Reactive machine control : a simulation approach using chi

Reactive machine control

a simulation approach using χ

A.T. Hofkamp

Voorkant: Tekening door G.J. Pouw te Naarden.
De tekening laat het interne kruiwerk van de molen te Zeddam (Gld) zien. Deze stenen torenmolen is tussen 1440 en 1450 gebouwd. Het is de oudste in bedrijf zijnde wind-korenmolen van Europa, en staat in de UNESCO top 100 van Nederlandse monumenten.
De molen is te bezichtigen aan de Bovendorpstraat 14 te Zeddam.
Voor meer informatie, zie http://welcome.to/torenmolen.nl .

Drukker: Universiteitsdrukkerij Technische Universiteit Eindhoven

/ stan ackermans institute, center for technological design

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Hofkamp, Albert Th.

Reactive machine control : a simulation approach using χ / by Albert Th. Hofkamp. - Eindhoven : Technische Universiteit Eindhoven, 2001. - Proefschrift. - ISBN 90-386-3012-3
NUGI 841
Subject headings: virtual machines / reactive machine control / development tools / simulation / chi language

Reactive machine control

a simulation approach using χ

PROEFONTWERP

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. R.A. van Santen, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 6 november 2001 om 16.00 uur

door

Albert Theo Hofkamp
geboren te Leeuwarden

Dit proefontwerp is goedgekeurd door de promotoren:

prof.dr.ir. J.E. Rooda
en
prof.dr. M. Rem

Copromotor:
dr.ir. J.M. van de Mortel-Fronczak

Preface

In March 1997, I started working at the Systems Engineering Group for my final project of OOTI[1]. I liked the work and the environment, and thus agreed to stay for a PhD on design for the design and implementation of a tool set for the development of machine control systems. Now, three years later, I have finished the project, had a lot of fun, and learned a lot.

Since I wrote this thesis, my name is on the front of this thesis. However, I am not the only person that worked on its contents. I would like to thank everybody involved for his or her assistance in this project. First of all, dr.ir. J.M. van de Mortel-Fronczak, prof.dr.ir. J.E. Rooda, and prof.dr. M. Rem for their help and time while writing this thesis.

I would like to thank dr.ir. W.T.M. Alberts, dr. G. Fábián M.Sc, and dr.dipl.-ing. G. Naumoski for their involvement in the compiler design and implementation, and ir. N.Z. Chen for constructing the simulation engine. Also, I thank dr. H. Geuvers for his help with the type checking algorithm, and correcting that part of my thesis.

A large number of people invested their time and energy in the paint-factory case study. Without them, the study would not have been possible. Dr.ir. D.A. van Beek defined the global requirements, ir. K.J. Eijsvogels came with the idea of a paint factory and performed the first simulations, ing. H.W.A.M. van Rooij, F.G.J. Soers, and F.A.J.C. van Stiphout created the physical version of the machine that ir. M.H.M. van Duin brought to life by designing and implementing the operational controller.

Last but not least, I thank Mieke, Corry, and Anja for their discussions about the challenges and problems of every day life. They provided the counterweight to prevent me from drifting into the technicalities of work.

In this thesis, Chapter 1 discusses the topic of this project, the design and implementation of reactive control systems for industrial machines using simulation. It introduces an approach to realize the construction of a controller, and discusses a number of important design decisions made in the project. One of these decisions is the simulation language. The chosen language χ is explained in the second chapter. This chapter also considers the differences between the real world around us, and the simulation world. Chapters 3

[1]Post-graduate designer programme *Software Technology*.

and 4 explain the design and implementation of the tool to perform the transformation from simulation to the real world. The fifth chapter discusses the paint-factory case study. This case study is the first larger experiment of the approach proposed in the first chapter. Chapter 6, the final chapter of this thesis, reflects on the work done and draws conclusions. Also, suggestions for next steps are made.

Summary

Customers want their products cheap and tailored to their needs. In the production process, this means that each product is more or less unique, yet they have to be manufactured in large enough quantities to keep the costs low. At the same time, the life time of a product is getting shorter. In order to sell as many products as possible, a short time-to-market is essential. To meet these demands, production machines must be constructed in a short time, with enough flexibility to manufacture each product variant in large enough volume. The short delivery time of machines and the requirements with respect to flexibility makes the design of these machines and their control systems increasingly complex. The Systems Engineering Group is therefore investigating new methods to design machines and their control systems. The focus in the research lies foremost on the design of the control system. In particular, methods and practices are devised that allow developers to construct machine control systems with high performance. Realizing this aim means that the controller and the machine have to be designed and used as a *two-unit*. That is, both are separate systems but they are so strongly linked that they cannot function independently.

Designing a machine and its controller as a two-unit is a highly complex task. Due to the strong inter-connections, a change in one system causes changes in the other system and vice versa. To manage the complexity, *virtual machines* are used. Instead of designing the construction of the machine and developing an implementation of the control system, a design of both systems is performed using simulation first. The machine and its control system are described in a specification and tested using simulation. Once the simulation gives satisfactory results, the design can be build. For the control system, the translation from simulation to implementation can be done automatically.

Specifying the machine and the control system enforces the developers to be precise about their ideas. That goes a long way in detecting and correcting potential problems early in the design process. Simulation of the virtual machine and controller allows the machine to 'come to life'. It enables study of the dynamic behaviour of the combination. The study provides insight in the design, and gives feedback about development steps done. Also, data from simulations may be used to guide the development process. For example, if the simulation data suggests that performance will be adequate, improving performance even further is not necessary.

This Ph.D. project on design aims to create practically usable means for performing case studies in the research of designing controllers for industrial machines using the virtual machine concept. In particular, a practically usable design technique is discussed, and tools to support the development process have been designed and implemented for the formal specification language χ.

The design technique presented in the thesis divides the development process in a number of vertical design steps, and a single horizontal design step at the end. In each vertical design step, a simulation model is designed and specified that is lower in abstraction level compared to the previous model. Simulation experiments are performed with the model, and the output of the experiments is studied. If the output is satisfactory, either the next vertical design step is made to further lower the abstraction level of the model, or the model has the right abstraction level in the sense that it uses the same interface as the real machine. In the latter case, the final horizontal design step is made. In this step, the model of the machine control system is translated to an implementation of the control system on a real-time operating system. Since the model of the controller is not modified, the implementation will function the same as the model, and since the interface of the machine model is the same as the interface of the real machine, the implementation will be able to control the real machine.

Two tools have been designed and implemented. The first tool is the simulation tool which enables simulation of a specification on a Unix system. The second tool is the translation tool. It translates a controller model to an implementation on the real-time operating system VxWorks. All tools have been written in C++, and consist of a compiler that compiles the specification, and a run-time system that executes the compiled model.

The VxWorks run-time system of the translation tool currently uses a single processor only, but it can easily be extended to run on a distributed system. An important part of the run-time system is the implementation of the communication algorithm. The χ specification language uses synchronous communication, whereas the VxWorks real-time operating system only provides asynchronous communication primitives. The implemented communication algorithm, based on Bagrodia [Bag89], performs the mapping of synchronous communication to asynchronous primitives. Several extensions have been added to the algorithm to make it fit the needs of the translated specification. Correctness of the implemented algorithm has been tested with a number of verifications, and a large number of Monte Carlo simulations.

As a test to see whether the virtual machine concept, the developed method, and the tools are useful in reaching the research goals, a case study has been performed. A machine called *the paint factory* and a control system have been developed using the design technique. Although the higher levels of control that optimize scheduling are not yet finished, the case study did produce a functioning machine with its control system. The use of the virtual machine concept seems to be a promising road towards the development of controllers for (complex) industrial machines.

Samenvatting

Klanten willen hun producten aangepast aan hun wensen en goedkoop. In het productie-proces betekent dit, dat elk product min of meer uniek is. Tegelijkertijd moeten ze wel in voldoende aantallen gemaakt worden om de kosten laag te houden. Bovendien wordt de levensduur van een product korter. Om zoveel mogelijk producten te verkopen is een korte time-to-market essentieel. Een korte time-to-market van een product betekent ook, dat de productie-machines snel te leveren moeten zijn. Machines moeten daarom snel te bouwen zijn en voldoende flexibiliteit bezitten, om met wisselingen tussen product-varianten te kunnen omgaan.
De kortere ontwikkeltijd van machines en de eisen ten aanzien van de flexibiliteit maken het ontwerp van deze machines en hun besturingssystemen complexer. De Systems Engineering groep onderzoekt daarom nieuwe methoden om machines en hun besturingssystemen te ontwerpen. De nadruk van het onderzoek ligt daarbij vooral op het ontwerpen van het besturingssysteem. Er worden methodes ontwikkeld die het voor ontwerpers mogelijk maken om machines en besturingen te bouwen die hoge prestaties leveren.
Om dit doel te verwezenlijken moeten de machine en zijn besturing als een *twee-eenheid* ontworpen en gebruikt worden. De machine en de besturing zijn twee losse systemen, maar ze zijn zo sterk met elkaar verbonden dat ze niet zelfstandig kunnen functioneren.
Het ontwerpen van een machine en zijn besturing is een complexe zaak. Door de sterke verbondenheid van beide systemen heeft het veranderen van het ene systeem gevolgen voor het andere en andersom. Om de complexiteit te kunnen beheersen wordt het idee van *virtuele machines* gebruikt. In plaats van het ontwerpen van de machine-constructie en het ontwikkelen van een implementatie van de besturing, wordt er eerst een ontwerp gemaakt van de systemen gezamenlijk door middel van simulatie. De machine en het besturingssysteem worden beschreven in een specificatie en geanalyseerd met behulp van simulatie-experimenten. Wanneer de simulatie-resultaten uitwijzen dat het ontwerp aan de gestelde eisen voldoet, kan het ontwerp gebouwd worden. Voor het besturingssysteem kan de vertaling van simulatie naar implementatie automatisch gedaan worden.
Het specificeren van de machine en de besturing dwingt de ontwerper om zijn ideeën precies te formuleren. Dat helpt veel in het ontdekken en corrigeren van potentiële problemen vroeg in het ontwerpproces. Simulatie van de virtuele machine en zijn besturing zorgt ervoor, dat de machine 'tot leven komt'. Dankzij simulatie is het mogelijk om het dynamische gedrag

van de combinatie te analyseren. De analyse levert inzicht in het ontwerp en geeft een terugkoppeling over ontwerpbeslissingen. Bovendien kan data uit simulatie-experimenten gebruikt worden om het ontwerpproces te sturen. Als de gegevens bijvoorbeeld aangeven dat de prestaties voldoende zijn, dan hoeven deze niet verder verbeterd te worden.

Het doel van dit proefontwerp is om praktisch bruikbare middelen te ontwikkelen voor het onderzoek naar het ontwerpen van machine-besturingen, gebruikmakend van virtuele machines. In het proefschrift wordt een ontwerpmethode behandeld, daarnaast heeft het promotieproject geresulteerd in het ontwerp en de implementatie van gereedschappen die het ontwerpproces ondersteunen voor de formele specificatietaal χ.
De ontwerpmethode voorgesteld in dit proefschrift verdeelt het ontwerpproces in een aantal verticale ontwerpstappen en één horizontale ontwerpstap aan het einde. In elke verticale stap wordt een simulatiemodel ontwikkeld en gespecificeerd op een lager abstractieniveau dan het vorige model. Dit model wordt gesimuleerd en de simulatie-resultaten worden bestudeerd. Als de resultaten voldoen aan de eisen, kan de volgende verticale ontwerpstap gedaan worden om het abstractieniveau verder te verlagen. Indien het abstractieniveau overeenkomt met de interface die gebruikt wordt bij de echte machine, wordt een laatste horizontale ontwerpstap gemaakt. Het model van de besturing wordt vertaald naar een implementatie op een real-time beheerssysteem (operating system). Omdat het model van de controller ongewijzigd blijft, zal de implementatie hetzelfde gedrag hebben als tijdens de simulatie. Omdat bovendien de echte machine dezelfde interface gebruikt als de machine in de simulatie, zal de implementatie de echte machine kunnen besturen.
Twee gereedschappen zijn ontworpen en gebouwd. Het eerste gereedschap is het simulatie-gereedschap, wat simulatie van een specificatie mogelijk maakt op een Unix systeem. Het tweede gereedschap is het vertaalgereedschap, wat een model van een controller vertaalt naar een implementatie op het real-time beheerssysteem VxWorks. Alle gereedschappen zijn geschreven in C++ en bestaan uit een vertaler (compiler) en een run-time systeem, wat het vertaalde model uitvoert. Het run-time systeem op VxWorks gebruikt nu slechts één processor, maar kan eenvoudig uitgebreid worden om te kunnen werken op een gedistribueerd systeem. Een belangrijk deel van het run-time systeem is de implementatie van het communicatie-algoritme. De χ specificatietaal gebruikt synchrone communicatie terwijl het VxWorks real-time beheerssysteem slechts asynchrone communicatie-primitieven biedt. Het geïmplementeerde communicatie-algoritme is gebaseerd op Bagrodia [Bag89], en vertaalt synchrone communicatie naar asynchrone primitieven. Er zijn een aantal uitbreidingen toegevoegd aan het algoritme om het passend te maken voor het vertaalde model. Correctheid van het gebouwde algoritme is getest door middel van een aantal verificaties en een groot aantal Monte Carlo simulaties.

Als een test of het virtuele machine concept, de ontwikkelde methode en de gereedschappen nuttig zijn in het bereiken van het onderzoeksdoel, is een casus (case study) uitgewerkt. Een machine, die *verffabriek* wordt genoemd, en het bijbehorende besturingssysteem zijn ontwikkeld, gebruikmakend van deze ontwerp-methode. Hoewel de hogere besturingslagen die de volgorde van orders optimaliseren nog niet af zijn, leverde de casus een functionerende machine en besturing. Het gebruik van virtuele machines lijkt een goede weg naar het ontwikkelen van besturingen voor (ingewikkelde) industriële machines.

CONTENTS

CHAPTER

ONE

Introduction

In the recent years, computer technology has been introduced in industrial machines on a large scale. The ever-decreasing size and cost has made this technology feasible for use in machine control systems. Also, the ever-increasing computing power of this technology opens new roads and new applications. For example, visual inspection of products can now be done automatically with a sophisticated program rather than manually. Manufacturers of industrial machines therefore extend the functionality of their machines to make them more attractive to customers. Computer technology also allows re-programming of the controller almost at the flick of a switch. Thus, it is much easier to adapt the machine to new situations; the machine has become much more flexible than its predecessor. To some extent, one can say that a machine is becoming a small factory in itself.

The market needs this flexibility. Our society is becoming more oriented towards the individual. Concepts like mass-customization[1] are becoming popular. These personalized products are however more complex to manufacture than products that cannot be personalized. In other words, more is demanded from the industrial machines.

In short, the introduction of computer technology increases the number of possibilities of the machine, and the market wants flexibility. The latter can be realized by clever use of the former. The question however is how to do this in a systematic and controlled way. More precise:

How do you systematically design a flexible industrial machine and its controller ?

The Systems Engineering Group has set itself the goal of finding an answer to this question. This project is one of the first steps towards solving this puzzle.

1.1 Control systems

A widely accepted model of a control system for an industrial system is [JM86], published by the National Bureau of Standards, nowadays known as NIST. This hierarchical production control system is a framework composed of five levels of control, as shown in Figure 1.1. The top level *facility* takes care of planning for the entire plant for a relative long planning horizon, but on a global level. Each level below takes orders from above,

[1] Every customer his own personalized product, while using mass-production manufacturing techniques.

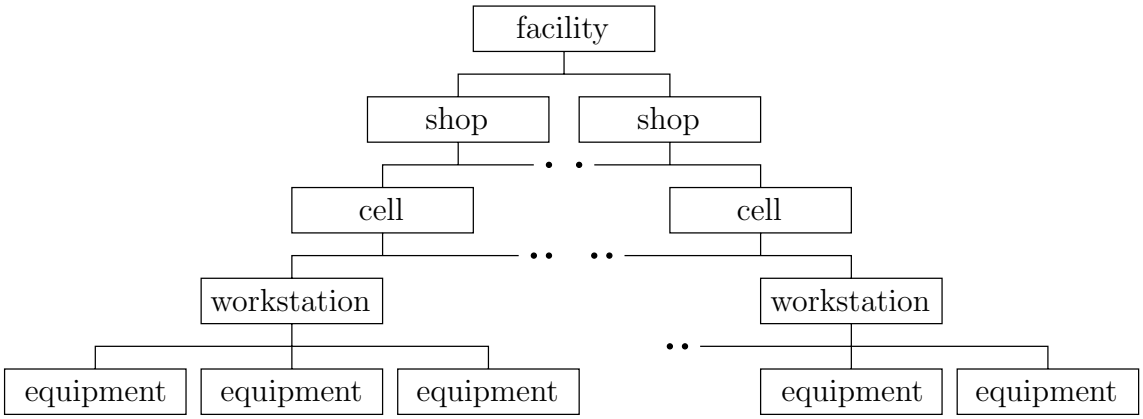

Figure 1.1: Model of a hierarchical control system by NIST.

makes a more detailed plan on a shorter planning horizon, and controls the level below using the computed plan. Below the *equipment* level, machines perform the manufacturing steps, thus realising the plan. Since production never completely follows the plan, status feedback is reported upwards between the levels. The feedback information can be used to adapt the plan, thus making optimal use of the industrial system.

As an example of such a framework we briefly describe the litho area of a chip factory using the above hierarchical control model. The *facility* level is the entire factory, as perceived by the clients and suppliers of the factory. This level accepts orders, agrees on delivery dates, and makes sure that the clients get the requested products. The internal organisation on the factory floor is controlled at the *shop* level. This level controls the various areas of the plant. In other words, it ensures that batches containing wafers are moved and processed in the correct area, and that materials needed for processing are also available.

One of these areas is the litho area. In this area, a number of litho modules and inspection machines are available. The *cell* control system routes the wafers to the machines, and makes sure that all the needed materials and chemicals for the particular wafer are available too at the same machine. For example, the masks necessary for the exposure process step should be on the same machine as the wafers that should be exposed.

Inside an individual litho module, a number of process steps are performed (coating, pre-bake, exposure, development, and post-bake). Routing of wafers through the process steps in a litho module is co-ordinated by the *workstation* control system. Each process step is performed by a machine, controlled by its *equipment* level control system. The machines themselves perform the process steps, and change the wafer surface.

In this project an initial step is taken towards a systematic approach of the development of machine control systems at equipment level in an industrial environment. Figure 1.2 shows a typical machine control system interacting with its environment. From above, orders are given to the machine control system. These orders are translated to actions

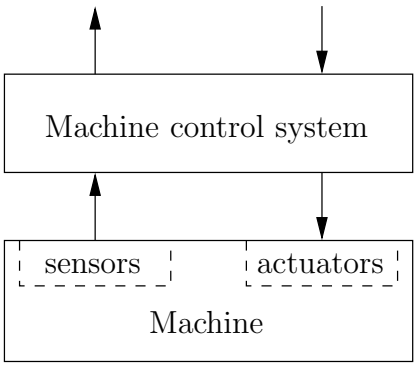

Figure 1.2: Model of a machine control system and its environment.

that must be done by the machine, and actuators are given orders to perform physical movements in the machine. Sensors in the machine report the status of the physical components in the machine back to the machine control system, and the latter uses this information as feedback, thus controlling the machine. Also, feedback on the status and progress of the order is given upwards to the higher level.

The end result of the development project of a machine control system is a functioning machine control system, controlling a physical machine. The control system thus has some real-time control properties such as timeliness. Obviously, these matters need to be addressed during the development of the control system. However, before reaching the point where real-time properties of the control system become important, the general design of the controller must be established. Especially for larger machines, this is a difficult subject. With the introduction of computers in machine control, machines have become much more flexible. They are no longer a mechanically-coupled number of parts, the parts can be controlled independently from each other. This delivers a lot of flexibility to the control system, but also makes designing a good controller more difficult because the various parts must co-operate as well as possible in order to utilize the increased flexibility. In a sense, machines can be seen as small factories with their own scheduling problems between the parts.

Another complicating factor is the environment of the machine and the controller, that is, the manufacturing process. For example, in some environments, the machine may be used in a batch system. In other environments, many small orders are manufactured, which requires a lot of switching between different products.

The main objective of a machine control system in an industrial environment is always to deliver maximal performance in the manufacturing process. In other words, the machine has to perform as well as possible in its environment. To achieve this goal, the controller has to be tailored to the environment. The design approach presented here considers the combination of controller and machine to be a *two-unit*. They are separate systems, but they should be developed and used together for the best results.

Development of the control system in this context involves profound knowledge of the

machine as well as its environment. With machines and environments as described above, this is too much knowledge to overlook completely. The systematic approach of the development process will thus focus on *understanding* of the situation.

1.2 Project goal

The goal of the research on embedded systems in the Systems Engineering Group is to develop a methodology to design controllers for (complex) industrial machines. Central concept in the research is the *virtual machine* concept. It means that the entire environment around the controller, including the machine, is simulated. In this simulation environment, the controller is developed. Section 1.3 explains these ideas in more detail.

This Ph.D. thesis aims to

> *create the practically usable means for performing case studies in the research of designing controllers for industrial machines using the virtual machines concept.*

'Create practically usable means' is the key-phrase here. The idea is to create means, such that researchers, engineers in industry, and students can perform practical experiments in the design of controllers, in order to get feedback on ideas in research.

The realization of this objective implies two 'deliverables':

- *A design technique.* The above description about development in a virtual environment is too abstract for students and engineers in industry. A more down-to-earth description is needed. This is called the design technique, and its description can be found in Section 1.3.

  Note that this technique is a first attempt. As research progresses, the design technique will quite likely be improved.

- *Tools to support the design technique.* A design technique is only viable for realistic case studies if it is supported by adequate, robust tools.

  Experimenters should concentrate on their objective of designing a controller, rather than worry about the limitations of the tools. Therefore, the tools should accept anything that is correct, even if it is big, ugly, or both. More about the tools can be found in Section 1.4.

An important factor in the construction of the technique and the tools are the capabilities of users. The intended audience in this project are designers with a Mechanical Engineering background. That means that knowledge of users about software-engineering techniques and formal methods is limited. The technique and the tools have to take this into account.

1.3 Design technique

The basic goal of the design technique is to provide a global framework in which the developer can organize his work. At the same time, it tries not to exclude any design

activity that may be considered useful. In other words, the design technique provides a general guide for the developer that can be tailored to a particular situation.

There are a number of reasons for this approach. First of all, trying to capture the entire design process is not feasible. Each machine control system has its own unique nature. This heavily influences the approach of designing the control system. Also, each developer has his own strong and weak points. The design technique should be flexible enough to allow each developer to use the approach that he prefers. Secondly, there is little or no experience with the development of design techniques for these kind of control systems. Further research must be done in order to decide on a good approach. Until that time, it seems foolish to eliminate any design approach. Also, it is more efficient to do research in this area after this project is finished, because tools are then available to perform realistic tests of different design approaches. Finally, the focus of the project lies on developing tool support rather than designing an optimal design technique.

The design technique is a framework to guide the developer in the design of the machine control system. The main focus of the technique is understanding the controller, the machine, and the environment.

One good way to get a good understanding of a subject is by *modelling* it. Modelling is a common engineering activity used in a lot of disciplines. For example, a new bridge is first modelled as a scale model and as a set of blueprints before construction starts. Software is designed and captured in graphical models like data flow diagrams. In fact, modelling is a technique that forces the developer to specify his thoughts. This aids in structuring the design [BRJ99]. Also, it confronts the designer with unclear areas in the design. Modelling is thus the foundation of the method.

Modelling is good for creating structure, but it gives little information about the dynamics of the system. Software-engineering techniques such as use-case diagrams or state diagrams do allow the developer to capture behaviour, but understanding the diagram as a dynamically changing object is a different matter, especially when the behaviour is complex. One technique to assist the developer in understanding the dynamics is *simulation* [Sha75]. With simulation, the design 'comes to life'. The developer can verify and correct his conceptual model of the dynamics by studying the results of the simulation. Since understanding the dynamics of the machine and its environment is crucial for development of good controllers, simulation is also part of the design method.

Even with modelling and simulation techniques, developers will make mistakes, and some of these mistakes will survive the design process and become bugs in the implemented controller. Obviously, there should be as few bugs in the implementation as possible. Paths to achieve this are extensive testing and formal verification. The former does not give full guarantees, the latter does provide correctness proofs at the cost of a lot of effort. Since failure of a controller for an industrial machine is very expensive, preventing bugs is extremely important. The method should thus allow the use of testing and formal verification techniques. However, the rigorousness of modelling and simulation techniques goes a long way in preventing bugs, see for example [Kar98]. Also, the machine control systems developed with the technique will not include mission-critical systems, for example

nuclear reactors or air planes. For these reasons, formal verification methods are currently considered to be a future extension to the design technique.

Finally, the end result of the development project using the design technique is a functioning controller. In other words, with the modelling and simulation techniques described above, a model is developed, and this model has to be ported to real hardware. That has to be done with *a)* as few changes as possible in the model, because every change may introduce an error, and *b)* little knowledge of software engineering and the real-time domain, because our users do not possess that knowledge.

The development path to the end result is as follows. First, a model is developed in a language formal enough for interpretation by a machine. The modelling process forces the developer to be precise. Since the language is formal enough for machine interpretation, a computer-based simulation of the model is possible. This simulation brings the model to life, allowing the developer to watch his design and check his assumptions. After enough iterations, the model is good enough to be implemented in the real world. Again using the ability to interpret the model by machine, the model is translated to an implementation. This implementation can then be used to control the real machine.

The above development procedure uses a number of assumptions. Two important ones are:

- The real-world machine is controlled in the same way as the simulated machine. The specified controller is translated to the implementation without functional changes. If the translated controller is to control the real machine, the latter should behave just like the simulated machine.

- The implementation should address the real-time properties needed. Numerous books are written on this subject. The properties listed below are from [HS91]. Others use similar lists. Some typical properties of a real-time control system are:

  - *Timeliness.* The system must react timely upon events from external processes.
  - *Simultaneity.* There are normally many external processes going on at the same time, and each of them must be controlled.
  - *Predictability.* The system must always (even under extreme conditions) produce predictable responses, both in the processing results themselves, and in the moments when the results become available.
  - *Dependability.* Embedded systems are used in environments where failure of the system may not only cause loss of data, but also endanger people and major investments. In order to minimize these risks, the embedded system must be made as reliable as possible.

  Timeliness is a property that needs to be verified already during the early stages of development. The modelling language does need the concept of time in order to allow this. Also, the translation to the implementation must preserve the timeliness

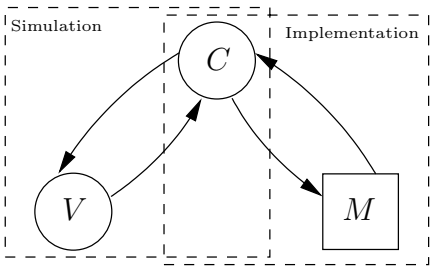

Figure 1.3: Simulation and implementation using virtual machine V, controller C, and real machine M.

property. The simultaneity property should be supported by the language. It cannot be expected that users understand how to incorporate simultaneous handling of multiple tasks in a sequential language. By introducing parallelism in the modelling language, simultaneity can be expressed in the model cleanly. Predictability of the implementation is taken care of by the modelling process. The rigorousness of this process ensures that the developer has at least seen and specified what happens in each situation. Dependability is a property that can only be ensured by systematic and precise working. The modelling and simulation process done by the developer does help here; a model in a formal language does not allow to leave parts of the system unspecified.

The models developed using the design technique look like Figure 1.3. The model consists of two components, the controller and the machine. While the design technique is used in the simulation domain, that is, models are developed and simulated, the controller C is connected with a *virtual machine* V. The latter is used as a representation of the machine in the model. With each iteration, more details are added to the controller, the machine component, or both, and behaviour of the combination is verified by simulating the model. During the translation to the real-world implementation, the virtual machine V is replaced by the real machine M, where M should be a real-world implementation of V. In the controller component, connections to machine V are replaced by connections to machine M, and the new combination can be executed and tested in the real world.

Early experiments show that a developer experiences the development process as three different stages, at least for relatively simple machine control systems:

1. *Conceptual design.* Starting from the requirements, early (abstract) models of the controller and the machine are written and simulated. The developer becomes acquainted with the global operation of the machine, and can check whether the given requirements can be fulfilled.

2. *Physical design.* Once the global working of the machine is understood, the developer can concentrate on designing the machine (its sensors and actuators), and the controller. Quite accurate estimates of the performance of the combination can be made,

and these estimates can be used to decide for example on hardware components or control strategy.

3. *Implementation and testing.* In this stage, the transformation to the real world is made. After it, the real machine is controlled by the controller. Correctness of the design (for example regarding reaction time) is verified by performing tests.

If the design technique is followed correctly, the last step should confirm that the machine and the controller are functioning correctly. If that is not the case, then the translation step to the implementation should be postponed, until the design is corrected by performing some extra iterations in the simulation domain.

From a larger distance, the design process first focuses on lowering the abstraction level of the design from requirements to a simulated controller, and then makes a translation step to an implemented controller. The first part is not uncommon in engineering; the principle can be found in many books about design, for example [vdKS98] uses 'doel', 'functie', 'structuur', and 'inrichting' (English translation: goal, function, structure, and layout). Also, [Bra93, Page 20] shows that several methods have the same global-to-detailed design-process phases, although the number of phases and the names of the phases differ with each method. The second part where the developed model is translated to an implementation is rare. The step from model to implementation is often made, for example in [Bri97], but these approaches start with an implementation in another language using the developed model as a guide. Aside from the fact that the developer needs to know at least two languages, there is a large chance of introducing errors into the implementation while implementing.

The design technique proposed in this project is depicted in Figure 1.4. In the figure, a rectangle represents a state of the design. Arrows between design states represent development steps or *design steps*. Development starts with a specification at a high level of abstraction. By modelling, simulating, and making design decisions, the design is made more concrete, that is, it is lowered in abstraction level. This is shown in the figure by the downward arrows. Such development steps are called *vertical design steps*. After enough vertical design steps have been taken, the abstraction level is low enough to implement the system in the real world. Translating to an implementation is called a *horizontal design step*, and is represented by the horizontal arrow in the figure. The step crosses the line between the simulation domain and the real-world domain. Unlike vertical design steps, it does not lower the abstraction level. It is the process of constructing an implementation as specified in the model. In other words, the horizontal design step creates a real-world version of the simulation model.

Language

The language used in modelling and simulation is very important in the design technique. It is the language in which developers express their design, so it must be easy to use and learn. Also, since it must be interpreted by a machine, it has to be a formal language.

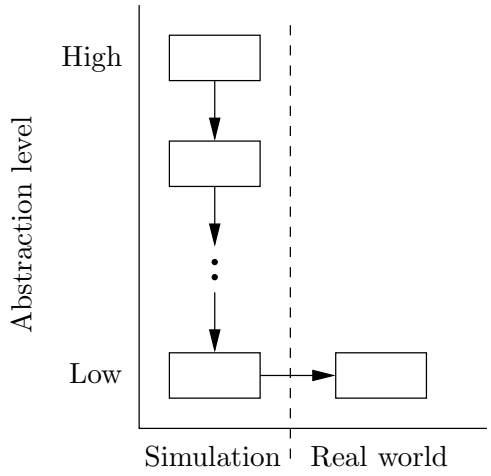

Figure 1.4: Design technique using simulation.

Since the implementation of a controller is a piece of software, it is tempting to use common software-engineering methods, for example described in [HP88, Coo91], or the newer object-oriented method UML [BRJ99]. These methods use diagrams to model the software, and manage to capture a part of the dynamics of the system, but none of them uses simulation which is a powerful technique for our users. Also, they do not treat the software and its environment as a two-unit, they concentrate on software only. For these reasons, the software-engineering methods have been dropped as modelling language.

The Systems Engineering Group already uses their own language χ for modelling and simulating, with a lot of success. Also, tools for simulation are already available. This makes the χ language the prime candidate. Nonetheless, it should be verified that χ is not a wrong choice.

χ is based on CSP [Hoa78], a formalism where multiple concurrently executing processes work together. The description of behaviour of processes themselves is based on process algebra, except an imperative style of programming is used to describe behaviour rather than a functional style. Data modelling in χ is done using constructs commonly found in modern programming languages, such as booleans, integer and real numbers, strings, tuples and arrays, lists, and sets. A more detailed description of the language can be found in Chapter 2. Also, χ fulfills the requirements mentioned previously. It has parallel execution, which allows easy specification of simultaneous tasks, and it has the concept of time, allowing timeliness properties to be expressed. Finally, the imperative style of programming and the powerful data handling constructs can be translated to a normal programming language, like C++.

For the implementation of the tools, a language is needed too. The existing simulation tool [NA98] consists of two parts, a compiler that translates χ to C++, and a run-time environment that performs the simulation. Both parts are written in C++.

In the horizontal design step, a similar strategy seems beneficial: a compiler for trans-

lating the χ model to the language understood by the implementation, and a tool in the run-time environment to support the translated code. For the compiler, an object-oriented programming language is better suited for the job, because the object-oriented paradigm provides better abstraction mechanisms. The cost of object-oriented techniques compared to the imperative paradigm is an increase in processing power and memory. Since the compiler is executed on a normal computer system with plenty of processing power and memory, these costs are not a problem. The two major programming languages in object-oriented programming are Java and C++. Since the compiler for the simulation tool is written in C++, the latter language gives better performance, and the author of this thesis has more experience with the language, the choice of C++ for the compiler will not be a big surprise.

For the run-time environment of the controller implementation, the situation is less clear. In embedded systems, small processors are common. Also, C is a common programming language, often with some processor-specific extensions. The Java language is also used in some cases, for example in set-top boxes. Despite this, the programming language C++ has been chosen for the run-time environment as well. C has been dropped because the project aims at the design of complex industrial machines. In this context, the cost of more processing power and more memory is not very important. Secondly, since the project is done as part of a research programme, it is expected that the software will be changed often. The higher abstraction of object-oriented languages is clearly an advantage then. Thirdly, C++ can be used in combination with C, which means that integration with software written in the latter language is possible. (To be fair, Java can also be used in combination with C, but C++ is designed for backwards compability.) Fourthly, availability of templates in C++ makes the handling of complex data types easier. Finally, in the run-time environment, performance plays a big role, and C++ delivers better performance than Java. Also, run-time maintenance like garbage collection is not performed.

1.4 Implementation of tools

In Section 1.2, it was concluded that tools were needed to support the design technique. The latter consists of two kind of steps; vertical design steps and horizontal design steps. The former takes place in the simulation domain, the latter crosses the border from the simulation domain to the real world. For each kind of design step, a computer-based tool is necessary:

- *A simulation tool.* A vertical design step is a transformation of a simulation model to a lower abstraction level. The design method aims at giving the engineer an understanding of the dynamic behaviour of the developed model. The best way to support this process, is by providing a simulator capable of showing the dynamic behaviour of the specification written by the engineer.

  Currently, there is no tool support for formally checking certain properties, for example (lack of) deadlock. Also, equivalence checking or testing between successive

Figure 1.5: Overview of the operation of a tool.

simulation models is not supported by tools. Such tools may be added in the future if the need arises in the research.

- *A translation tool.* A horizontal design step transforms a specification from the simulation domain to the real world. Tool support is needed to transform the specification to an executable with the same behaviour, running on an embedded controller.

Both tools take a χ specification, and produce an executable version of it that shows the specified behaviour. Since the tools should be usable in larger case studies, speed of execution of the result is important. Speed of the transformation from the specification to the executable format is not that important, because the conversion takes little time compared to the execution time of the result. The architecture of an existing simulation tool [NA98] has good properties in this respect, and this architecture is used as blue print for the design of the new tools. The idea is shown in Figure 1.5. Creating behaviour of the specification is done in two stages. First, the specification is compiled to an executable form. This executable form is then executed to obtain the behaviour. The benefit of this approach is that a large number of checks with respect to the correctness of the specification can be done during the compilation instead of during the execution. Furthermore, during the compilation stage, the program temporarily exists as specification in the intermediate implementation language C++, as discussed in the previous section.

The similarities between the simulation and the translation tools, the use of two stages to obtain behaviour of the specification, and the use of C++ as intermediate language also have its impact on the organization of the source code of the tools. In particular, both tools need

- a compiler front-end to parse and check the χ specification,
- a compiler back-end to generate C++ code suitable for execution,
- a run-time engine to 'generate' behaviour common for all specifications, and
- libraries to provide additional functionality.

The compiler front-end and back-end is a standard compiler construction approach. The run-time engine is highly coupled to the compiler back-end. It contains for example a

scheduler that chooses statements to execute. It also contains code to represent and compute data values during execution. The libraries contain code that provides functionality used only by some specifications. For example the square root function. In the case of the translation tool, the libraries also provide access to the physical machine. This is discussed in more detail in Section 3.2.

Libraries are also the means to add new functionality to the tools. This is used to experiment with new ideas. For example, recently, the wish to combine a χ simulation with other (third party) simulation tools has arisen. By building a new library that allows TCP/IP connectivity to other programs, experiments can be done to investigate this idea.

From maintenance point-of-view, it is advantageous to share as much source code as possible between both tools. The compiler front-end is completely shared by both tools. The run-time engine cannot be shared. The execution stage of both tools occurs at two different platforms, with different semantics (discussed in more detail in Chapters 2 and 3). Therefore, the run-time engines of both tools are separate pieces of source code. The code-generator of the compiler is tightly connected to the run-time engine, and exists therefore also twice (one generator for each tool). Libraries are common when they are platform-independent. For example, functions like the square root are common, but I/O interfacing functions exist only for the translation tool.

After the decision to construct six pieces of code (one compiler-frontend, two compiler-backends, two run-time engines, and one library), the next question was how to create all this functionality. An existing simulation tool ([NA98]) was available. Analysis showed that the global design ideas of the tool were good. Also, parts of the tool were re-usable, especially pieces of the run-time engine and the libraries. The compiler part was considered not re-usable, because

- The χ language had changed since the implementation of the old compiler. Concepts had been changed, or were added to the language.
- There was a fundamental design flaw in the type system of the compiler. χ uses both polymorphism and overloading. The type system of the old compiler is based on [Mil78]. This algorithm can handle only polymorphism. As a result, the type checking of the simulation compiler sometimes failed.
- Many features of the χ language and the translation to C++ were hard-coded in the compiler.

Especially the change of the type-checking system had a large impact on the existing compiler. Estimation of the amount of work showed that the benefit of re-use of existing compiler code was non-existent due to the huge amount of changes. That opened the path to creating a new compiler with an improved internal structure tailored to the new situation.

In the project, attention was foremost focussed on constructing a functional translation tool. The project started with constructing the compiler-frontend, then the run-time engine, followed by the code generator and the libraries.

Once the compiler-frontend was finished, the remaining simulation-tool functionality was implemented by Chen ([Che99b, Che99a]) as a separate sub-project, performed in parallel with the remaining three parts of the translation tool.

1.5 Reactive machine control

The title *reactive machine control* is basically a one-line description of the subject. Careful consideration of this description is necessary to give the readers a correct impression of its contents. Particularly important is the choice of the right jargon description. For this thesis, three candidates were considered close enough for a closer look.

Embedded is a recent term. Below are two definitions of the word.

> General-purpose systems are not designed for any specific applications but can be programmed to run different applications.
>
> In contrast, application-specific systems are designed for dedicated applications. ... As these systems are contained in a larger, and often, non-electric, environment, these are commonly referred to as *embedded systems*. [Gup95]

> An *embedded system* is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function. [Bar99]

Common to both definitions is the fact that the system has an application-specific purpose, and that other parts outside the system are related to it. A control system for a machine meets this definition, since the application is designed to control a particular machine. Also, the machine is a collection of parts that exists outside the control system and that has a relation to the system (without it, the machine would not function).

Real-time is another term which comes to mind when discussing machine control systems. The definitions address the fact that the system must be capable of influencing its environment on time.

> A *real-time computer system* may be defined as one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time. [Mar67]

> Whenever a computer system is required to acquire data, emit data, or interact with its environment at precise times, the system is said to be a *real-time computer system*. [LM87]

> *Real-time* systems are those which must produce correct responses within a definite time limit. Should computer responses exceed these time bounds, then performance degradation and/or malfunction results. [Coo91]

All machine control systems have to react on the machine within a certain time interval to prevent damage or degradation of performance of the machine.

Reactive is not often used in literature. It focuses on the fact that the system keeps on running indefinitely.

> *Reactive systems* are computer systems that continuously react to their environment at a speed determined by this environment. [Hal93]

> A *reactive system* is a system that maintains an ongoing interaction with its environment, as opposed to computing some final value on termination. [MP95]

A machine control system is reactive, since it continuously controls a machine at the speed of the machine without ever stopping (unless powered down).

Since the technique focuses on understanding rather than the combination of control system and its environment, 'embedded' is considered somewhat out of place. In the same way, timeliness of machine control systems is important, but not enough to use the word 'real-time' in the title. The ongoing dynamics of a machine control system is however a key-feature in understanding the design, making 'reactive' a very appropriate description for such systems in this thesis.

1.6 Thesis outline

This chapter explains the goal of the project in more detail, and also discusses the design technique proposed for the development of machine control systems. The next chapter explains the modelling language χ, how time is handled by the language and what happens when a χ specification is translated to the real-world domain. Chapters 3 and 4 describe the structure of the target system and the real-time environment. Also the translation done by the horizontal design step compiler is briefly described. As an example of usage of the design technique, the case study of 'the paint factory' is discussed in Chapter 5. Finally, concluding remarks, and possibilities for next steps are discussed in Chapter 6.

CHAPTER

TWO

The χ language

In this chapter, the χ specification language is informally introduced. More detailed descriptions of the language can be found in [BK00, Fáb99, NA98, Are96]. After the introduction of χ, the second part of this chapter discusses the semantics interpretation issue of the horizontal design step. A user develops a specification with simulation using χ semantics. The design technique described in the previous chapter should transparently transform the specification to an implementation using a horizontal design step. That implies that interpretation of the specification under real-world semantics should be predictable.

2.1 Language definition

Within the Systems Engineering Group, research is being done on the design of industrial systems. Since these systems are highly complex, mathematical theories like queueing theory cannot be applied adequately. Therefore, computer-based simulation is used as a design tool. Around 8 years ago, the simulation language in use at that time was not adequate any more, a new language was needed. It appeared that no existing simulation language met the needs of the application domain, and a decision has been taken to create a new simulation language, now known as χ. The language allows specification of discrete-event, continuous-time, and mixed discrete-event/continuous-time (hybrid) systems, thus covering a large range of industrial systems. The user is supported in the development of his model by a computer-based simulation tool, that allows verification and validation of the model. Meanwhile, the discrete-event part of the language has become stable, while the continuous-time part of the language is still being developed. Since this project only uses the discrete-event part of the language, references to 'the χ language' in this thesis always refer to the discrete-event part only, unless explicitly specified otherwise.

χ is a parallel language, it has concurrently executing processes that communicate with each other using synchronous communication channels. Its intended users are (mechanical) engineers and students with little or no training in formal methods.

Technically, the language is heavily based on CSP [Hoa78, Hoa85] and ideas from [vdS93]. It is a static language, the number of executing processes as well as the topology of the communication channels between the processes is fixed. Also, the type system is static (can be checked completely during compile time) and uses structural type equivalence (two types

are the same when their structure is the same). The static nature of the language allows the compiler tool to perform a lot of checks for errors.

For the user, χ is a small language with few concepts, which are as orthogonal to each other as possible. Much attention has been paid to making the language feel 'natural' to the user, but not at the cost of introducing difficult to understand constructs. Also, the user should be precise when specifying a model. This feeling of preciseness is designed into the language as well. The statically defined data types is one example, other examples are lack of type widening[1], and no support for 'unnatural' operations on data types, such as projection on a list.

In general, the language is considered to be a kind of mathematical tool for specifying industrial systems. Like in mathematics, short and clear descriptions are possible, not only for small systems but also (especially!) for complex ones.

Since the subject of this thesis does not concern the χ language itself, and other documents with more precise descriptions exist, this chapter only gives an informal introduction to the language. The formal language definition can be found in [BK00]. Below, the syntax and an informal description of the semantics of each construct is given. The next section discusses time-related aspects during an 'execution' of a specification.

The syntax of χ language constructs is defined by a set of production rules of a grammar. The general form of a production rule is $T ::= t_1\ t_2\ \ldots\ t_n$. It defines that the sequence of terminals and/or non-terminals t_1 through t_n can be rewritten to non-terminal T. The symbol ϵ denotes an empty sequence. Non-terminals are written in uppercase italics, keywords and symbols are written in roman, while more complex terminals (such as identifiers like *const-id* or *type-id*, or expressions[2] like e, b, or c are written in lowercase italics.

There may be more than one production rule with the same non-terminal at the left-hand side. This means that there may be more than one sequence of terminals and/or non-terminals that can be rewritten to the same non-terminal. The vertical bar is used as a shorthand notation for multiple production rules with the same non-terminal at the left-hand side. As an example, the two production rules $W ::= A$ and $W ::= b\ B$ specify that the non-terminal A as well as the sequence $b\ B$ may both be rewritten to the same non-terminal W. Using shorthand notation, both rules can be written in a single line $W ::= A \mid b\ B$. The χ language itself also uses the vertical bar as a symbol. In order to differentiate between both uses, the vertical bar used as a terminal in the language is written between quotes, as in '|'.

The top-level production rule of a specification is χ:

$$\chi \ ::= \ \chi\ \chi \mid \mathit{TD} \mid \mathit{CD} \mid \mathit{FD} \mid \mathit{PD} \mid \mathit{SD} \mid \mathit{XD}$$

[1] Type widening is the implicit conversion of a data type to a larger data type. For example in $d := 3$, where d is of type real. The integer number 3 is implicitly widened to 'fit' into the floating-point variable.

[2] In the implementation, an expression is a non-terminal rather than a terminal. However, the syntax of expressions is considered common knowledge and will not be discussed in this thesis.

non-terminals *TD*, *CD*, *FD*, *PD*, *SD*, and *XD* are *type definition*, *constant definition*, *function definition*, *process definition*, *system definition*, and *xper definition*, respectively. All these definitions are explained below.

Data types

χ is a high-level specification language, and the available data types reflect this. The user may use all types T, where

$$\begin{array}{lcl} T & ::= & \mathit{void} \mid \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \mathit{real} \mid \mathit{string} \\ & \mid & T^* \mid T^+ \mid T^n \mid T_0 \times T_1 \times T_2 \times \ldots \times T_m \\ & \mid & -T \mid ?\,T \mid !\,T \mid {\sim}\,T \\ & \mid & (T) \mid \mathit{type\text{-}id} \\ TD & ::= & \text{type } D \\ D & ::= & D, D \mid \mathit{type\text{-}id} = T \end{array}$$

On the first line, the basic data types are listed. The *void* data type is only used for synchronisation channels. The other basic data types are booleans, natural numbers, integer numbers, floating-point numbers, and strings, respectively. More complex data types may be constructed by using type operators. T^* constructs a list containing values of type T, T^+ does the same for sets, and T^n constructs statically sized arrays. The data type $T_0 \times T_1 \times T_2 \times \ldots \times T_m$ constructs a tuple where each field i ($0 \leq i \leq m$) has a possibly different type T_i. The third line describes production rules for constructing data types used for communication. By prefixing a type T with a dash, a communication channel capable of transporting data of type T is constructed. A process does not use channels directly, it accesses them through *ports* instead. The ?-operator constructs a port for receiving a value from a channel, the !-operator is used to construct ports for sending data, and the ~-operator is used to construct synchronization ports. On the fourth line, a production rule specifies that a data type may be grouped by surrounding it with brackets. Finally, a data type may be given a name by using a type definition (TD). The name serves as a shorthand notation for the type. Because χ uses structural type equivalence, using a name of a type in the specification is equivalent to inserting the type associated with the name at that point surrounded by brackets.

Constants

It is often convenient to use symbolic names as a representation of constants. χ facilitates this by the following production rules for constant definitions:

$$\begin{array}{lcl} CD & ::= & \text{const } X \\ X & ::= & X, X \mid \mathit{const\text{-}id} : T = e \end{array}$$

A constant definition CD is a list of definitions, each having an identifier, a type T and a value expressed in e. All values must be constant at compile time, they may only rely

on other constants. Using a identifier *const-id* in an expression is equivalent to using its associated value.

Functions

Some calculations are too complicated to be expressed in a single line, or are used at many different places in the specification. These calculations can be defined in a function, and be computed by calling the function by name:

$$
\begin{array}{lcl}
FD & ::= & \mathsf{func}\ \textit{func-id}\ FP \longrightarrow T = [\![\ \ L\ FS\ \ ]\!] \\
FP & ::= & () \mid (V) \\
V & ::= & V, V \mid I{:}T \\
I & ::= & I, I \mid \textit{var-id} \\
L & ::= & \epsilon \mid V\ \text{`}|\text{'} \\
FS & ::= & \text{skip} \mid e_1 := e_2 \mid \uparrow e \mid [\ FG\ ] \mid *[\ FG\ ] \mid *[\ FS\ ] \mid FS\,;\ FS \\
FG & ::= & [\!]_{1 \le i \le n}\, b_i \longrightarrow FS_i
\end{array}
$$

The function definition *FD* defines a function with a unique name *func-id*. The types of parameters needed by the function are specified in the formal parameter list *FP*. The return type of the function is specified by T. Inside the function, local variables may be introduced (production rule $L ::= V$ '|'), followed by a function statement *FS* which describes the algorithm of the calculation. Statements allowed in a function are:

- The *skip statement*, which does nothing.
- The *assignment statement*, which copies the value expressed by e_2 into the variable indicated by e_1.
- The *return statement*, which ends execution of the function and returns the value expressed by e to the caller.
- The *guarded statement*, which performs a non-deterministic choice between several alternatives. The statement continues execution with one of the statements FS_i for which the boolean guard b_i evaluates to true. χ requires at least one guard to evaluate to true. Also, a non-deterministic choice should not influence the resulting value of the function call.
- The *repetitive statement*, which is a repetitive version of the guarded statement. Repetitively, guards are evaluated, and a statement is non-deterministically chosen and executed, until all guards evaluate to false. At that moment the repetitive statement is finished, and execution continues with the statement following the repetitive statement. Note that repetitive non-deterministic choice is the only property of the statement. Other properties, for example fairness of choice are not part of the language. The form '$*[\ FS\ ]$' is a shorthand notation for '$*[\ \text{true} \longrightarrow FS\ ]$', and gives the user an endless loop.

- Finally, the semicolon concatenates two statements. Execution of this statement means execution of the statement before the semicolon, followed by execution of the statement after the semicolon.

A function in χ is a function in the mathematical sense. That means that a function *const-id* should always return the same value if it is called with the same parameter values. To enforce this, formal parameters are passed by value, and constructs with side effects (for example communication with another process) are not allowed. The only exception to this rule is the guarded statement which allows a non-deterministic choice. It is assumed that the user will take care of specifying functions with proper functional behaviour.

Process and system definitions

Process behaviour is defined in process definitions. The topology of multiple concurrently executing processes is specified in system definitions. This division between process specification and process instantiation is a separation of concerns. The former concentrates on the inner working of a process, while the latter concentrates on how the processes are connected to each other.

$$
\begin{array}{lcl}
PD & ::= & \mathsf{proc}\ \textit{proc-id}\ FP = [\![\ L\ PS\ ]\!] \\
PS & ::= & \mathrm{skip} \mid e_1 := e_2 \mid \mathrm{terminate} \mid E \mid [\ PG\ ] \mid *[\ PG\ ] \mid *[\ PS\ ] \mid PS\,;\ PS \\
C & ::= & c\,?\,e \mid c\,!\,e \mid c\,? \mid c\,! \mid c^{\sim} \\
E & ::= & C \mid \Delta e \\
PG & ::= & [\!]_{1 \le i \le n}\, b_i \longrightarrow PS_i \mid [\!]_{1 \le i \le n}\, b_i\,;\ E_i \longrightarrow PS_i \\
SD & ::= & \mathsf{syst}\ \textit{syst-id}\ FP\ = [\![\ H\ \text{`}|\text{'}\ N\ ]\!] \\
H & ::= & H, H \mid I{:}T \\
N & ::= & N \parallel N \mid \mathit{id}(e_1, e_2, \ldots, e_n) \\
XD & ::= & \mathsf{xper}\ = [\![\ \mathit{id}(e_1, e_2, \ldots, e_n)\ ]\!]
\end{array}
$$

The production rule *PD* specifies a process definition with name *proc-id* and statement *PS*. Despite the new name of the production rules, the statements are very similar to statements in a function definition. Below, only the new or changed statements are discussed.

- The *terminate statement* terminates execution. The statement is like an instantaneous self-destruct button. The statement is considered a kludge, and will probably be removed in the future.

- The *communication statements* listed in productions rules C are used to communicate with a *single* other process using a communication channel. In other words, communication channels are point-to-point connections between two processes. In all cases, c is an expression which evaluates to a port. The question and exclamation marks indicate the direction of communication (reception, respectively transmission). In the first two production rules, e is an expression used for data transfer. It respectively evaluates to a variable and a value. The remaining production rules are used

for synchronisation between processes. With synchronisation, no data is transferred. Rules $c?$ and $c!$ still have a sense of direction, while rule $c^{\sim}$ is a direction-less synchronisation.

- The *delta statement* rule Δe ($e \geq 0$) causes the process to sleep for e time units.
- The *guarded statement* and the *repetitive statement* have been extended. The first production rule of *PG* is the same as in the function definition. In the second production rule, the statement waits for communication and/or delta statements, before continuing execution. Effectively, the second form of the statement allows waiting for communication with another process and/or passing of time, which is why it is called the *selective waiting statement*, and the repetitive version is called the *repetitive selective waiting statement*.

The system definition *SD* creates concurrently executing processes (which may communicate with each other) using the parallel composition operator $\|$. The instantiation of a process is written as $id(e_1, e_2, \ldots, e_n)$. The process definition being instantiated is indicated by *id*. The values of the expressions e_i ($1 \leq i \leq n$) are used as initial values of the actual parameters of the process. Systems may be instantiated in the same way. Instantiation of a system means instantiation of the processes in its body. Nested instantiation of systems is allowed, recursive instantiation is not allowed.

The top-level instantiation used to instantiate the entire model is specified in the xper definition *XD*.

2.2 Time-related aspects

In the previous section, the syntax and informal meaning of each statement was explained. For understanding a specification, the order of executing statements in different processes is also important. χ is a language that has the concept of time. How time behaves in relation to execution of the statements is the topic of this section.

Language

The operational semantics of the discrete-event part of the χ language is formally described in [BK00]. Informally, the instantiated specification is translated to a tree of statements. Execution of a statement is performed by non-deterministically selecting a statement in the tree that can be 'executed' according to the rules of the semantics, followed by modifying the tree. Only one statement can be executed at each time, which naturally leads to *interleaving* semantics.[3]

Another view of an executing χ specification is to consider each instantiated process as a separate state machine. A transition in a state machine represents execution of a single statement. All state machines have local state. All those local states together

[3]Communication over a channel is regarded as a single distributed assignment here.

form the global state of the specification. At each time, only a single state machine can make a transition, thus resulting in interleaving semantics. This latter view is closer to an implementation, and will be used in this thesis.

Handling of *time* in χ is done by having different types of transitions. On the one hand there are time-less transitions where time is not progressing, while on the other hand some transitions cause non-zero progress in time. Choosing between transitions is done using the *maximal progress* principle. As long as time-less transitions are possible, one of these transitions is chosen and executed. When only time-progressing transitions exist, the non-empty set of those transitions is chosen, such that progress in time is as small as possible. When none of the state machines can perform a transition, the system is considered to be in a deadlock.

The above semantics means that most statements in a χ program do not cost time. In particular, calculations and communications with co-operating partners are performed time-less. This is known as the *synchrony hypothesis* [Hal93, BCG+97]. This abstract notion of time is very convenient during the design of a system, because it simplifies reasoning about its behaviour.

In the real world however, time-less activities do not exist, all activities cost time. Even worse, time is also progressing when you do nothing. In terms of state machines, one would say that a transition between states costs time and staying in a state also costs time.

Concurrency in the real world is in principle resolved using *true concurrency*. Multiple independent processors each perform transitions on their own. That means that it is possible to perform multiple state transitions at the same time. The tricky part here is that 'process' and 'processor' are not the same thing. The former is an abstract notion of a state machine with its local state, the latter is a physical resource that can make transitions in a state machine. The creation of a good mapping from process(es) to processor(s) is an area of research in the *real-time systems* and *embedded systems* disciplines. Below, the relation between time in the real world (wall-clock time) and time in a specification is discussed in the context of the simulation and translation tools.

Simulation

Several simulators have been built for the χ language in the past years, see [NA98, Fáb99]. These simulators adhere to the χ semantics. Since there are currently no real-world computer systems that adhere to this semantics, the *simulation time* perceived by the simulation of the model is completely decoupled from the time in the real world, the *wall-clock time*. In this way, the simulation time can be manipulated as desired within the constraints of the χ semantics. The advantages of this approach are *a)* execution of statements can be done completely time-less in simulation time, *b)* long expiration intervals can be executed in less wall-clock time than indicated by the interval, and *c)* simulation results become independent of the performance of the computer system being used, although the wall-clock time needed to get those results still depends on it.

In the past, the simulators have been mostly used to simulate manufacturing-control

systems. Machine-control systems can be simulated with the same simulators however. In principle, machine control systems are event-based. That means in a system as in Figure 1.3, the controller C waits until the machine *sends an event*, for example by communicating or by failing to communicate within a certain period. Upon the reception of an event, the controller starts *processing the event*, it calculates how it should react on the received event. Once the calculation is complete, the reaction is sent back to the machine.

Since the χ semantics uses maximal progress, it can be said that the simulated controller reacts infinitely fast to events.

Implementation

With the horizontal design step, a specification developed under χ semantics is interpreted under real-world semantics. The basic change when switching the semantics is that execution speed becomes finite. Processing of events suddenly costs time. In other words, the controller reacts only fast rather than infinitely fast under real-world semantics. The realization of any implementation means basically deciding about the mapping of processes to processors, along with other properties of the processors, like performance and/or reliability. As said before, this decision should be made by using research results in the field of real-time systems and/or embedded systems.

2.3 The horizontal design step

Since time in χ is handled differently than time in the real world, interpretation of the same specification under both semantics will create different behaviour. These differences must be small enough to make the horizontal design step a useful activity. Below are a number of approaches that may be employed to reach this objective.

Formal approach

The formal approach to deal with this issue is to change the semantics of χ. In particular, try to find a semantics χ' that defines behaviour of statements such that it mimics real-world behaviour more closely. A relatively simple change in this direction is the introduction of passage of time while communicating, like in [Hoo91]. A much more elaborate change is for example [LMW99]. They simulate the execution of a program on a computer, down to a simulation of the processor caches and internal buses of the processor. The big disadvantage of this approach is that the semantics becomes more difficult to understand. More difficult semantics also means that it is more difficult to understand the behaviour of the specification. Exactly this understanding is crucial to advance in the design of the highly complex industrial machines, at which the design technique is aimed. That means that changing the semantics may be helpful in capturing the real-world semantics, but at the same time, it hampers development of the earlier stages. Since the earlier stages in the

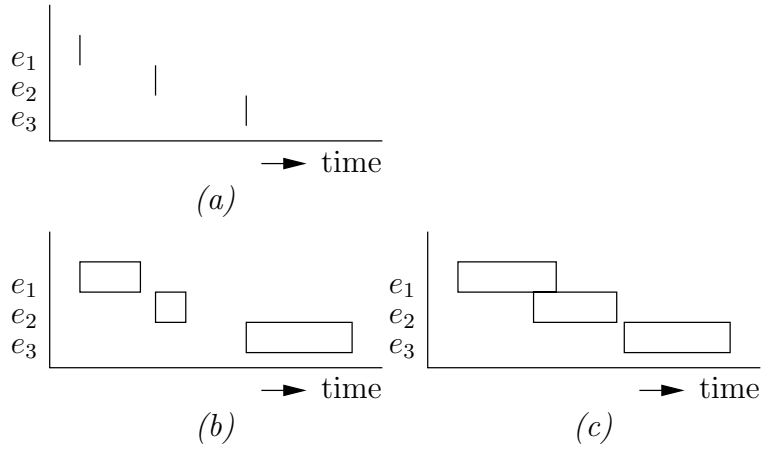

Figure 2.1: Timing of execution of a specification.

form of vertical design steps are considered more important than an easy horizontal design step, this approach has been abandoned.

Assume approach

Another approach is to simply assume that the execution time of a controller to process an event can be neglected compared to the response time of a machine. In other words, the response time of a controller is so small that it may be safely ignored. This assumption makes development, in particular the horizontal design step, much easier. It basically states that execution is sufficiently fast to assume that it is infinitely fast, which is exactly the assumption made by the χ semantics. The horizontal design step can thus simply be implemented by porting the specification from the simulation environment to the real-world environment.

The disadvantage of this approach is the assumption itself. In particular, is the assumption valid, and if not, what then? Figure 2.1 shows what happens when interpreting the same specification under χ semantics and under real-world semantics. In Figures 2.1a through 2.1c, execution of three events e_1, e_2, and e_3 is shown. The events are listed vertically, time progresses horizontally. Execution of the event is shown with a bar in the figure. When the event is received, the bar starts. When calculation of the response for the event is finished, the bar ends. In 2.1a, χ semantics is used for the execution. Since that semantics uses the synchrony hypothesis, each event starts and is executed entirely at that moment. The result is thus an infinitely small bar, which is drawn as a vertical line. When the same specification is interpreted using real-world semantics, time progresses during execution of the event, and pictures like 2.1b or 2.1c come into existence. The starting time of the event is different from the ending time, and thus a bar with a non-zero length appears. The length of the bar indicates how fast the event is handled. Faster execution means shorter bars. A few points can be made about this figure:

- If the real-world system would have infinite execution speed, then the bars would be infinitely small, which would produce the same picture as Figure 2.1a.
- On a relatively fast system, execution of one event is finished before the next event is received. A picture like 2.1b emerges. At all times, the system is either idle or executing exactly one event.
- On a slower system, bars in the picture become longer. Also, if events arrive faster than they are executed, the execution of different events may overlap each other, which means that the system is busy executing multiple events at the same time. As an example, see Figure 2.1c.

In the latter case, the execution of several events at the same time may cause events to interfere with each other, and cause unspecified behaviour with respect to the χ semantics. As an example, consider the following specification:

$$
\begin{aligned}
\mathsf{func}\ f() \longrightarrow \mathrm{nat} &= [\![\ldots]\!] \\
\mathsf{proc}\ P(a\colon !\mathrm{nat}) &= [\![\ x\colon \mathrm{nat} \mid x := f()\,;\ a\,!\,1 \ ]\!] \\
\mathsf{proc}\ Q(b\colon !\mathrm{nat}) &= [\![\ \Delta 4\,;\ b\,!\,2 \ ]\!] \\
\mathsf{proc}\ R(a, b\colon ?\mathrm{nat}) &= [\![\ x\colon \mathrm{nat} \mid [\ a\,?\,x \ [\!]\ b\,?\,x\ ]\,;\ !\,x \ ]\!] \\
\mathsf{syst}\ S() &= [\![\ a, b\colon -\mathrm{nat} \mid P(a) \parallel Q(b) \parallel R(a, b) \ ]\!] \\
\mathsf{xper} &= [\![\ S() \ ]\!]
\end{aligned}
$$

For simplicity, it is assumed that execution of the program starts at time $\tau = 0$.[4] Under χ semantics, execution of all statements is timeless. Therefore, process R receives an event at time $\tau = 0$ from process P. R handles the event by executing the communication, and printing 1 on the output. At time $\tau = 4$, process Q receives a time-out event from the delta statement. This is also the only statement of Q being executed, because the communication with process R cannot take place.

Under real-world semantics, execution of any statement costs time. To simplify the explanation, it is assumed that the execution time of statements in the processes may be ignored compared to the execution time needed to compute the result of function f. The execution time of this function is represented with t_f. If $t_f < 4$, then the first event is received at time $\tau = t_f$, and the expiration event of the the delta statement is received at time $\tau = 4$. Handling both events is done in the same way as under χ semantics. If $t_f > 4$, then the delta statement expires before the computation of f finishes. Communication using channel b becomes possible before communication using channel a. Execution of the first event at time $\tau = 4$ of R therefore prints 2 instead of 1, which is incorrect with respect

[4] This is not a limitation, since χ uses relative time-outs.

to the semantics of the χ specification. When $t_f = 4$, then an unpredictable choice is made between one of the above executions.

Obviously, a specification interpreted under real-world semantics should show specified behaviour only. This can be done in the following ways:

- Assume that the system is handling at most one event at any time. In other words, assume that nothing bad happens.
- Perform some form of monitoring on the system. Monitoring the behaviour of the system enables us to detect potential problems.
- Enforce execution of a single event at any time by adding a layer between the machine and the controller. The layer only allows passage of one event at a time. The controller executes the event and returns a response, the layer then passes the next event, etc. Since at most one event is passed to the controller, the controller will behave as specified.

  Whether delaying events is dangerous depends on the type of control being used. One type of control is *feedback control*, where the controller must react immediately. For example, a motor must be stopped when a package has reached the end of a conveyor. If the controller would not react immediately to the sensor at the end of the conveyor, the package would drop on the floor. The other type of control is *supervisory control*. Events are sent as an acknowledgement that a certain operation is finished. In the previous example, the motor would be stopped by a lower layer of control (for example, the sensor is hard wired to the motor), and the machine control system then gets an acknowledgement that a new package has arrived. With supervisory control, it is not dangerous when the controller reacts too late. Reacting too late will only degrade the performance of the machine.
- Write a robust specification, in other words, one that makes no assumptions about the order of events. The above example may be a correct χ specification, it is not necessarily a good model of a controller. In fact, it is quite unlikely that a customer would be satisfied with a controller that breaks if events come in an unexpected order.

  In the normal practice of writing specifications, a user develops models that can withstand an unexpected order of events.

The last one of these solutions indicates that the problem of interference may be smaller than we think. None the less, there should be a mechanism to verify that the specification is indeed robust.

Splitting the horizontal step

A third approach is splitting the horizontal design step. Performing the entire horizontal design step in one time seems too much. By splitting the step into multiple smaller steps, it may become easier to tackle the problem:

- One way to 'prepare' the specification for the real world is to add delta statements to the statements in the specification, to reflect the passage of time in the real world. Simulation of the extended specification may predict behaviour of the controller in the real world.
- Another way is to insert an intermediate step between the χ simulation and the real-world execution. For example, a simulation could be performed using a semantics somewhere between the χ semantics and the real-world semantics. If simulation under that semantics behaves correctly, some confidence in the correctness of the specification is gained.
- Finally, it is possible to simulate the specification using the real-world semantics by running the simulation on the implementation platform. In other words, the transformation to the implementation is made, but the virtual machine is not replaced by the real machine. The resulting implementation thus contains the controller as well as the virtual machine, yet real-world semantics is used for interpretation.

Each way has its own advantages and disadvantages. However, common to all ways is that interpretation of the specification is performed either with a changed program or with changed semantics. It is unclear what success or failure of this interpretation really means for the correctness of the specification.

The construction of a perfect horizontal design step implies a very thorough understanding of real-world semantics. At this moment, nobody has achieved this goal, at least not such that a well-defined relation between the χ semantics and the real-world semantics can be defined within the project. Development of that knowledge is very interesting, but outside the scope of this project. For this reason, this project uses the 'assume nothing bad happens' approach to deliver a first operational version of the development method.

Then, by doing a number of case studies, the impact of the change in semantics can be evaluated.

CHAPTER

THREE

The real-time platform

In the design approach explained in the first chapter, the developer of a control system can perform a horizontal design step from simulation to real-world implementation. In order to allow the users to perform this step, the translation should be automated as much as possible. Before before building a computer tool, it is necessary to define the structure of the target environment. In this and the next chapter, the structure and its implementation are explained.

Figure 3.1 gives a global view of the target environment. At the bottom is the machine which is being controlled by the target system. The target system interacts with the machine using sensors and actuators, and controls the machine. The system itself consists of three parts, the *real-time operating system*, *run-time support*, and the generated χ *implementation*. The last two parts together are also called the *machine control application*, since they form an application for the real-time operating system.

The remainder of the chapter discusses the target environment in a bottom-up manner. First, the choice of the real-time operating system is discussed, followed by the design issues of the machine control application. In Sections 3.4 and 3.5, the implementation of the run-time support layer respectively the χ implementation, is discussed. The chapter is concluded with some possible directions for future extensions.

3.1 The real-time operating system

An operating system is responsible for managing resources of a computer system and providing access to its resources for applications being executed on the computer. Real-time operating systems are special in the sense that they are tailored to managing resources on small and heavily customized computer systems. Also, they allow access to resources such that hard real-time guarantees can be given for carefully designed applications.

In the area of machine control, it is important to have hard real-time guarantees, as it allows construction of correct controllers which react to sensor readings within a pre-defined time limit, thus allowing actuators to react within a pre-defined time limit. Timely reaction of the actuators ensures that behaviour of the machine can be limited, thus allowing the machine as a whole to behave in a controlled way.

For simple machine control systems, for example switching a current on and off at pre-defined times, it is not necessary to have a real-time operating system. The controller itself

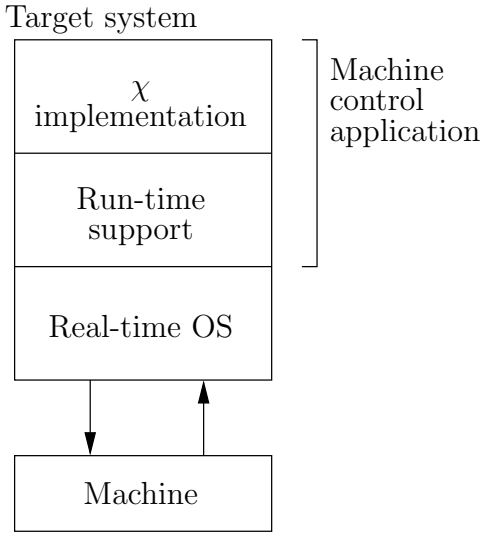

Figure 3.1: The target system and its machine.

can manage its resources. This project however aims to be capable of handling complex control systems as well. In these systems, timing requirements are more complex. Also, it is very likely that the machine control application is not the only application being executed on the system. In such an environment, the machine control application can no longer decide by itself how and when it should use the resources, a real-time operating system must make those decisions instead. For this reason, a real-time operating system is necessary in the target system.

There are about three different ways to obtain an operating system for the target:

1. Build a real-time operating system from scratch,

2. Use a freely available real-time operating system, or

3. Buy a real-time operating system from a commercial vendor.

The first option was dropped almost immediately, since writing a real-time operating system is a complex and time-consuming job, and is quite likely big enough to fill the entire project. Creating only an operating system does not cover the objective of the project since it does not enable the design step from simulation to implementation. Also, there are already many real-time operating systems in the world and there is no compelling reason why an existing real-time operating system would not be sufficient for the job.

By dropping the first option, the decision was made to use an existing operating system. Because the functionality expected from the real-time operating system was small (it should support C++, semaphores, and threading), and most operating systems seem to support this functionality, the search for the 'right' operating system was not very extensive.

The operating system considered for the second option was *RTLinux*[1]. The 'normal' Linux platform has been used for several years in the group both for development of software and for simulations. Also, the RTChi project done earlier in the group [dBvdBCP97] provided positive experiences with this real-time platform. Commercial solutions considered for the third option (buying an operating system from a commercial vendor) were *QNX*[2] and *VxWorks*[3]. The former was already available within the group complete with all documentation, the latter is used extensively in the industry. In order to make a decision on which operating system to obtain, the requirements from the first chapter are carefully considered.

- The first requirement is that the entire project must be completed within the two years available for the project.

  This is another reason why the first option (writing our own operating system) is not feasible. However, it does not make a firm choice between using a freely available or a commercial operating system.

- The second requirement is that the real-time platform must be viable in an industrial environment. The design approach aims to be capable of designing complex industrial applications. The only way to test whether this aim is reached, is by designing controllers for machines actually used in the industry, and verify that those machines function satisfactory.

  Since the research group does not have enough resources (people and money) to do this within the university, industrial partners are being included in this part of the research. Therefore, the operating system chosen in the project has to be a viable option to them as well.

  All operating systems under consideration are known to be used in industry ([KZ96], articles in *Linux Journal*[4], and the industry and suppliers themselves). This fact is not really helpful in making a choice between them. A strong point in favour of a commercial solution is that industrial partners can buy support for the operating system. This argument supports the candidates QNX and VxWorks.[5]

  Also, the operating system should be capable of handling real complex machines. It is expected that the research will start with relatively simple machines, and then move towards more complex machines. Much effort would be wasted if the solution chosen now appears to be incapable of handling the more complex systems.

[1] For more information, point your browser to http://www.rtlinux.org/ .

[2] QNX is a product of *QNX Software Systems Ltd.*

[3] VxWorks is a product of *Wind River Systems, Inc.*

[4] *Linux Journal* is a monthly magazine dedicated to the Linux operating system and its cousin RTLinux. For more information, see http://www.linuxjournal.com/, or its publisher SSC at http://www.ssc.com/ .

[5] In the meantime, commercial support for RTLinux has also become available. That means that this operating system has become more viable to industrial partners.

- The third and last requirement being considered, is the ability to use the platform for educational purposes. It creates the need to be able to have multiple instances of the platform. None of the options above would create any major problems, except that a commercial solution may be costly.

The above considerations and requirements make no firm choice between using a freely available operating system, or buying a commercial solution. For this reason, technical facilities provided by free and commercial operating systems are weighed as well.

- *RTLinux* [Yod, Epp97] is a small piece of software which is positioned between the hardware and the Linux operating system. Hard real-time tasks are handled by RTLinux, and the remaining idle time is given to the Linux operating system. The philosophy behind it is 'small is beautiful'. In other words, RTLinux only delivers the minimal functionality (hard real-time tasks, their scheduling, and some interfacing to non-hard real-time tasks), everything else, including non-hard real-time tasks, is deferred to the normal Linux operating system, which provides a stable and very complete Unix environment, including networking and graphical applications.

  The system is open source, still under development, but has already been applied successfully in the industry. Also, RTLinux has been successfully used in our RTChi experiment.

- *QNX* [QNX96] is a micro kernel for the PC, based entirely on message passing. The usual distinction between operating system and application does not exist on this system, everything is a task. The distinction between programs in kernel space and programs in user space does exist, but since it is a micro-kernel, many traditional operating system tasks are executed in user space rather than in kernel space. The micro-kernel thus allows for a very modular and flexible system, including switching to a distributed system, because message passing across the network is handled transparently by the operating system.

- *VxWorks* [Win97] is a Unix-like, POSIX-compliant, hard real-time, multi-tasking platform, delivering all the usual programming features, such as multi-tasking, IPC, and RPC. It also supports a long list of different (multi-)processor architectures, and an implementation of all the usual TCP/IP services. Also, the system can be bought with extensive debugging and monitoring facilities.

All three operating systems provide a development environment, and the possibility to program the system in C and C++.

Because of its maturity, its multi-processing capabilities, the large number of supported processor families, and the familiarity of industrial partners with VxWorks, the latter has been chosen as operating system for the target system.

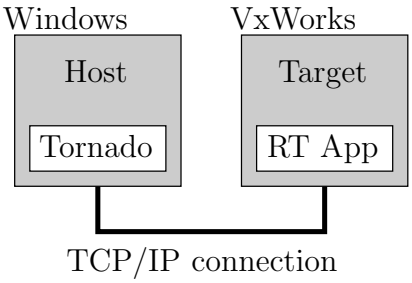

Figure 3.2: Overview of a VxWorks environment.

VxWorks

Wind River systems (http://www.wrs.com/) sells the real-time operating system *VxWorks* as well as the development environment *Tornado* for construction of the applications. The general structure is shown in Figure 3.2. The minimal setup consists of two computer systems, a *host* on which development of the real-time application takes place, and a *target system* (also known as *target*) on which the developed real-time application executes.

The host system uses a normal desktop operating system, such as Windows[6]. On the host, the development environment Tornado is installed. This environment provides the developer with the various tools needed for the development, testing, and debugging of the real-time application.

The target system is a dedicated machine for executing the real-time application. The VxWorks system can be delivered for a large range of processors, which allows for a lot of freedom in the choice of hardware for the target. Depending on what hardware and memory-space available is in the target, the system can be configured to include hardware drivers for example for a hard disc, or an ethernet device, application programming interfaces (API's) for facilities like asynchronous I/O, semaphores, message queues, memory management, queued signals, scheduling, and clocks and timers, networking protocols based on TCP/IP like telnet, FTP, SNMP, and HTTP, and testing and debugging facilities which can deliver detailed information about what is happening during the execution of the program.

The developed application and the VxWorks kernel can be linked together and be put in ROM to form a dedicated application that controls the embedded system.

3.2 Machine control application

On top of the real-time operating system lies the machine control application. While the name suggests it is a single application, it is in fact a framework which allows execution of any χ specification that a user will write. This framework consists of two layers. The top layer is the χ implementation layer. It is generated by the χ compiler from a specification

[6]Windows (in various flavours) are products of *Microsoft*.

written by a developer. The run-time support layer at the bottom provides an environment for the generated χ implementation. The environment uses operating system services to provide generic core services to the generated program, in particular

- it manages processes of the χ implementation,
- it provides a point-to-point synchronous communication service (communication using channels), and
- it translates time units used by the χ implementation to time ticks used by the operating system.

Besides these core services, the χ implementation needs other services as well. For example, an implementation of the data types used is necessary. Also, specifications need an interface to access the hardware they control. These services are deliberately not covered by the run-time support layer. Instead, the χ implementation uses separate libraries which (may) interface directly with the operating system. The main reason for not including these services is flexibility. By not including them in the layer, experiments with these services are easier to perform.

The decision to exclude hardware access from the core services means that a χ implementation has to control the physical machine by using function calls. Since communication statements are used to model hardware communication during simulation, this decision means that those communication statements have to be replaced by function calls during the horizontal design step. Unfortunately, the current χ language does not provide enough hooks to perform this replacement automatically. Therefore, the user has to manually replace the communication statements that model hardware access.

While the manual conversion is a disadvantage, the technique to do it has already been tried in the RTChi project. In that project, the technique worked quite nicely. Also, it was not clear what type of hardware would be used in the experiments. In order to be as generic as possible, the function-call mechanism was chosen as method to perform hardware access in this project.

Aside from the hardware-access conversion and the change in semantics which cannot be avoided (as explained in Section 2.3), the user should be able to use all language constructs in the simulation and in the implementation. This has the advantage that machine control specifications are not limited in the allowed language constructs, thus making research to find the best way to model controllers easier.

This chapter continues with an explanation of the structure of the machine control application followed by a discussion of a number of important services provided by the run-time support layer. The synchronous communication service (also provided by the support layer) is explained in the next chapter, since it is a relatively complicated core service.

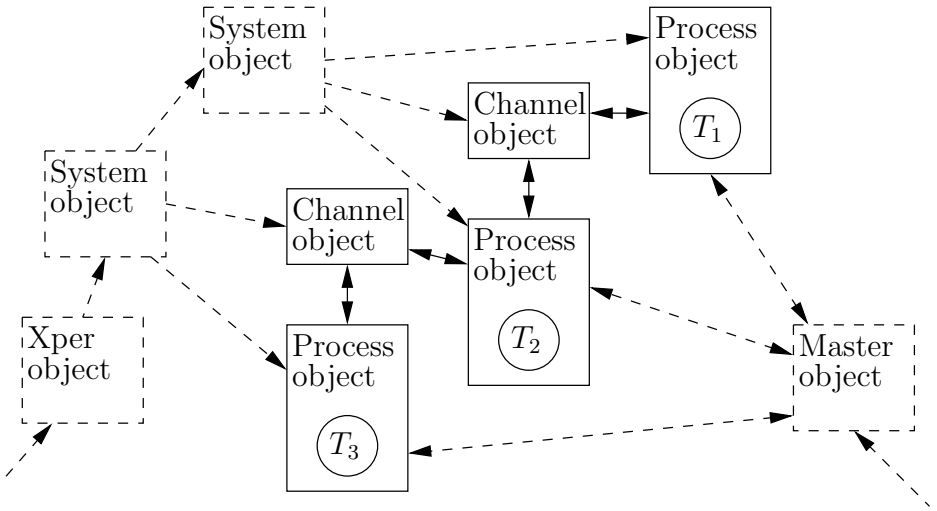

Figure 3.3: Data structures of an executing machine control application.

3.3 Data structures of the machine control application

The structure of data in an application explains a lot about the internal operation of that application. Therefore, it is important to discuss the data structures used in a machine control application executing a χ specification. A graphical representation is shown in Figure 3.3. The functionality of a χ process and the value of its local variables is kept inside a *process object*. Execution of this functionality (thus creating dynamic behaviour of the process) is done by constructing VxWorks tasks T_i within each process object in the target system. In the figure, three process objects with their tasks are drawn. Channels between processes only contain data, they are passive objects. This data is stored in *channel objects* in the implementation. Communicating processes rely on channel objects to forward their requests and answers to their communication partner. The communication protocol used by the process objects to decide which processes communicate is explained in the next chapter.

The part of the figure with solid lines (tasks, process objects, and channel objects) implements the behaviour of the χ specification. The arrows between the objects show which object has knowledge of which other objects. For example, the process object with task T_3 only knows about the channel object placed directly above it. That channel object knows about the process object, and the process object with task T_2.

Theoretically, it is possible to calculate the set of instantiated processes with their connected communication channels during compilation, and construct only the instantiated processes in run-time. That would mean that the expressions used to compute the processes to instantiate would either have to be calculated during compile time, or code must be generated to compute the values of the expressions during startup of the program. Since the expressions used in the instantiation-part of the specification can be arbitrary

complex, compile-time calculation is considered too complex. That means that code must be generated to calculate the instantiation of the processes. In that case, it is easier to keep the hierarchy during compilation, and create the machine control application recursively on the target starting from xper, until all process objects have been constructed. In the figure, the instantiation hierarchy is shown at the left with dashed lines. The *xper object* was the starting point of the application. The incoming arrow means that the environment knows the address of this object. The xper object constructs either a single system object or a single process object. System objects construct more system objects or process objects. In this way the instantiation hierarchy is traversed, until all process objects at the leafs of the hierarchy have been created. In the figure, the hierarchical relation between the xper object, the system objects, and the process and channel objects is shown with arrows.

As can be seen in the figure, the application is distributed over several tasks. Each task is only concerned with its local behaviour as specified in the χ program. That means that there is no task that knows what all processes are doing, and that can perform operations which affect the entire application, such as stopping the application. Since such operations are needed, an object that handles the global co-ordination has to be added in the implementation. This object is called *master object*. It is depicted at the right-hand side of the figure. Like the xper object, this object is known by the environment to allow processes outside the application to distribute information or events to the entire application. Also, all process objects have knowledge about its address, allowing them to request coordinated global activities.

Currently, the master object handles starting and stopping of the application, and provides the ability to get mutual exclusive access to shared resources such as the screen and the keyboard.

3.4 Run-time support

The horizontal design step has to be possible for each correct χ specification. That means that in fact many potential machine control applications exist. Therefore, the design described here can be considered a framework rather than a single application. The run-time support layer provides common functions (services), while the χ implementation layer, generated from the χ specification by the χ compiler, uses those functions and implements the behaviour specified in the specification. As a good engineering principle, the common part of the functions should be kept separate from the generated part. C++ provides the object-oriented inheritance mechanism for implementing this principle. The common code is kept in base classes, while the changing functionality is kept in derived classes. In the machine control application, the run-time support layer is stored in base classes. The χ compiler generates derived classes which implement the requested behaviour in the χ implementation layer.

The master object class and the process base class of the run-time support layer will be explained in more detail below.

Master object

As explained in the previous section, the master object *a)* distributes information and events to tasks of the application, and *b)* coordinates activities of tasks at a global level. In the current implementation, information about priority and stack-size of processes is distributed. Also, the events of starting and stopping of the control application are passed on. Finally, the object coordinates access to the screen and the keyboard, which are considered shared devices.

Having a single object in the application that is accessible by all tasks is not in line with the idea of having a distributed, localised set of tasks. Even worse, it may become a bottleneck in the system if too many tasks try to access it at the same time. Below, it is explained why having a single master object will not cause disturbance of the normal control cycle (that is, the response time of processes in the machine control application to events detected within the machine will not change significantly).

- *Startup of the application.* The master object is flooded with requests, since each process requests a stack size and a execution priority. Also, each process registers itself as existing in the application.

  At this moment, the machine control application is not executing. The flood of requests is thus not affecting the normal control cycle.

- *Shutdown of the application.* Each process unregisters itself, which creates a lot of requests at the master object.

  Just as during startup, the machine is not being controlled at this moment, which means that the normal control cycle is not affected.

- *Access to the keyboard.* The keyboard is a shared device, which means that a process needs to get mutual exclusive access before being allowed to read characters from the keyboard.

  In a typical machine control application, only one process will attempt reading from the keyboard at the same time, because if multiple processes compete with each other, the operating system will unpredictably choose one process to get access first, which means that the input typed by the user goes to an unpredictable process, which is typically unwanted behaviour.

  Also, a human sitting behind the keyboard is extremely slow compared to the time needed to get mutual exclusive access from the central master object, so the latter time interval can be neglected.

- *Access to the screen.* Just as the keyboard, access to the screen must also be requested before writing anything to the screen.

  First of all, if many processes would attempt to write to the screen more or less at the same time, then the screen would display the output faster than a human would be able to read (which after all is the main purpose of printing something to the

screen). Also, the process of actually making text appear on the screen by accessing the screen driver takes more time than getting mutual exclusive access.

In the last two cases, it should be noted that in general, a machine control application has a single keyboard and a single screen.[7]. That means that no matter how many processes want to use them, they will always have to use the same hardware. By locating the master object in the same machine, the overhead to get mutual exclusive access becomes negligible compared to the actual I/O operation.

Last but not least, it is not wise to assume that devices are infinitely fast. A modeller should assume that devices such as the screen or the keyboard may be slow, and ensure that it does not affect the normal control cycle, for example by temporarily buffering output.

With the above functionality and design considerations in mind, the class definition of the master object shown in Figure 3.4 will not contain surprises. For each function of the master object, some methods exist. Upon construction of the object, a default process priority and stack-size is given. The values of these parameters are retrieved by each process through the GetPriority() and GetStackSize() methods. Also, all process objects register and unregister themselves through the Register() respectively Unregister() methods, so they are known by the master object. This knowledge is used by the environment during execution of the StartTasks() and Shutdown() methods, which start respectively stop the application. Finally, the Claim...() and Release...() method pairs coordinate (mutual exclusive) access to standard input and standard output (the keyboard, respectively the screen).

Process base class

The run-time support layer provides common services, used by process objects in the χ implementation layer. These common services exist in the form of methods in the process base class CVxWorksProc, shown in Figure 3.5. In the constructor, the name of the process, the address of the global master object and positioning information of its process definition in the χ specification (used when a fatal error is reported to the user) are stored. The Claim...() and Release...() methods are used to get access to standard input and output (the call is forwarded to the central master object).

From outside the process, there are two types of objects which can enter this process object, the master object, and channel objects. The Start() and Shutdown() methods are entry points for the master object to start, respectively stop the task associated with this process object. Also, channel objects forward calls from the other process. The IncomingComm() method is used when a remote process wants to communicate with this object, and the Answer() method is used to deliver the final answer to this process (see Chapter 4 about communication for more information).

In the protected part of the base class are methods used by the local (derived) process object. The InitFunction() is optional, it performs initialisation before the normal control cycle is started. The virtual Run() method implements the behaviour of the process in a derived process class. The next three methods (Sleep(), Terminate(), and CurTime()) also

[7]At least, in our experiments.

```
typedef set<CVxWorksProc*, less<CVxWorksProc*> > CProcSet;

class CVxWorksMaster
{
public:
    CVxWorksMaster(int iPrio, int iStksize);
    virtual ~CVxWorksMaster();


    // Starting and stopping of tasks:
    void StartTasks();
    bool Shutdown();

public:
    // Request task information
    int GetPriority(const string& sProcname);
    int GetStackSize(const string& sProcname);

    // Task registration
    void Register(CVxWorksProc *pP);
    void Unregister(CVxWorksProc *pP);

    void ClaimInput();       // Request and release mutual
    void ReleaseInput();     // exclusive access to the keyboard
    void ClaimOutput();      // Request and release mutual
    void ReleaseOutput();    // exclusive access to the screen
private:
    CProcSet cProcesses;     // Set of processes

    int iPriority;           // Default priority of a process
    int iStack;              // Default stack size of a process

    CMutexSem cReportSem,
              cRegisterSem,  // Used for registering processes
              cInputSem,
              cOutputSem;    // Used for cin/cout mutual exclusion
};
```

Figure 3.4: Master-object class definition.

```
class CVxWorksProc
{
public:
    CVxWorksProc(const string& sProcName, CVxWorksMaster *master,
                                                const string& sPos);
    virtual ~CVxWorksProc();

    void ClaimInput();
    void ReleaseInput();
    void ClaimOutput();
    void ReleaseOutput();

    // Called from pMaster
    void Start();                       // Start the VxWorks task
    bool Shutdown(int iNotThisTask);  // Kill the VxWorks task

    CComAnswer IncomingComm(CSync *pChannel);
    void Answer(CComAnswer cAns);
protected:
    virtual void InitFunction();      // Perform initialisation
    virtual void Run()=0;             // Start execution

    void Sleep(CTime cT);             // Delay for the indicated period
    void Terminate();                 // Shutdown everything
    static CTime CurTime();           // Request time from the system

    CComProto cComProto;              // The communication protocol
private:
    string sPosition, sProcName;
    CVxWorksMaster *pMaster;

    int iTaskId;                      // Task id of the VxWorks task
};
```

Figure 3.5: Process-object base-class definition.

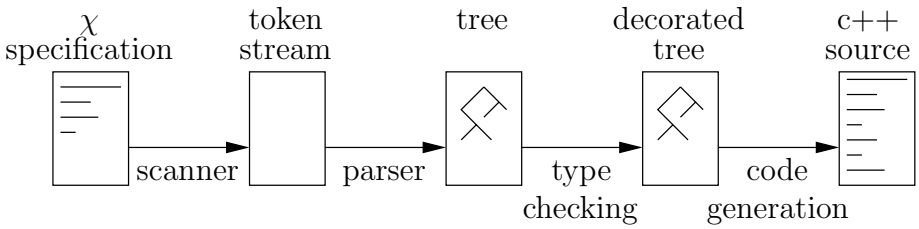

Figure 3.6: Overview of the phases in the χ compilation process.

provide services to the derived class. Finally, the cComProto object implements primitives for the selective waiting and sequential communication statements.

3.5 χ implementation

The remaining part is the implementation of the behaviour as described by the χ specification. This part is different for each specification. The implementation code is generated by a program called the χ *compiler*. This compiler takes the specification, and translates it to equivalent C++ code. The generated code uses the services provided by the run-time support part, for example by deriving concrete process classes from the process base class.

By compiling the generated code, and linking it with the pre-compiled files from the run-time support part, an implementation with the same behaviour as specified in χ becomes available[8]. The implementation can then be downloaded and started, to control the real machine.

The χ compiler

As explained above, the χ compiler translates a χ specification to equivalent C++ code that can be used on the target system. To the developer of a machine control application, the translation from a specification to an implementation on the real-time platform is an auto-magic[9] step.

To this project however, the compiler is crucial in making the horizontal design step possible for our users. On the other hand, from a compiler-construction point-of-view, the compilation process from a χ specification to C++ code is quite standard. As shown in Figure 3.6, the compilation process consists of several standard phases. In the first phase, the *scanner* converts the text of the χ specification to a token stream, which is read by the *parser* to construct a parse tree. The *type-checking* phase decorates the tree with extra information (for example, data-type information is added in the expression nodes). The

[8] With the changed semantics as explained in the second chapter.

[9] An *auto-magic* process is an automatically executed process, which the author does not want to explain in detail, because it is too ugly, too complicated, or just too simple [Ray96].

last phase is *code generation*, which generates a C++ source from the decorated tree to match the environment expected by the run-time support part.

Type checking

The most interesting phase in the compiler is the type-checking phase. This phase checks the entire specification for semantical errors, and decorates the parse tree for the code-generation phase. The type-checking phase checks definitions specified in the source in the following order:

1. Constant definitions,
2. Type definitions,
3. Function definitions,
4. Process definitions, and finally
5. System and xper definitions.

In other words, first all constant definitions are checked, then all type definitions, etc. The reason for this order is that χ is considered to be a mathematical language to describe industrial systems, rather than a programming language. Therefore, the order in which definitions in the specification are listed has no influence on their interpretation. By checking in the above order, the definition of elements is checked before they are used. For example, function definitions are checked before process definitions, because the latter may use function applications in their definition.

Throughout the type-checking phase, the χ type system is implicitly used. This system defines what data type an expression in χ has. The language uses the following conventions for data types

- It is a statically typed language, all types are decided during compile time.
- The language uses structural type equivalence, two data types are the same when their structure is the same.
- Expressions may not have a polymorphic data type at the end of the compilation.
- Both polymorphism and overloading are allowed in the language.
- Last but not least, data types are not related to each other. For example, the data type of natural numbers is not related to the data type of integers. A consequence of separating data types is that the language does not use casting. For example, the literal list $[1.0, 2]$ would be correct if the value 2 would be casted to its equivalent real value 2.0. χ however does not perform casting, and considers the list semantically incorrect. The user may however use conversion functions for changing the type of data.

The piece of software that derives a type of an expression in its context is at the core of the type-checking phase.

The basic type-checking algorithm used in previous versions of the χ compiler is based upon [Mil78]. This algorithm is capable of handling polymorphism (also known as *generic polymorphism*) while deciding the data type of expressions. Meanwhile, allowing overloading (also known as *ad-hoc polymorphism*) of library functions had become desirable in χ. Therefore, a new checking algorithm for deriving types in expressions was needed. Unfortunately, this was not as easy as it sounds. Apparently, the combination of no casting, but with use of both generic and ad-hoc polymorphism was sufficiently non-standard to belong to the set of problems for which no off-the-shelf solution exists. Therefore, a solution had to be devised. Cardelli [Car87] describes a general approach to solve a set of type equations with the presence of polymorphism. Extending this idea with overloading, and having it solved by a computer seemed a feasible option. Also, [DdlBS99] proved that it could be done in Prolog. The article describes an algorithm to generate a set of Prolog typing rules which (after solving by a Prolog interpreter) defined the type of a Prolog expression.

Defining the data type resolving problem precisely, and designing an algorithm to solve the type equations was the next step. Ideally, after defining the algorithm, some properties should be proven about it, for example correctness and termination properties. However, since time was limited, proving properties could take a long time, and some confidence was built with the algorithm while it was being constructed, it was decided to take an engineering approach instead. Rather than formally proving the properties, the algorithm was tested on a few tricky examples. Also, by reasoning about the mechanism of the algorithm it was deduced that key-properties like termination would very likely be correct. With this knowledge, the algorithm was implemented and then tested again to verify the results as well as the implementation. The tests gave satisfactory results. Also, after integrating the algorithm into the compiler, users found the data-type checking of the χ compiler accurate. This means that errors reported by the compiler are indeed errors in the specification, and constructs considered to be correct by the compiler are not found to be faulty. In the latter case, it should be noted that the type checking done by the C++ compiler on the generated code serves as an extra check on the correctness of the generated code. In this respect, it is convenient to have users that do not understand C++. Their first reaction when confronted with a C++ error is to report it rather than trying to fix it.

Users also found the algorithm to be slow in some cases. One user even believed that the compiler sometimes got stuck in an endless loop. Investigation of this behaviour revealed that this behaviour was a property of the algorithm. An optimization to prevent this problem has been developed and implemented. The precise description of the type-checking problem, the devised algorithm to solve the type equations, and the changes made to the algorithm to ensure proper performance are described in more detail in Appendix A.

Code generation

The translator tool performs the translation of χ constructs to C++ code. For each language construct in Chapter 2, a translation has been designed and implemented. In

this section, the most interesting translations are discussed.

- *Type definitions* are completely eliminated. The identifier is replaced by the data type that it represents. This is allowed because χ uses structural type equivalence.

- Each χ *data type* is translated to a C++ data type. Basic data types such as *bool*, *nat*, and *string* have a native C++ equivalent. The χ types *nat* and *int* are both translated to the same int type to prevent conversions as defined in the C++ language.

  Constructed data types are translated by instantiating a template. Except for tuples, all constructed data types have a single template class. For example, the list data type $real^*$ is translated to CList⟨double⟩[10].

  For tuples $T_0 \times T_1 \times \ldots \times T_m$, a single template class is not enough because m is variable between different tuples, and the length of an argument list of C++ template classes is fixed. For this reason, the template class itself is also generated. Generating template classes may create a consistency problem when using tuple data types in libraries, since the latter are compiled at a different time compared to the compilation of a χ specification. At this moment, the problem is non-existent since no library function uses tuples. In the future, the χ compiler will be modified to generate tuple template classes on demand to assist the developer to prevent such consistency problems.

- *Constants* are implemented by replacing the constant identifier with its expression.

- *Functions* are compiled to C++ functions.

- The *guarded statement* is implemented with if-then-else statements. This implementation is correct with respect to the semantics because the χ language allows all decision policies.

- The *terminate statement* indirectly calls the CVxWorksMaster::Terminate() method which shuts the entire control application down.

- *Communication statements* are implemented as selective waiting statements with a single alternative with a true guard and no statements to execute after the communication succeeds.

- The *delta statement* is implemented as a call to a delay function available in the *VxWorks* libraries. The argument of the delta statement is interpreted as the number of seconds to wait. Because the delay function takes an integer number of clock ticks, a conversion of the argument to the number of clock ticks is performed. That means that the difference in waiting time between the specified argument and the actual number of clock ticks may be half a clock tick. For very small values of the argument, this difference may become significant. However, the delta statement is normally used

[10] This C++ notation means that CList is a templated class using the type double as its argument.

to detect time-out, and small values for the argument are not common with this type of use.

- In the language semantics, the *(repetitive) selective waiting statement* uses a non-deterministic choice between alternatives. In the implementation, it is attempted to be as cheap in execution as possible. For this reason, $\Delta 0$ (wait for 0 time units) is given priority over communication.

  If communication should be performed (because there is no $\Delta 0$), verifying whether a communication partner exists is assumed to be infinitely fast. If no partner exists, the process executing the selective waiting statement blocks until the delay expires or until a partner announces itself.

  The decision to assume infinite speed in determining whether communication can take place is taken because the statement behaves closer to the semantics in this way. A user has the guarantee that at least all communication options are tried. The assumption is not unreasonable given the fact that a selective waiting statement normally has only a few alternatives, testing is relatively fast (a subroutine call in a single processor environment), and time-out values are normally quite long because they are used for detecting failure to respond.

- The *interleaving semantics* of χ is not preserved. The semantics defines that a χ statement is entirely executed in a single step. In the implementation, a single χ statement is typically translated to 10–20 C++ statements[11]. A single C++ statement is in its turn translated to multiple machine-language statements. Scheduling of statements is done by the *VxWorks* scheduler which uses pre-emptive scheduling. Therefore, switching of tasks while in the middle of executing a single χ statement is likely to happen. However, the changing part of the state of a task is not externally visible. Semaphores are used to take care of presenting a correct and consistent state to the outside world, for example, while communicating over a channel.

- Also not preserved is the *timeless-nes of executing statements*. As explained in the second chapter, this is wanted behaviour. However, it should be noted that referring to the current time in a statement may behave in unexpected ways. For example $[\ \tau = \tau \longrightarrow \text{skip}\ ]$[12] may fail because the value of the clock may change while the guard expression is evaluated.

- *Synchronous communication* is not really synchronous. There is always one side that concludes first that communication should take place (details of how this decision is

[11] Roughly measured in the simulation tool.

[12] This is a guarded statement with one alternative. The τ refers to the current time. The χ semantics specifies that this statement is executed at a single point in time, therefore the boolean guard always evaluates to *true* and the guarded statement can always choose the alternative. In the implementation, time progresses while evaluating the guard, therefore the current time may change between the two reads, causing the guard to evaluate to *false*. The latter results in having a guarded statement without a selectable alternative, which is a violation of the semantics.

reached, is discussed in the next chapter).

Data transfer over the channel takes place after reaching an agreement to communicate. The actual transfer is synchronized by semaphores. First the receiver supplies an address where the value should be written into, then the sender computes the value, writes it, and synchronises back to the receiver that the data transfer has been performed. Note that transfer using pointers is better than transferring the actual value from sender to receiver, because it saves one copy-operation of the (large) value.

Unfortunately, in an implementation of guarded statements and selective waiting statements, a choice must be made which rule to use in selecting an alternative. In principle, every choice is wrong. Whatever rule is used, users can always use their experience with the implementation to tailor their models. Therefore, users should be educated in not falling into this trap rather than trying to outsmart a user in an implementation.

3.6 Future extensions

The current implementation of the target system and the χ compiler is quite complete. The most notable exceptions are reading and writing to file (due to lack of a file system on the target), and random numbers. This functionality does however fit in the design of the target system in the sense that the design allows inclusion of these features.

Also, the machine control application is currently limited to computer systems with a single processor. A quite likely extension is distribution of the control application to several machines, connected to each other by a network. χ supports this extension already with its notion of parallel executing processes. The design of the target system is also prepared for this extension by not having a central engine. Instead, everything is decided as locally as possible by the processes themselves. This allows for a scalable system when extending to distributed control systems.

The target system is also prepared for extension of the research towards real-time behaviour. At the moment, χ considers each process equally important. Therefore, each task gets the same priority in the target system. In the future, some tasks in a controller may be considered more important than other tasks. For example, a process that monitors the emergency switch may need more attention than a task that calculates optimal schedules. By assigning different priorities to different processes, these ideas can be tested. To implement this extension, only the registration procedure of process objects will need to be extended.

CHAPTER

FOUR

Synchronous communication

The χ language uses synchronous communication to transfer information between processes. In the implementation however, both the computer hardware and the real-time operating system only provide asynchronous communication primitives. Therefore, a conversion from synchronous communication to asynchronous communication has to be performed as part of the horizontal design step. This conversion can be done in three ways:

- The user converts the χ specification to an equivalent specification that uses asynchronous communication.
- The χ compiler performs the conversion.
- The run-time support layer implements a synchronous communication service using asynchronous communication primitives.

The big advantage of the first solution is that the user has full control over the use of communication. That allows for tailoring of the translation to the specific circumstances known to exist in the specification. For example, buffer processes in the specification may be eliminated, because the communication channel may be used as buffer. The downside of the solution is that the developer has to ensure that the model remains correct. Also, to exploit the advantages offered by the solution, the developer needs to have a good understanding of the use of asynchronous communication. In some early case studies [Wie98, Kam99], this appeared to be a major obstacle for our users.

To make the horizontal design step feasible for our users, a simpler solution is needed. The second and third solutions for the conversion are much better in that respect. They shift the burden of the conversion from the developer to the tools. In fact, with these solutions, the developer does not need to know about the conversion. The disadvantage of being less flexible in the use of communication is a cheap price to pay for the increase in usability. Deciding whether to use the second or the third solution is mainly a matter of deciding where the conversion takes place. The second solution is slightly more flexible in the sense that the compiler may be able to take advantage of generating more optimal code in a number of cases, at the expense of a more complex compiler. Since generating optimal code is not the first priority in this project, the third solution of including support for synchronous communication in the run-time support layer has been chosen.

The next step in implementing the communication service is to decide how the service should be implemented. One approach is to make a single object in the target system responsible for combining communication requests. Each task reports its progress with respect to communication to this central object, and queries whether communication can take place. Another approach is trying to localize communication as much as possible. Rather than having a central object, the tasks themselves exchange information with each other, until they reach agreement.

The centralized approach with a single object is simple to implement, but it lacks scalability. Also, distribution of the control system across several computer systems may become more difficult due to communication bandwidth requirements between the computers. As a result, the centralized approach has been rejected in the project.

For the implementation of the distributed synchronous communication service, several algorithms exist in the literature. The article of Buckley and Silberschatz [BS83] gives an overview of previous solutions and provides criteria for choosing an algorithm:

- As few processes as possible should be involved in reaching a decision on whether or not to communicate.
- Processes should need as little information as possible to reach a decision.
- With the assumption that processes execute at a non-zero rate, and matching communication possibilities exist, the decision to communicate should be made within a certain time period.
- Processes should communicate as few messages as possible.

These criteria look sound and were adopted as criteria for comparing solutions with each other. It should be noted that that the last criterion currently has no big impact in the implementation. Our target environment is a single-processor system with multiple tasks in a shared-memory environment. Sending a message in this system means performing a subroutine call, which is very fast. However, if the system is extended to distributed control (that is, the machine control application becomes distributed over multiple computer systems), the last criterion will become very important. For this reason, the last criterion is also used in the decision of which algorithm to use.

All previous algorithms reviewed by Buckley and Silberschatz fail to meet one or more of the above criteria. Their own algorithm does adhere to the criteria, but is found to be incorrect in [KS97]. Luckily, in the meantime, Bagrodia [Bag89] has devised another, slightly simpler and more efficient algorithm which also complies with the above criteria. This algorithm has therefore been chosen to serve as a base for resolving communication in the target system.

In the next section, the algorithm of Bagrodia is summarized. Section 4.2 continues with an explanation of the extensions added to the core algorithm to make it usable for the implementation of synchronous communication in χ. Finally, the implementation of the adapted algorithm and the verification of it is discussed.

4.1 Bagrodia

This section summarizes the distributed synchronous point-to-point communication algorithm by Bagrodia [Bag89]. In particular, the more formal parts of the algorithm are not discussed here.

In a group of asynchronously executing processes, each process has a *unique process identifier* p_i. Each process does not terminate, and is either *active* or *idle*. In the active state, the process is busy performing calculations, and is not willing to communicate. After some time, the process needs to exchange information with another process. At that moment, the process switches from active state to idle state. In the idle state, the process tries to communicate with another process. After it succeeds in communicating with another process, the process goes back to active state, performs calculations, etc.

Communication between two process p_i and p_j is called an *interaction*, and is written as (p_i, p_j). All interactions which may take place (where both processes are idle), are said to be *enabled*. Interactions which are not enabled are considered to be *disabled*. Since a single process can only commit to a single interaction each time, enabled interactions which 'use' the same process cannot happen at the same time. These interactions are *in conflict* with each other. As an example, interactions (p_1, p_2) and (p_1, p_3) between three different processes p_1, p_2, and p_3 are in conflict with each other, because process p_1 can only participate in one of the interactions.

The distributed algorithm of Bagrodia has the following properties:

- *Safety* property: Processes do not (simultaneously) commit to interactions that conflict with each other.
- *Liveness* properties:
  - If p_i commits to interaction (p_i, p_j), p_j has either committed to the interaction or will eventually do so.
  - Every enabled interaction is eventually disabled.

In the article, a proof is given for these properties.

The algorithm

For an interaction (p_i, p_j) to take place, both p_i and p_j have to commit to the interaction. Committing to an interaction can only be done after both processes have successfully concluded a negotiation process. The negotiation process can only be started by the processes that 'owns' the interaction. Ownership of the interaction is defined as ownership of a token associated with the interaction.

In other words, for every interaction, there is a token. A token t, associated with interaction (p_i, p_j), is either owned by process p_i or by process p_j. The process owning the token can initiate the negotiation process for the interaction. During the negotiation

process, ownership of token may change. That means that the next time the interaction is attempted, the other process starts the negotiation process.

The algorithm uses tokens rather than interactions in the decision process. It requires that tokens are fully ordered. In other words, for each pair of tokens t_i and t_j, the relation $t_i < t_j$ is properly defined. The article simplifies the proofs by making interactions and tokens the same. For example, the token associated with interaction (p_i, p_j) is written as (p_i, p_j). At the start, each token is given to one of the processes involved in the interaction.

A process P_1 initiates communication by sending the smallest token possessed by it to the process indicated by the associated interaction. For example, if the smallest token possessed by process P_1 is (p_1, p_2), then this token is sent to process P_2. If the receiving process is idle, it responds to this request with a **commit** message. In that case, the token is sent back, and communication is considered successful. If the process does not want to communicate, it responds with a **refuse** message. The token is captured by the receiving process and stored for future use. Upon receiving the refuse message, the initiating process P_1 again selects the smallest token, and attempts again to communicate, etc.

Unfortunately, due to the asynchronous character of exchanging messages, it is possible that after the initiating process P_1 has sent the request to P_2, but before it receives an answer, another token is received by P_1 from a third process P_3. Reception of this token gives process P_1 a choice. If process P_2 refuses the request, it can acknowledge the request from P_3, and vice versa. However, P_1 can only decide how to answer to P_3 after it has received an answer from P_2. Waiting for an answer is not an option, because P_2 may be busy negotiating with P_3, thus creating deadlock. To prevent deadlock, the algorithm uses a smaller/bigger rule. If the incoming request from the third process has a token bigger than the token being used by the process itself, the incoming request is refused. Else, a **delay** message is returned which means as much as 'please hold'. In the example where process P_1 is busy with token (p_1, p_2), and process P_3 requests communication with the former process using token (p_1, p_3), the first process will deny the request of P_3 if $(p_1, p_3) > (p_1, p_2)$. Otherwise, it will send a delay message to P_3. Once the process gets a commit or a refuse answer from its own request, it sends a final refuse respectively commit message to the process that was delayed.

With the bigger/smaller rule, deadlock cannot happen. Also, sending a refuse message to a process with a bigger token does not mean elimination of the possibility of communicating with that process, because sending a refuse message also means capturing the token. That means that the process sending the refusal gets the ability to initiate communication with the same process. In the example, if P_1 would refuse communication with P_3, it captures the token (p_1, p_3). Then, if P_2 returns a refuse message to P_1, the latter tries to communicate with a different process. It takes the smallest token owned by it for example (p_1, p_3), and sends a request to communicate.

There are two cases not discussed yet that may occur while executing the algorithm. The first case is that more processes may attempt to communicate with P_1 after P_3 has been delayed. Those requests are always refused. The second case is that a process attempting to communicate finds all its potential partners busy. With each request, the process looses

a token, until it runs out of tokens. At that point, the process cannot initiate requests to communicate anymore and has to wait until it gets an incoming request, which it then acknowledges.

4.2 Communication in χ

The algorithm of Bagrodia discussed above provides a solid base for deciding which χ processes will communicate with each other. Before it can be applied in the communication service in the target system, the handling of guards, channels, and data transfer must be addressed. Also, an interface to the χ implementation layer is needed. In the discussion below, only the selective waiting statement is taken into account. The sequential communication statement is treated as a selective waiting statement with a single alternative. During the discussion, two deviations with respect to the semantics are introduced. These are discussed in more detail at the end of this section.

Starting with the interface for the selective waiting statement, the user considers the selective waiting statement as a choice mechanism between one or more communication and/or delta alternatives, each with its own guard. The interface to this mechanism in the run-time support layer uses the following five steps to realize the behaviour:

1. Initialise data structures for storing the alternatives.
2. Add all useful alternatives for which the guard evaluates to true.
3. Make a choice from the given alternatives.
4. Execute the code associated with the choice if necessary. For example, perform the data transfer of the communicated value.
5. Continue execution of statements at the correct point in the program.

If at least one communication alternative exists during execution of the third step, the algorithm of Bagrodia is used to decide which alternative is chosen. Outside the third step, the task is considered to be in active state, it is busy calculating. Other processes that attempt to communicate with it at that time, are refused.

The second, third and fourth steps are explained in more detail below. 'Adding a useful alternative' means that all alternatives are added, except alternatives that cannot be chosen and duplicate alternatives. Alternatives that cannot be chosen are delta alternatives with a timeout value larger than the smallest value. Duplicate alternatives are delta alternatives with a timeout value equal to the smallest value, and multiple communication alternatives that attempt to communicate over the same channel except one. For example, in the following selective waiting statement

$$
\begin{array}{ll}
[& \text{true}\,;\ \Delta 2 \ \longrightarrow\ \ldots \\
[] & \text{true}\,;\ \Delta 1 \ \longrightarrow\ \ldots \\
[] & \text{true}\,;\ \Delta 1 \ \longrightarrow\ \ldots
\end{array}
$$

$$
\begin{array}{l}
[\!] \ \text{true};\ s\,!\,1 \ \longrightarrow \ \ldots \\
[\!] \ \text{true};\ s\,!\,2 \ \longrightarrow \ \ldots \\
[\!] \ \text{true};\ c\,!\,3 \ \longrightarrow \ \ldots \\
]
\end{array}
$$

with s and c both ports to communicate natural numbers, the first alternative is eliminated because it is never chosen. Either the second or third alternative is kept, while the other is considered a duplicate alternative, and thrown away. Likewise, either the fourth or fifth alternative is stored and the other is thrown out since both alternatives use the same port. The last alternative is stored because it uses a different port. Step three thus makes a choice between waiting one time unit, communicating over port s, or communicating over port c. The current implementation throws away all duplicates, except the first one.[1]

In the third step, a final selection is made between the following four cases, based on the available delta and communication alternatives from the second step:

3a The smallest timeout value of the delta alternative is less than 0. In this case, the statement violates the χ semantics. An error is reported and execution of the program is aborted.

3b The smallest timeout value of the delta alternative is 0. The χ semantics leave freedom to either communicate or to select the delta alternative. Since the latter is cheaper in execution, it is chosen rather than attempting to communicate.

3c There is neither a delta alternative nor a communication alternative to choose from. For the selective waiting statement, this is a violation of the semantics. Like above, an error is given and execution is aborted. For the repetitive selective waiting statement, this situation means the end of the repetition. Execution proceeds by executing the next statement.

3d Either the smallest timeout value of the delta alternative is larger than 0, or there is no delta alternative. In the latter case, the timeout value is considered to be ∞. The algorithm of Bagrodia is used to check whether it is possible to communicate. If communication is not possible within the timeout limit, the delta alternative is chosen. How the decision to choose the delta alternative is made, is explained later in this section.

This step is relatively straightforward because a lot of potential troublesome choices have already been eliminated in the second step. At this point, an alternative has been chosen. Timeout alternatives are already dealt with in the algorithm by waiting the specified time interval for communication, but when a communication alternative with data transfer has been chosen, the actual data transfer has not been done. In that case, this data transfer is performed in the fourth step. The sender writes the result of the expression evaluation

[1] The χ semantics allows any policy, therefore a different policy can be implemented without breaking the semantics.

to memory using an address supplied by the receiver. Synchronisation of these activities is ensured by semaphores. Then, execution of the next χ statement is started.

The χ semantics only demands that a non-deterministic choice is made between alternatives with a true guard in a selective waiting statement. Giving $\Delta 0$ alternatives priority over communication, throwing away duplicates and unreached delta alternatives, and using a non-fair communication-decision algorithm therefore meet the requirements of the language.

Above, it is stated that Bagrodia is used to select between communication alternatives, and a timeout. While this is true in general, some extensions to the algorithm have been silently assumed in order to simplify the explanation above. With knowledge of the global solution as discussed above, the time has come to explain the details of guards, communication channels, and the handling of timeout. Except for the treatment of timeout, the approach to add these extensions to the core Bagrodia algorithm has been based on recommendations in [Bag89].

- *Guards.* As in χ, most CSP variants use guards to enable and disable communication alternatives. As noted by Bagrodia, the algorithm is easily extended by only requesting and committing tokens for which its guard evaluates to true. Tokens for which the guard evaluates to false are never sent and are refused when another process uses them.

- *Channels.* By making tokens and interactions equivalent, Bagrodia assumes a single interaction between two processes. In χ, this assumption is not valid. The language uses channels as communication medium, and only excludes channels that connect a process with itself. In other words, in χ it is possible that two different channels connect the same pair of processes with each other. The reason for assuming a single interaction between processes is however for ease of proof only. There is no structural limitation to the connection of channels, as long as each interaction has a unique token associated with it, and tokens are fully ordered. Therefore, by associating a χ-channel with a token, channels can be added to the algorithm. Fully ordered tokens are ensured by comparing the address of the channel object associated with the token.

- *Timeout.* Bagrodia assumes that a process wanting to communicate will wait until communication is possible. In χ, delaying can be limited to some upper limit by including delta alternatives in the selective waiting statement. As a result, the algorithm of Bagrodia cannot accommodate delta alternatives without change.

  Adding timeout means that a process may go from active state to idle state to wait for communication, and then after some time autonomously decides to go back to active state. The implementation assumes that a process can decide whether or not communication is possible within the period prior to the timeout. In other words, it can try all available communication options before the timeout interval expires. Therefore, at the moment the timeout expires, the process has no tokens left and is

waiting until some other process initiates communication with it. In this situation, no harm is done by deciding autonomously to go back to active state. The process is after all not negotiating with any other process, nor has any other process indicated that it is interested in communicating with the waiting process. Thus timeout can be chosen under the following conditions: *a)* the process has no outstanding requests for which it has not received a final answer, *b)* it has not received a commit to one of its requests, and *c)* it has refused all requests coming from other processes. Under these conditions, it is safe to switch back from idle state to active state.

It should be noted that the above description deviates from the official χ semantics in two ways, aside from the difference in behaviour of time. The first deviation is made in step 3b. In this step, an alternative with timeout value 0 is given preference above communication. The reason for this decision is that a choice has to be made in an implementation, and there is no compelling reason for not choosing the cheap alternative first.

The second deviation from the semantics lies in the handling of timeout alternatives. Specifically, it is assumed that the decision to communicate with another process can be made within the timeout interval. Obviously, for small timeout periods this may not be true. However, in the application domain, timeout alternatives are used for fault detection. If, for some reason, the realized timeout period is slightly longer than specified because the decision cannot be reached in time, nothing bad happens.

4.3 Implementation

The algorithm designed above needs to be linked into the architecture of the target system. This is achieved by filling in the details of the channel objects, and the cComProto object in the base process class.

Rather than listing the C++ code here, the algorithm is explained using a more compact notation. The data in channel objects and process objects is presented in tuples. For example, the data of a channel is stored in tuple (p_0, p_1, p_{owner}). For function and procedure calls, the object-oriented notation object.method() is used. Definitions of the methods use χ notation, in combination with object-oriented conventions for access to data and methods of the object. For example, the line c.changeowner$(p) = [\![\ p_{owner} := \text{other}(p)\ ]\!]$ defines the method *changeowner* on channel c. Inside the body of the method, parameter p, the data stored in the channel tuple, and the methods associated with the channel are available without prefixing them. For example, the method *other* in the body is in fact c.other.

Below, the channel objects are described first, followed by the process objects. The reason for this order is that channel objects are easier to understand than process objects. However, since channels and processes closely work together, both should be studied at the same time in order to understand them.

Channel

Data in a channel is a tuple (p_0, p_1, p_{owner}). It contains references to its connected processes p_0 and p_1. p_{owner} is the owner of the token of the channel. Storing the owner of a channel inside the channel tuple originates from the wish to get rid of moving tokens back and forth between processes. The reason for this is that a process may be connected to many channels, which means that it requires a lot of storage to store all the tokens that it may own. Since this holds for all processes, a lot of that storage is wasted (not in the last place because a token is always associated with exactly one process). To reduce the storage requirements, the list of tokens is distributed from within the process to its connected channels in the form of an owner field.

On each channel c, the following methods are defined:

$$
\begin{aligned}
c.\text{owner}(p) &= [\![\uparrow(p = p_{owner})]\!] \\
c.\text{changeowner}(p) &= [\![p_{owner} := \text{other}(p)]\!] \\
c.\text{other}(p) &= [\![[\ p = p_1 \longrightarrow \uparrow p_0 \ [] \ p = p_0 \longrightarrow \uparrow p_1 \ ]]\!] \\
c.\text{remotecomm}(p) &= [\![\uparrow c.\text{other}(p).\text{remotecomm}(c)]\!] \\
c.\text{finalanswer}(p, a) &= [\![c.\text{other}(p).\text{finalanswer}(c, a)]\!]
\end{aligned}
$$

The first method is a consequence of the above decision to store ownership in the channel. Instead of checking whether a process owns the token, it must verify that it owns the channel using the function $c.\text{owner}(p)$, before it can issue a communication request for that channel. Another consequence of the above decision is that a channel needs to have a way to change its ownership. The method *changeowner* takes care of this functionality. The *other* method returns the process not specified as parameter. It allows a process connected to the channel to find the process connected at the other port of the channel. The two functions, $c.\text{remotecomm}(p)$ and $c.\text{finalanswer}(p, a)$ perform a forwarding service of process methods to the other process connected to the channel. (The $c.\text{other}(p)$ part returns the process connected to the channel that is not p. On that process, the method is called.)

Last but not least, both processes have access to the data in a channel, which makes the channel a shared data structure. To prevent data corruption, the channel must be claimed by a process using a semaphore before using the channel. The operations on this semaphore are written as operations on the channel itself, so '$\text{P}(c)$' means 'perform the P operation on the semaphore of channel c'.

Process

A process has a tuple $(s, dm, cm, c, dc, bs, ans)$ to store its data. This data is shared between several processes, and must be mutually exclusively accessed. Like channels, obtaining and releasing this lock is denoted as semaphore operations on the process. The fields in the tuple have the following meaning:

- State s of the process has four possible values:

  active The process is busy calculating, and not interested in communicating with another process.

  trying The process is trying to establish communication (it has sent a request, but has not yet got an answer).

  pending The process has been asked to wait, and has not received a final answer yet.

  idle The process is waiting for another process to initiate communication with it.

- Delta set dm contains information about alternatives using a delta event. The set is either empty, or contains a single element (d, i), with d the value of the time-out expression for the i^{th} alternative.

- Communication set cm contains information about communication alternatives. Each element is a tuple (c, i), with c the channel used in the i^{th} alternative. The set is ordered on channel, because the algorithm needs an in-order traversal of channels.

- Channel c is being tried for establishing communication. This data is only valid when $s \in \{\textbf{trying}, \textbf{pending}\}$.

- Delayed channel dc is a set containing at most one channel c'. Existence of this channel in the set means that another process has tried to initiate communication using channel c'.

- Semaphore bs is used for internal synchronisation between the p.doselwaiting and the p.remotecomm routines. The semaphore must be initially taken. Use of this semaphore is demonstrated in the example on Page 57.

- In the ans field, the final answer is stored for this process when it is in pending state (the process is blocked on semaphore bs at that moment, and is not able to accept the value immediately).

Besides the tuple, there are also a number of methods associated with each process p. These methods follow the five steps described in Section 4.2. Note that in the sequel, $c \in cm$ is a shorthand notation for $\exists\, x\colon (c, x) \in cm$.

$$
\begin{aligned}
p.\text{clear} &= [\![\ dm := \emptyset\,;\ cm := \emptyset\ ]\!] \\
\\
p.\text{adddelta}(d,i) &= [\![\ [\ dm = \emptyset \longrightarrow dm := \{(d,i)\} \\
&\quad [\!]\ dm = \{(d',i')\} \wedge d < d' \longrightarrow dm := \{(d,i)\} \\
&\quad [\!]\ dm = \{(d',i')\} \wedge d \geq d' \longrightarrow \mathit{skip} \\
&\quad]\ ]\!] \\
\\
p.\text{addcomm}(c,i) &= [\![\ [\ c \notin cm \longrightarrow cm := cm \cup \{(c,i)\}\ [\!]\ c \in cm \longrightarrow \mathit{skip}\ ]\ ]\!] \\
\\
p.\text{finalanswer}(c,a) &= [\![\ \mathrm{P}(p)\,;\ ans := a\,; \\
&\quad [\ ans = \textbf{ack} \longrightarrow \mathit{skip} \\
&\quad [\!]\ ans = \textbf{refuse} \longrightarrow c.\text{changeowner}(p) \\
&\quad]\ ; \mathrm{V}(bs)\,;\ \mathrm{V}(p) \\
&\quad]\!]
\end{aligned}
$$

The *clear* routine performs the first step by clearing the delta set dm and the communication set cm. The second step, adding new communication and delta alternatives to their sets is done using the *addcomm* respectively *adddelta* methods. In case of adding a delta alternative, the algorithm is only interested in one of the delta alternatives with the smallest time-out value, so the add method does not even store other alternatives. Multiple communication alternatives using the same channel are also silently deleted, except for one alternative. The *finalanswer* method is a routine to deliver a delayed answer to process p. This routine also adjusts the ownership of the channel if necessary.

For the actual handling of communication, two other methods are needed. The first and easiest method is for handling incoming communication requests from other processes:

$$
\begin{aligned}
&p.\text{remotecomm}(rc) = \\
&[\![\ \mathrm{P}(p) \\
&;\ [\ s = \textbf{active} \longrightarrow \mathrm{V}(p)\,;\ \uparrow \textbf{refuse} && \text{// line \#R1} \\
&\quad [\!]\ s = \textbf{idle} \longrightarrow [\ dc = \emptyset \wedge rc \in cm \longrightarrow dc := \{rc\}\,;\ \mathrm{V}(bs)\,;\ \mathrm{V}(p)\,;\ \uparrow \textbf{ack} && \text{// line \#R2} \\
&\qquad\qquad\quad [\!]\ dc \neq \emptyset \vee rc \notin cm \longrightarrow \mathrm{V}(p)\,;\ \uparrow \textbf{refuse} \\
&\qquad\qquad\quad] \\
&\quad [\!]\ s = \textbf{trying} \vee s = \textbf{pending} && \text{// line \#R3} \\
&\qquad \longrightarrow [\ dc = \emptyset \wedge rc \in cm \longrightarrow [\ rc > c \longrightarrow \mathrm{V}(p)\,;\ \uparrow \textbf{refuse} \\
&\qquad\qquad\qquad\qquad\qquad [\!]\ rc \leq c \longrightarrow dc := \{rc\}\,;\ \mathrm{V}(p)\,;\ \uparrow \textbf{wait} && \text{// line \#R4} \\
&\qquad\qquad\qquad\qquad\qquad] \\
&\qquad\quad [\!]\ dc \neq \emptyset \vee rc \notin cm \longrightarrow \mathrm{V}(p)\,;\ \uparrow \textbf{refuse} \\
&\qquad\quad] \\
&\quad] \\
&]\!]
\end{aligned}
$$

Depending on the activities of process p as indicated by the state s, which channel c is used by p, and whether or not another process has already requested communication (stored in dc), the request is acknowledged, refused or delayed.

The second routine is the main routine. It performs the selection between the alternatives as explained in the third step.

p.doselwaiting =
⟦ [$dm = \emptyset \longrightarrow t := \infty$ // line #S1
  ▯ $dm = \{(d, i)\} \longrightarrow t := d$
  ]
; [$t < 0 \longrightarrow$ Report fatal error
  ▯ $t = 0 \longrightarrow \uparrow$ Alternative i, $(t, i) \in dm$
  ▯ $t > 0$
    $\longrightarrow$ P(p) // line #S2
      ; [$cm = \emptyset \wedge dm = \emptyset \longrightarrow$ V(p); $\uparrow$ No alternative chosen
        ▯ $cm = \emptyset \wedge dm \neq \emptyset \longrightarrow$ V(p); Δd; $\uparrow$ Alternative i, $(t, i) \in dm$
        ▯ $cm \neq \emptyset$
          $\longrightarrow s :=$ **trying**; $dc := \emptyset$ // line #S3
            ; ∗[$c \in cm$ // Note: c iterates over set *cm*
                 $\longrightarrow$ V(p); P(c) // line #S4
                   ; [¬c.owner(p) $\longrightarrow$ V(c)
                     ▯ c.owner(p)
                       $\longrightarrow a :=$ c.remotecomm(p) // line #S5
                         ; [$a =$ **ack** $\longrightarrow$ V(c); P(p); $s :=$ **active** // line #S6
                                 ; [$dc = \emptyset \longrightarrow$ *skip*
                                   ▯ $dc = \{c'\}$ // line #S7
                                     $\longrightarrow$ P(c'); c'.finalanswer(p, **refuse**); V(c')
                                       ; $dc := \emptyset$
                                   ]
                                 ; V(p); $\uparrow$ Alternative i, $(c, i) \in cm$ // line #S8
                           ▯ $a =$ **refuse** $\longrightarrow$ c.changeowner(p); V(c) // line #S9
                           ▯ $a =$ **wait** // line #S10
                             $\longrightarrow$ P(p); $s :=$ **pending**; V(c); V(p)
                               ; P(bs); P(p) // line #S11
                               ; [$ans =$ **ack**
                                   $\longrightarrow s :=$ **active**
                                     ; [$dc = \emptyset \longrightarrow$ *skip*
                                       ▯ $dc = \{c'\}$
                                         $\longrightarrow$ P(c'); c'.finalanswer(p, **refuse**); V(c')
                                           ; $dc := \emptyset$
                                       ]
                                     ; V(p); $\uparrow$ Alternative i, $(c, i) \in cm$
                                 ▯ $ans =$ **refuse** // line #S12

```
                                        ⟶ s := trying; V(p)
                                    ]
                                ]                                          // end a = wait
                            ]                                              // end c.owner(p)
                        ; P(p); [ dc = ∅     ⟶ skip                        // line #SE
                                [] dc = {c'} ⟶ s := active; dc := ∅; V(p)
                                             ; P(c'); c'.finalanswer(p, ack)
                                             ; V(c'); ↑Alternative i,  (c', i) ∈ cm
                                ]
                    ]                                                      // end repetition c ∈ cm
                ; s := idle; V(p); P(bs, Δt); P(p); s := active            // line #SI
                ; [ dc = ∅     ⟶ V(p); ↑Alternative i,  (t, i) ∈ dm
                  [] dc = {c'} ⟶ dc := ∅; V(p); ↑Alternative i,  (c', i) ∈ cm
                  ]
            ]                                                              // end cm ≠ ∅
        ]                                                                  // end t > 0
]
```

The above method uses the steps *3a* through *3d* to perform the selection. Note that inside the repetitive statement that iterates over the channels in cm, the process is released and claimed again to give other processes a chance to request communication through the p.remotecomm() method. When the process runs out of channels to try, it ends the iteration, sets its state to **idle**, and performs $\mathrm{P}(bs, \Delta t)$ operation. This is a special P operation on semaphore bs with an upper time limit Δt. This primitive is provided by VxWorks.

Example

As an example of how the implemented communication algorithm works, consider the following χ specification:

$$
\begin{aligned}
\textsf{proc}\ P(c{:}\sim \text{void}) &= [\![\ c\sim\ ]\!] \\
&= \\
\textsf{proc}\ X(c, d{:}\sim \text{void}) &= [\![\ [\ c\sim [\!]\ d\sim\ ]\ ]\!] \\
&= \\
\textsf{syst}\ S() &= [\![\ c, d{:}\ -\text{void} \mid P(c) \parallel P(d) \parallel X(c, d)\ ]\!] \\
&= \\
\textsf{xper} &= [\![\ S()\ ]\!]
\end{aligned}
$$

In this specification, two processes P attempt to communicate with process X. The χ semantics specify that synchronization will take place, either using channel c or using channel d. After translation of the specification, downloading the result, and instantiating the machine control application, the process objects and channel objects as shown in Figure 4.1. In the figure, three big boxes and two small boxes are connected with each

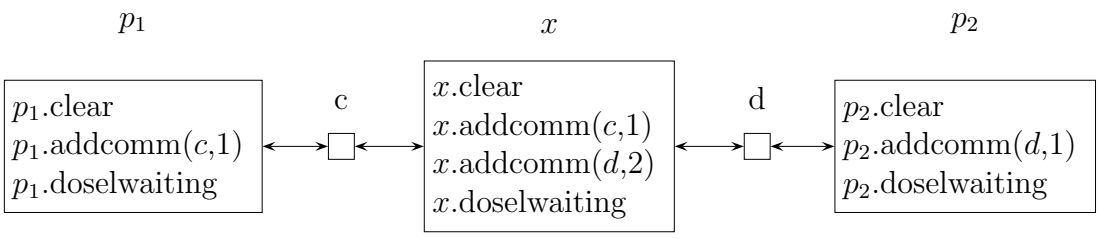

Figure 4.1: Translated χ communication example.

process object p_1	channel object c	process object x	channel object d	process object p_2
p_1.s: **active**		x.s: **active**		p_2.s: **active**
p_1.dm: $\emptyset$		x.dm: $\emptyset$		p_2.dm: $\emptyset$
p_1.cm: $\{(c,1)\}$		x.cm: $\{(c,1), (d,2)\}$		p_2.cm: $\{(c,1)\}$
p_1.bs: *taken*	c.owner: p_1	x.bs: *taken*	d.owner: p_2	p_2.bs: *taken*
p_1: *free*	c: *free*	x: *free*	d: *free*	p_2: *free*

Table 4.1: Data values of the process and channel objects just before attempting to communicate.

other. The big boxes represent process objects. At the left, one instantiated process P is shown. For reference purposes, this object is called p_1. In the middle, the instantiated process object x is shown, and at the right, the second instantiation of P is displayed. The latter is called p_2. Inside the process objects, the generated code for handling the communication is listed. The small boxes between the process objects represent channel objects. Between p_1 and x is channel object c, and between processes x and p_2 is channel object d. Note that in reality, an xper object, a system object, and a master object also exist[2], but these are not shown to improve clarity of the figure.

Associated with each object are some data and a semaphore. These are initialized during instantiation of the objects, and with the calls to the *clear* and *addcomm* functions, as shown in the generated code in the figure. Table 4.1 shows the value of all relevant variables of the objects just prior to calling the *doselwaiting* function in each process object. The state of each process object follows directly from the execution of the *clear* and *addcomm* functions. The state s of the process is **active**, since the process has not started communicating yet; the delta map *dm* is empty, because there are no delta alternatives available; the communication map *cm* contains the alternatives added with the *addcomm* function call(s); semaphore *bs* is in its initial state; the semaphore protecting the data of the process object (that is, the variables explained above) is free for use.

The state of the channel objects only contains the current owner of the channel in

[2]As shown in Figure 3.3, on page 33.

variable *owner*. Its value is constructed during instantiation of the channel object. The other data of the channel is not shown here for clarity.

Below, one sequence of execution for the three tasks in the process objects is described. It is a fairly complex sequence that covers a lot of the code of the implementation. In particular, delaying the final answer to a process occurs. The starting point of the example is that all three process objects are about to call the *doselwaiting* function, and the values of the relevant variables are as listed in the table.

The execution is split in seven (numbered) pieces. The final result is that processes p_2 and x communicate with each other, and that process p_1 becomes blocked due to lack of a communication partner. For the purpose of demonstrating the algorithm, the scheduler that selects tasks to execute makes some unfortunate choices. This results in a number of failed attempts to communicate before being successful.

At this level of detail, it is easy to lose track of the context. Therefore, a small piece of text in italics font introduces what will happen in each piece of execution in terms of the algorithm.

1. *Process p_2 attempts to communicate with process x, but fails because x is still in active state.*
   The scheduler starts the execution in task p_2.[3] It enters the *doselwaiting* function, and sets local variable t to ∞ on line #S1, since p_2 has no delta alternatives. Execution continues on line #S2 by obtaining a lock on the local data of the process object, and concluding that it should attempt to iterate over the communication alternatives. On line #S3, it switches to the **trying** state, indicating that it is trying to communicate with other processes, and it initializes the *dc* variable. It sets the c variable to the first (and only) communication alternative. On line #S4, the process data is released for use by other processes, and a lock on the channel object is taken. Once the lock is obtained, the tasks concludes that it is the owner of the channel (d.owner $= p_2$), and it queries process x whether communication is possible on line #S5 (d.other(p_2) reduces to x, execution therefore resumes in x.remotecomm(d)). After grabbing a lock on the data of process x, it is concluded on line #R1 that process x is not willing to communicate. The lock is released, and the return value **refuse** is returned, and put into local variable a of process p_2. Execution proceeds on line #S9. The owner of channel d is changed from process p_2 to process x, and the channel data is released. The execution of the task continues with checking whether another process tried to communicate with p_2. On line #SE, the own process data is locked again (it was released on line #S4). Then, *dc* appears to be empty, and execution is continued with the next alternative. In this case, there is no next alternative, so line #SI is executed next. The state of the process is set to **idle**, meaning 'I am waiting for someone to communicate with me', the process data is released, and the process blocks on its synchronization semaphore p_2.bs.

[3] Just as with process objects, the tasks are given names that correspond with the names of the objects.

2. *Process x attempts to communicate with process p_1, but fails because x does not own channel c. The task then proceeds by attempting communication over channel d.*
   After task p_2 blocks, the scheduler selects task x for execution. The tasks also starts executing *doselwaiting*. Execution proceeds along lines #S1 and #S2 to line #S3, where the state of process object x is changed to **trying**. In this example, channel c is considered smaller than channel d, therefore, task x will first attempt to communicate using channel c. On line #S4, the semaphore of channel c is taken. Inspection of the ownership of the channel then reveals that task x is not the owner, and the channel semaphore is released again. Execution then proceeds at line #SE. Since no other process has attempted to communicate, dc is still empty, and the task goes on with channel d on line #S4. On that line, the task releases the lock on its own process data, and claims the channel data of channel object d. Since process p_2 gave ownership of the channel to x, the task finds that it owns the channel, and execution continues from line #S5 to p_2.remotecomm(d).

   However, before task x gets a chance to execute this function, it is interrupted by the scheduler, and a switch is made to task p_1.

3. *Process p_1 attempts to communicate with task x, and is told to wait for a final answer from x.*
   Task p_1 also starts with executing the *doselwaiting* function. On line #S3 it sets its state to **trying**. Channel c is selected, and since process p_1 owns it, it 'jumps' over the channel to execute x.remotecomm(c) on line #S5. It acquires the lock on the process data of x, and determines that process x is in the trying state on line #R3. The rc argument has value c, and task x is busy using channel d. Since channel c is smaller than channel d, the second alternative is chosen (line #R4). x.dc is filled with the channel used by p_1, the lock on the process data of x is released, and the return value **wait** is returned, and put into variable p_1.a on line #S5. This return value means that process x is currently not able to answer. Using the returned answer, the alternative on line #S10 is chosen. The task reclaims its own process data, sets its state to **pending**, and releases channel c (thus allowing process x to use the channel for delivering the answer). Next, it releases its own process data, and blocks on the synchronization semaphore p_1.bs until the answer is delivered.

4. *Process x is successful in establishing communication with process p_2. After returning with this result, it sends a refusal to process p_1 to indicate it is not interested.*
   With process p_1 blocked, only task x is able to proceed. The scheduler switches to this task, and continues the execution. The task owns the data of channel d, and is about to call p_2.remotecomm(d). In that function, it claims the process data of process p_2. Then it inspects the state of process p_2, and detects that it is idle. Also, p_2.dc is still empty, and process p_2 wants to communicate using channel d ($dc = \emptyset \wedge rc \in cm$). Communication is thus possible between process x and process p_2. The alternative on line #R2 is chosen, task x stores channel d in p_2.dc as a note to process p_2, it releases task p_2 by releasing semaphore p_2.bs, and it releases the process data of p_2.

Finally, it returns **ack** to indicate success. The return value is stored in x.a on line #S5.

Since the return value indicates success, the alternative on line #S6 is chosen. Process x releases the channel (channel d), and claims its own process data to check whether anything happened while it was busy establishing communication over channel d with process p_2. It finds that x.dc contains c on line #S7. Another process has apparently attempted to establish communication. It claims the data of channel c, and sends a refuse message by executing c.finalanswer(x,**refuse**). Channel c forwards the call to procedure p_1.finalanswer(c,**refuse**). In this procedure, task x claims the process data of process p_1, stores the refusal message in p_1.ans, changes the ownership of the channel, and releases p_1.bs, thus allowing task p_1 to resume execution. The procedure then ends, Execution continues after the call in x.remotecomm, directly below line #S7. Channel c is released, the delayed channel variable x.dc is cleared, the own process data is released on line #S8, and the number of the selected communication alternative (alternative number 2 in this case) is returned, which also ends the execution of the fragment of code of task x.

5. *Process p_1 runs out of communication options, and becomes blocked, waiting for a communication partner.*
   During the execution of task x, both process p_1 and process p_2 were awakened by a synchronization on the internal semaphore. The scheduler picks process p_1 first. This task was blocked on p_1.bs, waiting for a final answer from task x on line #S11. Since task x released the semaphore in the p_1.finalanswer procedure, taking the p_1.bs semaphore succeeds for task p_1. The next step is to reclaim the own process data, and inspect p_1.ans. The answer appears to be a refusal, and execution therefore continues on line #S12. After changing the state of the process back to **trying**, the task notices that no other process has attempted to communicate with it on line #SE. The process has tried all its communication options without any success. On line #SI it changes its state to **idle** and it blocks on p_1.bs indefinitely.

6. *Process p_2 notices that it has succeeded.*
   The task scheduler switches to task p_2 that became unblocked during the the execution of task x in the previous part. The task wakes up after taking the p_2.bs semaphore on line #SI. It claims its own process data, switches to the **active** state, takes the delayed channel from p_2.dc that caused the process to wake up, and returns the associated number of the alternative (1 in this case). Since the function call was the last statement in the process object, the execution of process p_2 terminates.

The final result is that processes p_2 and x agree on having synchronized with each other, while process p_1 is waiting for a communication partner.

4.4 Verification of the implementation

Resolving communication in the target system is a vital functionality. It is important that it works correctly under all conditions. In other words, the implementation as proposed in Section 4.2 should always work correctly.

Starting point of the verification is [Bag89], where safety and liveness properties of the core algorithm are proven. From the core algorithm to the implementation, the following additions have been made:

- Guards have been added,
- Channels have been added,
- Autonomous switching of tasks from idle state to active state has been added,
- Ownership of channels has been moved to the value of a shared variable in a channel object,
- Process objects have been introduced containing shared data, and
- Semaphores have been introduced to prevent corruption of shared data, and to synchronize behaviour.

The latter two items are consequences of constructing an implementation with concurrently executing tasks, using semaphores to synchronize behaviour.

Assuming that the original proofs in the article are correct, the first three additions are not difficult. In fact, the first two items are even explicitly mentioned as obvious extensions in the article. The third item is also not hard: If no other process has seen you being in idle state, autonomously switching back to active state will not influence the correctness of the algorithm. Therefore, showing correctness of the algorithm with only the first three additions is almost the same as done in the article.

The part of the implementation with the biggest risk of being incorrect is in the handling of semaphores, and in the protection of shared data. By having a single P or V operation wrong, processes may become blocked indefinitely, or data may become corrupted. *Verification of the implementation therefore means foremost showing that the handling of semaphores and the sharing of data is done correctly.*

A model checker seems suitable for this type of verification, in this project, the tool spin [Hol91, Hol97] has been used. With the choice to use a model checker, the next question is what should be checked. In Figure 4.2, five interesting configurations of χ processes are shown. The first configuration tests communication between two processes. This is the smallest configuration possible. The second configuration tests what happens when a third process can interfere, and the third configuration tests what happens when a fourth process is added. On the second line, two configurations are shown that check correct handling of cycles in communication options. Also, in these configurations two communications can be performed independently. Despite the fact that Bagrodia already

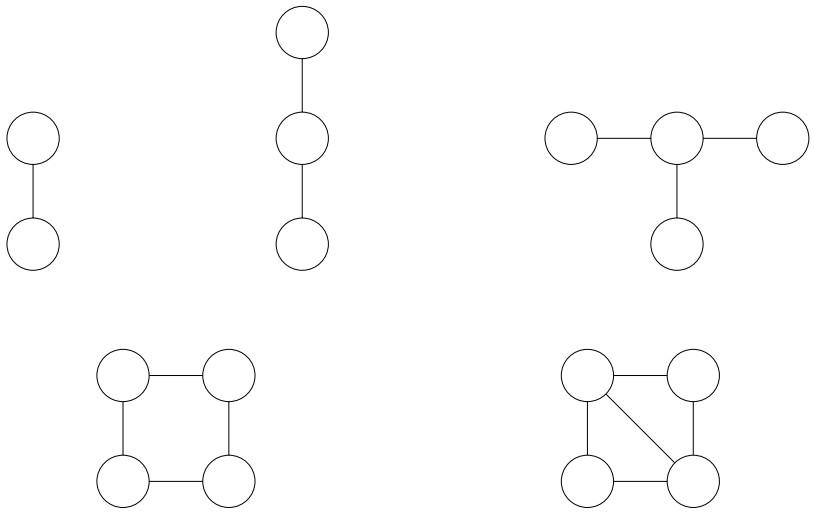

Figure 4.2: Interesting test cases for verification.

proved that his algorithm can cope with cyclic configurations, the cases have been included because they are excellent tests whether semaphores are released properly.

The configurations should handle communication between processes correctly independent of

- initial ownership of each channel,
- relative order between channels,
- activation or de-activation of communication options within each process, and
- existence of delta alternatives.

In the explanation of the algorithm, ownership of each channel was assigned to one of its processes. The implementation should work correctly, independently of what assignment was initially made. Along the same lines, an order between each pair of channels was assumed. The implementation should work in every case. The last two items check that adding guards or delta alternative do not harm the correctness.

In the above configurations, it is assumed that each process attempts to communicate once. Other interesting tests are that the implementation allows multiple sequential communications, one communication after the other over the same channel, without getting into a deadlock, or getting corrupted data.

After coding the implementation in the Promela language used by the spin model checker, actual verification of the configurations was started. Code can be found in Appendix B.2. The first configuration with two processes and one channel was performed successfully. Adding non-deterministic choices for the ownership of the channel and values for the guards gave no problems; the model checker finished that verification request successfully as well. Note that non-deterministic choices for the ownership of the channel was

not strictly necessary, due to symmetry in the configuration, all cases were already covered without it.

Trying to verify the second configuration proved to be impossible within the machine limits however, even without non-deterministic choices and maximal state-space compression. Apparently, the more complex execution patterns (with three processes, it is possible that one process is asked to wait for a final answer), and the interleaving possibilities of three processes were too much for the model checker.

Despite a number of attempts, verification of a configuration containing more than two processes has not been successfully concluded. The model checker consistently ran out of its 1GB memory before the entire state space was explored. On the positive side, it did not find errors during its execution.

Since complete verification appeared to be too much, a switch to bitstate verification was made. Results were positive, but coverage was too low to conclude anything.

Failure of bitstate verification exhausted the formal verification options of the model checker. Instead of attempting verification, the implementation was tested with over 215 million Monte Carlo simulations. Since random testing has no problems with large state spaces, a model with four processes was used. The code is shown in Appendix B.3. All simulations performed were successfully concluded. That does give some confidence, but no conclusive proof.

The final answer to the question 'Is the implementation correct?' is therefore 'I do not know'. It works in practice, and verification attempts did not find errors, but those two facts are not strong enough for a conclusive positive answer to the question. On the other hand, the large number of successfully executed random simulations do give some confidence in the correctness.

Although this project ends with a non-conclusive answer, the correctness of the implementation of the algorithm is still crucial for the system. In particular, before the system can be used in more critical systems, a positive answer to the question has to be found. Therefore, in the future, a feasible way to answer the question must be found.

Epilogue

It is interesting to investigate why the verification effort ended at this point, despite the inconclusive answer. To understand this, it is necessary to understand the goal of a designer. In general, a designer aims to deliver a solution with the highest possible quality within the limits of the project. With that goal in mind, the prospect of having a formally proven correct communication algorithm is very appealing to a designer. From that goal also follows that a designer is, to a large degree, solution driven. That is, he is primarily interested in his own particular solution. This explains why the verification effort targeted the implementation rather than, for example, some abstract version of it. In this case, the decision to target the implementation was supported by the fact that the core algorithm was already published. In that article, the existence of a number of important properties

was already established. This provided a solid foundation for the implementation, and made it feasible to investigate properties of the implementation.

The verification effort was mainly intended to proof correctness of the implementation in the sense that it would show that no errors had been made in the development steps from the core algorithm to the implementation. In this case, the formal verification needed more resources than available; three instances of the algorithm created a state space explosion within the model checker beyond machine limits. With formal verification out of reach, the steps from formal verification to bit state verification to Monte Carlo simulations are logical. All attempts try to verify as much of the implementation as possible.

Since formal verification is not possible with the implementation, a conclusive answer cannot be given. The question arises how to proceed. In this case, other verification options with a model checker do exist, but they are not interesting with respect to the goal, either because they verify something different than the implementation or because they are only possible after degrading the quality of the solution.

The reasoning to this conclusion is based on the fact that the current implementation is too big to handle for the model checker. There are two ways to solve this verification problem. Either implementation details are left out, or behaviour of the implementation is limited. The former way is the strategy to verify an abstract version of the algorithm. By abstracting from details, the state space becomes smaller and the model checker should be able to prove correctness. In this case, Bagrodia already showed that the abstract algorithm was correct. The major new addition of the implementation described in this thesis is that the algorithm is implementable in terms of semaphores. By abstracting away from the implementation, the verification work of Bagrodia would be repeated rather than taken to the next level.

The latter way of limiting the behaviour of the implementation is also not a feasible option. The idea is to take the limits of the model checker as constraints, and adjust the implementation until the model checker can cope with it. Since much of the state space explosion is caused by concurrent execution of multiple instances of the algorithm, this strategy would mean minimizing concurrency. The solution found in this way would be formally correct, but at the same time it also defeats the purpose of having a distributed algorithm. This option is therefore also not useful.

That exhausts verification approaches with a model checker. Other verification options for checking an implementation do not seem to be available in the sense that they need more work than possible in the project. For example, manual verification is possible, except that the number of proofs is too large to be feasible within the project.

The conclusion is therefore that no next steps exist to give a conclusive answer within the project. The verification effort therefore ends after performing the Monte Carlo simulations with the conclusion that there is no proof of correctness of the implementation.

CHAPTER

FIVE

Case study

As an example of how the design technique may be applied in practice, a machine named the *paint factory* was designed and constructed as a test. The case study started with a global idea of the machine and should end when the entire machine works. At the time of writing, the case study project is still under development. However, a control system for the machine has been designed and implemented using the design technique and the tools. Therefore, it seems justified to make some observations and draw some conclusions, based on the current state of the project.

Even though the design technique was used in particular for the development of the controller of the machine, other parts of the machine in the development process also seem to make vertical design steps, as will be shown below.

It should be kept in mind that the goal is to see *how simulation techniques, and in particular the design technique is used in practice*. That means that in the project, the designer was free to use the design technique in any way he considered useful. Conclusions were drawn afterwards. This way of performing the case study gives less specific information about the design technique and the tools, but more information about how a machine control design problem is solved in practice using the virtual machine concept.

5.1 Choice of the case

With proper tool support for both the vertical and the horizontal design steps, the design technique described in the first chapter can be applied by users. However, before a user can develop a machine, there must be a concrete idea about the machine to develop.

As a first requirement, in order to get meaningful results, the machine being designed in the case study should come close to machines typically used in an industrial environment. That means that the machine being designed and built should be more than a toy problem. On the other hand, since the case study is done with a limited number of people, it is not possible to develop a really large machine. A second requirement is that, while the case study should deliver a machine that demonstrates the usefulness of the approach, the machine itself should be useful as well in research and in education. In other words, designing and building the machine and then throw it away, because its existence is enough proof that it can be built, is out of the question.

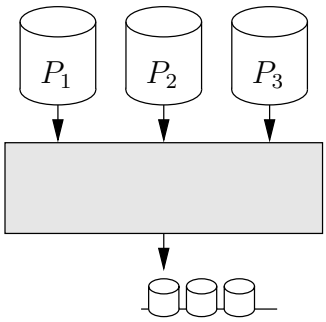

Figure 5.1: Global view of the paint factory.

The first requirement more or less rules out existing small examples, like the bottling system or elevator system in [Ove87], or the water tank system of [Are96]. Usefulness in research as well as demonstrating the usefulness of a specification language for machine control implies that the machine must have a 'difficult' problem which can be properly solved in a specification language like χ. Usefulness in research means that the machine must be in a domain which is not entirely understood yet from a modelling point of view. That requirement boils down to designing a machine typically used in the process industry. In that type of industry, a lot of research is done on how to apply χ well. In a few years, we may be able to apply those results at machine-control level. With certainty of being able to use the machine in research, use of the machine for educational purposes comes for free. If we don't completely understand the machine now, then students should be able to learn from the machine for the years to come, even if only parts of the machine are used by them.

We asked a lot of people to think of ideas for this new machine. The idea now known as *the paint factory* was chosen. The basic idea of the machine is shown in Figure 5.1. At the top are three vessels P_1, P_2, and P_3 containing predefined primary colours of paint. Customers can request small canisters of paint with any colour. The paint factory mixes the primary colours to get the requested colours, and fills the canisters with the requested colour in the correct order.

Physically, the machine has the size of a large table. Paint with different colours is 'implemented' by coloured water, vessels have a diameter of 14 cm, and a height of 18 cm, and canisters of paint are physically plastic coffee cups. Tubes between vessels have the same size as a garden hose, and are transparent to show the flow of the various colours of paint. Note that despite the small size of the machine, the parallel nature and complexity of a typical machine in the process industry has been preserved.

5.2 Highlights

Rather than describing every step of the development, only the interesting highlights of the development process are shown. Also, application of the horizontal design step is shown in

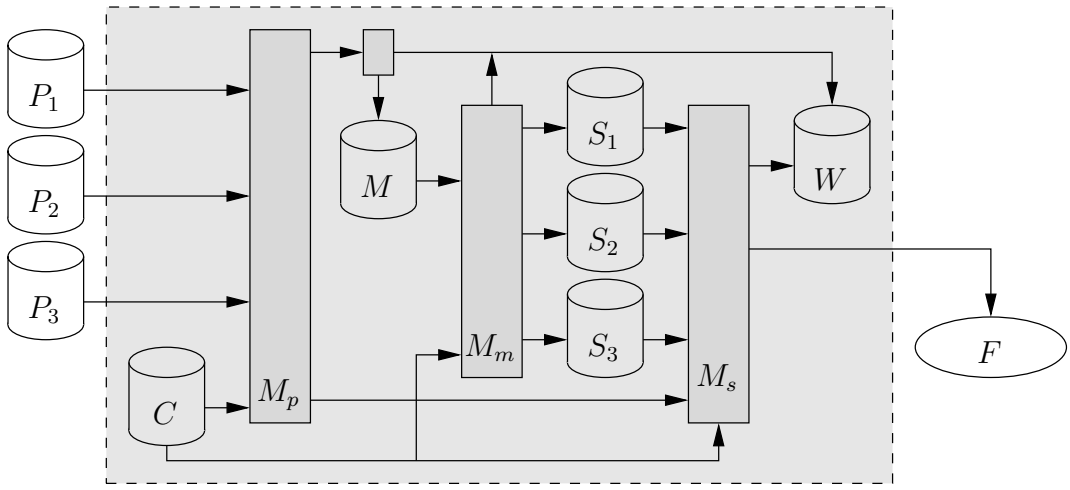

Figure 5.2: Internal structure of the paint factory.

more detail for a part of the controller. The observations and conclusions about the case study are discussed in Section 5.3 and some parts of Chapter 6.

Overview of development

From the initial idea of the machine as described above, an internal structure as depicted in Figure 5.2 was made. The light-gray area contains the internals of the paint factory. From primary vessels P_1, P_2, or P_3, paint is pumped to mixing vessel M. After mixing two or more primary colours, the paint is stored in storage vessel S_1, S_2, or S_3. At the appropriate time, the paint is then put into the canisters in the turn table in filling station F. The dark-gray areas are manifolds, which connect incoming tubes to outgoing tubes. The layout of the manifolds is not decided at this stage. To preserve the colour of the paint in the system, vessels, manifolds, and tubes, have to be cleaned before a different colour of paint may be transported through them. For this purpose, vessel C with cleaning liquid has been added. Vessel W is used to collect liquid waste.

Also, an initial simulation model of the machine was built. This simulation model used for example vessels with infinite size, manifolds were not modelled, and the customer could choose between six colours only.

With the internal structure and the simulation results, the decision was taken to start the physical construction of the machine. A more detailed layout was constructed, and various pieces of the physical machine were made and roughly put together. The design of the internals of the manifolds became important. An initial design of them (not shown here) was found to be incorrect. By creating a number of alternatives, and by performing some calculations to verify them on correctness, the manifolds were designed as shown in Figure 5.3. The manifolds from the previous figure are shown here with dark-gray areas surrounded by a dashed line. The machine uses a combination of 3/2 valves, 2/2 valves,

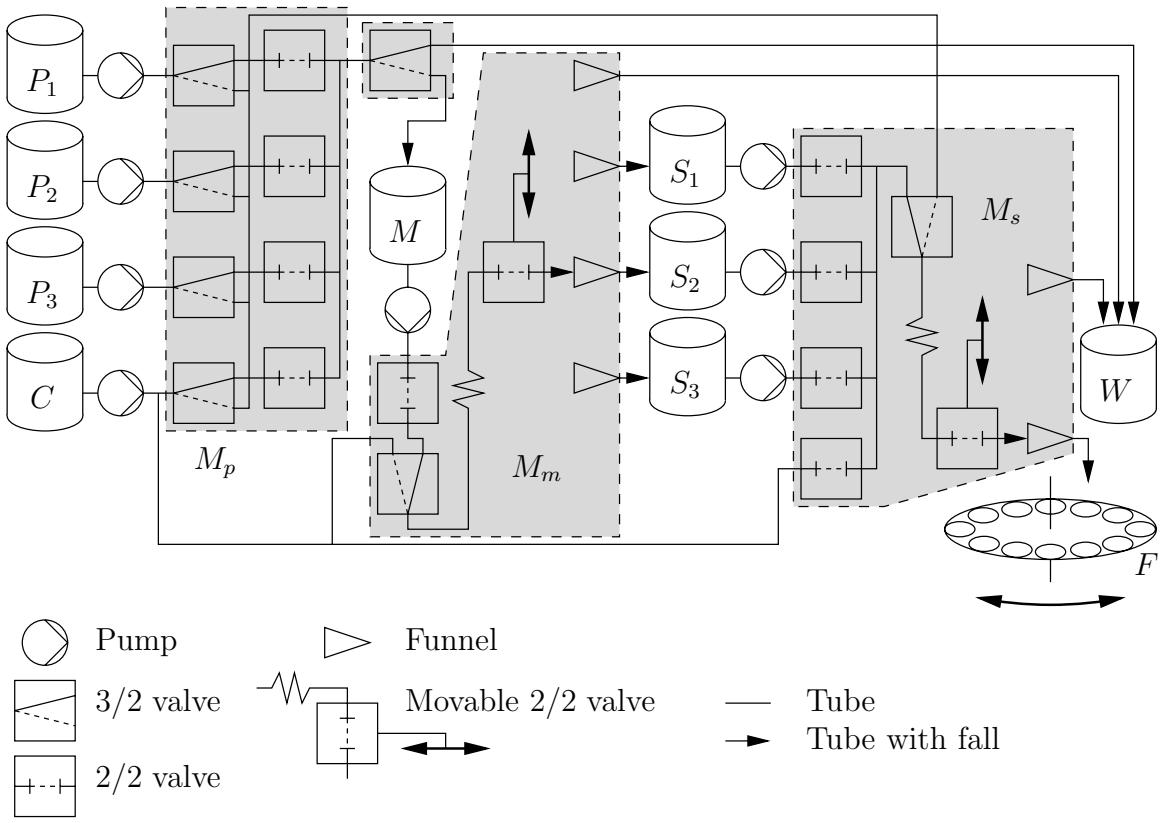

Figure 5.3: Detailed layout of the paint factory.

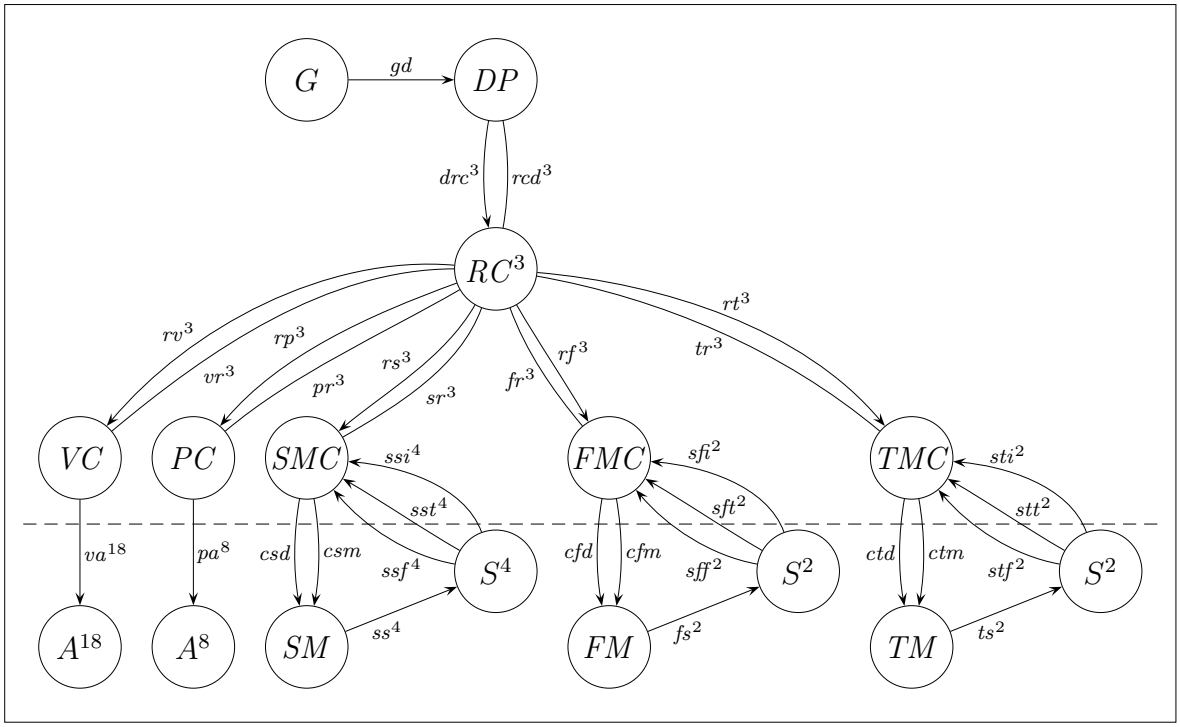

Figure 5.4: Paint factory simulation model in [vD00].

and movable valves. These valves select between two exits, switch a flow on and off, and choose between n exits respectively.

With the detailed layout in place, the manifolds can be constructed. The next step in making the machine come alive, is designing and implementing a controller for the machine. The simulation model depicted in Figure 5.4 shows the structure of the controller. Each circle is a process. The name of its definition is shown inside the circle. Some names are raised to the power of a integer number. That means that there are actually that many processes in existence instead of one. Channels connect the processes with each other. Channels with data transfer are indicated by an arrow, channels without data transfer are indicated by a line. Like processes, some channels are actually an array of channels. The number of channels is indicated by the number associated with the name of the channel. The horizontal dashed line across the figure indicates the border between controller and virtual machine. Above the line are the controller processes, below the line are the processes modelling the machine. From the bottom upwards, and from left to right, A^{18} are the 18 valves being controlled, A^8 are the 8 pumps, and SM is a motor that moves the valve above the storage vessels in manifold M_m. S^4 represents the 4 sensors that detect presence of the valve at each position. Process FM is similar to SM, but is positioned above the filling station in manifold M_s. TM is the motor that rotates the turn table. Its two sensor processes S detect correct positioning of a cup, and a zero position respectively. For each type of hardware, a hardware controller exists. The 18 valves are controlled by

a single valve controller VC, and PC is the pump controller responsible for controlling all pumps. Processes SMC, FMC, and TMC control the storage motor, filling-station motor, and turn-table motor respectively. In order to transport paint from one vessel to another vessel, a number of hardware controller must be given commands. These commands must be synchronized as well. For example, if the storage motor needs to be moved to a different storage vessel, the valve of manifold M_m should be closed until the motor has arrived at the correct position. Co-ordinating the actions performed by the hardware is the responsibility of resource controller RC. Since the paint factory allows at most three paint-movement jobs to be processed in parallel, there are three resource controllers. The dispatcher process DP makes sure that each job being processed uses different resources. Finally, the generator process G 'generates' orders to move paint from one vessel to another. Note that the generator is operational in nature, it sends jobs to move paint in a specified quantity between two specified vessels. This is quite different from the original idea of a customer ordering a number of canisters of paint with a specified colour. The reason for this change is that it became clear that the translation from customer orders to jobs is a scheduling problem which was too difficult to deal with at that time. Scheduling was postponed until later, and the operational control of the machine was developed first instead. Later, the scheduling of customer orders can be put on top of the operational controller.

The horizontal design step

The vertical design step lowers the abstraction level of a design. This activity is well known within the group as 'developing a simulation model'. The horizontal design step on the other hand, is relatively new. This case study is the first larger test of the design technique, therefore it is interesting to see how this step has been made.

It appears that the horizontal design step was not as horizontally executed as prescribed by the design technique. Several changes slipped in between the last simulation model and the implementation model. To illustrate this, the SMC controller process definition before and after the horizontal design step is discussed.

The purpose of SMC is to control the manifold M_m (see Figure 5.3). The manifold switches the output of the mixing vessel to the waste vessel, or to one of the three storage vessels. Physically, it is a cart with a 2/2 valve moving on a track, as schematically shown in Figure 5.5. At both ends of the track, limit switches are positioned that prohibit movement of the cart off the track. Also, along the track, four sensors detect the presence of the cart when it is correctly positioned above one of the vessels. The names of the sensors and limit switches are also shown in the figure.

Since the cart and valve are physically attached to each other, the discussion below about movement and positioning uses cart and valve interchangeably.

Figure 5.6 lists the SMC controller in the last simulation model before the horizontal design step. The controller gets its orders through channel rs, and acknowledges its readiness for receiving the next order through sr. The orders contain instructions to move the valve to one of the four positions above the vessels. Channels csd and csm are used

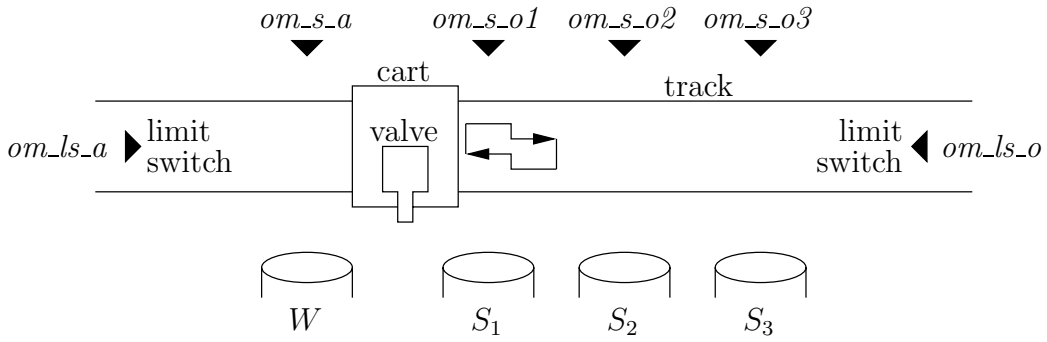

Figure 5.5: Schematic overview of the valve track controlled by the *SMC* controller.

```
proc SMC( rs: (?smorder)^3,  sr: (~ void)^3,  csd: !dir,  csm: !mon
        , ssf, sst: (~ void)^4,  ssi: (?bool)^4
        ) =
[[ b: bool,  i, m: nat,  k: smorder
| b := false;  i := 0
; *[  i < 4 ∧ ¬b ⟶ ssi.i ? b;  i := i + 1 ]
; [ b   ⟶ m := i − 1
  [] ¬b ⟶ csd ! true;  csm ! true;  sst.3 ~;  m := 3;  csm ! false
  ]
; i := 0;  *[  i < 3 ⟶ sr.i ~;  i := i + 1 ]
; *[ i ⟵ [0..3): rs.i ? k
     ⟶[ k = m ⟶ skip
        [] k ≠ m ⟶ csd !(k > m);  csm ! true;  sst.k ~;  csm ! false;  m := k
        ]
      ; sr.i ~
   ]
]]
```

Figure 5.6: Simulation model of the *SMC* controller in [vD00].

to set the direction of the cart and control movement respectively. Reports of the correct positioning of the valve comes from four sensors, which may be accessed through channels *ssf*, *sst*, or *ssi*. (For an explanation of having this many channels for the sensors, see *sensor modelling* at page 82.) The controller has no information about what happens with the cart between the positions. Note that the limit switches are not used in the simulation model.

After switching the machine on, the first job of the controller is thus to initialize properly by finding the current position of the valve. By probing all four sensors, it is determined whether the valve is located at a position above the vessels. If it is not, the valve must be somewhere between the sensors. In the latter case, by moving the cart towards the last position and waiting until the valve is detected by the corresponding sensor $sst.3$, the controller 'finds' the valve. The cart is stopped, and the controller is ready to receive orders. This is acknowledged upwards by synchronizing over all channels *sr*. From that moment, the controller repeatedly takes an order to move from position m to position k, it starts movement of the valve in the right direction until the valve arrives, and acknowledges the arrival upwards by synchronising readiness to receive a new order. In short, the controller does not take limit switches into account, and it moves the cart to a known position before accepting orders.

The implementation model of the same controller is shown in Figure 5.7. The basic steps taken are the same as in the simulation model. First initialization takes place, then the controller switches to normal operation. However, the code looks completely different. First of all, in the implementation model, all processes of the virtual machine have been deleted. Also, communication on actuator channels and sensor channels has been replaced by function calls to control the actual hardware. Controlling hardware means that the controller has to know about hardware addresses where the sensors and actuators are situated. In the controller, sensor addresses are stored in the constants in six-tuple *st*, and actuator addresses are stored in constants *omdir* and *om*. The four sensors used to position the valve above a vessel are addressed using $st.1$ through $st.4$, The remaining two sensors refer to the limit switches at both ends of the track. This leads to the second main change in the controller. The valve is assumed to be anywhere on its track, including between a limit switch and an outermost position. If the valve hits a limit switch, the movement has to be reversed. This reversing of direction makes movement of the valve more involved. Therefore, the controller is not attempting to move the valve during initialisation, as done in the simulation model. Instead, it only quickly tries to find the valve at one of the six positions. If it cannot find the valve, it marks it as 'unknown' using the assignment $cp := 99$. Next, it announces its readiness for taking orders by synchronizing over *sr*, followed by waiting for an order to move the valve from position *cp* to position *rp*. Upon receiving the order, the direction for the valve to move to is set or reset accordingly by Setbit(h, *omdir*), respectively Resetbit(h, *omdir*). Also, the limit switch at the end of the track which may become active is indicated in *ls*. Movement of the valve started with Setbit(h, *om*). The controller then monitors the sensor at the requested position as well as the limit switch, until the valve arrives at either one. If it arrives at the limit switch, the direction of the valve is reversed, and the controller resumes monitoring. If the valve

```
proc SMC(rs: (?smorder)^3, sr: (~ void)^3, ob: ?optohandle) =
[[ rp, cp, ls, rc, i: nat, b, c: bool, h: optohandle, st: (nat^2)^6
| st := ⟨ om_ls_a, om_s_a, om_s_o1, om_s_o2, om_s_o3, om_ls_o ⟩
; rc := 99; i := 0; b := false; ob ? h
; *[ i < 6 ∧ ¬b ⟶ b := Readbit(h, st.i); i := i + 1 ]
; [ b ⟶ cp := i − 1 [] ¬b ⟶ cp := 99 ]
; *[[ rc = 99 ⟶ i := 0; *[ i < 3 ⟶ sr.i ~; i := i + 1 ]
     [] rc ≠ 99 ⟶ sr.rc ~
     ]
   ; [ rc ← [0..3): rs.rc ? rp ]
   ; *[rp ≠ cp
        ⟶[ rp > cp ⟶ c := Setbit(h, omdir); ls := 5
           [] rp < cp ⟶ c := Resetbit(h, omdir); ls := 0
           ]
          ; c := Setbit(h, om); b := false
          ; *[ ¬b ∧ ¬Readbit(h, st.rp) ⟶ b := Readbit(h, st.ls) ]
          ; [ b ⟶ cp := ls [] ¬b ⟶ cp := rp ]
      ]
   ; c := Resetbit(h, om)
   ]
]]
```

Figure 5.7: Implementation model of the *SMC* controller in [vD00].

arrives at the requested position, movement is turned off, the controller acknowledges the arrival of the valve upwards, and waits for the next order to move the valve.

5.3 Observations and conclusions

The paint factory with its controller as described above, is able to physically transport paint from one vessel to another, and from a vessel to the turn table. After adding a scheduling layer and a user interface, the machine will meet the goal of the case study. Adding these layers of software is not necessary for evaluating the design technique for the design of machine control systems, because the layers do not introduce new concepts in control of the hardware. Below, a number of observations about the case study are discussed.

Vertical design step If the changes caused by the switch from simulation to the implementation in the last specification are ignored, three simulation models have been developed. The first one was an initial model with infinitely big vessels, and no manifolds, the second one was a complete operational controller of the machine, and the third one was the same model, but with the introduction of more details, like the limit switches. In this sequence, lowering of abstraction level can be seen between successive models. In other words, vertical design steps are made in the development of the machine control system.

Vertical design steps appear elsewhere as well. Looking at Figure 5.1, Figure 5.2, and Figure 5.3 as a sequence in the development process, it is clear that in each figure the abstraction level is lowered further. Vertical design steps have thus been made in the design of the machine as well. Some of these steps are not based on simulation, but for example on performing calculations of properties of possible manifold layouts. The vertical design steps can thus be useful in the design of a machine as well.

Horizontal design step Evaluating the horizontal design step is more difficult. After all, only one horizontal design step has been made in the entire project. Therefore drawing conclusions from the case study regarding the horizontal design step is somewhat hazardous.

With the above in mind, the first conclusion is that the technique seems to work. After all, a functioning operational controller exists. A more detailed analysis shows that the horizontal design step was not exactly horizontal. Besides the step from simulation to implementation, limit switches have been added and a different way of initializing was chosen. The latter changes are in fact vertical design steps. In other words, the horizontal design step made was in fact diagonal in nature, since it contains both horizontal and vertical design step activities.

Deviating from the design technique is not wrong; the technique is intended as a guide rather than a strait jacket. If the developer wants to deviate, there is probably a good reason. In the case study however, after finishing the operational controller,

the developer himself considered the diagonal design step an error. The next time, he would do more vertical design steps before making the step to the physical machine.

Distribution of control In the paint factory, but in other experiments as well, the control system is often distributed across several control processes that work together. Apparently, developers consider this a more natural way to specify the controller than in a single controller process. This observation strengthens the believe that a specification language aimed at specifying these kind of systems should have parallel executing processes.

Layering of control Another property often found in controllers is layering of functionality. For example in Figure 5.4, controller *SMC* controls a single cart on a track, while the control process *RC* controls the entire paint factory for a single pump order to pump a specified amount of paint through the factory. At a higher level of abstraction, layering of control also takes place. As explained in the previous section, the controller designed is operational in nature, and a scheduler that transform customer orders to pump orders is put on top of the former.

This layering is natural in the design of complex controllers. It occurs in manufacturing control systems as well, for example, see the hierarchical model of a manufacturing-control system in [JM86], shown in Figure 1.1. It is nice to see that the design technique allows this kind of layering. It gives more confidence that the technique will be scalable to larger machines.

Other observations made in the case study are the use of function calls to address the physical machine, and the modelling of sensors. The conclusions drawn from these observations go beyond the case study, and are therefore discussed in the next chapter.

Notably lacking are observations about the simulation and translation tools. During the project, no special attention was paid to them, they were used just like any other tool. Apparently, the tools performed well enough not to be noticed, and/or the modelling problems were far bigger than the tool problems.

It should be noted that the designer performing the case study had experience with simulator tools. The magic step from a 'dead' specification on paper to a 'living' simulation was already known to him. Also known to him was the development of a simulation model, that is, performing vertical design steps. Furthermore, at the surface, there is no big difference between a tool that performs a vertical design step and one that performs a horizontal design step. Both tools start with a specification, and end with an executable program. The only major difference between both tools is that the generated executables run on different platforms. In other words, on the surface there were no major changes in the approach or the tools.

Even though it sounds odd here, the fact that the designer did not notice the tools should be considered a good sign. Basically it means that the tools were used as tools in the case study, rather than as objects that need special care. This is much like the use of for example, a text editor. How it is used, and how well it performs, are irrelevant

questions in day-to-day use. The tool is simply assumed to perform its function without being noticed.

A number of smaller changes and extensions have been made to the tools, such as building a proper I/O library and fixing the random number generator. Also, in the mean time, more experience has been gained with the tools, and except for the translation of communication to function calls (see the next chapter for more details), the tools function satisfactorily, not only in research projects, but also in education.

CHAPTER
SIX

Conclusions

In this project, a design technique for the design and implementation of machine control systems has been proposed. Also, to make the technique feasible for realistic case studies, tools have been developed to provide computer support for the developers. In this chapter, the project is reflected upon, and suggestions for next steps are made.

In the next sections, the design technique, the modelling of machine control systems, virtual machines, and the languages and tools used, are evaluated.

However, before discussing the various details, the overall conclusion of the project is that the design technique seems to work. As demonstrated in the case study, machines like the paint factory can be designed and implemented using the design technique and the tools. The immediate result of the project is that it has become practically feasible to design and implement controllers for machines. This capability forms the basis for conducting further experimental research towards better design techniques for machine control systems.

6.1 The design technique

The design technique is a framework that guides developers during the development process of a machine control system.

The basic idea of the technique is understandable to our users. The available tools allows them to scale the design technique up to realistic applications. The technique is however far from perfect. Especially the horizontal design step needs a lot of research.

Below, both types of design steps are discussed in more detail.

Vertical design step

A vertical design step is a single iteration in the development cycle of a simulation model. In the step, the abstraction level of the model is reduced. In the Systems Engineering Group, developing simulation models is a known art. In other words, a lot of experience has been gained with vertical development steps already. Until now, the iterations just did not have an explicit name.

In the case study, development steps were made by computing different alternatives rather than simulating them. Since the steps lower the level of abstraction, such steps

should be marked vertical design steps too. The current definition of vertical design steps is clearly too narrow.

At first this was considered a surprising result, because it was expected that only simulation was used to make design decisions. However, looking back at the source [vdKS98], the method is presented as a generic method. In other words, it was already promised that the development steps would be applicable in more situations. The surprise is that apparently, there is currently little understanding of our own approach of developing models. How to deal with this discovery is still unclear. However, when improving the design technique, this issue should be addressed.

Horizontal design step

In the horizontal design step, a simulation model of a control system is transformed to an implementation controlling a real machine.

One of the weak points of the horizontal design step is its reliability. Users applying the design technique expect that after performing the horizontal design step, everything works as modelled. In practice however, it is the moment where the modelling errors come to the surface. Often, they are small errors like using the wrong value for the direction of a motor. For example, writing true to an output causes a cart to move to the left instead of the expected movement to the right. These small errors can however cause major havoc in the machine. The attention of further research in the design technique should therefore focus on making this step more reliable.

Discussed below are two aspects of the predictability of the horizontal design step. Also, there is a short explanation on how the expressive power of the selective waiting statement may be preserved during the horizontal design step.

Accuracy of the machine model

In the horizontal design step, the virtual machine in the simulation model is blindly replaced with the real machine. If this step is to be successful, the virtual machine has to be accurate. More precisely, the model must have actuators and sensors that react the same as in the real machine. In the ideal case, this property should hold at each moment in time. For any realistic machine, this requirement is quite likely too complex to fulfill. At a low enough level of detail, no two machines are the same, let alone that an abstract description of a machine would be able to capture the behaviour exactly. Fortunately, all that is needed is a model that is accurate enough. The question is however, what is accurate enough, and how can one verify this property. This will be an important topic of research to make the horizontal design step more reliable.

Post-horizontal phase

The current tool implementation of the horizontal design step only performs a transition from simulation to implementation. In particular, it does not pay attention to optimiza-

tions in time or space.

One of the first objectives after the transition should be to verify correctness of the resulting implementation. This has to be done using testing techniques, since formal verification against a physical machine is difficult to do. Also, in some embedded systems, minimizing usage of time and space is important.

For these reasons, it may be useful to extend the design technique with a post-horizontal phase, where testing and tailoring of the implementation can take place.

Transformation of communication

In the current tool implementation, the user has to replace machine communications in the controller with function calls to read and write the hardware status of the physical machine. This has been a design decision of the tool. Hardware access has many forms, and in order not to exclude possibilities, the most flexible and easily accessible form of hardware addressing has been chosen. The price that a user pays for this decision is the loss of a selective waiting statement with multiple communication choices. For example, in a controller model, the following fragment waits for a response from the machine on ports a and b:

$$
\begin{array}{l}
\ldots \\
;\,[\ a\,{\sim} \longrightarrow \ldots \\
\ \ [\!]\ b\,{\sim} \longrightarrow \ldots \\
\ \ ] \\
\ldots
\end{array}
$$

In the implementation, the synchronization channels a and b are replaced by hardware access at ha respectively hb. The following fragment thus comes into existence:

$$
\begin{array}{l}
\ldots \\
;\,u := \text{false};\ v := \text{false} \\
;\,*[\ \neg u \wedge \neg v \longrightarrow u := \text{Readbit}(ha);\ v := \text{Readbit}(hb)\ ] \\
;\,[\ u \longrightarrow \ldots \\
\ \ [\!]\ v \longrightarrow \ldots \\
\ \ ] \\
\ldots
\end{array}
$$

Two extra variables u and v temporarily store the values read from the physical machine. These variables are then used to decide the flow of the program.

The above is an example of a standard replacement pattern. A similar pattern exists for the case of a one or more communication choices and one or more time-out choices. Since these patterns are standard, it is possible to code them into the compiler tool for the horizontal design step. Another alternative that can be considered is to leave the fragment untouched, and change the communication choices in the run-time support layer of the target system. In the latter alternative, the controller still uses communication primitives to access the machine. These are coupled to the real machine in the run-time support layer. The latter solution is conceptually cleaner, but requires some fundamental changes in the

implementation. At this moment, no decision has been made which of the alternatives should be used.

6.2 Modelling

Although the Systems Engineering Group has been developing simulation models of industrial systems for many years, modelling a machine control system with its machine is a new subject. With the availability of tools both for the vertical and horizontal design steps, it has become possible to get experience in the modelling, simulation, and implementation of machine control systems.

One of the goals is to get standard solutions for standard concepts. Just like a buffer and a machine are standard in any manufacturing control system, so are an actuator and a sensor standard in machine control systems. Rather than re-inventing the solution each time, these concepts should be developed once and be available for everybody.

Sensor modelling

Coming up with a good model of a sensor for simulation is not simple. In particular, it appears to be impossible to have a simple sensor model for *continuous polling*. With continuous polling, the controller should react immediately upon the sensor. The sensor is however a passive component, and not capable of sending a signal to the controller at the moment the latter should react. The controller thus has to continuously query the value of the sensor until it becomes active and then take immediate action.

For the purpose of this discussion, it is assumed that the controller has to monitor a button. As soon as the button is activated, the controller should perform some action. This leads to the following χ specification:

$$
\begin{aligned}
&\mathsf{proc}\ U(pb\colon !\,\mathrm{bool}) = [\![\ pb\,!\,\mathrm{false};\ \Delta 1;\ pb\,!\,\mathrm{true}\ ]\!]\\
&\mathsf{proc}\ C(lb\colon ?\,\mathrm{bool}) = [\![\ v\colon \mathrm{bool} \mid v := \mathrm{false};\ *[\ \neg v \longrightarrow lb\,?\,v\ ];\ \ldots\ ]\!]\\
&\mathsf{proc}\ B(pb\colon ?\,\mathrm{bool},\ lb\colon !\,\mathrm{bool}) = [\![\ b\colon \mathrm{bool} \mid pb\,?\,b;\ *[\ pb\,?\,b\ [\!]\ lb\,!\,b\ ]\ ]\!]\\
&\mathsf{syst}\ S = [\![\ pb, lb\colon -\mathrm{bool} \mid U(pb) \parallel B(pb,\ lb) \parallel C(lb)\ ]\!]\\
&\mathsf{xper}\ \ = [\![\ S()\ ]\!]
\end{aligned}
$$

There are three processes running in this specification. The user process U simulates a user pressing the physical button (channel pb) after one second, a controller process C that continuously polls the logical button (channel lb) until it is activated and then performs some (unspecified) action, and finally the button process B that transforms the physical position of the button to a logical value. Like a normal button, the physical position can change at any moment, and the logical value can be queried at any moment.

The above specification is a logical representation of reality. Most buttons work in this manner, and polling is an accepted method of querying status. The simulation of this specification will however never reach the point where the button is pressed by the user. The reason for this behaviour is the assumption of the synchrony hypothesis for χ

models. This hypothesis assumes infinite computing resources by only progressing time when every process is blocked. The infinite computing resources enable instantaneous response of a modelled system to events, which is a good property for reasoning about a complex system. However, in the case of continuous polling, it uses its infinite computing resources to perform an infinite number of polling operations before progressing in time. In other words, it is stuck. It should be noted that this problem exists for any language using the synchrony hypothesis, not only χ is affected.

There are two ways to solve the above problem. One solution is to further lower the abstraction level of polling, the other solution is to stop lowering the abstraction level before reaching the above 'solution'.

Further lowering of the abstraction level means introducing the concept of finite resources into the specification. In reality, things like infinite computing resources and infinite bandwidth in the communication channel do not exist. Therefore, it can be justified that after a polling cycle in the controller or after usage of the communication channel, a small amount of progress in time should be made. By introducing either of these concepts, time will progress in the simulation, and eventually reach the moment where the simulated user presses the physical button.

The more abstract version of the above specification can be reached by realizing that the value of the button is not of interest to us; the controller is only interested in the moment that the button is pressed down. Thus, rather than communicating the value of the button over the channel, the channel should communicate 'the button is pressed down'. If the button is not depressed, the channel should block. In this version, the controller will wait for a communication with the button. Since the channel blocks until the user presses the button, time will progress in the simulation.

Neither of the solutions is really logical for modelling polling. The more abstract version uses a button model that is too rich compared to reality, the introduction of finite resources is against the philosophy of the modelling language. The whole idea of the synchrony hypothesis is that the developer should not worry about details such as finite resources, these worries should be postponed until later in the development process.

Errors and exceptions

Handling of, and recovery from error conditions is a difficult concept in machine control systems. The basic idea is that upon detection of an error in the operation of the machine, the machine must go to a safe state. This is often enforced by hardware construction of the machine. For example, the emergency switch is wired to cut the power of the machine when activated. The controller must detect the error, and decide how to handle it. After resolving the error, a switch back to normal operation of the machine must be made in a well-defined manner, thus resuming production.

While the above explanation is short and easy to understand, the implications in terms of the controller specification are huge. Occurrence of errors is by definition unpredictable. That means that at any point in the control program an error may occur from which

recovery must be possible. Recovery often means a radical step outside the normal flow of control. Other languages use exceptions to handle such matters. The current definition of χ has no exceptions, although [vBR96] describes an extension to add exceptions to the language. However, before deciding to include exceptions in the language, error handling and recovery in machine control systems should be carefully studied in order to be able to judge whether exceptions are the right answer to the problem.

6.3 Virtual machines

As briefly touched in the first chapter, a virtual machine is the part of a simulation model that represents the machine hardware being controlled. In the design technique, the virtual machine is developed using vertical design steps, and then replaced by the real machine in the final horizontal design step. Within the design technique, the role of the virtual machine is thus limited to use as a representation of the machine hardware during the vertical design steps. However, the virtual machine provides a number of hooks in the development process which are very interesting to explore in a future research project.

First of all, a virtual machine is a formal, unambiguous description of a machine. That makes it a very good candidate to use as a reference document between the group of machine hardware developers and the group of control system developers, which may co-exist in a project which uses concurrent-engineering techniques.

Secondly, a virtual machine is still a valid description of the real machine after performing the horizontal design step. In other words, the virtual machine may still be used as a test bed for modifications in the controller. This opens the possibility to make enhancements or to fix bugs in the controller and test the changes on the virtual machine, before requiring costly machine time to implement and test the changes on the real machine. Of course, the virtual machine is only valid as long as the real machine is not modified. If the real machine does get modified, the virtual machine should be modified accordingly in order to keep the virtual machine in sync.

6.4 Languages and tools

In this section, the choice of languages and tools is discussed. Also, some indications of consequences are given if the controller implementation would use a different language or a different real-time operating system.

Using χ as modelling language has been a correct decision. The ability to use the years of experience in modelling manufacturing-control systems with this language made the introduction of the new modelling subject easier. Students did not have to learn a new language, and could discuss their problems with other students without language barriers. χ has shown to be a good modelling language to tackle the complexity of machine control systems in our experiments. It is expected that the language will be able to deliver enough expressive power to tackle complex control systems as well.

The use of object-oriented techniques has aided the design of both the compiler and the run-time environment. More importantly, it had a positive impact on maintenance and extension of the tools as well, especially on the compiler tool. Bugs could be located quickly and extensions could be introduced by locally modifying the implementation.

As for implementation languages, using C++ in the compiler tool has also been a good decision. Design and implementation of the compiler went fluently, for a large part due to experience of the previous implementations and the object-oriented design. The type checker was the most difficult hurdle, because there was no off-the-shelf algorithm. The design of the compiler tool has shown to be good. Major changes in the implementation have not been necessary. For example, in the mean time, the code generator for simulations has been added. This has been done without changing other parts of the compiler. The compiler has also proven to be accurate in its error reporting.

Using C++ in the run-time support environment of the controller implementation has been beneficial to the project, because it enabled the re-use of existing implementations of χ data structures. The main weak point in the current implementation of the environment is the lack of a formal proof of the correctness of the communication protocol.

How well the environment will perform in real industrial projects is not yet known. Especially the choice of the implementation language remains a point of concern. However, should the need arise, switching to a different implementation language is not insurmountable. For example, switching to Java should be straightforward due to the object-oriented design, except for the use of templates. Replacements for the latter can however be generated by the compiler. Switching to C is more work because that language also lacks object-oriented concepts, but there are no fundamental obstacles that prohibit the switch to a different implementation language.

The real-time operating system VxWorks functions as expected. Currently, only a small part of its capabilities is actually used. That means that it is not difficult to switch to a different operating system now. On the other hand, VxWorks does provide a number of tools that may become useful in the future.

The overall conclusion of the project is that a base line has been drawn. A design technique has been proposed that is understandable and feasible to the users. Working tools exist to enable users perform realistic case studies.

On the other hand, the research on this subject has only just started. With the case study, it has become clear that there is currently little understanding of the development process itself, especially the horizontal design step.

With the available tools, the next step is to learn modelling of machine control systems. Also, fitting the design technique into a real industrial design project with the use of concurrent engineering techniques, as well as exploring the use of the virtual machine concept are big challenges for the future.

APPENDIX

A

The type-checking mechanism

In this appendix, the type-checking mechanism used in the χ compiler is explained. How this mechanism fits in the other parts of the compiler is explained in the section about the χ compiler on page 39.

The global working is first demonstrated with a small example. After the example, the algorithm is explained in a bottom-up fashion. It starts with the core algorithm to perform pattern matching on types. The pattern-matching algorithm is used to solve the types of a set of type variables. This algorithm is described next.

After implementation, the performance of the type-check algorithm was too poor for practical use. By making some small modifications, the performance was improved to an acceptable level. The final section describes these modifications.

A.1 Introduction

The type-checking process in the compiler decides which data types are used in an expression. It works in three phases.

1. In the first phase, the compiler generates a set of *type variables*, and associates each type variable with a *type set*. The latter describes possible solutions for the type variables. The compiler also generates a number of *type rules*. These rules describe the relation between different type variables.

2. In the second phase, the set of type sets and type rules is used to deduce a type for each type variable. This is done by logic reasoning. By combining type sets with each other using the type rules, the sets are reduced until no reduction is possible any more, or until a type set becomes empty. In the former case, the solution is found, in the latter case, it is concluded that no valid solution exists.

3. In the third phase, if a solution is found, the types found by the algorithm are checked for being unique and non-polymorphic, and they are copied into the abstract syntax tree of the compiler. If the solution is not unique or it is polymorphic, the expression is considered to be incorrect. The user should modify the expression.

This appendix focuses mainly on the second step.

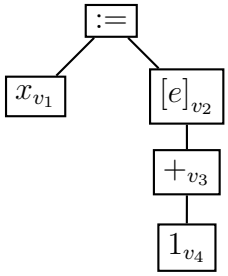

Figure A.1: $x := [+1]$ as a tree structure.

Before proceeding to the example, a few words about types. In χ, a number of basic types exist, such as *nat* or *bool*. More complex types can be constructed by applying operators on types. For example, the $*$-operator constructs a list from a given type. In this appendix, that nested structure is important, because equality of types in χ is defined as structural equality. Unfortunately, the structure of types is not easily seen when using the χ syntax. For this reason, this appendix uses an expression syntax. A type can be seen as an expression. The basic types are the constants, the operators are functions. This tree structure is then written as a set of nested function calls. For example, a list of integers is written in χ as 'int$*$'. Here, it is written as 'list(int)'. The only exception to this rule is for functions. For clarity, the return type is indicated by separating it from the arguments by an arrow. For example, the addition function of two naturals has the type '$(\text{nat}, \text{nat} \rightarrow \text{nat})$'. In this appendix, function identifiers f, g, etc are used as generic type operators.

As an example of the above process, consider the statement $x := [+1]$, where x is declared as a variable of type int*. The right-hand side of the assignment has the same type. The constant 1 has nat as type. The unary +-function converts the constant to the same value, but with an int type. The square brackets construct a list, containing the integer constant, thus resulting in the same int* type. Since the types at the left-hand side and right-hand side match, it is a correct χ assignment statement. Below is an explanation of how the compiler reaches this decision. The statement is stored in the abstract syntax tree as shown in Figure A.1. The subscripts v_i in each node are type variables assigned to the nodes in this part of the tree.

The first phase of the type-check algorithm is to generate type rules and type sets for the assignment statement. For the assignment statement, the rules and sets are listed in tables A.1 and A.2 respectively. As explained, the type rules describe relations between different type variables. For example, $(A.1)$ says that if v_1 is type T, then v_2 should also be type T and vice versa. Type sets specify which types may be used to fulfill the rules. Each element in a type set may be used as a type in the result. A polymorphic value in an element (written here as Greek letters α, β, etc), allows any type to be used in its place. For example, a valid type for v_2 would be list(bool) or list(list(int)) according to $(A.5)$.

$$v_1 = v_2 \quad (A.1)$$
$$v_2 = \text{list}(v_r) \quad (A.2)$$
$$v_3 = (v_4 \rightarrow v_r) \quad (A.3)$$

Table A.1: Type rules of Figure A.1

$$v_1 \in \{\text{list(int)}\} \quad (A.4)$$
$$v_2 \in \{\text{list}(\alpha)\} \quad (A.5)$$
$$v_3 \in \{(\text{nat} \rightarrow \text{int}), (\text{int} \rightarrow \text{int}), (\text{real} \rightarrow \text{real})\} \quad (A.6)$$
$$v_r \in \{\beta\} \quad (A.7)$$
$$v_4 \in \{\text{nat}\} \quad (A.8)$$

Table A.2: Type sets of Figure A.1

Note that names of polymorphic values are unique within a single element of a type set only.

Resolving the equations for the type variables in the second phase can for example be done by first applying $(A.1)$ and $(A.4)$. The only common type between v_1 and v_2 is list(int). v_1 and v_2 thus both become this type. Using $(A.2)$, type set $(A.7)$ becomes $v_r \in \{\text{int}\}$. Finally, applying $(A.3)$ results in $v_3 \in \{(\text{nat} \rightarrow \text{int})\}$. At this moment, no further reduction is possible, and the second phase terminates.

The third phase checks the existence of a unique non-polymorphic type for each type variable, and copies them to the abstract syntax tree.

A.2 Type matching

The core algorithm of the second type-checking phase is reaching a decision whether two (type) expressions p and p' match, and if so, what the resulting type is. This functionality is delivered by the type-matching function $\mathcal{M}(p, p', \Sigma)$. Parameters p and p' are two type-patterns which may contain 'holes' in the form of polymorphic values. It is assumed that both patterns do not share polymorphic values, thus if a polymorphic value α is used in pattern p, then it is not used in pattern p' and vice versa. In the implementation, this requirement is fulfilled in the type-resolving algorithm. The third parameter Σ of the type-matching function is a symbol table of substitutions in both patterns. A substitution has the form α/x, which means that the polymorphic value α should be replaced by the pattern x.

Testing whether a substitution for α is available in the symbol table is written as $\alpha/x \in \Sigma$. This expression is true when the substitution exists. If it is true, x becomes bound to the replacement pattern to allow usage of this pattern.

Symbol tables can be concatenated, as in $\Sigma \cdot [\alpha_1/x_1, \ldots, \alpha_n/x_n]$. The list of substitutions between the square brackets define a symbol table containing the substitutions, the dot operator concatenates both tables together. The concatenation is defined only for non-overlapping symbol tables, that is, the concatenation $\Sigma_1 \cdot \Sigma_2$ is only allowed if $\forall_i \colon \alpha_i/x_i \in \Sigma_1 \Rightarrow \alpha_i/y_i \notin \Sigma_2$.

Finally, application of substitutions in a pattern p from symbol table Σ is written as

Σp, and is defined as

$$\Sigma p = \begin{cases} \Sigma x, & \text{if } p = \alpha,\ \alpha/x \in \Sigma \\ \alpha, & \text{if } p = \alpha,\ \alpha/x \notin \Sigma \\ f(\Sigma x_1, \Sigma x_2, \ldots, \Sigma x_n) & \text{if } p = f(x_1, x_2, \ldots, x_n) \end{cases}$$

The result of the type-matching function is a tuple $\langle \Sigma', b \rangle$. The first field contains the updated symbol table Σ', the second field is either **match** or **fail**. The former value means that a match is found, the latter means that no match has been found.

The type-matching function has four cases, depending on whether the first and second parameters are a function or a polymorphic value. For each of the cases, the relation between the function parameters and its result is described. Besides the operations on the symbol table listed above, a test whether a polymorphic value α is used in pattern p is also needed in the description. This is written as $\alpha \in p$.

$$\begin{aligned} &\mathcal{M}(f(p_1, \ldots, p_n), f'(p'_1, \ldots, p'_m), \Sigma) \\ &= \begin{cases} \langle \Sigma, \textbf{fail} \rangle & \text{if } f \neq f' \vee n \neq m \\ \langle \Sigma, \textbf{match} \rangle & \text{if } f = f' \wedge n = m \wedge n = 0 \\ \langle \Sigma_n, b \rangle & \text{if } f = f' \wedge n = m \wedge n > 0 \end{cases} \\ &\quad \text{with } \langle \Sigma_i, b_i \rangle = \mathcal{M}(p_i, p'_i, \Sigma_{i-1}), \Sigma_0 = \Sigma, b = \begin{cases} \textbf{match} & \text{if } \forall_{1 \leq i \leq n} \colon b_i = \textbf{match} \\ \textbf{fail} & \text{otherwise} \end{cases} \end{aligned} \tag{A.9}$$

$$\begin{aligned} &\mathcal{M}(f(p_1, \ldots, p_n), \alpha, \Sigma) \\ &= \begin{cases} \mathcal{M}(f(p_1, \ldots, p_n), x, \Sigma) & \text{if } \alpha/x \in \Sigma \\ \langle \Sigma \cdot [\alpha / f(p_1, \ldots, p_n)], \textbf{match} \rangle & \text{if } \alpha/x \notin \Sigma \wedge \alpha \notin \Sigma f(p_1, \ldots, p_n) \\ \langle \Sigma, \textbf{fail} \rangle & \text{if } \alpha/x \notin \Sigma \wedge \alpha \in \Sigma f(p_1, \ldots, p_n) \end{cases} \end{aligned} \tag{A.10}$$

$$\begin{aligned} &\mathcal{M}(\alpha, f(p_1, \ldots, p_m), \Sigma) \\ &= \mathcal{M}(f(p_1, \ldots, p_m), \alpha, \Sigma) \end{aligned} \tag{A.11}$$

$$\begin{aligned} &\mathcal{M}(\alpha, \alpha', \Sigma) \\ &= \begin{cases} \mathcal{M}(x, \alpha', \Sigma) & \text{if } \alpha/x \in \Sigma \\ \mathcal{M}(\alpha, x, \Sigma) & \text{if } \alpha'/x \in \Sigma \\ \langle \Sigma, \textbf{match} \rangle & \text{if } \alpha = \alpha' \\ \langle \Sigma \cdot [\alpha/\alpha'], \textbf{match} \rangle & \text{if } \alpha/x \notin \Sigma \wedge \alpha'/x \notin \Sigma \wedge \alpha \neq \alpha' \end{cases} \end{aligned} \tag{A.12}$$

The first case (A.9) occurs when trying to match two 'normal' patterns. They match if they have a common prefix f and f', and when all sub-patterns p_i, p'_i match. The other three cases occur when a polymorphic value is encountered. If this happens in combination with a normal pattern, the polymorphic value is substituted from the symbol table Σ if available, or an entry is added to the table. In the latter case, an extra condition has been added to prevent infinitely large type patterns. The last case of polymorphic values in the

type-matching function is (A.12), where two polymorphic values are matched with each other. If either of them can be substituted, the substitution takes place; if they are equal, there is always a match; otherwise they must be equal to each other for a match, so one value is assigned as pattern for the other value.

It should be noted that in the first case, the description states that all sub-patterns matches $\mathcal{M}(p_i, p'_i, \Sigma_{i-1})$ should be attempted, even when a failure to match has already been detected. This part of the definition of $\mathcal{M}$, although seemingly inefficient, is intended, and is necessary for improving the performance of the type-check algorithm later.

A.3 Type-variables resolving

With the type matching function $\mathcal{M}$, the value of type variables v_i in an expression can be decided, like in the example at the beginning of this chapter.

More formally, the problem to solve is to find a unique value for each of the type variables $v_1, v_2, \ldots, v_n$. Allowed values for type variables are kept in associated typesets $V_1, V_2, \ldots, V_n$. Each value in a typeset may contain polymorphic values, but the number of values in a typeset is always finite. It is assumed that all type sets are non-empty, otherwise there would be no valid solution for the associated type variable.

Also available is a set of type rules[1]

$$u_0^1 = F_1(u_1^1, u_2^1, \ldots, u_{p_1}^1),\ u_0^2 = F_2(u_1^2, u_2^2, \ldots, u_{p_2}^2),\ \ldots,\ u_0^m = F_m(u_1^m, u_2^m, \ldots, u_{p_m}^m)$$

The variables u are instances of type variables v_i, therefore each u_k^j has a typeset associated with it. Also, F_j is a compact notation for some type pattern using type variables $u_1^j, \ldots, u_{p_j}^j$.

The basic reduction step for type sets using rule j is as follows:

$$\begin{array}{ll}
\forall\, i \in \{j_0, \ldots, j_k\}: & V'_i := \emptyset; \\
\forall\, x_{j_0} \in V_{j_0}, \ldots, x_{j_k} \in V_{j_k}: & \text{if } \mathcal{M}(v_{j_0}, F_j(v_{j_1}, \ldots, v_{j_k}), [v_{j_0}/x_{j_0}, \ldots, v_{j_k}/x_{j_k}]) = \langle \Sigma, \mathbf{match} \rangle \\
 & \text{then } \forall\, i \in \{j_0, \ldots, j_k\}: V'_i := V'_i \cup \{\Sigma v_i\}; \\
\forall\, i \in \{j_0, \ldots, j_k\}: & V_i := V'_i
\end{array}$$

This basic reduction step is repeatedly attempted for all type rules $j \in \{1, \ldots, m\}$ until none of the typesets V_i changes any more.

As service to the user, the algorithm is prematurely aborted when a type set becomes empty after the update. An empty type set means that there is no valid value for its associated type variable. This can happen when the user specifies an incorrect expression. By stopping the algorithm prematurely, the algorithm can give a more precise position where the error is detected. Note that continuing the algorithm would not create illegal solutions. As soon as one type set becomes empty, applying the reduction step to a rule

[1]The general form of a single type rule would be $F_j(t_1^j, \ldots, t_{p_j}^j) = G_j(u_1^j, \ldots, u_{q_j}^j)$. While it is possible to generalize the type-checking algorithm to this form, it gives unnecessary flexibility and complicates the optimization discussed later. Therefore, this general form is not used here.

that uses the empty set, will cause distribution of the empty solution to all other used type sets in that rule, until all type sets V_i are empty. At that point, no further reduction is possible, and the algorithm will terminate.

For the algorithm to be useful, it should have a number of properties. One important property is that the result should be independent of the order in which type rules are applied to the type sets.

This property holds, because the reduction step updates all type sets to the biggest common solution. For example, if $U = \{a, b\}$ and $V = \{b, c\}$, and the type rule $u = v, u \in U, v \in V$ is applied, then afterwards $U = V = \{b\}$. Both type sets U and V are changed as a result of applying the rule. In other words, when applying a rule, the type sets are updated 'in all directions'. Also, unless at least one type set objects in the sense that it does not give a match, the element is preserved. Therefore, no solution is thrown away, unless there is 'proof' that the solution will not deliver a match. The latter basically means that the type rules cannot be applied in the wrong order. For example, if the algorithm would again apply the same type rule as above using the modified type sets, then both type sets U and V will not change.

A.4 Performance

While the type-checking algorithm discussed above produces correct results, it is also too slow for realistic applications. This is caused by the fact that the algorithm tries to find matches for a type rule by trying each possible combination of elements from the type sets. Consider the following example, where V_0 denotes $\{f(\text{int}, \text{int}, \text{int}, \text{int}, \text{int})\}$, and V_i denotes $\{\text{nat}, \text{int}, \text{real}\}$ for $1 \leq i \leq 5$:

$$\begin{aligned} v_0 &\in V_0 \\ v_i &\in V_i, \quad (1 \leq i \leq 5) \\ v_0 &= f(v_1, v_2, v_3, v_4, v_5) \end{aligned}$$

Applying the rule to these type sets produces the following sequence of matching attempts[2] in the second phase:

[2] Or some permutation of this sequence.

Step	v_0	v_1	v_2	v_3	v_4	v_5	Result
1	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	nat	nat	nat	nat	nat	No match
2	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	int	nat	nat	nat	nat	No match
3	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	real	nat	nat	nat	nat	No match
4	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	nat	int	nat	nat	nat	No match
			⋮				
121	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	nat	int	int	int	int	No match
122	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	int	int	int	int	int	Match
123	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	real	int	int	int	int	No match
			⋮				
243	$f(\text{int},\text{int},\text{int},\text{int},\text{int})$	real	real	real	real	real	No match

In other words, one match out of 243 attempts. Although this is bad, the real bad news is that the algorithm slows down exponentially. In realistic χ specifications, it is not uncommon to have for example 13 or more parameters in the top-level system definition, instead of the 5 parameters shown here. That means that users would have to wait a few hours or days before the algorithm is finished, not something that the average user is willing to do.

There are two ways to reduce the problem. Either the initial size of the type sets is made smaller before the second phase of the algorithm begins, or the matching algorithm is made smarter. The former solution means that the generator creating the type sets and type rules becomes more intelligent, it should take the context of a type set into account. In some sense, this means that the reduction of the type sets is partly moved to the first phase of the type-check algorithm. Besides the fact that the approach violates the separation of concerns, it also complicates the generation phase. In particular, it is very difficult to ensure that all cases are covered, and no errors have been made.

The other solution of a smarter reduction during type matching in the second phase is a far better solution. The basic idea is that elements in type sets that will not contribute to a solution should be eliminated as soon as possible. The effects of this solution are dramatic. If, for example, type set V_1 could be changed to contain {int} only, a 66% reduction in the number of matching attempts is realized. The results get even better when the same trick is applied to type sets V_2 through V_5 as well. Also, last but not least, implementing this optimization does no harm to the generality of the algorithm, and is cheap (both in time and code).

The optimization uses the fact that the left-hand side (LHS) of a type rule consists of a single type variable, the fact that the type-matching function continues to attempt matching sub-patterns in (A.9) even after a matching failure has been detected, and the reduction property of a type set.

The approach used to implement this general idea is based on the fact that matches are searched between the LHS and the right-hand side of the rule. Thus, in order to find a match for some combination of values for v_i ($i \geq 1$), there has to be a 'correct' element in the type set for the LHS (V_0). Since finding a match is a recursive algorithm, this holds

both for the top-level of both type patterns, and for the levels below. This means that for v_i ($i \geq 1$) to have a 'correct' value (to result in a match), there has to be a 'correct' type pattern at the LHS. For the example this means, that in order to find a match for $v_5 = \text{nat}$, there has to be an element in V_0 where the fifth parameter is 'nat'. So, if for some value v_i ($i \geq 1$), each element of V_0 causes a failure to match for the sub-pattern v_i, then that value of v_i will never match due to the reduction property of a type set. In the example, since there is no element in V_0 with 'nat' as fifth parameter, $v_5 = \text{nat}$ will never match and can safely be discarded. Implementing this mechanism allows the following sequence of the example

Step	v_1	v_2	v_3	v_4	v_5	Result
1	nat	nat	nat	nat	nat	All nat elements eliminated
2	int	int	int	int	int	Match
3	int	int	int	int	real	$v_5 = \text{real}$ eliminated
4	int	int	int	real	int	$v_4 = \text{real}$ eliminated
5	int	int	real	int	int	$v_3 = \text{real}$ eliminated
6	int	real	int	int	int	$v_2 = \text{real}$ eliminated
7	real	int	int	int	int	$v_1 = \text{real}$ eliminated

In only seven calls to the matching function, the unique solution is found and the algorithm terminates.

The implementation of this mechanism is not difficult. Since the LHS of a rule is limited to a single value from V_0, trying all possible combinations at the LHS of the rule can be implemented by iterating v_0 over V_0 as the innermost loop. The matching function $\mathcal{M}$ is modified to tag matches to type variables (so, for example if $\mathcal{M}(f(p_1, \ldots, p_n), \alpha, \Sigma) = \langle \Sigma', \text{match} \rangle$, then α is tagged as 'matched'). Before entering the innermost loop, tags to v_i ($i \geq 1$) are reset, and after the loop the tags are examined. If there is no tag on v_i ($i \geq 1$), then no match occurred for any element from V_0, and the value used by v_i can be eliminated.

After implementation of this optimization, the execution speed of the χ compiler improved noticeably. With large type-check problems such as with 13 parameters or more, the χ compiler does slow down, but compilation times stay in the order of a few seconds, mainly due to having to write the generated C++ code to disk or network. Compared with the C++ compiler that is executed next to convert the generated code to machine language, this is lightning fast.

APPENDIX

B

Promela code and verification results

The synchronous communication algorithm explained in Chapter 4 has been the subject of verification and simulation efforts (also explained in that chapter). The code below is a listing of the Promela programs used in these efforts, in order to allow verification of the verification and simulation results.

The algorithm itself is specified in algorithm3.m4. That file is used both in the verification and simulation efforts, with some small changes to tailor it. These changes are described in the next section. The files 2p1c.m4 and 3p2c.m4 are used to construct test cases with 2 processes and 1 channel, and 3 processes and 2 channels respectively. The file t46simulation.m4 is the test case with 4 processes and 6 channels, used in the simulation effort.

For compactness and readability, the macro processor *m4* is used to generate the promela code read by *spin*. The commands to perform the verification or simulation are shown just before the listing of each test case.

B.1 Algorithm

The algorithm3.m4 file implements the synchronous communication algorithm as described in Chapter 4. The following aspects can be changed:

- *Maximal number of channels of a process.* This number is set by the *MAXCOMMS* definition from the test case. Its value should match the number of alternatives in the *TryComms* macro. The file below is for maximal 3 channels connected to a process. To change this number, look for "assert(MAXCOMMS==3)" below, and follow the directions in the comment.

- *Initial ownership of channels.* As part of the initialization, each channel is initially assigned to a process. For verification of the algorithm, this should be decided non-deterministically. However, in some cases deterministic assignment suffices. Since the latter is a large reduction in the state space, the latter should be used whenever possible. The initialization is performed in *Chan_Init*. The file below uses non-deterministic initialization. To change it to deterministic initialization, search the macro definition, and follow the directions in the comment.

```
/* communication3.m4
** Generic communication code for processes with max 3 channels
**
** For a different maximal number of channels, find "assert(MAXCOMMS==3)"
** And change the macro to match MAXCOMMS.
*/

/* ---------------------------------------------------------------------
** Defines
*/
#define Semaphore  bool
#define SemFree    0
#define SemTaken   1

#define Answer     byte
#define AnsWait    100
#define AnsAck     108
#define AnsRefuse  109

#define ProcState  byte
#define Active     200
#define Trying     201
#define Pending    202
#define Idle       203

#define NOT_USED   255

/* ---------------------------------------------------------------------
** Structures
*/
typedef Channel {
        byte pab[2];
        Semaphore chansem;
        bit powner;     /* pab[powner] is owner */
};

typedef DeltaStorage {
        short deltavalue; /* In reality, this is a real value */
        byte altnumber; /* Number belonging to this alternative */
}

typedef CommStorage {
        byte channelidx;
        byte altnumber;
}

dnl Process storage
typedef Storage {
        DeltaStorage ds;        /* Mimimal delta alternative value) */
        bit numdeltas;          /* Number of delta's (either 0 or 1) */
```

```
        CommStorage cs[ MAXCOMMS ]; /* Comm alternatives */
        byte numcomms;          /* Number of comm alternatives */
}

typedef Process {
        ProcState s;
        byte c;         /* Current channel idx */
        byte dc;        /* Delayed channel. 255=NOT-USED */
        Answer ans;
        Storage stor;
        Semaphore procsem;
        Semaphore bs;
}

/* ----------------------------------------------------------------------
** Globals
*/
Channel channels[ NUMCHANNELS ]

Process processes[ NUMPROCESSES ]

/* ----------------------------------------------------------------------
** Semaphores
*/

define(Sem_Init,$1=$2)dnl
dnl 1=sem, 2=initial value

define(Sem_P,atomic { ($1==SemFree); $1=SemTaken })dnl
dnl 1=semaphore

define(Sem_V,atomic { assert($1==SemTaken); $1=SemFree })dnl
dnl 1=semaphore

/* Channel semaphores
*/
define(Chan_P,Sem_P(channels[$1].chansem))dnl
dnl 1=chanidx

define(Chan_V,Sem_V(channels[$1].chansem))dnl
dnl 1=chanidx

/* Process semaphores
*/
define(Proc_P,Sem_P(processes[$1].procsem))dnl
dnl 1=process-idx

define(Proc_V,Sem_V(processes[$1].procsem))dnl
dnl 1=process-idx

/* ----------------------------------------------------------------------
```

```
** Channel definitionss
*/

define(Chan_Init,channels[$1].pab[0]=$2; channels[$1].pab[1]=$3;
                if
                :: true -> channels[$1].powner=0
                :: true -> channels[$1].powner=1
                fi;
                Sem_Init(channels[$1].chansem,SemFree)
                /* assert($2 != $3) */ )dnl
dnl 1=channel-idx, 2=process_a-idx, 3=process_b-idx
dnl $2 should not be $3
dnl
dnl For deterministic initial ownership, replace the if statement with
dnl channels[$1].powner=0;

define(Chan_Owner,($2==channels[$1].pab[channels[$1].powner]))dnl
dnl 1=chanidx 2=procid of caller

/* ------------------------------
** Process definitionss
*/

define(Proc_Init,Sem_Init(processes[$1].procsem,SemFree);
                Sem_Init(processes[$1].bs, SemTaken);
                processes[$1].s = Active; processes[$1].c=NOT_USED;
                processes[$1].dc=NOT_USED; Storage_Init(processes[$1].stor))dnl
dnl 1=process-idx

define(Proc_ClearCommDelta,Storage_Init(processes[$1].stor))dnl
dnl 1=process-idx

define(Proc_AddDelta,Storage_AddDelta(processes[$1].stor,$2,$3))dnl
dnl 1=process-idx, 2=delta-value, 3=alternative number

define(Proc_AddComm,Storage_AddComm(processes[$1].stor,$2,$3))dnl
dnl 1=process-idx, 2=channel-idx, 3=alternative number

define(Proc_FinalAnswer, Proc_P($1); processes[$1].ans = $3;
                        if
                        :: ($3==AnsAck) -> skip
                        :: ($3==AnsRefuse)
                           -> run ChanChangeOwner($2,$1,mydummy); mydummy?dummy
                        fi;
                        Sem_V(processes[$1].bs); Proc_V($1))dnl
dnl 1=procidx, 2=chanidx, 3=theanswer

define(Proc_Stor,processes[$1].stor)dnl
dnl 1=procidx

/* ------------------------------
```

```
** Alternatives storage
*/

define(Storage_Init,$1.numdeltas=0; $1.numcomms=0)dnl
dnl 1=storage structure

define(Storage_AddDelta,if
                        :: ($1.numdeltas == 0)
                           -> $1.ds.deltavalue=$2; $1.ds.altnumber=$3; $1.numdeltas=1
                        :: ($1.numdeltas == 1 && $1.ds.deltavalue<=$2) -> skip
                        :: ($1.numdeltas == 1 && $1.ds.deltavalue>$2)
                           -> $1.ds.deltavalue=$2; $1.ds.altnumber=$3
                        fi)dnl
dnl 1=storage structure, 2=deltavalue, 3=alternative number

define(TryComms,/* assert(MAXCOMMS==3); */
                if
                :: ($1.numcomms>=1 && $1.cs[0].channelidx==$2) -> $3
                :: ($1.numcomms>=2 && $1.cs[1].channelidx==$2) -> $3
                :: ($1.numcomms>=3 && $1.cs[2].channelidx==$2) -> $3
                :: else -> $4
                fi)dnl
dnl 1=storage-structure 2=ch-idx 3=true-activity 4=false-activity
dnl This macro should be adapted if MAXCOMMS changes.
dnl There should be an alternative for each i in this macro, for 0<=i<MAXCOMMS
dnl of the form ":: ($1.numcomms>=(i+1) && $1.cs[i].channelidx==$2) -> $3"
dnl with "(i+1)" and "i" replaced by its value

define(Storage_AddComm, TryComms($1,$2,skip,$1.cs[$1.numcomms].channelidx=$2;
                                    $1.cs[$1.numcomms].altnumber=$3;
                                    $1.numcomms++;
                                    assert($1.numcomms<=MAXCOMMS)))dnl
dnl 1=storage structure, 2=channelidx, 3=alternative number


/* ---------------------------------------------------------------------
** Channel processes
*/

proctype ChanChangeOwner(byte chanidx; byte procid; chan res)
{
        /* Only the owner should call this process & only if protected */
        atomic {
                assert(Chan_Owner(chanidx,procid));
                assert(channels[chanidx].chansem == SemTaken)
        }
        channels[chanidx].powner = 1-channels[chanidx].powner;
        res!0   /* Return something random */
}

proctype ChanFinalAnswer(byte chanidx; byte procid; chan res; Answer theans)
```

```
{
        chan mydummy = [0] of { bit };
        bit dummy;

        /* May only be called when taken */
        assert(channels[chanidx].chansem == SemTaken);

        if
        :: (channels[chanidx].pab[0] == procid)
           -> Proc_FinalAnswer(channels[chanidx].pab[1],chanidx,theans)
        :: (channels[chanidx].pab[1] == procid)
           -> Proc_FinalAnswer(channels[chanidx].pab[0],chanidx,theans)
        fi;
        res!0        /* Return something random */
}

proctype ChanRemoteComm(byte chanidx; byte procid; chan res)
{
        chan myres = [0] of {Answer};
        Answer cr;

        /* May only be called when taken & when owned */
        atomic {
                assert(channels[chanidx].chansem == SemTaken);
                assert(Chan_Owner(chanidx,procid));
        }

        if
        :: (channels[chanidx].pab[0] == procid)
           -> run ProcRemoteComm(channels[chanidx].pab[1], chanidx, myres)
        :: (channels[chanidx].pab[1] == procid)
           -> run ProcRemoteComm(channels[chanidx].pab[0], chanidx, myres)
        fi;
        myres?cr; res!cr
}

/* ---------------------------------------------------------------------
** Process processes
*/

proctype ProcRemoteComm(byte procidx; byte chanidx; chan res)
{
        Proc_P(procidx);
        if
        :: (processes[procidx].s == Active) -> Proc_V(procidx); res!AnsRefuse
        :: (processes[procidx].s == Idle)
           -> if
              :: (processes[procidx].dc == NOT_USED)
                 -> TryComms(Proc_Stor(procidx),chanidx,
                                processes[procidx].dc=chanidx;
                                Sem_V(processes[procidx].bs);
```

```
                                Proc_V(procidx); res!AnsAck
                               ,Proc_V(procidx); res!AnsRefuse)
            :: (processes[procidx].dc != NOT_USED)
               -> Proc_V(procidx); res!AnsRefuse
            fi
        :: (processes[procidx].s == Trying || processes[procidx].s == Pending)
           -> if
              :: (processes[procidx].dc == NOT_USED)
                 -> TryComms(Proc_Stor(procidx)
                             ,chanidx,if
                                      :: (chanidx > Proc_Stor(procidx).
                                                cs[ processes[procidx].c ].
                                                channelidx)
                                        -> Proc_V(procidx); res!AnsRefuse
                                      :: (chanidx <= Proc_Stor(procidx).
                                                cs[ processes[procidx].c ].
                                                channelidx)
                                        -> processes[procidx].dc=chanidx;
                                           Proc_V(procidx); res!AnsWait
                                     fi
                             ,Proc_V(procidx); res!AnsRefuse)
              :: (processes[procidx].dc != NOT_USED)
                 -> Proc_V(procidx); res!AnsRefuse
              fi
        fi
}

define(Refuse_DelayedChan,if
                          :: (processes[$1].dc == NOT_USED) -> skip
                          :: (processes[$1].dc != NOT_USED)
                             -> Chan_P(processes[$1].dc);
                                run ChanFinalAnswer(processes[$1].dc,$1,mydummy,
                                                    AnsRefuse);
                                mydummy?dummy;
                                Chan_V(processes[$1].dc);
                                processes[$1].dc = NOT_USED
                          fi)dnl
dnl 1=procidx

/* return values: 7=fatal error
** 9=no alternative chosen
*/
proctype Proc_DoSelWait(byte procidx; chan res)
{
        byte curchan,result;
        chan myans = [0] of {Answer};
        Answer a;
        chan mydummy = [0] of { bit };
        bit dummy;

        atomic {
```

```
            result = 199;   /* To be sure of an error if not assigned */
            /* Paranoia checking */
            if
            :: (Proc_Stor(procidx).numcomms == 0) -> skip
            :: (Proc_Stor(procidx).numcomms == 1) -> skip
            :: (Proc_Stor(procidx).numcomms == 2 &&
                    Proc_Stor(procidx).cs[0].channelidx
                    < Proc_Stor(procidx).cs[1].channelidx) -> skip
            :: (Proc_Stor(procidx).numcomms == 3 &&
                    Proc_Stor(procidx).cs[0].channelidx
                    < Proc_Stor(procidx).cs[1].channelidx
                && Proc_Stor(procidx).cs[1].channelidx
                    < Proc_Stor(procidx).cs[2].channelidx) -> skip
            :: else -> assert(0)
            fi
    }

    if      /* if 0 */
    :: (Proc_Stor(procidx).numdeltas == 1 &&
        Proc_Stor(procidx).ds.deltavalue < 0)
         -> atomic { result=7; goto swdone  } /* Negative delta !! */
    :: (Proc_Stor(procidx).numdeltas == 1 &&
        Proc_Stor(procidx).ds.deltavalue == 0)
         -> atomic { result=Proc_Stor(procidx).ds.altnumber; goto swdone }
    :: (Proc_Stor(procidx).numdeltas == 0 ||
         (Proc_Stor(procidx).numdeltas == 1 &&
          Proc_Stor(procidx).ds.deltavalue > 0))
         -> Proc_P(procidx);
            if      /* if 1 */
            :: (Proc_Stor(procidx).numcomms == 0 &&
                Proc_Stor(procidx).numdeltas == 0)
                                                  /* no alternative available */
                 -> Proc_V(procidx); atomic { result=9; goto swdone }
            :: (Proc_Stor(procidx).numcomms == 0 &&
                Proc_Stor(procidx).numdeltas == 1)
                 -> Proc_V(procidx);
                    /* DELTA Proc_Stor(procidx).ds.deltavalue */
                    atomic { result=Proc_Stor(procidx).ds.altnumber; goto swdone }
            :: (Proc_Stor(procidx).numcomms > 0)
                 -> processes[procidx].s = Trying;
                    processes[procidx].c = 0;
                    do      /* iterate over each channel */
                    :: (processes[procidx].c >= Proc_Stor(procidx).numcomms)
                         -> break
                    :: (processes[procidx].c < Proc_Stor(procidx).numcomms)
                         -> curchan = Proc_Stor(procidx).
                                      cs[ processes[procidx].c ].
                                      channelidx;
                            Proc_V(procidx); Chan_P(curchan);
                            if      /* if 2 */
                            :: (!Chan_Owner(curchan,procidx)) -> Chan_V(curchan)
```

```
:: (Chan_Owner(curchan,procidx))
   -> run ChanRemoteComm(curchan,procidx,myans);
      myans?a;
      if        /* if 3 */
      :: (a==AnsAck)
           -> Chan_V(curchan); Proc_P(procidx);
              processes[procidx].s = Active;
              Refuse_DelayedChan(procidx);
              result=Proc_Stor(procidx).
                     cs[ processes[procidx].c ].
                     altnumber;
              Proc_V(procidx); goto swdone
      :: (a==AnsRefuse)
           -> run ChanChangeOwner(curchan,procidx,
                                  mydummy);
              mydummy?dummy; Chan_V(curchan)
      :: (a==AnsWait)
           -> Proc_P(procidx);
              processes[procidx].s = Pending;
              Chan_V(curchan); Proc_V(procidx);
              Sem_P(processes[procidx].bs);
              Proc_P(procidx);
              if        /* if 4 */
              :: (processes[procidx].ans == AnsAck)
                 -> processes[procidx].s = Active;
                    Refuse_DelayedChan(procidx);
                    result=Proc_Stor(procidx).
                           cs[ processes[procidx].c ].
                           altnumber;
                    Proc_V(procidx); goto swdone
              :: (processes[procidx].ans == AnsRefuse)
                 -> processes[procidx].s = Trying;
                    /* Channel owner already changed */
                    Proc_V(procidx)
              fi /* fi 4 */
      fi /* fi 3 */
fi; /* fi 2 */
Proc_P(procidx);
if /* if 5 */
:: (processes[procidx].dc == NOT_USED) -> skip
:: (processes[procidx].dc != NOT_USED)
   -> processes[procidx].s = Active;
      curchan = processes[procidx].dc;
      processes[procidx].dc=NOT_USED;
      Proc_V(procidx); Chan_P(curchan);
      run ChanFinalAnswer(curchan,procidx,mydummy,
                          AnsAck);
      mydummy?dummy; Chan_V(curchan);
      if
      :: (Proc_Stor(procidx).numcomms>=1 &&
          Proc_Stor(procidx).cs[0].channelidx==curchan)
```

```
                    -> atomic { result=Proc_Stor(procidx).cs[0].
                                        altnumber; goto swdone }
                 :: (Proc_Stor(procidx).numcomms>=2 &&
                     Proc_Stor(procidx).cs[1].channelidx==curchan)
                    -> atomic { result=Proc_Stor(procidx).cs[1].
                                        altnumber; goto swdone }
                 :: (Proc_Stor(procidx).numcomms>=3 &&
                     Proc_Stor(procidx).cs[2].channelidx==curchan)
                    -> atomic { result=Proc_Stor(procidx).cs[2].
                                        altnumber; goto swdone }
                 :: else -> assert(0)
                 fi
        fi; /* fi 5 */
        processes[procidx].c++;
od;
processes[procidx].s = Idle; Proc_V(procidx);
if /* if 6 */
:: (Proc_Stor(procidx).numdeltas == 0)
   -> if
      :: Sem_P(processes[procidx].bs) -> skip
      :: timeout -> /* FORCED END OF THE SW */
                assert(processes[procidx].dc == NOT_USED);
                atomic { result=6; goto swdone }
      fi
:: (Proc_Stor(procidx).numdeltas == 1)
   -> if
      :: Sem_P(processes[procidx].bs) -> skip
      :: (processes[procidx].bs == SemTaken) -> skip
      fi
fi; /* fi 6 */
Proc_P(procidx); processes[procidx].s = Active;
if /* if 7 */
:: (processes[procidx].dc == NOT_USED)
   -> Proc_V(procidx);
      atomic { result=Proc_Stor(procidx).ds.altnumber;
               goto swdone }
:: (processes[procidx].dc != NOT_USED)
   -> curchan = processes[procidx].dc;
      processes[procidx].dc=NOT_USED;
      Proc_V(procidx);
      if /* if 8 */
      :: (Proc_Stor(procidx).numcomms>=1 &&
          Proc_Stor(procidx).cs[0].channelidx==curchan)
         -> atomic { result=Proc_Stor(procidx).cs[0].
                           altnumber; goto swdone }
      :: (Proc_Stor(procidx).numcomms>=2 &&
          Proc_Stor(procidx).cs[1].channelidx==curchan)
         -> atomic { result=Proc_Stor(procidx).cs[1].
                           altnumber; goto swdone }
      :: (Proc_Stor(procidx).numcomms>=3 &&
          Proc_Stor(procidx).cs[2].channelidx==curchan)
```

```
                                -> atomic { result=Proc_Stor(procidx).cs[2].
                                                        altnumber; goto swdone }
                            :: else -> assert(0)
                            fi /* fi 8 */
                        fi /* fi 7 */
                fi /* fi 1 */
        fi; /* fi 0 */
swdone:
        res!result
}
```

B.2 Verification

In this section, two test cases used in the verification effort are listed. The first test case has two processes and one channel, the second test case has three processes and two channels. The verification result of each test case is listed directly after the test case itself.

Starting with the test case of two processes and one channel, the following commands perform the verification:

```
$ m4 2p1c.m4 > 2p1c.prom
$ spin -a 2p1c.prom
$ gcc -O2 -o 2p1c -DMEMCNT=28 -DCOLLAPSE -DSAFETY pan.c
$ ./2p1c -m330
```

```
/* 2p1c.m4
** Verification of 2 processes, 1 channel
*/
define(NUMCHANNELS,1)dnl
define(NUMPROCESSES,2)dnl
define(MAXCOMMS,3)dnl

define(CH1,0)dnl

/* Make initial ownership deterministic. Both cases are checked because
** the configuration of the processes is symmetric.
*/
include(communication3.m4)


proctype Pi(byte procidx; byte ch; chan ichan)
{
        chan myans = [0] of { byte };
        byte i;
```

```
        Proc_ClearCommDelta(procidx);
        Proc_AddComm(procidx,ch,1);
        run Proc_DoSelWait(procidx,myans);
        myans?i;
        assert(i==1 || i==6);
        ichan!i
}

init
{
        chan p0r = [0] of { byte };
        chan p1r = [0] of { byte };
        byte i0,i1;

        atomic {
                Chan_Init(CH1,0,1);
                Proc_Init(0); Proc_Init(1);
        }
        run Pi(0,CH1,p0r); run Pi(1,CH1,p1r);
        p0r?i0; p1r?i1;

        atomic {
                if
                :: (i0==1) -> assert(i1==1)
                :: else ->
                fi;

                if
                :: (i1==1) -> assert(i0==1)
                :: else ->
                fi;
        }
}
```

The verification ended successfully. The output of the verifier is shown below, except for the list of unreached states. These states handle the case when a third process interferes with a communication attempt between two processes. Since this verification attempt only has two processes, interference does not occur, and the states that handle interference are not used.

```
(Spin Version 3.3.4 -- 9 September 1999)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid endstates       +
```

```
State-vector 191 byte, depth reached 136, errors: 0
  194262 states, stored
  206470 states, matched
  400732 transitions (= stored+matched)
    5704 atomic steps
hash conflicts: 75842 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
39.435  equivalent memory usage for states (stored*(State-vector + overhead))
6.859   actual memory usage for states (compression: 17.39%)
        State-vector as stored = 23 byte + 12 byte overhead
1.049   memory used for hash-table (-w18)
0.008   memory used for DFS stack (-m330)
8.012   total actual memory usage

nr of templates: [ globals procs chans ]
collapse counts: [ 23128 20 156 104 97 604 48 7 ]
```

A more interesting test case is three processes connected together with two channels. This test case should use non-deterministic initial ownership of channels to cover all cases. As explained in Chapter 4, this verification fails, even with deterministic assignment of channels.

The following commands perform the verification:

```
$ m4 3p2c.m4 > 3p2c.prom
$ spin -a 3p2c.prom
$ gcc -O2 -o 3p2c -DMA=276 -DMEMLIM=1000 -DCOLLAPSE -DSAFETY pan.c
$ ./3p2c -m500
```

```
/* 3p2c.m4
** Verification of 3 processes, 2 channels
**
** O---O---O
*/
define(NUMCHANNELS,2)dnl
define(NUMPROCESSES,3)dnl
define(MAXCOMMS,2)dnl

define(CH01,0)dnl
define(CH02,1)dnl

/* MAXCOMMS!=3, change the lines in communicationX.m4 !!
** Initial ownerhsip of channels should be non-deterministic, but even with
** deterministic initial ownership, the state space is too big
```

```
*/
include(communication2.m4)

/* Both ends */
proctype Pi(byte procidx; byte cha)
{
        chan myans = [0] of { byte };
        byte i;

        Proc_ClearCommDelta(procidx);
        Proc_AddComm(procidx,cha,1);
        run Proc_DoSelWait(procidx,myans);
        myans?i;
        assert(i==1)
}

proctype X(byte procidx; byte cha; byte chb)
{
        chan myans = [0] of { byte };
        byte x;
        bit i,j;

        Proc_ClearCommDelta(procidx);
        Proc_AddComm(procidx,cha,0);
        Proc_AddComm(procidx,chb,1);
        run Proc_DoSelWait(procidx,myans);
        myans?x; atomic { assert(x==0 || x==1); i=x; x=0; }
        Proc_ClearCommDelta(procidx);
        Proc_AddComm(procidx,cha,0);
        Proc_AddComm(procidx,chb,1);
        run Proc_DoSelWait(procidx,myans);
        myans?x; atomic { assert(x==0 || x==1); j=x; x=0; }
        assert(i!=j)
}

init
{
        chan p0r = [0] of { byte };
        chan p1r = [0] of { byte };
        chan p2r = [0] of { byte };
        byte i0,i1,i2;

        atomic {
                Chan_Init(CH01,0,1); Chan_Init(CH02,0,2);
                Proc_Init(0); Proc_Init(1); Proc_Init(2)
        }
        run X(0,CH01,CH02); run Pi(1,CH01); run Pi(2,CH02);
}
```

The verifier fails due to lack of memory. The output fragment with respect to this event

is:

```
Depth=306 States=7.8e+07 Transitions=1.92303e+08 Nodes=7017620 Memory=927.950
Depth=306 States=7.9e+07 Transitions=1.94801e+08 Nodes=7075341 Memory=937.371
pan: out of memory
        9.39009e+08 bytes used
        102400 bytes more needed
        1.04858e+09 bytes limit (2^MEMCNT)
```

The verifier also outputs the following memory statistics:

```
Stats on memory usage (in Megabytes):
25020.841 equivalent memory usage for states (stored*(State-vector + overhead))
937.854 actual memory usage for states (compression: 3.75%)
1.049   memory used for hash-table (-w18)
0.012   memory used for DFS stack (-m500)
939.009 total actual memory usage
```

Note that the verifier has been given 1000MB (`-DMEMLIM=1000`), while it actually used 939MB. The difference between these two numbers comes from the fact that the latter number is the upper memory limit enforced by the operating system. In other words, the verifier aborts because it is not given more memory.

B.3 Simulation

After exhausting the verification options, a switch has been made to Monte Carlo simulations, in order to test the implementation as much as possible. Unlike the verification, non-determinism is not a problem here. In fact, more non-determinism gives the opportunity to more different behaviour, and that increases the chances of finding bugs.

The Promela program used in the simulation contains four processes and six channels (each process is connected to each other process). Also, initial ownership of channels is assigned non-deterministically.

The Promela code is generated by executing:

```
$ m4 t46simulation.m4 > p.prom
```

A single simulation is performed by executing:

```
$ spin -nX p.prom
```

with `X` a number from the range 1–217200000, which is used as the initial seed for the random number generator of spin. All simulations ran to completion without errors.

```
/* t46simulation.m4
** Top level test file used for simulation
** Initial ownership of channels was non-deterministic
**
*/
define(NUMCHANNELS, 6)dnl
define(NUMPROCESSES,4)dnl
define(MAXCOMMS,3)dnl
dnl USE DIFFERENT communicationX.m4 if MAXCOMMS!=3
dnl
dnl #channels, #processes, max #communications in a single process

dnl Aliases for channels and processes
define(CH01,0)dnl
define(CH02,1)dnl
define(CH03,2)dnl
define(CH12,3)dnl
define(CH13,4)dnl
define(CH23,5)dnl

define(P0,0)dnl
define(P1,1)dnl
define(P2,2)dnl
define(P3,3)dnl

/* ---------------------------------------------------------------------
** Include the communication algorithm
**
** Be sure to modify the initial ownership of channels to non-deterministic
** for simulation
*/

include(‘communication3.m4’)


/* ---------------------------------------------------------------------
** Instantiation of 4 processes connected with 6 channels (1 channel between
** each pair of processes)
*/

proctype P(byte procidx; byte cha, chb, chc; chan ichan)
{
        chan myans = [0] of {byte};
        byte i;
        bool b0,b1,b2,b3;

        /* Proc_P() ? */
        Proc_ClearCommDelta(procidx);
        /* b0; delta 2 -> i:=0 */
        if
        :: true -> Proc_AddDelta(procidx,2,0); b0=true
```

```
        :: true -> b0=false
        fi;
        /* b1; cha~ -> i:=1 */
        if
        :: true -> Proc_AddComm(procidx,cha,1); b1=true
        :: true -> b1=false
        fi;
        /* b2; chb~ -> i:=2 */
        if
        :: true -> Proc_AddComm(procidx,chb,2); b2=true
        :: true -> b2=false
        fi;
        /* b3; chc~ -> i:=3 */
        if
        :: true -> Proc_AddComm(procidx,chc,3); b3=true
        :: true -> b3=false
        fi;
        run Proc_DoSelWait(procidx,myans);
        myans?i;

        atomic {
                if
                :: (i==0) -> assert(b0)
                :: (i==1) -> assert(b1)
                :: (i==2) -> assert(b2)
                :: (i==3) -> assert(b3)
                :: (i==6) -> assert(!b0 && (b1 || b2 || b3))    /* FORCED END OF SW */
                :: (i==9) -> assert(b0==false && b1 == false &&
                                        b2 == false && b3 == false)
                fi
        }

        ichan!i
}

init
{
        chan p0ret = [0] of { byte };
        chan p1ret = [0] of { byte };
        chan p2ret = [0] of { byte };
        chan p3ret = [0] of { byte };
        byte p0i,p1i,p2i,p3i;

        d_step { /*assert(NUMCHANNELS==6);*/
                Chan_Init(CH01, P0, P1); Chan_Init(CH02, P0, P2);
                Chan_Init(CH03, P0, P3); Chan_Init(CH12, P1, P2);
                Chan_Init(CH13, P1, P3); Chan_Init(CH23, P2, P3);

                /*assert(NUMPROCESSES==4);*/
                Proc_Init(P0); Proc_Init(P1); Proc_Init(P2); Proc_Init(P3)
        }
```

```
        run P(P0,CH01,CH02,CH03,p0ret);
        run P(P1,CH01,CH12,CH13,p1ret);
        run P(P2,CH02,CH12,CH23,p2ret);
        run P(P3,CH03,CH13,CH23,p3ret);
        p0ret?p0i; p1ret?p1i; p2ret?p2i; p3ret?p3i;

        atomic {
                if
                :: (p0i==1) -> assert(p1i==1)
                :: (p0i==2) -> assert(p2i==1)
                :: (p0i==3) -> assert(p3i==1)
                :: else -> skip
                fi;

                if
                :: (p1i==1) -> assert(p0i==1)
                :: (p1i==2) -> assert(p2i==2)
                :: (p1i==3) -> assert(p3i==2)
                :: else -> skip
                fi;

                if
                :: (p2i==1) -> assert(p0i==2)
                :: (p2i==2) -> assert(p1i==2)
                :: (p2i==3) -> assert(p3i==3)
                :: else -> skip
                fi;

                if
                :: (p3i==1) -> assert(p0i==3)
                :: (p3i==2) -> assert(p1i==3)
                :: (p3i==3) -> assert(p2i==3)
                :: else -> skip
                fi
        }

}
```

BIBLIOGRAPHY

[Are96] N.W.A. Arends. *A Systems Engineering Specification Formalism*. PhD thesis, Technische Universiteit Eindhoven, 1996.

[Bag89] Rajive Bagrodia. Synchronization of asynchronuous processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, October 1989.

[Bar99] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly, January 1999.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design of embedded systems: The POLIS approach*. Kluwer academic publishers, 1997.

[BK00] V. Bos and J.J.T. Kleijn. Formalisation of a production systems modelling language: the operational semantics of χ core. *Fundamenta Informaticae*, 41(4):367–392, 2000.

[Bra93] L.E.M.W. Brandts. *Design of industrial systems*. PhD thesis, Technische Universiteit Eindhoven, 1993.

[Bri97] Klaas Brink. *Interfacing Control and Software Engineering: a formal approach*. PhD thesis, Delft University of Technology, 1997.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[BS83] Gael N. Buckley and Abraham Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[Che99a] N.Z. Chen. Design and implementation of the chi 0.6 kernel. Systems Engineering report SE 420219, Technische Universiteit Eindhoven, July 1999. Detailed Design Document for the Chi 0.6 redesign project.

[Che99b] N.Z. Chen. Redesign of the χ kernel. OOTI report, Technische Universiteit Eindhoven, June 1999.

[Coo91] J.E. Cooling. *Software Design for Real-time Systems*. Chapman and Hall, 1991.

[dBvdBCP97] A. Gouder de Beauregard, B. van den Broek, R. Caelers, and R Penners. Migration of χ towards a real-time operating system. Technical report, Technische Universiteit Eindhoven, 1997. Department of Mathematics and Computing Science.

[DdlBS99] B. Demoen, M. Garcia de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In J. Edwards, editor, *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228. Springer-Verlag, January 1999.

[Epp97] Jerry Epplin. Linux as an embedded operating system. *Embedded Systems Programming*, October 1997.

[Fáb99] Georgina Fábián. *A Language and Simulator for Hybrid Systems*. PhD thesis, Technische Universiteit Eindhoven, 1999.

[Gup95] Rajesh Kumar Gupta. *Co-synthesis of hardware and software for digital embedded systems*. Kluwer academic publishers, 1995.

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer academic publishers, 1993.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[Hol97] Gerard J. Holzmann. The spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hoo91] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems*. PhD thesis, Technische Universiteit Eindhoven, 1991.

[HP88] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing, 353 West 12th Street New York, New York 10014, 1988.

[HS91] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing predictable real time systems*. Kluwer academic publishers, 1991.

[JM86] Albert T. Jones and Charles R. McLean. A proposed hierarchical control model for automated manufacturing systems. *Journal of Manufacturing Systems*, 5(1):15–25, 1986.

[Kam99] J. C. A. Kamp. χ-modellen simuleren met vxworks. Onderzoeksopdracht, Technische Universiteit Eindhoven, January 1999. SE 42201.

[Kar98] Pim Kars. *Formal Methods in the Design of a Storm Surge Barrier Control System*, pages 353–367. Number 1494 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[KS97] Devendra Kumar and Abraham Silberschatz. A counter-example to an algorithm for the generalized input–output construct of CSP. *Information Processing Letters*, 61(6):287, 28 March 1997.

[KZ96] Andrew J. Kornecki and Janusz Zalewski. *Real-Time Ssystems Education*, pages 73–79. IEEE Computer Society Press, 1996.

[LM87] Peter D. Lawrence and Konrad Mauch. *Real-time microcomputer system design*. McGraw-Hill, inc, 1987.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation and Electronic Systems*, 4(3):257–279, July 1999.

[Mar67] James Martin. *Design of Real-Time Computer Systems*. Prentice-Hall, 1967.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17:348–375, 1978.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems*. Springer-Verlag, 1995.

[NA98] G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. PhD thesis, Technische Universiteit Eindhoven, 1998.

[Ove87] R. Overwater. *Process and Interactions. An approach to the modelling of industrial systems*. PhD thesis, Technische Universiteit Eindhoven, 1987.

[QNX96] QNX Software Systems Ltd., 175 Terence Mathews Crescent, Kanata Ontario, K2M 1W8, Canada. *QNX Operating System: System Architecture*, 1996.

[Ray96] Eric S. Raymond, editor. *The New Hacker's Dictionary*. MIT Press, third edition, 1996.

[Sha75] Robert E. Shannon. *Sytems Simulation, the art and science*. Prentice-Hall, 1975.

[vBR96] D. A. van Beek and J. E. Rooda. A new mechanism for exception handling in concurrent control systems. *European Journal of Control*, 2(2):88–100, 1996.

[vD00] M.H.M. van Duin. From simulation using χ to implementation using vxworks, a case: The paint factory. Master's thesis, Technische Universiteit Eindhoven, 2000. SE 420225.

[vdKS98] H.H. van den Kroonenberg and F.J. Siers. *Methodisch ontwerpen*. Educatieve Partners Nederland, 1998. (Dutch).

[vdS93] Jan L.A. van de Snepscheut. *What computing is all about*. Springer-Verlag, 1993.

[Wie98] F. W. Wiekmeyer. Using vxworks for workstation control. Resarch project report, Technische Universiteit Eindhoven, December 1998. SE 420205.

[Win97] Wind River Systems, Inc. *VxWorks, Programmer's Guide*, first edition, 1997. VxWorks version 5.3.1.

[Yod] Victor Yodaiken. The rtlinux manifesto. http://www.rtlinux.org/.

INDEX

Curriculum vitae

Albert Theo Hofkamp was born on the 25th of July, 1967 in Leeuwarden. In 1985, he finished the *atheneum* in Leeuwarden, followed by a study in Electrical Engineering, which was successfully finished in 1989. After serving in the Army, he studied Computer Science at the University of Twente from 1991 to 1995. He graduated within the Tools Group of TIOS (Tele-Informatics and Open Systems) on the subject A static semantics checker for LOTOS, and wrote *lcr*. After graduating, he continued with the OOTI programme (Postgraduate designer course Software Technology) at the Eindhoven University of Technology (TUE). The final project Data modeling in χ for this programme was done at the department of Mechanical Engineering, with the Systems Engineering Group at the same university.

In December 1997, he started his PhD project on the design of real-time systems.

Titles in the IPA Dissertation Series

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ.* Faculty of Mechanical Engineering, TUE. 2001-13