other. Different control strategies can be tried on a single plant model or different plant models can be used to validate a single controller implementation.

Cost effectiveness. There are many variables that determine the cost effectiveness of a method or technique in practice. The method comparison in the first part of this thesis has shown that two aspects are dominant: the availability of domain and method knowledge. Detailed insight into the application domain as well as detailed knowledge of the methods used is required in order to be effective. Both aspects are typically not available in a single person and it is the task of the system architect to bring the relevant experts together. Then again, the end result will principally be determined by the quality of the people actually performing the work. This has not been studied in this thesis. Nevertheless, some observations can be made on cost effectiveness. First of all, the VDM++ language changes proposed in Chapter 3 have significantly reduced the model size while increasing its capabilities. This has a positive impact on the time required for model construction and maintenance. Second of all, the use of abstract and high-level system models in the early life-cycle, as proposed by the BODERC project, has been successfully applied in industry. For example, Orbons states in [47] that the "HappyFlow" modeling approach has enabled Océ to skip a complete physical machine-build iteration cycle, saving many man-years of effort.

Adoption in industry. Industrial applicability is obviously closely related to the issue of cost effectiveness mentioned above. But formal description techniques seem to have difficulty reaching the main stream of system engineering even despite the fact that there is sufficient evidence to demonstrate its positive impact and relevance [72]. Based on personal experience, this problem is intrinsic to the term "formal method".

Popular belief in industry is that a PhD is required in order to use these techniques effectively. This myth is perhaps strengthened by the connotation caused by the word "formal". It seems to imply that these techniques are an all-or-nothing approach aiming at proving absolute correctness. This is of course not true. As already mentioned in Chapter 3, VDM++ in particular has been applied in a pragmatic style leading to several very successful industrial applications [30]. Formal languages excel in abstraction, which is considered to be a critical success factor required in problem solving [63]. But the effort spent on modeling should be balanced with the insight gained otherwise the technique will not be adopted in industry [29]. Learning a new formal notation is usually considered a high hurdle. But the training effort required is in general on par with learning any new programming or specification language and does certainly not require a PhD. For example, Felica Networks, a subsidiary of Sony Corporation recently reported on the successful development of a firmware application for a new integrated circuit for which they trained 50 engineers during one week in VDM++. The 150 man year project produced a 700 page executable specification in VDM++ and 10 million test cases providing near-perfect test coverage. The project was completed on time, within budget and with a considerable higher measured quality than earlier releases of the same product [65], while only one external VDM++ expert assisted the newly trained engineers in part time. The solutions proposed in this thesis do not significantly increase the learning time.

The second issue is related to the word "method". Formal description techniques are disruptive to the current engineering practice because effort is shifted towards the start of the project [91]. Typically, more time will be spend on modeling and analysis as compared to traditional design. This usually makes project managers nervous because progress is in general difficult to measure. This activity is often even considered unproductive by managers, in particular when concrete design artifacts are lacking or only available at the very end. They do not dare to rely on the premise that the implementation and test phases are usually significantly shorter and more predictable because of the higher quality of those initial design artifacts. A solution to this problem is to embed formal techniques, as a step-wise approach with identifiable intermediate deliverables, into a commonly accepted development process. The higher upfront investment cost will otherwise simply not be accepted. This issue is addressed in Chapter 5, where such an embedding of techniques into an industrial development process is proposed.

Another important point is the scalability of the method and tools. It would be foolish to claim that the solution presented in this thesis is the “silver bullet”, the antidote to all problems in embedded systems design. On the contrary, it is specifically targeted at embedded control systems, possibly consisting of several interconnected computing nodes. It would be very hard if not impossible to analyze massively parallel applications, such as for example image processing or large-scale wireless sensor networks, even though the language is probably sufficiently expressive to describe such systems. Neither does it guarantee to provide hard bounds to the timeliness properties. The simulation based technique used cannot guarantee complete coverage of the state space of the model. Exhaustive techniques, such as model checking, can provide hard bounds but only under specific circumstances such as limited model size and complexity, in particular for hybrid and stochastic models. Hence, the simulation based technique is preferred in this thesis because it is has a better chance of scaling up towards industry needs. In early life-cycle multi-disciplinary system-level modeling it is better to have an approximate answer than no answer at all, in particular in support of an iterative design process. Additional deductive or exhaustive analysis techniques such as interactive theorem proving or model checking can be used on (parts of) the model at a later stage if more accuracy is required. This is also good engineering practice because the amount of effort and skill involved in performing these particular tasks is usually much larger than the light-weight simulation-based modeling approach proposed in this thesis. It is not advisable to spend this kind of effort if the model is not at least order of magnitude correct. The co-simulation interface presented in Chapter 4 increases the time required to perform the system simulation. But the insight gained from the improved analysis outweighs the performance loss.

But will the solution proposed in this thesis ever be used in industry? The current maintainers of VDMTOOLS, CSK Systems Corporation in Japan, have already adopted the results presented in Chapter 3 and it is available in version 8.0 which has been officially released[1] in July 2007. CSK has already indicated that the continuous time interface described in Chapter 4 will also become part of their product. This will at least enable the industrial uptake of the research results presented in this thesis.

7.3 Future work and outlook

An important point not yet addressed in the previous section is the ability to adapt the methodology to the ever changing and increasing needs in the embedded systems domain. This thesis has shown that it is both possible and fruitful to combine engineering methods that seem to have only very little in common at first glance. The integrated solution leverages the analysis potential and removes the methodology lock-in that many

[1] VDMTOOLS is available free of charge from *http://www.vdmtools.jp/en*.

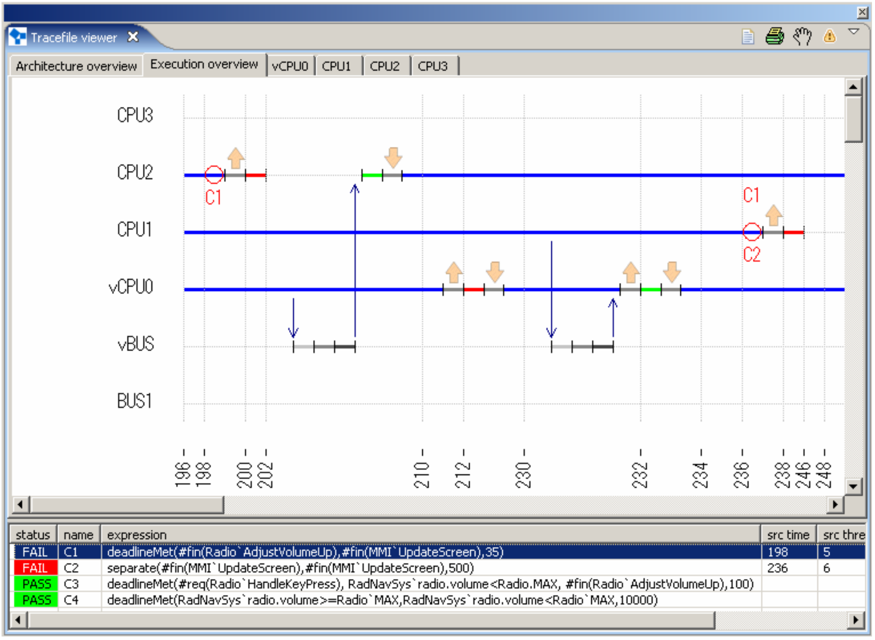
practitioners face and seem unable to break. But does the solution presented here cause a vendor lock-in? Are we forced to use VDM+ and Bond-graphs to reach these results? Well, in fact, this is not the case. The reconciled semantics presented in Chapter 4 is not specific to the tools used, neither is the extended semantics shown in Chapter 3. This is also confirmed by results from related work presented in [41, 77]. It is possible to replace VDM++ by another discrete event simulation technique and to replace Bond graphs by another dynamic systems modeling approach, although we have not provided proof in this thesis. Currently, an attempt is made in the VIEWCORRECT project to reach comparable results using POOSL and 20-SIM and CSK Systems Corporation is considering combining VDM++ and SCILAB.

An obvious future work activity would be tool improvement both in terms of performance and capabilities. The former particularly concentrates on the simulator and the co-simulation interface and the latter concentrates on enhanced visualization and support for additional scheduling techniques. Three additional directions for future research work have been identified due to feedback received on exposing our results to our peers in academia and industry.

1. First of all, the VDM++ notation presented in Chapter 3 can again be extended quite easily on the syntactic level to describe probabilistic properties of a system. For example, we use the `duration` and `cycles` statements to specify the timing behavior of (a part of) the model. Currently, these constructs take only a single parameter which is typically used to denote the worst-case response time. Instead, an argument pair can be used to capture the expected best and worst-case response times. A value is chosen from this interval at simulation time according to some predefined selection strategy. This could be a global setting for the interpreter or it can be described locally, for example by adding a third parameter. This parameter could for example be a higher-order operation that implements the selection strategy, for example: best-case always, worst-case always, pseudo random selection according to some distribution function or even context-aware selection functions that mimic caching behavior. Initial experiments have shown that these extensions are feasible and would provide results comparable to those reported in [36], however the consequences for the operational semantics and the simulation speed have not been investigated.

2. The second direction for future work is to decouple the specification of the validation property from the model itself. The usual approach is to include a programmable observer inside the model. Hence, there is no distinction between the model and the observer as can be seen from the definition of *evalPrintJob* in the *Supervisor* class presented in Section 6.3.3. This increases the model complexity unnecessarily and causes a model maintenance issue since there may be many of these properties and they are likely to change often. The approach proposed is to specify so-called validation conjectures over system traces. These traces are constructed on-the-fly by the simulator and contain both observable and internal behaviors of the system, such as operation invocations with their associated parameter values, and also state changes, such as assignments to class instance variables. The validation conjectures are analyzed during the simulation run and the result can be visualized after the simulation run is complete, as is shown in Figure 7.1. The top part of the screen shows the execution trace of the model in terms of task and communication activity per resource, as was shown in Chapter 3. The bottom part of the screen is new and lists the validation conjectures

and their status. The execution trace is centered when the user selects a particular validation conjecture. The circles on the traces of resources CPU2 and CPU1 indicate the begin and end points that apply to failing validation conjecture C1. The engineer can use this diagram to locate the cause of the problem. This research direction seems promising and first results from this approach have been published in [32].



status	name	expression	src time	src thre
FAIL	C1	deadlineMet(#fin(Radio`AdjustVolumeUp),#fin(MMI`UpdateScreen),35)	198	5
FAIL	C2	separate(#fin(MMI`UpdateScreen),#fin(MMI`UpdateScreen),500)	236	6
PASS	C3	deadlineMet(#req(Radio`HandleKeyPress), RadNavSys`radio.volume<Radio.MAX, #fin(Radio`AdjustVolumeUp),100)		
PASS	C4	deadlineMet(RadNavSys`radio.volume>=Radio`MAX,RadNavSys`radio.volume<Radio`MAX,10000)		

Figure 7.1: Visual presentation of validation conjectures and their state

3. A similar problem occurs when dependability of a system is under study. Disturbance models need to be added in a controlled and repeatable way. The usual approach is to copy and modify an existing model in order to describe the disturbance. This usually causes model maintenance issues, in particular when several failure modes are analyzed simultaneously. The co-simulation interface proposed in Chapter 4 provides a potential solution to this problem, since it is positioned exactly at the sensor-actuator interface. The behavior of both connected simulators is defined by the information exchanged over the interface. It is therefore the ideal location to interfere, but without the direct need to change either model. For example, it is possible to change both temporal as well as state properties of the information exchanged over the interface during simulation. Signals may be suppressed, delayed or even injected additionally, for example to model erratic behavior of sensors or actuators. Similar, the values exchanged can be modified on-the-fly, e.g. to represent “stuck-at-*x*” symptoms. In other words an explicit fault model is added to the simulation interface. This direction seems promising and first results from this approach have been published in [4].

The longer term challenge is to close the gap between simulation, model checking and formal proof, to enhance the level of rigor far beyond what can be provided with the solution proposed in this thesis. I believe this requires additional work in two main directions. First of all by developing support for the automated mapping of models between different paradigms. This would enable us to unleash different validation and verification strategies on a set of consistent models with a common semantics. The area of model driven architecture (MDA) shows promising results but the mapping support is currently mainly syntactical while the real challenges are at the semantic level. The area of program abstraction has demonstrated some spectacular results in recent years using dedicated uni-directional mapping approaches for specific semantic aspects of some model. I believe these viewpoints need to merge in order to get to the next generation of robust tool support for multi-disciplinary and complex system design. Secondly, the analysis techniques themselves need to be improved substantially in order to provide the scalability and flexibility required to meet the demands of industrial size problems. This requires developing better (faster) analysis algorithms and capturing the heuristics of well-known modeling strategies and analysis optimizations available today. But a prerequisite to all these directions is continued research in the area of language semantics and unification, as proposed by Henzinger and Sifakis in [53].

Outlook. There is genuine interest from both the academic and industrial communities to continue research along the lines suggested in this thesis. For example, the Overture[2] project was started several years ago and it is currently gaining momentum fast. The aim of the project is to develop a set of open-source Eclipse [3] plug-ins to support research on VDM++ and related notations. Several MSc and PhD projects are lined up in different countries on related topics and international workshops are organized on a regular basis. CSK Systems Corporation have taken their role as maintainer of VDMTOOLS very seriously, meanwhile creating a significant user community in Japan. They are also actively involved in the Overture project, in particular to provide lessons learnt from industry. This information is used to focus on-going research by means of maintaining a strategic research agenda. The Overture community is also active in the Grand Challenges initiative on the Verified Software Repository [4], where work is started on the Mondex electronic purse, the POSIX fault-tolerant flash file system and last but not least PACEMAKER. The latter has been modeled with the VDM++ language extensions described in Chapter 3 [25].

The tools developed in this thesis are available at *http://www.overturetool.org* and the models are available at *http://www.marcelverhoef.nl*.

[2] See *http://www.overturetool.org*.

[3] See *http://www.eclipse.org*.

[4] See *http://www.fmnet.info/vsr-net/* and *http://www.cas.mcmaster.ca/sqrl/pacemaker.htm*.

Bibliography

[1] Emile Aarts and Stefano Marzano. *The New Everyday – Views on Ambient Intelligence*. 010 Publishers, 2003. Koninklijke Philips Electronics NV.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] Frank Ambrosius. Modelling and distributed controller design of the BodeRC paper-path setup. Master's thesis, University of Twente, department of Electrical Engineering, Mathematics and Computer Science, January 2007. Appeared as technical report 003CE2007. On-line available at *http://www.ce.utwente.nl*.

[4] Zoe Andrews, John Fitzgerald, and Marcel Verhoef. Resilience modelling through discrete event and continuous time co-simulation. In *Proceedings of the Dependable Systems Network - DSN'07*, 2007.

[5] F. Bacelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons Ltd, August 1992.

[6] Jan Beckers, Maurice Heemels, Bjorn Bukkems, and Gerrit Muller. Effective industrial modeling: The example of happy flow. In Gerrit Muller and Maurice Heemels, editors, *Model-based design of high-tech systems*, pages 77–88. Embedded Systems Institute, 2006. See also [47].

[7] Gerd Behrmann, Alexandre David, and Kim Gulstrand Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[8] Gerd Behrmann, Alexandre David, Kim Gulstrand Larsen, John Håkansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Third International Conference on the Quantitative Evaluation of Systems (QEST'06)*. IEEE, 2006. This paper is available on-line at *http://dx.doi.org/0-7695-2665-9/06*.

[9] A.J. Bennet, A. J. Field, and M. C. Woodside. Experimental Evaluation of the UML Profile for Schedulability, Performance and Time. In *UML2004 - The Unified Modeling Language*, volume 3273 of *Lecture Notes in Computer Science (LNCS)*, pages 143–157. Springer, 2004.

[10] JPL Special Review Board. Report on the loss of the Mars Polar Lander and Deep Space 2 missions. Technical Report JPL D-18709, Jet Propulsion Laboratory, March 2000. Available on-line at *http://klabs.org/reports.htm*.

[11] Barry W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986. On-line available at *http://doi.acm.org/10.1145/12944.12948*, also appeared in the May 1988 issue of IEEE Computer.

[12] Egor Bondarev, Michel Chaudron, and Peter de With. Quality-oriented design space exploration for component-based architectures. Technical report, Technical University of Eindhoven, Department of Mathematics and Computer Science, February 2006.

[13] P.C. Breedveld. Multibond-graph elements in physical systems theory. *Journal of the Franklin Institute*, 319(1/2):1–36, 1985.

[14] J.F. Broenink and G.H. Hilderink. A structured approach to embedded control systems implementation. In *International Conference on Control Applications, CCA*, pages 761–766. IEEE, September 2001. Available on-line at *http://dx.doi.org/10.1109/CCA.2001.973960*.

[15] Frederick P. Brooks. *The Mythical Man-Month*. Essays on Software Engineering. Addison-Wesley, 1995. Reprint of the original 1975 book.

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

[17] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[18] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe*, pages 190–195, 2003.

[19] ControlLab Products. 20-sim, See also *http://www.20sim.com*.

[20] Henk Corporaal. Embedded system design. In *PROGRESS White Papers*, pages 7–27. Stichting Technische Wetenschappen, 2006. Available on-line at *http://www.stw.nl/Programmas/Progress*.

[21] CSK. Development guidelines for real-time systems using VDMTOOLS. Technical report, CSK Systems Corporation, 2007. Available on-line at *http://www.vdmtools.jp/en*.

[22] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley, July 1999.

[23] Menno M.C.M. de Hoon. Performance analysis of distributed real-time embedded systems. Master's thesis, Technical University Eindhoven, January 2006.

[24] Derek J. Andrews and Peter Gorm Larsen and Bo Stig Hansen and Hans Brunn and Nico Plat and Hans Toetenel and John Dawes and Graeme Parkin and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[25] Hugo Daniel dos Santos Macedo. Validating and understanding the Boston Scientific PACEMAKER requirements. Internship report, Minho University, Braga, Portugal, 2007. Available at *http://www.overturetool.org*.

[26] Bruce Powell Douglas. *Real-Time UML Workshop for Embedded Systems*. Embedded Technology. Newnes - Elsevier, 2007.

[27] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[28] ESA. PSS-05-10 guide to software verification and validation. Technical report, European Space Agency, ESA Publication Division, ESTEC, Noordwijk, The Netherlands, 1994. ISSN 0379-4059.

[29] John Fitzgerald and Peter Gorm Larsen. Balancing insight and effort: the industrial uptake of formal methods. *Formal Methods and Hybrid Real-Time Systems*, 4700:237–254, 2007. *http://dx.doi.org/10.1007/978-3-540-75221-9_10* .

[30] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005. See also *http://www.vdmbook.com*.

[31] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[32] John S. Fitzgerald, Peter Gorm Larsen, Simon Tjell, and Marcel Verhoef. Validation support for distributed real-time embedded systems in VDM++. In *Proceedings of 10th IEEE International Symposium on High Assurance System Engineering (HASE)*, pages 331–340. IEEE, 2007. Available on-line at
http://doi.ieeecomputersociety.org/10.1109/HASE.2007.76 .

[33] Bastian Florentz. Inside architecture evaluation: Analysis and representation of optimization potential. In *Sixth IEEE/IFIP Working Conference on Software Architecture (WISCA)*, Mumbai, India, January 2007.

[34] 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, pages 329–348. Springer, 2007. Appeared earlier as [37]. Available on-line at
http://dx.doi.org/10.1007/978-1-4020-6149-3 .

[35] Oana Florescu. *Predictable Design for Real-Time Systems*. PhD thesis, Technische Universiteit Eindhoven, 2007. ISBN 978-90-386-1654-4.

[36] Oana Florescu, Menno de Hoon, Jeroen Voeten, and Henk Corporaal. Probabilistic modelling and evaluation of soft real-time embedded systems. In *Proceedings of SAMOS 2006*, volume 4017 of *Lecture Notes in Computer Science*, pages 206–215, 2006. *http://dx.doi.org/10.1007/11796435_22* .

[37] Oana Florescu, Jeroen Voeten, Marcel Verhoef, and Henk Corporaal. Reusing real-time systems design experience through modelling patterns. In *Forum on specification and Description Languages (FDL)*. ECSI, 2006. Received the best

paper award at FDL 2006. This paper is available on-line at *http://www.es.ele.tue.nl/premadona/publications/FVVC06.pdf*.

[38] H.J.M. Freriks, W.P.M.H. Heemels, G.J. Muller, and J.H. Sandee. Budget-based design. In *Model-based design of high-tech systems*, pages 59–76. Embedded Systems Institute, 2006. See also [47].

[39] H.J.M. Freriks, W.P.M.H. Heemels, G.J. Muller, and J.H. Sandee. On the systematic use of budget-based design. In *Systems Engineering: Shining Light on Though Issues*, Proceedings of the 16^{th} Annual International INCOSE Symposium. INCOSE, 2006. Paper is available on-line at *http://www.incose.org/ipub*.

[40] Marc Constantijn Willem Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Technical University Eindhoven, October 2002. On-line available at *http://www.es.ele.tue.nl/∼mgeilen/publications/thesis.pdf*.

[41] Luiza Gheorghe, Faouzi Bouchhima, Gabriela Nicolescu, and Hanifa Boucheneb. Formal definitions of simulation interfaces in a continuous/discrete co-simulation tool. In *Proc. IEEE Workshop on Rapid System Prototyping*, pages 186–192. IEEE Computer Society, 2006. This paper is on-line available at *http://doi.ieeecomputersociety.org/10.1109/RSP.2006.18*.

[42] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings of the Fifth Euromicro Workshop on Real-time Systems*, pages 118–123. IEEE, 1993.

[43] Systems Integration Requirements Task Group. *ARP 4754: Certification Considerations for Highly Integrated or Complex Aircraft Systems*. Aerospace Recommended Practice. SAE International, April 1996. *http://www.sae.org*.

[44] Ernst Hairer, Syvert P. Nørsett, and Gerhard Wanner. *Solving ordinary differential equations I : Nonstiff problems*. Springer, second edition, 1993.

[45] Ernst Hairer and Gerhard Wanner. *Solving ordinary differential equations II : Stiff and differential-algebraic problems*. Springer, second edition, 1996.

[46] Arne Hamann, Rafik Henia, Razvan Racu, Marek Jersak, Kai Richter, and Rolf Ernst. SymTA/S - Symbolic Timing Analysis for Systems. In *Work In Progress session - Euromicro Workshop on Real-time Systems*, 2004.

[47] Maurice P.M.H. Heemels and Gerrit J. Muller, editors. *Boderc: Model-based design of high-tech systems*. Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, NL, 2006. Available on-line at *http://www.esi.nl/boderc*.

[48] W.P.M.H. Heemels, L. Somers, P.F.A. van den Bosch, Z. Yuan, B. van der Wijst, A. van den Brand, and G.J. Muller. The key driver method. In *Model-based design of high-tech systems*, pages 27–42. Embedded Systems Institute, 2006. See also [47].

[49] W.P.M.H. Heemels, L. Somers, P.F.A. van den Bosch, Z. Yuan, B. van der Wijst, A. van den Brand, and G.J. Muller. The use of the key driver technique in the design of copiers. In *Proceedings of the International Conference on Software and Systems Engineering and their Applications (ICSSEA)*, 2006.

[50] W.P.M.H. Heemels, E.H. van de Waal, and G.J. Muller. A design methodology for high-tech systems. In *Model-based design of high-tech systems*, pages 11–26. Embedded Systems Institute, 2006. See also [47].

[51] Martijn Hendriks and Marcel Verhoef. Timed automata based analysis of embedded systems architectures. In *Workshop of Parallel and Distributed Real-Time Systems (WPDRTS)*. IEEE, 2006. This paper is available on-line at
http://dx.doi.org/10.1109/IPDPS.2006.1639422.

[52] Dan Henriksson. *Flexible Scheduling Methods and Tools for Real-Time Control Systems*. PhD thesis, Lund Institute of Technology, Department of Automatic Control, December 2003. See also *http://www.control.lth.se/truetime/*.

[53] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2006. This paper is available on-line at
http://dx.doi.org/10.1007/11813040 1.

[54] Jozef Hooman, Nataliya Mulyar, and Ladislau Posta. Coupling Simulink and UML models. In B. Schnieder and G. Tarnai, editors, *FORMS/FORMATS 2004*, pages 304–311, 2004.

[55] Jozef Hooman and Mark van der Zwaag. A semantics of communicating reactive objects with timing. *Software Tools for Technology Transfer*, pages 97–112, 2006.

[56] Jozef Hooman and Marcel Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In *Festschrift in honor of Willem-Paul de Roever*, LNCS Festschrift Series. Springer Verlag, 2008. (to appear).

[57] Johann Hörl and Bernhard K. Aichernig. Validating voice communication requirements using lightweight formal methods. *IEEE Software*, 13–3:21–27, May 2000.

[58] ITRS. International technology roadmap for semiconductors. Available on-line at *http://public.itrs.net*, 2007.

[59] Chris W. Johnson. The natural history of bugs: Using formal methods to analyse software related failures in space missions. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2005.

[60] Jim Johnson. *My Life Is Failure*. Standish Group International, Inc., 2006. Co-author of the original 1994 CHAOS report. See *www.standishgroup.com*.

[61] Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics: Modeling and Simulation of Mechatronic Systems*. Wiley-Interscience, third edition, 2000.

[62] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *ASAP '97: Proc. of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, page 338, Washington, DC, USA, 1997. IEEE Computer Society.

[63] Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):37–42, April 2007.

[64] Philippe Kruchten. *The Rational Unified Process – An Introduction*. Object Technology Series. Addison-Wesley, 1999.

[65] Taro Kurita, Toyokazu Oota, and Yasumasa Nakatsugawa. Formal specification of an embedded IC for cellular phones. In *Proceedings of Software Symposium 2005*, pages 73–80. Software Engineering Association of Japan, June 2005. Only available in Japanese.

[66] Kevin Lano. Logic specification of reactive and real-time systems. *Journal of Logic and Computation*, 8(5):679–711, 1998.

[67] Peter Gorm Larsen. Ten years of historical development: "bootstrapping" VDMTOOLS. In *Journal of Universal Computer Science*, volume 7, pages 692–709. Springer, 2001.

[68] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*, pages 604–618. VDM Europe, Springer, March 1991.

[69] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Number 2050 in Lecture Notes in Computer Science (LNCS). Springer, 2001.

[70] Jim Ledin. *Simulation Engineering - Build Better Embedded Systems Faster*. Embedded Systems Programming. CMP Books, 2001.

[71] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. ARIANE 5 – flight 501 failure – report by the inquiry board. Technical report, European Space Agency, July 1996. Available on-line at *http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf* .

[72] Tiziana Margaria, Bernhard Schätz, and Marcel Verhoef. Formal methods going mainstream: Costs, benefits, experiences. *BCS-FACS FACTS*, 2006(2):34–38, September 2006. Report on the ForTIA Industry Day at FM'05. Available on-line at *http://www.bcs-facs.org/newsletter/facts200609.pdf* .

[73] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[74] Gordon E. Moore. Cramming more components onto integrated circuits. In *Electronics*, volume 38. April 1965.

[75] Paul Mukherjee, Fabien Bousquet, Jerome Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In Juan Bicarregui and John Fitzgerald, editors, *The Second VDM Workshop*, September 2000.

[76] Gerrit Muller. *CAFCR: A Multi-view Method for Embedded Systems Architecting; Balancing Genericity and Specificity*. PhD thesis, Technische Universiteit Delft, 2004. Available on-line at *http://www.gaudisite.nl*.

[77] Gabriela Nicolescu, H. Boucheneb, L. Gheorghe, and F. Bouchhima. Methodology for efficient design of continuous/discrete-events co-simulation tools. In James Anderson and Ralph Huntsinger, editors, *High Level Simulation Languages and Applications - HLSLA*, pages 172–179. SCS, San Diego, CA, 2007.

[78] H. P. Oosterom. On the verification of real-time distributed embedded control systems. Master's thesis, University of Twente, Department of Electrical Engineering, August 2006. On-line available at *http://www.ce.utwente.nl/rtweb/publications/*.

[79] Simon Perathoner. Evaluation and comparison of performance analysis methods for distributed embedded systems. Master's thesis, Politecnico di Milano, March 2006. Appeared as technical report MA-2006-05 at ETH Zürich, TIK laboratory.

[80] Simon Perathoner, Ernesto Wandeler, and Lothar Thiele. Timed automata templates for distributed embedded system architectures. Technical Report 233, ETH Zurich, November 2005.

[81] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proceedings of EMSOFT'07*, pages 193–202. ACM, 2007.

[82] Colin Potts. Software-engineering revisited. *IEEE Software*, 10(5):19–28, 1993.

[83] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated statecharts - a lightweight formal approach. In *FASE 2000 - Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science (LNCS)*, pages 127–146. Springer, 2000.

[84] Kai Richter, Marek Jersak, and Rolf Ernst. A formal approach to MpSoC performance verification. *IEEE Computer*, 36(4):60–67, April 2003.

[85] Peter J. Robinson. *Hierarchical object-oriented design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[86] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. Republished version of the original 1970 IEEE WESCON paper. *http://portal.acm.org/citation.cfm?id=41765.41801*.

[87] Heico Sandee. *Event-Driven Control in Theory and Practice - Trade-offs in software and control performance*. PhD thesis, Technische Universiteit Eindhoven, 2006.

[88] Heico Sandee, Maurice Heemels, Gerrit Muller, Peter van den Bosch, and Marcel Verhoef. Threads of reasoning: A case study in printer control. In *Systems Engineering: Shining Light on Though Issues*, Proceedings of the 16^{th} Annual International INCOSE Symposium. INCOSE, 2006. Paper is available on-line at *http://www.incose.org/ipub*.

[89] Alberto Sangiovanni-Vincentelli. Successive refinements of communication functions and architectures in system design. In *Design Automation and Test in Europe*, 2006. Hot Topic Session – Network the next "Big Idea" in design?

[90] SC-167. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992. *http://www.rtca.org*.

[91] Donna C. Stidolph and James Whitehead. Managerial issues for the consideration and use of formal methods. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2085 of *LNCS*, pages 170–186. Formal Methods Europe, Springer, 2003.

[92] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[93] F. W. Vaandrager. De ingebouwde informatica. Katholieke Universiteit Nijmegen, Heyendaalseweg 135, 6525 AJ Nijmegen, December 1996.

[94] Manuel van den Berg, Marcel Verhoef, and Mark Wigmans. Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice – proceedings of the VDM workshop at FM'99*, pages 85–93, September 1999.

[95] Peter van den Bosch, Oana Florescu, Marcel Verhoef, and Gerrit Muller. Modeling of performance. In *Model-based design of high-tech systems*, pages 101–114. Embedded Systems Institute, 2006. See also [47].

[96] Peter van den Bosch, Gerrit Muller, Marcel Verhoef, and Oana Florescu. Modeling of hardware software performance of high-tech systems. In *Proceedings of the* 17^{th} *Annual International INCOSE Symposium*. INCOSE, 2007. Paper is available on-line at *http://www.incose.org/ipub*.

[97] Piet van der Putten and Jeroen Voeten. *Specification of Reactive Hardware / Software Systems*. PhD thesis, Technical University Eindhoven, 1997.

[98] Marcel Verhoef. On the use of VDM++ for specifying real-time systems. In John Fitzgerald, Peter Gorm Larsen, and Nico Plat, editors, *Towards Next Generation Tools for VDM: Contributions to the First International Overture Workshop*, CS-TR 969, pages 26–43. School of Computing Science, Newcastle University, June 2006. This technical report is available on-line at *http://www.cs.ncl.ac.uk/research/pubs/trs/papers/969.pdf*.

[99] Marcel Verhoef and Jozef Hooman. Evaluating embedded system architectures. In *Model-based design of high-tech systems*, pages 151–159. Embedded Systems Institute, 2006. See also [47].

[100] Marcel Verhoef and Peter Gorm Larsen. Interpreting distributed system architectures using VDM++ a case study. In Brian J. Sauser and Gerrit Muller, editors, *Proceedings of the Conference on System Engineering Research - CSER 2007*, Hoboken, NY, 2007. Stevens Institute of Technology.

[101] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006. Available on-line at *http://dx.doi.org/10.1007/11813040_11* .

[102] Marcel Verhoef, Peter Visser, Jozef Hooman, and Jan Broenink. Co-simulation of distributed embedded real-time control systems. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods - IFM*, volume 4591 of *Lecture Notes in Computer Science*, pages 639–658. Springer, 2007. This paper is available on-line at *http://dx.doi.org/10.1007/978-3-540-73210-5_33* .

[103] Peter Visser, Jan Broenink, and Job van Amerongen. Design trajectory and controller-plant interaction. In Gerrit Muller and Maurice Heemels, editors, *Model-based design of high-tech systems*, pages 205–214. Embedded Systems Institute, 2006. See also [47].

[104] P.M. Visser and J.F. Broenink. Controller and plant system design trajectory. In *Conference on Computer Aided Control Systems Design, CACSD*, pages 1910–1917. IEEE Control Systems Society, 2006. This paper is available on-line at *http://www.ce.utwente.nl*.

[105] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal of Software Tools for Technology Transfer (STTT)*, 8(6):649–667, November 2006. This paper is available on-line at *http://dx.doi.org/10.1007/s10009-006-0019-5* .

[106] J.G. Ziegler and N.C. Nichols. Optimum settings for automatic control. In *Proc. of the American Society of Mechanical Engineers*. ASME, 1942. Republished in the Dynamic Systems, Measurements, and Control DSCD Journal in 1993, vol. 115, pp. 220-222.

Samenvatting

Computers zijn niet meer weg te denken uit ons dagelijks leven. De personal computer is voor veel mensen een belangrijk gereedschap geworden, het Internet een voornaam communicatiemedium en de spelcomputer een bron van vertier. In minder dan vijftig jaar tijd heeft de computer zeer veel invloed gekregen op de kwaliteit van ons leven. Vaandrager [93] constateert dat met name door de spectaculaire daling in prijs en grootte van computerapparatuur het gebruik van dit soort technologie de laatste decennia een ongekende vlucht heeft genomen. Dit geldt met name voor de categorie van de zogenaamde *ingebedde systemen* (naar het Engelse *embedded systems*) waarbij computertechnologie in producten is ingebouwd en daarmee de functionaliteit geheel of gedeeltelijk bepaalt; de computer en het product zijn onlosmakelijk met elkaar verbonden. Denk daarbij aan de wasmachine, video recorder, DVD speler, foto- en videocamera en natuurlijk de mobiele telefoon.

In dit proefschrift worden ingebedde systemen beschouwd die een fysisch proces controleren en besturen. Denk daarbij aan besturing van een volautomatisch productieproces zoals een waferstepper of een moderne digitale printer. Belangrijke eigenschappen van deze klasse van regelsystemen zijn de hoge mate van autonomie en tijdsdruk. De computer neemt zelfstandig beslissingen op basis van metingen en corrigeert het fysische proces zonder directe tussenkomst van de mens. De taak van de computer is om onder alle omstandigheden het fysische proces binnen vooraf bepaalde grenzen in een gedefinieerde toestand te houden. Vaak moet relatief veel rekenwerk worden uitgevoerd om een eventuele correctie te bepalen. Bovendien speelt de reactietijd een cruciale rol, denk daarbij bijvoorbeeld aan de airbag in de auto. Dit zijn taken waarin de computer excelleert; de mens ontwerpt de receptuur, het regelalgoritme dat door de computer wordt uitgevoerd.

Ondanks het feit dat computers steeds sneller en krachtiger worden blijft het ontwerpen van ingebedde regelsystemen zeer uitdagend. Enerzijds wordt dit veroorzaakt door de constante toename van eisen die worden gesteld aan dit soort systemen, anderzijds omdat door software oplossingen elders kosten bespaard kunnen worden. Steeds vaker wordt gekozen om specifieke ontwerpeisen in de regelaar zelf op te lossen. Vaak omdat dit de enige plek is waar een grote mate van flexibiliteit geleverd kan worden op een laat moment in de systeemontwikkeling, de software is immers eenvoudig te wijzigen.

Het ontwikkelen van nieuwe systemen is een constante afweging tussen drie aspecten: tijd, geld en kwaliteit. In de markt van consumenten en kapitaalgoederen staan met name tijd en geld constant onder druk. De productievolumes zijn weliswaar hoog maar de verkoopmarges zijn vaak relatief laag. Er is dus een groot economisch belang om eerder dan de concurrentie met een nieuw product op de markt te komen. Maar kan dat zonder afbreuk te doen aan kwaliteit en, als afgeleide daarvan, functionaliteit? Deze zogenaamde *time-to-market* druk heeft in het verleden meermalen geleid tot spectaculaire mislukkingen. De belangrijkste uitdaging is om de juiste balans te vinden tussen deze aspecten en dat blijkt buitengewoon lastig.

Dit geldt in belangrijke mate ook voor ingebedde regelsystemen. Daar liggen een aantal problemen aan ten grondslag. Ten eerste, de ontwikkeling van computer hardware gaat veel sneller dan de ontwikkeling van regelsystemen die op deze technologie zijn gebaseerd. Ongeveer iedere 18 maanden kunnen we spreken van een totaal nieuwe hardware generatie, de systeemontwikkelingscyclus van een nieuw product is over het algemeen veel langer. Bovendien is er een duidelijke trend waarneembaar naar heterogene en gedistribueerde hardware. Deze zogenaamde *system-on-chip* oplossingen combineren analoge, digitale en hoog-frequent elektronica met meerdere, via een intern netwerk gekoppelde, processoren in één geïntegreerde schakeling. Opvallend daarbij is dat de ontwerpgereedschappen voor deze nieuwe generatie computerhardware duidelijk achterblijft, hetgeen de ontwikkeling van systemen extra compliceert. Ten tweede, de continue economische druk om productkosten zo laag mogelijk te houden dwingt de ontwerper om te werken op het randje van de technische haalbaarheid. Maar hoe kan de ontwerper deze beslissingen goed nemen als de mechanica, elektronica en de regelaar nagenoeg gelijktijdig ontwikkeld worden om de doorlooptijd te beperken en de systeemeisen vaak nog niet eens duidelijk zijn op het moment dat de belangrijkste architectuurbeslissingen genomen moeten worden? Eén van de belangrijkste problemen daarbij is de a-priori validatie van deze beslissingen in de (voor)ontwerpfase en de gevolgen van potentiële wijzigingen gedurende de levenscyclus. Dit zijn aspecten die in dit proefschrift aan de orde komen.

Het ontwerpen van ingebedde regelsystemen is bij uitstek een multi-disciplinair vraagstuk. Specifieke kennis van werktuigbouwkunde, regeltechniek, elektrotechniek en informatica is onontbeerlijk en de interactie tussen deze vakgebieden is bepalend voor het behaalde eindresultaat. En dat blijkt in de praktijk moeizaam, met name voor systeemaspecten die discipline overstijgend zijn zoals bijvoorbeeld betrouwbaarheid, robuustheid, energieverbruik en snelheid. In de huidige beroepspraktijk blijkt ontwerpen vaak disciplinegewijs te zijn ingericht en pas tijdens de systeemintegratiefase komt dit multi-disciplinaire aspect aan bod en de interactie tot stand. Slechts zelden leidt een optimale oplossing binnen één discipline tot het bereiken van het optimum op systeemniveau en vaak wordt dit probleem pas tijdens de integratiefase vastgesteld. Met andere woorden, de consequenties van de genomen ontwerpbeslissingen zijn pas laat in het ontwerpproces zichtbaar. Dit leidt vervolgens tot kostbare correcties en projectuitloop. Eén van de kernproblemen is dat de gebruikte disciplinespecifieke ontwerptechnieken fundamenteel van elkaar verschillen en dat de ontwerpers zich concentreren op verschillende type problemen en daarvoor hun eigen werkwijze hebben ontwikkeld. Er is geen synergie.

Het gebrek aan dialoog op systeemniveau in de vroege ontwerpfase tussen de verschillende ontwerpdisciplines is één van de uitdagingen waarvoor in het BODERC project, waarvan dit onderzoek deel uit maakt, een oplossing werd gezocht. Het doel van dit proefschrift is om te bepalen of er methoden en technieken bestaan, of te definiëren zijn, die een oplossing bieden voor dit probleem. De uitdaging daarbij is om de effectiviteit van een dergelijke oplossing aan te tonen door toepassing op een casus van enige omvang, die is geïnspireerd op een industrieel ontwerpprobleem: het papierpad van een hoogvolumeprinter.

Als startpunt voor dit onderzoek is in hoofdstuk 2 gekozen om een aantal bestaande ontwerp- en analysetechnieken toe te passen op een relatief eenvoudige casus: het ontwerp van een autoradionavigatiesysteem. De doelstelling van deze fase van het onderzoek was enerzijds ervaring opdoen met het modelleren van dergelijke problemen en anderzijds het toetsen van de vraag of een aantal specifieke ontwerpeisen beantwoord kon worden met de beschikbare technieken. Ook was vergelijking van de resultaten, zowel in kwalitatieve als kwantitatieve zin mogelijk omdat telkens dezelfde casus werd beschouwd. Voor zover bekend is het de eerste keer dat een dergelijk vergelijkend onderzoek op deze wijze is uitgevoerd. In deze studie is gekeken naar Modular Performance Analysis (MPA), Symbolic Timing Analysis for Systems (SymTA/S), UPPAAL, POOSL en VDM++.

Frappant genoeg leidde met name de vergelijking van de modellen en de analyseresultaten tot zeer interessante inzichten. Zo werd de incompleetheid van de originele probleemstelling meermalen aangetoond, maar werden ook subtiele fouten in de gemaakte modellen en de tools ontdekt. Sommige methoden zijn sterk in het vinden van een exacte oplossing maar hebben daar veel tijd voor nodig. Andere methoden zijn sterk in het snel vinden van een goede eerste orde benadering. In veel gevallen bleek dat de sterke en zwakke kanten van diverse methoden elkaar kunnen compenseren. Dit is ook vaak nodig omdat de uitkomst vaak onevenredig sterk beïnvloed wordt door de zwakke kant van een techniek. De keuze van de juiste techniek in een bepaalde ontwerpfase kan dus van grote invloed zijn op de behaalde effectiviteit. Het modelleren en analyseren van een probleem in twee of meer methoden zorgt er in ieder geval voor dat de gevonden resultaten altijd kritisch zullen worden beschouwd. In de praktijk wordt vaak te snel de conclusie getrokken dat het gevonden antwoord ook de juiste is.

Eén van de leermomenten uit het vergelijkend onderzoek is dat het expliciet maken van de computer hardware architectuur in een model van de ingebedde software allesbehalve eenvoudig is. Het beschrijven van deze relatie tussen hardware en software is weliswaar mogelijk, maar het aanbrengen van wijzigingen, hetgeen veelvuldig gebeurt in de vroege ontwerpfase om snel ontwerpafwegingen te kunnen maken, kost zeer veel tijd en is bovendien foutgevoelig. In hoofdstuk 3 van dit proefschrift wordt daarom een voorstel gepresenteerd om één van de technieken, VDM++, aan te passen om dit probleem op te lossen. Daartoe wordt zowel de syntax als de semantiek van deze formele specificatie taal aangepast. VDM++ kent een synchroon executiegedrag waarbij er altijd maximaal één taak tegelijk actief is in het model, gebaseerd op beschikbaarheid van één enkele processor. De belangrijkste wijziging in de semantiek is dat nu ook asynchroon executiegedrag wordt toegestaan en dat er meerdere processoren kunnen zijn die elk een eigen actieve taak kunnen hebben. Belangrijk daarbij is dat de betekenis van de bestaande modellen nog steeds wordt ondersteund. Maar de uitbereiding maakt het nu mogelijk om de computer hardware architectuur, die mogelijk bestaat uit meerdere processoren en netwerken, expliciet te benoemen en daarmee te redeneren over distributie van software en de invloed op het totale systeemgedrag. Wijzigingen in de computer hardware architectuur zijn eenvoudig door te voeren zonder dat daarvoor het model van de software aangepast hoeft te worden.

In hoofdstuk 4 wordt deze onderzoekslijn nog een stap verder doorgetrokken. De modellen die met de aangepaste VDM++ notatie worden beschreven, kunnen door middel van discrete event simulatie nader worden bestudeerd. De omgeving waarin deze ingebedde regelsystemen werken, met andere woorden de fysische werkelijkheid, laat zich ook uitstekend formeel beschrijven, bijvoorbeeld door middel van differentiaalvergelijkingen of bondgrafen. Het gedrag van deze fysische systemen kan dus ook worden bestudeerd door middel van simulatie, zij het in het continue tijddomein. In hoofdstuk 4 wordt een extra wijziging in de semantiek van VDM++ gepresenteerd die consistente co-simulatie van de discrete regelaar, gespecificeerd in VDM++, mogelijk maakt waarbij de fysische werkelijkheid mathematisch is beschreven, bijvoorbeeld met behulp van bondgrafen. Hiervoor is ook prototype tooling ontwikkeld die wordt toegepast op een simpel maar relevant voorbeeld: een watertank controller. Deze geïntegreerde formele semantiek van VDM++ en bondgrafen, en de bijbehorende tools, vormen de kern van het resultaat van dit onderzoek. Bovendien blijkt de ontwikkelde geïntegreerde formele semantiek ook toepasbaar voor andere methoden en technieken.

In hoofdstuk 5 wordt het ontwikkelproces van ingebedde regelsystemen beschouwd waarbij de focus nadrukkelijk ligt op het inzichtelijk maken van de multi-disciplinaire vraagstukken die met name aan het begin van de ontwikkelcyclus spelen. Welke, vaak informele, technieken kunnen worden ingezet om deze ontwerpdialoog te stimuleren en te structureren? Een drietal technieken wordt daarbij beschouwd: de *key-driver method*, *threads of reasoning* en *budget-based design*. Op welke wijze vindt ontwikkeling van regelsystemen en software momenteel in de praktijk plaats? Hoe verhouden deze aanpakken zich tot elkaar en welke rol spelen formele ontwerptechnieken in deze processen? Zijn ze in elkaar te passen of sluiten ze elkaar uit? Welke impact hebben deze processen in de praktijk en hoe wordt de juiste balans tussen proces en product bereikt? Wat moet er gedaan worden in welke fase? Aan deze vragen wordt in hoofdstuk 5 aandacht besteed.

De resultaten van het onderzoek uit de hoofdstukken 3, 4 en 5 komen samen in hoofdstuk 6. Hierin wordt het papierpad van een hoogvolumeprinter nader bestudeerd. Zowel het ontwerp van het model van de fysische werkelijkheid als het ontwerp van de ingebedde regelaar komt daarbij uitgebreid aan bod. Bijzondere aandacht wordt gegeven aan het validatieproces dat werd gevolgd tijdens de ontwikkeling, waarbij werd gewerkt volgens de processen beschreven in hoofdstuk 5. In de eerste fase werd de co-simulatie interface toegepast zoals beschreven in hoofdstuk 4. In de tweede fase werd uit het VDM++ model direct C++ code gegenereerd die vervolgens als *software-in-the-loop* applicatie kan worden gesimuleerd. In de eerste twee fasen werd een 3D visualisatiemodel gekoppeld aan de simulator om het dynamisch gedrag van het systeem inzichtelijk te maken. In de derde en laatste stap werd de gegenereerde C++ code gecompileerd voor het target platform en op de proefopstelling getest. Daarbij zijn metingen verricht die vervolgens zijn vergeleken met de uitkomst van de simulaties, met als eindresultaat een regelaar die conform verwachting presteert. Daarbij viel op dat reeds in de eerste fase diverse discipline-overstijgende problemen opgespoord konden worden omdat de co-simulatie niet leidde tot het gewenste resultaat. Bovendien bleek dat de continue validatieaanpak inderdaad bijdraagt tot het actief en relatief eenvoudig beheersen van de ontwerp- en implementatierisico's.

De conclusie van het onderzoek gepresenteerd in dit proefschrift is dat een methode voor het multi-disciplinair ontwerpen van ingebedde gedistribueerde regelsystemen inderdaad beschikbaar is. Met behulp van twee bestaande technieken, VDM++ en bondgrafen, is aangetoond dat relatief compacte en eenvoudig onderhoudbare modellen gemaakt kunnen worden van zeer complexe systemen. Daarvoor werd de syntax en semantiek van VDM++ aangepast en een geïntegreerde semantiek voor continue tijd co-simulatie ontwikkeld, ondersteund door prototype tools. Met behulp van deze tools is het mogelijk om de ontwikkelde modellen te inspecteren. Deze simulaties geven weliswaar geen bewijs van absolute correctheid maar de resultaten ondersteunen wel degelijk de ontwerpdialoog in de vroege fase van het ontwerp. De effectiviteit van deze aanpak is door middel van een relevante casus aangetoond. Inmiddels zijn de resultaten van hoofdstuk 3 ook beschikbaar in een commercieel verkrijgbaar product, sinds versie 8.0 is deze functionaliteit namelijk ook beschikbaar in VDMTOOLS.

De tools en modellen die ontwikkeld zijn in het kader van dit proefschrift zijn beschikbaar op *http://www.overturetool.org* en op *http://www.marcelverhoef.nl*.

Curriculum Vitae

Marcel Henri Gerard Verhoef was born at Papendrecht on 5 August 1968. He attended the Willem de Zwijger Scholengemeenschap where he obtained his Atheneum-B degree in 1986. He moved to Delft University of Technology where he studied Computer Science and Aerospace Engineering. He became a student assistant at the Computer Science department in 1991, where he worked on the *Delft VDM-SL Front-End*. He moved to the Institute of Applied Computer Science, IFAD, at Odense, Denmark for 9 months in 1992 to obtain his MSc thesis called "*A Constructive Static Semantics for the IPTES Meta-IV Specification Language*". This work was supervised by professor Jan van Katwijk and involved specification and implementation of a type-checker for the ancestor language to VDM++. Some of the MSc results ended up in VDMTOOLS and are still in use today. He needed to go into the obligatory military service in November 1992 and finally graduated from Delft University of Technology in September 1993. He served in the army for 14 months and was honorably discharged in January 1994 reaching the rank of "Eerste Luitenant". He joined the COMBINE project at the faculty of Civil Engineering at Delft University of Technology as a research assistant. This project, lead by professor Godfried Augenbroe, was sponsored within the EC Joule programme and was aimed at integrating design tools for developing energy efficient buildings. He was responsible for implementing a large product data model in an object-oriented database and implementing several on-line and off-line interfaces, for example to leading CAD tools such as Bentley Microstation and Autodesk AutoCad. He moved to industry at the end of the project in 1996 and started working for BSO/Origin. He worked on several commercial and large-scale software engineering projects for customers such as the European Space Agency, Shell Nederland and the Dutch Department of Defense. In 1999, he moved with several colleagues to form Chess Information Technology, a new company in the Chess group. Emphasis shifted towards complex mission-critical and embedded systems and he worked as a systems architect for customers such as the Bloemenveiling Aalsmeer, European Space Agency, Océ Technologies and Siemens VDO Automotive. He worked on the BODERC project from March 2003 to March 2007, while still being employed by Chess who became full partner in the project. Currently, he is part of the Chess innovation team where he consults customers and the other Chess business lines on embedded system architecture issues. He is involved in the Quasimodo EU Seventh Framework research project where he works on verification and validation of large-scale wireless sensor networks using model checking technologies. He is a member of KIVI, IEEE, ACM, INCOSE and ForTIA and he is treasurer of Formal Methods Europe. He likes recreative running, spinning and he plays squash and golf. He maintains a web-site with an up-to-date overview of his publications: *http://www.marcelverhoef.nl*.

Acknowledgements

First of all, I would like to thank Frits Vaandrager and Jozef Hooman for accepting me as a PhD candidate “beyond-the-ordinary” at the Radboud University Nijmegen. It was a pleasure working with you both and I am particularly grateful for the mental support you provided in times of need. I truly admire your patience and mentoring skills!

Second of all, I would like to thank Jan Laagland and Rene Hodde, joint owners and executive directors of Chess Information Technology, now Chess, and Siebren de Vries, for giving me the opportunity to step out of the commercial business for four years to chase my own dream.

This manuscript would not have seen the light of day without the warm support of the large number of the people I have had the pleasure to meet and work with:

- I would like to thank Ernesto Wandeler, Lothar Thiele, Paul Lieverse, Martijn Hendriks, Simon Perathoner, Kai Richter, Menno de Hoon, Oana Florescu, Jeroen Voeten, Henk Corporaal, Egor Bondarev, Peter Gorm Larsen and Erik Oosterom for their in-depth discussions and contributions to the method comparison presented in Chapter 2.

- I would like to thank my co-authors, Jozef Hooman and Peter Gorm Larsen for their contribution and the anonymous reviewers of FM 2006, Søren Christensen, John Fitzgerald, Finn Overgaard Hansen, Shin Sahara, Erik Gaal and Evert van de Waal for their valuable comments and support for the paper presented in Chapter 3.

- I would like to thank my co-authors, Peter Visser, Jozef Hooman and Jan Broenink for their contribution and the anonymous reviewers of IFM 2007, Zoe Andrews, Job van Amerongen, Peter van den Bosch, Erik Gaal, Peter Gorm Larsen and Frits Vaandrager for their valuable comments and support for the paper presented in Chapter 4.

- Chapter 5 and 6 would simply not have been possible without the help of Peter Visser, Frank Ambrosius and Jan Broenink. I would like to thank Heico Sandee, Peter van den Bosch, Maurice Heemels and Gerrit Muller for the BODERC group work which was instrumental to the Threads of Reasoning paper [88]. Many thanks to Peter Gorm Larsen for the in-depth discussions on the VDM++ development process for embedded real-time systems.

- I owe many thanks to the manuscript committee members, prof. dr. Bart Jacobs (chair), prof. dr. Lothar Thiele, prof. dr. ir. Peter Gorm Larsen, dr. ir. Jeroen Voeten and dr. ir. Jan Broenink for final approval of this manuscript and providing me with many detailed corrections, suggestions and questions which have helped me to improve and fine-tune this work.

I would like to thank John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee and Nico Plat for giving me the opportunity to contribute to the book *Validated Designs for Object-Oriented Systems* [30]. It has been a joy and very worthwhile experience! In particular I want to mention the incredible hospitality I have enjoyed from Peter and Margit Larsen at their home in Søften, Denmark, where I have spent several weeks over the last few years working with Peter on the book and parts of this thesis. Those were memorable occasions that I will never forget!

Of course, I want to acknowledge all the people involved in the BODERC project at the Embedded System Institute for the stimulating discussions and open atmosphere in the project. I want to thank my fellow PhD students Heico Sandee, Marieke Cloosterman, Björn Bukkems, Peter Visser and Oana Florescu, and the ESI staff members Frans Beenker, Gerrit Muller, Bauke Sijtsma, Jan-Matthijs Wijnands, Anget Mestrom and Ed Brinksma. But also the participants from the project partners: Jan Beckers, Peter van den Bosch, Zhaorui Yuan, Hennie Freriks, Paul van den Bosch, Maarten Steinbuch, Henk Nijmeijer, Henk Corporaal, Job van Amerongen, Jan Broenink, Jeroen Voeten, Maurice Heemels, Nathan van de Wouw, Rene van de Molengraft, Evert van de Waal, Erik Gaal, Berry van der Wijst, Jandit van Doorn, Adriaan van den Brand and Lou Somers.

Thank you Silvian de Jager for making the extraordinary graphic design for the cover of this manuscript!

Special thanks to Marcelle van Valkenburg, who has convinced me to write the magic letter to ESI that started it all in November 2002.

And last, but certainly not least, my parents: Gerrit and Dikkie, to whom I dedicate this thesis and my sister Bernadette, my family and friends, for their love, patience and support.

Titles in the IPA Dissertation Series since 2006

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming*

DSP applications. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of

Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01