University of Twente

EEMCS / Electrical Engineering
Control Engineering

Design Space Exploration for Fieldbus-based Distributed Control Systems

Matthijs ten Berge

M.Sc. Thesis

Supervisors
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
ir. B. Orlic

August 2005

Report nr. 029CE2005
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Summary

Development of embedded control systems at the Control Engineering chair takes a structured approach. It is supported by a toolchain consisting of both software and hardware tools that can be used in the various stages of the design process. The available tools range from modeling and simulation software to a hardware-in-the-loop simulator.

However, support for the implementation of control systems on *distributed* control systems is limited. Distributed systems consist of multiple processor nodes, interconnected by a fieldbus (or other communication solutions). As the fieldbus is part of the control loop, it will affect the behavior of the control system. With the existing toolchain, this effect is unknown until the complete control system is realized and measurements can be performed.

Implementation of *embedded* control systems can be complex and time consuming, as the implemented software solution must be validated on the actual target hardware. Embedded software is often difficult to debug, this depends on the debugging features of the development tools for the embedded platform.

The goal of this project is to design a new tool, which can be integrated in the existing toolchain, to aid in the verification and design space exploration during the implementation phase of a distributed embedded control system. This new tool, the *Design Space Exploration (DSE) simulation environment*, enables the simulation of a complete control system on a PC. It can be used during the implementation phase of the development process, and allows the designer to verify the correctness of the control system implementation and to choose the optimal parameters for all degrees of freedom (this is the design space exploration).

The DSE simulation environment is based on the Communicating Threads (CT) C++ library, which is an implementation of the Communicating Sequential Processes (CSP) language. This library supports a number of different platforms, which makes it possible to test software targeted at embedded hardware on a normal PC.

The DSE simulation environment incorporates a modified version of the TrueTime network simulator, allowing simulation of distributed systems. Support for network communication is further extended by the addition of Remote Channels to the CT C++ library. These Channels can be used to turn a normal CT Channel into a networked variant. In the Remote Channel concept a separation is made between parts that are specific to a single channel, to the communication hardware or to a single communication interface. This separation makes the addition of support for new hardware relatively easy, and it allows multiple channels to be routed over the same network interface, independent of the network hardware used.

To get the results from a simulation, a logging implementation has been made. Although this implementation is sufficient for use in simulations, it is not suitable for real-time logging. A logging concept is presented, which can be used both for real-time and simulation logging.

A demonstration setup, consisting of a distributed controller on two ADSP processor boards and the Linix plant, was simulated with the DSE simulation environment. This case shows how a simulation of a complete distributed control system can be constructed. Furthermore a smaller simulation setup was created, which shows an example of how the DSE simulation environment can be used for measurements on different communication protocols.

Future work would include a better integration of the simulation environment with the other tools available, such as gCSP as a tool to graphically structure simulations. Extending the functionality of the DSE simulation environment, for instance by adding support for more network hardware or by adding support for simulation of execution time, is also recommended.

Samenvatting

De leerstoel Control Engineering gebruikt een gestructureerde benadering voor het ontwerp van embedded regelsystemen, welke wordt ondersteund door hardware- en softwaregereedschappen, die worden ingezet tijdens de verschillende stadia in het ontwerpproces. De beschikbare hulpmiddelen variëren van modelleer- en simulatiesoftware tot hardware-in-the-loop simulatoren.

Ondersteuning voor de implementatie van *gedistribueerde* regelsystemen is echter beperkt. Gedistribueerde systemen zijn systemen bestaande uit meerdere processors, die met elkaar verbonden zijn door middel van een veldbus (of andere communicatiemiddelen). Omdat de veldbus deel uitmaakt van de regellus, zal het gedrag van het totale regelsysteem erdoor worden beïnvloed. Met de bestaande verzameling van ontwerphulpmiddelen worden de gevolgen van deze beïnvloeding pas duidelijk nadat het hele regelsysteem is gerealiseerd en er metingen aan kunnen worden verricht.

De implementatie van een *embedded* regelsysteem kan complex en tijdrovend zijn, vooral doordat de resulterende softwareoplossing op het echte doelsysteem moet worden gevalideerd. Dit is zeer afhankelijk van de mate waarin de ontwikkelsoftware voor het doelsysteem hierin ondersteuning biedt.

Doel van dit project is om een nieuw gereedschap te ontwikkelen, dat geïntegreerd kan worden in de bestaande verzameling gereedschappen, waarmee de verificatie en design space exploration van een gedistribueerd embedded regelsysteem kan worden vereenvoudigd. Dit nieuwe gereedschap, de *Design Space Exploration (DSE) simulatieomgeving*, maakt het mogelijk om een volledig regelsysteem te simuleren op een PC. De simulatieomgeving kan gebruikt worden tijdens de implementatiefase van het ontwerp, waardoor mogelijkheden worden geboden om reeds tijdens deze fase de implementatie te verifiëren en optimale parameters te kiezen voor alle vrijheidsgraden in het ontwerp (dit is de design space exploration, ofwel de verkenning van de ontwerpruimte).

De DSE simulatieomgeving is gebaseerd op de Communicating Threads (CT) C++ bibliotheek, een implementatie van de Communicating Sequential Processes (CSP) taal. De bibliotheek ondersteunt een aantal platformen, waardoor het mogelijk wordt om software die bedoeld is voor een embedded processor te kunnen testen op een gewone PC.

De DSE simulatieomgeving gebruikt een aangepaste versie van de TrueTime netwerksimulator, waarmee het mogelijk is om gedistribueerde systemen te kunnen simuleren. De ondersteuning voor netwerkcommunicatie is verder uitgebreid door Remote Channels toe te voegen aan CTC++. Met deze Channels kan een gewoon kanaal eenvoudig worden omgezet in een netwerkvariant. In het Remote Channel concept is een scheiding aangebracht tussen de delen die specifiek zijn voor een enkel kanaal, voor de communicatiehardware of voor een enkele communicatie-interface. Door deze scheiding is het eenvoudig om ondersteuning voor nieuwe hardware toe te voegen. Bovendien kunnen meerdere kanalen dezelfde netwerkinterface gebruiken, onafhankelijk van de gebruikte netwerkhardware.

Om resultaten uit een simulatie terug te krijgen is de mogelijkheid om gegevens te loggen toegevoegd. De gebruikte implementatie biedt voldoende functionaliteit voor simulatie, maar is ongeschikt voor het loggen in real-time. Een alternatief concept, wat zowel in simulaties als in real-time situaties bruikbaar is, wordt daarom gepresenteerd.

Een demonstratieopstelling, bestaande uit een gedistribueerde regelaar op twee ADSP systemen en de Linix opstelling, is gesimuleerd met de DSE simulatieomgeving. Dit voorbeeld laat zien hoe een simulatie van een compleet gedistribueerd regelsysteem opgezet kan worden. Verder is een kleine simulatie gemaakt die laat zien hoe de simulatieomgeving ook voor eenvoudige simulaties kan worden gebruikt, bijvoorbeeld om metingen aan verschillende communicatieprotocollen uit te voeren.

Een betere integratie van DSE met de overige gereedschappen wordt aanbevolen, bijvoorbeeld met gCSP om simulaties grafisch te construeren. Verder kan functionaliteit worden toegevoegd, zoals ondersteuning voor netwerkhardware of voor het meenemen van executietijd in de simulaties.

Preface

This report forms the conclusion of my Electrical Engineering study at the University of Twente.

I am very thankful for everyone who helped me during this project. Special thanks go to Bojan Orlic, who has been of great help throughout the whole project. I also would like to thank Marcel Groothuis for his assistance, especially when numerous problems showed up during the development of the demonstration setups. The cooperation with Mark Huijgen has always been very pleasant, not only during this project, but also in several other projects throughout the study.

A final word of thanks goes to my parents for their unconditional support during my study.

Matthijs ten Berge

Hengelo, August 2005

Table of contents

1 Introduction

1.1 Concurrent systems

As embedded systems become more and more complex, methods are needed to structure these systems. Defining a hierarchy and properly structuring the concurrency within this hierarchy can help dealing with the growing complexity.

In *concurrent programming* the task of the complete system is split up in blocks, where each block (process) has its own specific sub-tasks. Some of these blocks can run in parallel, others are to be run after their predecessor has finished or when a certain condition is met. The interrelationships between the blocks are denoted in a *concurrency structure*, which can be captured and analyzed via some formal method or process algebra, for instance in CSP (see below).

When the concurrency is not limited to single computer systems, but instead is spread over two or more computers, the term *distributed computing* is used. Where single-processor systems can only mimic parallel execution of software, for example by occasional switching between the processes, distributed systems can execute software truly in parallel. The different parts of a system need to communicate in order to achieve the desired system behavior. Therefore the communication between the participating computers is also assumed to be part of the distributed system. In most industrial applications all communication is performed over a fieldbus. *Fieldbus* is a generic term used to describe a common communication system for control systems or field instruments, hence its name. In this report the terms fieldbus and network are used interchangeably.

The *Communicating Sequential Processes* language (CSP) is a language in which concurrent systems can be described and analyzed algebraically (Hoare, 1985). Occam is a parallel programming language based on a subset of CSP and customized for a specific type of processors – transputers (Welch *et al.*, 1993), which are nowadays obsolete. The CSP descriptions can be translated directly into Occam source code. Such a direct translation is however not possible into sequential languages like C, C++ or Java, as these languages lack a number of concepts.

Support for these concepts is added by using the *Communicating Threads* (CT) library (Hilderink *et al.*, 2000). The CT library provides the necessary concepts in an object-oriented and structured way. Variants of the library are available for C, C++ and Java and are named accordingly: CTC, CTC++ and CTJ. The C++ version of the library has been the starting point for a rather drastic redesign (Orlic and Broenink, 2004). As this project is based on this redesigned version, only the C++ variant of the library is used.

1.2 Design Trajectory

A specific design trajectory for embedded control systems development was defined in (Broenink and Hilderink, 2001). This design trajectory, as shown in Figure 1-1, consists of several phases.

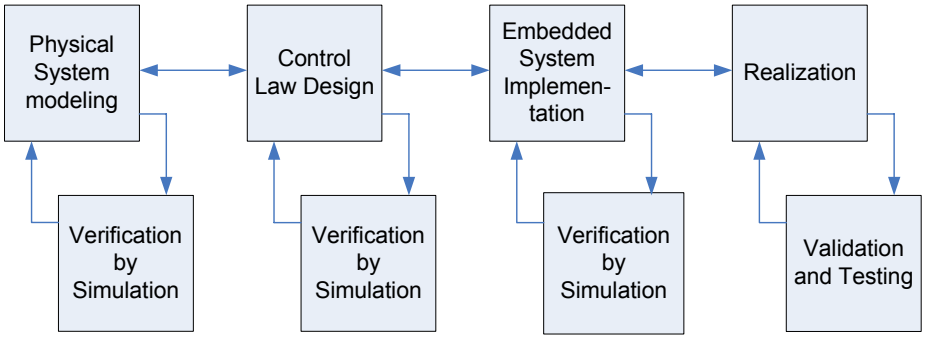

Figure 1-1 Design trajectory for embedded control systems

In the first phase, the *physical system modeling* phase, models are designed that describe the dynamic behavior of the physical system. The results are verified by simulating the behavior of the model and comparing the simulation results with the physical system. This process can be iterative, by correcting the errors found during simulation in a new modeling cycle.

In the second phase, the *control law(s)* are developed. The result is again verified by simulation to check whether the desired control system behavior is achieved. Again this is an iterative process, which can partially be automated via iterative optimization functions such as curve fitting or error minimization.

In the third phase in the design process, the *implementation phase*, the designed control law(s) need to be converted into a software implementation. In case of a distributed system this phase also involves *process allocation*, which is the distribution of the processes over the available processor nodes. In the implementation phase choices are made for the design options in the systems, such as the type of network, the needed bit rate and the assignment of priorities.

In the last phase the system is *implemented* on the target system(s). This can be done stepwise by using techniques like *hardware-in-the-loop* (HIL) simulation (Visser *et al.*, 2004). In this phase the result can be validated, i.e. it must be assured that the requirements for the intended use of the control system have been fulfilled, for instance by testing the behavior of the completed system under various circumstances.

During the complete design trajectory the designer is exploring the design space. Every design contains a number of parameters, the *degrees of freedom*, which can be chosen by the designer. The complete design space consists of all possible combinations of these parameter values. Evaluating the effect on system performance of different combinations is called *design space exploration*, as (part of) the design space is searched for the optimal combination.

1.3 Design tools

Some phases of the design trajectory are covered by design tools. Modeling of physical systems and design of control laws is covered by 20-sim. 20-sim is a modeling and simulation program developed by Controllab Products B.V. (CLP, 2002). The first two design phases can be covered completely by this program. Furthermore its code generation functionality has a role in the third phase. Another design tool is gCSP (Jovanovic *et al.*, 2004). It provides a graphical language in which the concurrent structure of the design can be described. It can also generate code to implement this structure. The latest versions of gCSP now also support combining the generated blocks of code from 20-sim and gCSP into a usable program automatically. Simulation in the third design phase is not supported by existing tools. The final design phase is supported by HIL-simulation, where the plant is replaced by a combination of hardware and software that simulates the behavior of that plant. The realization of the control software is mostly supported by software development tools from the manufacturers of the target systems.

So far, all related previous projects implemented either simple control systems executed on single processor systems, or more complex distributed systems but with a fixed hardware topology and predefined process allocation. However, none of those projects did take into account a need for early design space exploration. Such a design space exploration, during the third phase of the design trajectory, is needed in order to identify optimal or suboptimal hardware and network topologies and mappings of software processes and communications to processing nodes and links prior to the actual implementation.

The research project that is immediately related to this project (Orlic and Broenink, 2003) was initially intended to deal with the influence exerted by networks in CSP based distributed systems. However, due to the existing gap in the tool support for the design methodology, this project also deals with the support for design space exploration in CSP based distributed systems.

1.4 Outline of the report

To fill the gap in the tool support for simulation in the embedded system implementation phase, a simulation environment is developed. This simulation environment is also referred to as the DSE simulator, which stands for *design space exploration*, and is described in chapter 2 of this report. As mentioned, the networks are also part of a distributed system and need to be simulated. The part of the simulation environment that simulates these networks is presented in chapter 3.

In order to efficiently explore the design space for a distributed application that uses fieldbuses, one should be able to switch to different network types easily. The implementation of external channels in the CT library, which are used to let processes on separate computers communicate, did not provide this ability. Therefore a new implementation of these channels was developed. This new implementation, called *Remote Channels*, is described in chapter 4. Implementation details of the Remote Channels can be found in appendix A.

Obtaining results from a simulation, in whatever form, is an important aspect of a simulation. But also in an application running on the final target processor it can be important to observe parts of the program. These data-gathering activities all fall under the denominator of *Logging* and are described in chapter 5.

In order to test the simulation environment with a real example, a setup has been build using two Digital Signal Processor (DSP) boards from Analog Devices and a hardware plant, called Linix. This setup and its simulation with the new simulation environment are presented in chapter 6. This chapter also contains results of the performance measurements on the Remote Channel protocols. The model of the control system is shown in appendix B.

The conclusions and recommendations can be found in chapter 7.

2 DSE Simulation Environment

2.1 Motivation

The design trajectory for embedded systems is shown again in Figure 2-1. Some tools that can be used in the phases are indicated. It is assumed that 20-sim is part of the toolchain and that it will be used during the first two design phases. 20-sim can be used for both the design and simulation steps in these phases.

The design trajectory shown in the previous chapter (see Figure 1-1) also shows a *verification by simulation* during the implementation phase of the design. Besides some formal checking methods, the toolchain however does not provide means for simulation during this phase. The simulation environment presented in this chapter is meant to fill this gap. Some arguments why such a simulation environment is needed are given below.

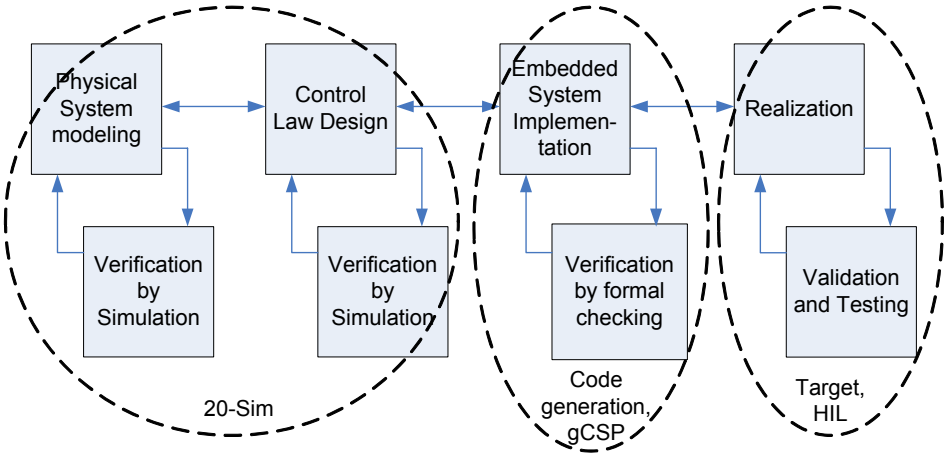

Figure 2-1 Design trajectory for embedded control systems

All embedded system design phases should provide simulation support

Simulation in the *physical system modeling* and *control law design* phases (see Figure 2-1) can be useful to verify the correctness of physical and control system models and its parameters. It is possible to generate code from the obtained models, for instance by using the code generation features of 20-sim. The generated code can be customized for a specific hardware platform by defining template files containing platform specific code, which is for instance done for the ADSP platform in (Mocking, 2002). The simulator in these design phases is however not capable of verifying the generated *software* implementation of a model in the context of the given hardware platform.

The embedded system *implementation* phase is currently only supported by the 20-sim code generation and a tool for structuring concurrency, the gCSP tool (Jovanovic *et al.*, 2004). In the current state of development, the gCSP tool can aid in entering a design graphically and it can generate code from this design. Code blocks generated from 20-sim can be embedded using process wrappers, and supplemental processes can be added. The concurrency structure of the design can be entered by defining compositional and communication relationships between processes and grouping them into constructs. Code for the executable, as well as the code for formal CSP model checking, can be generated for the given designed process architecture. No simulation or analysis, other then formal checking for concurrency problems, can be performed by the gCSP tool as it is now.

The embedded system *realization* phase already deals with the real target hardware. Another project, focused on this phase (Visser *et al.*, 2004), exists, that deals with the implementation of *HIL* (hardware-in-the-loop) simulation and plant realization. In the embedded system realization phase, special focus is put on supplemental tools, for instance to provide logging functionality. The obtained measurement results can then be compared to simulation runs from previous phases. Although it can

provide valuable feedback to the designer, it is a kind of post-game approach, in the last phase of the design trajectory.

An overall system simulation, capable of capturing the influence that embedded system implementation exerts on the behavior of the overall system, is also needed in an earlier phase of the design trajectory. The *implementation* phase is the earliest phase where this overall system simulation can be performed.

Validation and Testing on the target is expensive

Testing the generated code and its surrounding software is very well possible on the real target, as the software is ultimately meant for that target. Of course the hardware must already be available, which is not always the case. Design iterations on the real target can come at high costs. When changing a parameter in the software, a lot of time is spent on recompiling and downloading the new code into the target.

Changing a parameter like the type of fieldbus on a real target is another example of design space exploration at high costs: changing the network type does not only require software changes, but also changes in hardware, which costs both time and money. This can range from simply exchanging a hub for a switch to a full redesign of the target with different hardware components. As a general rule it can be assumed that design changes in a later phase of the design process are more expensive than in earlier phases, see for instance (Douglass, 2003).

In-target debugging can be hard

Depending on the target, it can be difficult to debug the software just by running it on the target. Some targets provide good debugging support: PC104 stacks running Linux for instance provide networking, a login console and a debugger installed. Other targets do not provide debugging support at all (mostly microcontrollers) or at high costs. These costs can be both time (stepping through the code of an ADSP-21992 takes about two seconds for each step) and money (for example in-circuit device emulators for microcontrollers).

2.2 Embedded system simulation

With the embedded system simulation environment an extra sub-phase in the design trajectory is introduced (see Figure 2-2). Its intended place in the design trajectory is at the *embedded system implementation* phase. It can be expected that in the future the gCSP tool will be enhanced with

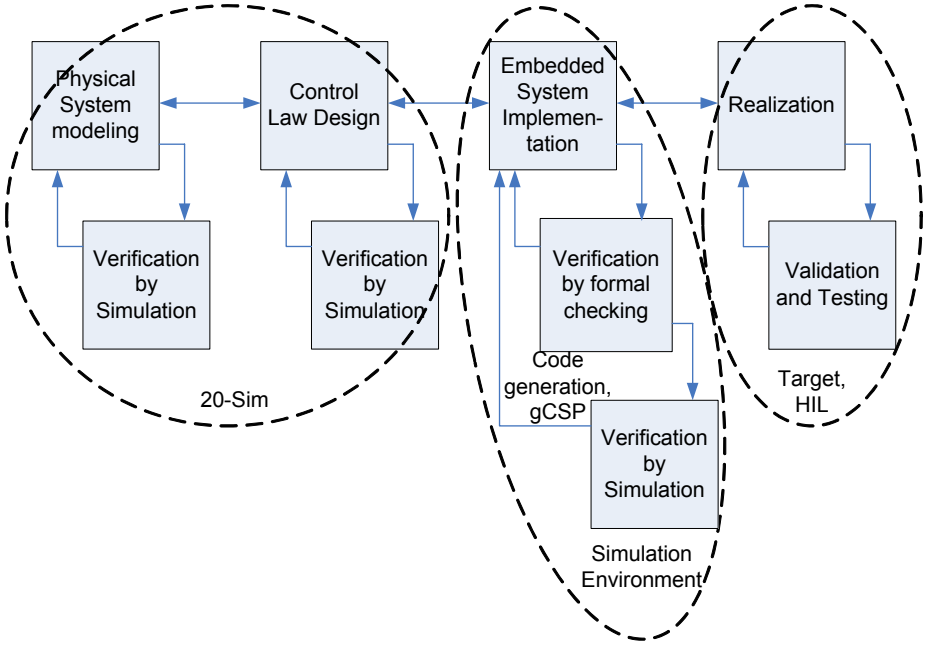

Figure 2-2 Design trajectory with added DSE Simulation Environment

support for this simulation framework. However, since the simulation framework is being developed in parallel with gCSP tool, it is directly building upon the code obtained by 20-sim code generation.

After code is generated from the simulation model, it can be combined with the other software components, just as if the implementation is to be tested on the target system. But instead of running on the target, the software can run in a simulated environment on a normal PC. The simulator environment provides a number of components with functionality needed for a simulation run. These components are discussed in this chapter. The term 'simulation environment' is more appropriate than 'simulator', as the resulting executable runs independently, as opposed to a simulation model or script, which is executed by a simulator. As such, the developed components do not make up a simulator, but they create an environment in which the simulation can be executed.

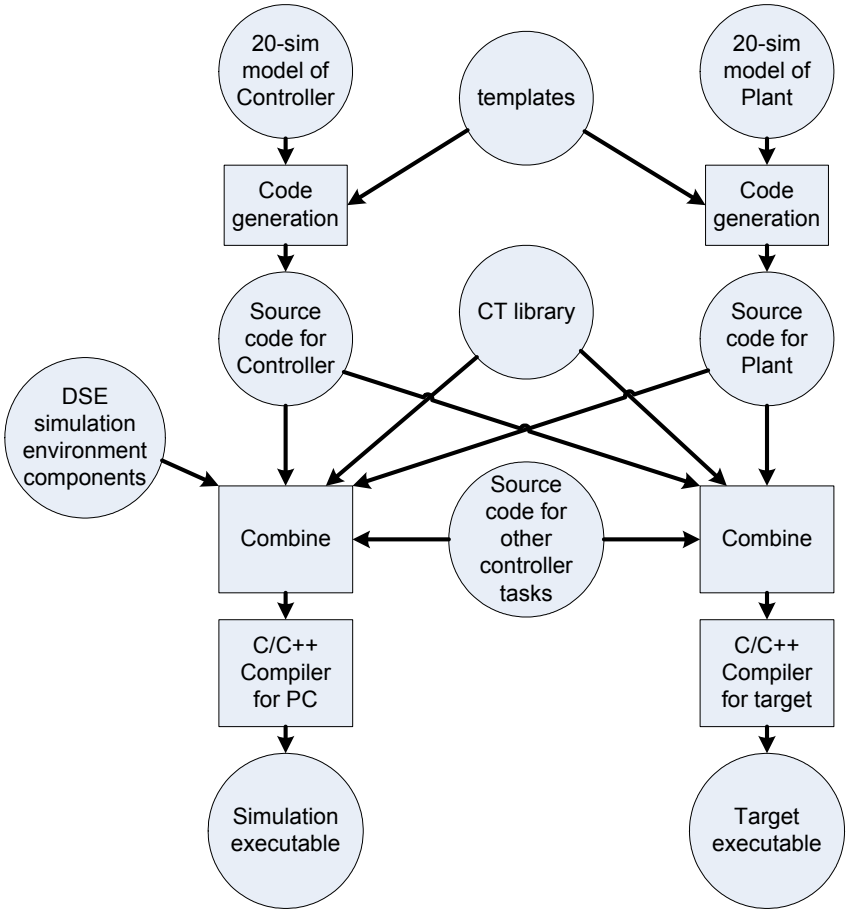

Figure 2-3 Steps to create executables for simulation or for the target

2.2.1 Available simulators

A number of simulators and related tools for real-time and control system co-design is already available in research communities. A good overview is given in (Henriksson *et al.*, 2005). This project is especially interested in simulation of distributed systems, with a focus on the influence of the networks. Therefore, especially appealing was to investigate whether existing network simulators can be reused. Several research projects such as 'ns2', a network simulator from the VINT project (VINT, 2005), 'Real', a network simulator from Cornell University (Keshav, 1997) and TrueTime (Henriksson and Cervin, 2003) were investigated.

The design of a new simulation environment from scratch however had some important advantages:

Simulations are C++ programs

All parts of a simulation are written in C++. This gives a lot of design freedom and eliminates the need to learn yet another language: ns2 for instance uses OTcl, 'Real' uses NetLanguage and TrueTime

uses a combination of Matlab code, Simulink models and MEX functions. Furthermore, the resulting executable can run stand-alone, with little overhead compared to scripted (interpreted) simulators. It is possible to use existing C++ development tools, debuggers and profiling tools. The latter can for instance be used to get insight in how the processing load is balanced over the nodes.

Coupling with 20-sim

A tight coupling with 20-sim is necessary to easily move on from the model-based simulation design phase to the code-based simulation. The code-generation tool of 20-sim can provide this coupling.

Use of CSP

The simulation environment is based on the CT C++ library, an implementation of the CSP theory (Hoare, 1985). In the current design flow, CT C++ is already often used to finally implement the software. By using it also in the simulation environment, the transition from source-code-based simulation to verification on the target will be smooth. Because CT supports several platforms, development can be done on the platform of choice. Thanks to the structure of a CSP program, it is very easy to assign processing tasks to a different processor node, just by moving part of the process hierarchy to a different location on the tree and replacing local channels with remote channels or vice versa.

2.2.2 Basic assumptions and principles

The simulation is based on the idea that a complete distributed system can be represented by adding higher layers to the existing process hierarchy (see Figure 2-4 and Figure 2-5). That part of the application that should be executed on one single node is wrapped in a process. As this process represents the complete node, it is named after this node (*ProcessorNode1* and *ProcessorNode2* in the example shown in the figure).

Networks are also represented as processes (*NetworkSimulator1* and *NetworkSimulator2* in the example). All communication channels that were local to a certain node remain local inside the wrapper process of that node. Instead of communicating with network interface hardware, all remote communication channels are now communicating with the processes that represent the network(s). The main idea is to keep all executable code intact and only add additional processes to simulate the networks.

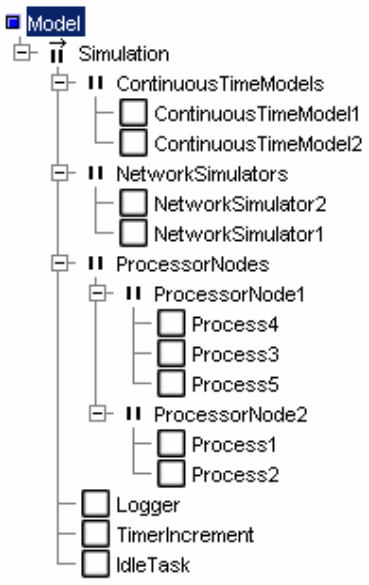

Figure 2-4 Typical simulation layout

Beside processes representing networks it is also possible to add processes that represent a plant (the example in Figure 2-4 has two *ContinuousTimeModels*, which will be discussed in more detail in section 2.3.1). All these higher-level processes are grouped in one (Pri)Parallel construct representing

the whole system. The complete system also contains some extra components, such as the *Logger* and the *TimerIncrement*. The *Logger* provides data logging functionality and the *TimerIncrement* is part of the timing implementation of the simulation. These components are discussed in detail later, in section 2.3.2 and 2.4.

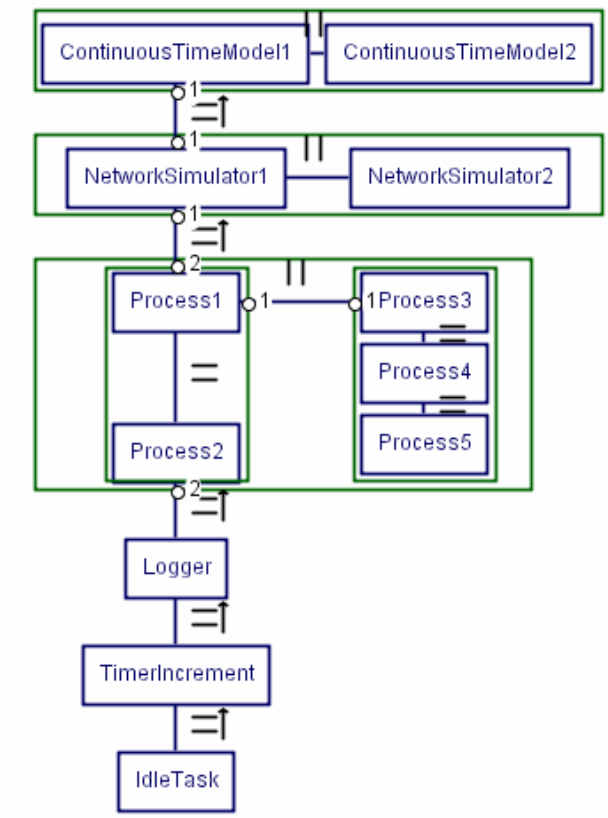

Figure 2-5 Simulation layout block diagram

Since focus is put on the influence of networks on distributed system simulation, the decision was made to assume that major overhead is induced by network delays and that the computation overhead is negligible compared to any network delays. Due to significant difference in speed between processing and communication devices, this assumption is a valid first approximation for most real life systems.

2.2.3 Execution timing

As stated in the previous section, the network delays are assumed to be large compared to the execution time of the code. However, for simulations with relatively fast networks, or no networks at all, the execution time of the processor nodes should be taken into account. As an exception, when both inputs and outputs of the code blocks are timed, the code execution time will not influence the behavior of the system: reading the inputs and updating the outputs then occurs at fixed intervals, independent of calculation delays.

The reason that the code execution time is not part of the current DSE simulation environment is twofold.

- First it requires a redesign of the timer structure. The simulator needs to be able to determine on which processor node a certain process is running. As processor nodes can possibly contain large constructs with processes, and these processes can also be moved to other processors as part of a design iteration, this task is, although possible, not trivial. The timer structure should then keep track of the execution time of the different processor nodes, as they all run in parallel, and block the execution of nodes that would otherwise get too much ahead of the others.

Cooperation with the CT scheduler is also needed, because a distinction must be made between processes on different or on the same processor node: processes on different nodes can run truly in parallel, while processes on the same processor node cannot run in the time already consumed by other processes on that same node. The Channels also need to be aware of the timing issues: when determining the order of arrival of reader and writer, the simulation time of both reader and writer must be taken into account. Possibly the concept of the TrueTime kernel simulator (Henriksson and Cervin, 2003) can be used.

- Furthermore it requires the execution time of all relevant code blocks to be measured or estimated. Measurements cannot be done on a random desktop PC; they have to be done on the real target processor(s), as the architecture, instruction set and execution speed will most likely differ. Measuring the execution time is hindered by instruction and data caching, interrupts and conditional branching in the code, because they all influence the execution times. On the other hand, estimation of execution times requires building complex detailed models for every target hardware platform.

As a result of the untimed code execution, it is not possible to have for instance a process with an endless loop in a processor node construct. Such a process will run forever, causing the TimerIncrement process never to be reached. As a workaround for this kind of situations, one could make such a process timed, by letting it block on a timer channel on every iteration of the loop.

2.3 Detailed Simulation framework

The DSE simulation is based on a CSP hierarchy tree, like the example shown in Figure 2-4, and implemented using the CT C++ library. The way the different components are arranged in the main PriPar construct ensures proper timing of the simulation, as will be shown in section 2.3.2, *Simulation process and its timing*.

2.3.1 Continuous-time models

On top of the hierarchy are the continuous-time models. These are the numerical models representing the physical process (plant) which will be controlled by the embedded system. As the simulation is based on discrete-time simulation, the continuous-time models are approximated by discretizing them at sufficient small time steps. As a rule of thumb, the time steps of the discretized model should be ten times as small as that of the other, discrete, components in the system (Groothuis, 2004). In this project however, a more accurate method was applied. Instead of updating the states and rates of the plant much faster than those of the other, discrete components, the states and rates are updated only when that is needed according to the plant dynamics or as result of actions of other discrete components.

In the examples, discretizing is performed by the 20-sim code generation feature, based on Euler's method. This method normally produces models with a constant time step. Besides the normal model calculations at fixed time steps, extra calculations can be performed when it seems appropriate. Typically this will be the case when a discontinuity in the model is expected, for instance when a discrete controller updates an input of the model. The method of triggering these extra calculations is addressed in section 2.3.2: *SimTimerChannel advanced features*.

Variable-step Euler discretizing

Ordinary Euler discretizing (see Figure 2-6) works by determining the rate-of-change of the model states at a time t_0. These rates are multiplied by the configured time step size t_{step} and added to the states at t_0 to obtain an estimate of the states at t_2 ($=t_0 + t_{step}$). Within the interval $t_0 \ldots t_2$ the rates are assumed to be constant. Because of the fixed step size, all calculations to determine the states at t_2 can already be performed while the simulation is still at t_0. Depending on the dynamics of the model, the time step can be chosen such that the rates indeed do not significantly change during one time step.

However, when such a model is combined with for instance a discrete model running at a different time step, the inputs of the continuous-time model could change drastically at time t_1 (somewhere between t_0 and t_2, see Figure 2-7), due to the discrete model updating its outputs. At t_1, the estimated states need to be updated for the partial step between t_0 and t_1, and the rates need to be recalculated for the new situation.

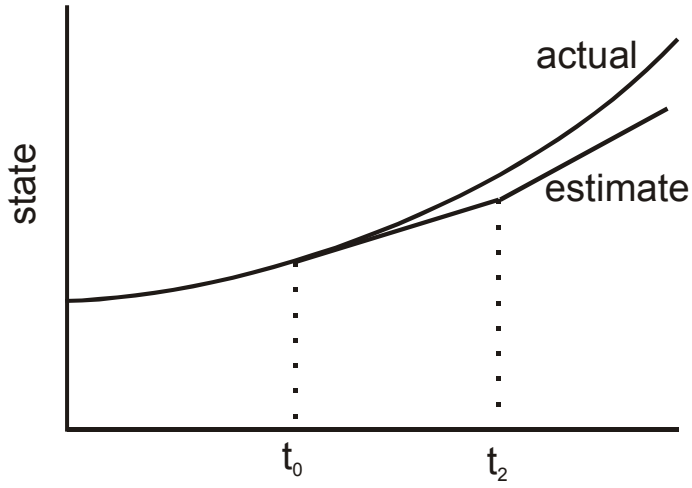

Figure 2-6 Euler discretization

As the time step t_0-t_1 is not known in advance, it is impossible to calculate the next states while still at time t_0. The only thing that can already be calculated then is the rate-of-change. At time t_1, the actual time step that is taken is known and can be used to calculate the states at t_1. After updating the states, the new rates can be calculated. This is also how the simulation wrapper around the continuous-time models implements the variable-time Euler method (this implementation can be found in the file linixplant/linixplant.cpp in the *linixsim* example directory).

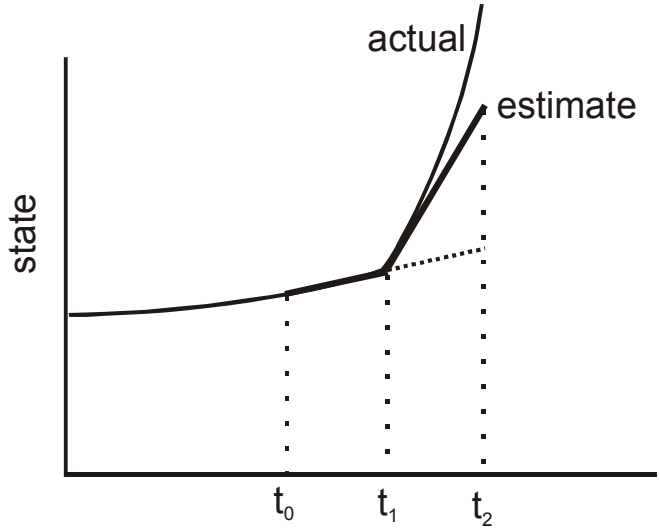

Figure 2-7 Recalculating the rate at a discontinuity

Using the presented variable-step mechanism, a simulation with one discrete controller model and one continuous-time plant model, both running at a different time step, result in really accurate results compared to the 20-sim simulation of the same system, something which could not be achieved with the fixed-step Euler models. The 20-sim simulator internally also uses a variable step size mechanism. Although its implementation details are unknown, the results are identical. In the scope of this project a special 20-sim template for the presented variable-step Euler method was made. Combining the variable-step approach with other integration methods supported by 20-sim was not explored, since this issue is not a primary concern of this project.

2.3.2 The SimTimer

As already mentioned, the DSE simulation uses discrete time steps. One global object, the *SimTimer*, is used to keep track of this simulation time. The SimTimer does not directly show up in the process hierarchy (Figure 2-4), as it is a passive object. Instead, the active object *TimerIncrement*, which is a process and as such does show up in the process hierarchy, is used to execute the SimTimer's functionality.

Processes are not allowed to use the SimTimer directly. As CSP only allows processes to communicate using channels, the processes should create a channel to the SimTimer. This *SimTimerChannel* encapsulates the interface to the timer (see Figure 2-8).

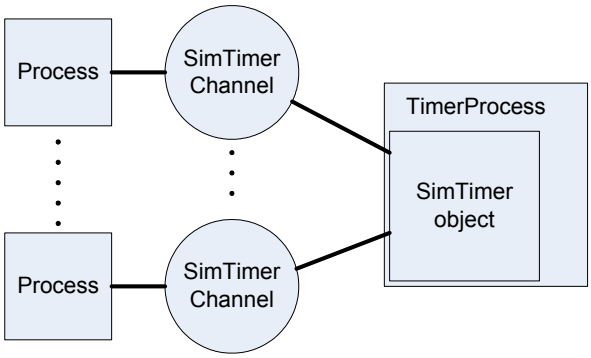

Figure 2-8 SimTimerChannels

The SimTimer should be an active object, as in CSP Channels should have processes (active objects) on both sides. The SimTimer is however a passive object, 'borrowing' the thread of control from the IncrementTimer process (see *TimerProcess* in Figure 2-8) This separation, which could cause some confusion, has a reason. It makes it very easy to change the 'source of execution' (i.e. where the SimTimer borrows its thread of control from) to for instance a different process, a hardware interrupt function or an operating system callback function. A purpose for this is explained in section *The SimTimer in real applications*, further on this section.

SimTimerChannel basic features

Reading from the SimTimerChannel will return the current simulation time. This communication will never block, as such it can be seen as a read from an overwrite channel.

Writing to the SimTimerChannel can however be a blocking communication. Blocking occurs if the written time value lies in the future (with respect to the simulation time). The writing process will then block on the SimTimerChannel until the requested simulation time has become present. If, on the other hand, the written time value is already in the past, the channel communication will not block, but return immediately.

DSE simulation processes and its timing

The blocking behavior of the SimTimerChannel is utilized by organizing all processes in a PriPar construct: the processes with highest priority, the continuous-time models, will initially be the first processes allowed to run. They will perform a model calculation step and then write the time value of their expected next run to their SimTimerChannels. As this requested time is in the future, the channels will cause the continuous-time model processes to be blocked. Now the processes with the next highest priority, the network simulators, will be executed (see Figure 2-9). Similarly to the continuous-time models, the network simulators will also do a simulation step and block on the SimTimerChannels.

This process continuous until all continuous-time models, network simulators, nodes and the Logger are blocked on a channel. Whether these are all SimTimerChannels, or also regular channels, does not matter: all processes are blocked and no single process can make any progress anymore. In normal

situations (except when there are external interrupts) this situation is called a deadlock. But in the simulation there is still a process left that is runnable: the TimerIncrement process.

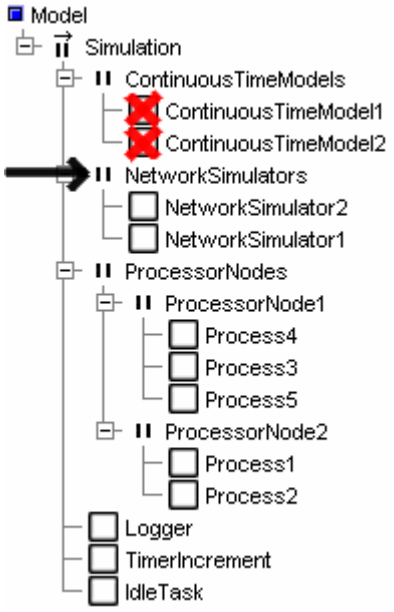

Figure 2-9 Continuous-time models are blocked, simulation continues with the Network Simulators

The TimerIncrement process requests the SimTimer to increment the simulation time. Out of all requested times on the blocked SimTimerChannels, the SimTimer selects the one that is most near, i.e. the smallest time increment. It increments the simulation time to this new value. The SimTimerChannels which now meet the current time are unblocked, the others are left blocked.

Basically this is the way passage of time is often expressed in simulation environments. First, all processes are allowed to execute without real passage of the time. Once all processes are blocked on either time channels or communication events, the time will be incremented to the next valid point.
As the unblocked processes have a higher priority than the TimerIncrement process, program execution will continue with those.

This was the reason for introducing a PriParallel instead of a Parallel composition. The TimerIncrement process should be preempted after each time update and will not make extra time increments until all other processes are blocked again. In order for this construction to function properly, the TimerIncrement process should allow a context switch to take place after each timer increment. In the CT library framework this can be done for instance by using a sequence of calls to the enterAtomic and exitAtomic functions or by using the yield function. If this is not done, or if for instance the PriPar's implementation is not correct, the TimerIncrement process will remain the active process. It will then step through the complete simulation time in one run, leaving the other processes no chance to do their work.

SimTimerChannel advanced features

In order to implement for instance the variable-step model calculations (introduced in section 2.3.1), or a bounded wait for an event (for instance a timeout when waiting for an incoming packet, see chapter 4 about Remote Channels), additional functionality is needed. More specific, a method is needed for external processes to reactivate processes blocked on SimTimerChannels, prior to the previously intended time point of their release. So a blocking wait on a SimTimerChannel needs to be ended prematurely. Two methods are provided to achieve this.

The first method is by using a SimTimerChannel *earlyRelease*. Any process, except for the blocked process itself of course, can earlyRelease a SimTimerChannel. The channel is unblocked immediately and, depending on the relative priority of the processes, the unblocked process might be executed immediately. As the current Channel interface in the CT C++ library does not allow return values for channel operations, it is not trivial to let the process know that is was earlyReleased. Besides obtaining this information by changing the Channel interface definition in the library, or by using the exception mechanism, the process can also perform a read on the SimTimerChannel to obtain the current time, and compare it with the time it had requested in the *write* operation.

The earlyRelease method is used in the *RemoteLinkDriver* (see also chapter 4) to handle incoming packets and timeouts: first a packet is sent, and then the process writes the packet timeout time to its SimTimerChannel. If no acknowledging packet arrives, the time specified in the SimTimerChannel will eventually expire and the sending process is released, knowing that a timeout has occurred. On the other hand, if an acknowledgement packet is received, the SimTimerChannel will be earlyReleased, allowing the incoming packet to be processed immediately.

The second method is by using a *globalEarlyRelease*. A process can request the SimTimer to iterate over all available SimTimerChannels to see whether it can earlyRelease them. On creation of a SimTimerChannel, one has to specify explicitly that the channel is susceptible to a globalEarlyRelease. The SimTimer will skip channels that do not specify this property, and channels that are not blocked.

This method is used to perform the extra simulation steps in continuous-time models. Any time when a discrete model delivers a new input value to a continuous-time model, is should request a globalEarlyRelease. The continuous-time models will then be interrupted. They read the current time and perform a model calculation based on this time.

The SimTimer in real applications

Despite its name, the use of the SimTimer is not restricted to simulations. It can also be used in real (real-time) applications. On the side of the processes nothing needs to be changed, as the SimTimerChannel forms an abstract, platform independent interface to the timer. Only the software component which increments the timer needs to be changed.

Incrementing the time is no longer performed by a process, as in the DSE simulation, but by a real timer implementation. The SimTimer can for instance be incremented from a hardware timer interrupt service routine or from a callback function which is called by the Windows multimedia timer. The SimTimer object is now not longer a passive object, but it has its own thread of control, namely the thread that executes the interrupt or callback function.

Instead of incrementing the timer up to the nearest time, the timer is now incremented with fixed time steps, representing the step size of the timer source. So if, for instance, a hardware timer is running at one kilohertz, the SimTimer is incremented a millisecond each step. It is even possible to update the timer from more than one source of time; as long as those sources do not interrupt each other (interrupt nesting). In this case, the routines should update the SimTimer using absolute time values instead of steps relative to the previous value.

An example of the use of the SimTimer in a real application is the *Linix* demonstration setup, which will be discussed in chapter 6. It uses one of its hardware timer modules to generate an interrupt. The SimTimer gets updated from that interrupt routine. The timer is used via two SimTimerChannels, one to wait for the next run of the controller model, the other to provide the Logger with values for its timestamps.

Units of time

By default, the SimTimer uses a double-precision floating point type to represent time. In this case the unit of time is one second. Some parts of the DSE simulation environment, like the network simulator and the continuous-time models, rely on this unit. Using a floating point number to represent the time

has the advantage that it scales to the needed time range, i.e. it is both possible to use very small time units (like microseconds) or very large time units (like hours) with full precision. A combination of both small and large units however, causes rounding errors to occur. This disadvantage has to be taken care of, for instance by never comparing two time values to be equal. Instead, they should be compared to be 'almost equal', thereby allowing a small percentage of deviation.

The SimTimer can also be configured to use a different type as its time unit, namely an integer. This setting can not be combined with the network simulators or continuous-time models without some modifications. For real applications however it can be somewhat more convenient. The unit of time can for instance be chosen to correspond to one hardware timer tick, i.e. the timer interrupt routine increments the SimTimer by one. Rounding errors like with floating point numbers do not occur. Instead, a timer overflow can be expected, but this can be handled easily.

The Linix demonstration setup mentioned already will work with either configuration. Depending on the configured time type, it chooses the correct step size to update the timer and to wait for the next model run.

2.4 The Logger

The Logger is used to transfer results of the simulation to an external data processing application. It is discussed in detail in chapter 5. The low-priority position of the Logger in the simulation hierarchy (see Figure 2-4) might seem somewhat awkward, as storing the results has a high priority in simulation, i.e. in a simulation it is more important to store all results than to run the simulation at high speed. However, with an eye on real-time execution on the target, where timely execution is of higher priority than logging all results, it is a logical choice.

First consider the simulation case. At a certain time the processes in the tree are executed. They will generate some log information, which will be stored in a buffer. No data can be transmitted yet, as the Logger will not get scheduled because of its low priority. At a certain point all processes are finished with this iteration and they are all blocked. Now the Logger will be executed and it will transmit the entire contents of the log buffer to the outside world (more on this in chapter 5). Only when all data has been sent, it will block and give over control to the TimerIncrement process, which will initiate a new time step.

Provided that the log buffer is large enough, which will not pose any difficulties as no significant memory limitations exist on most development machines, this process layout can guarantee that all data is logged, at the cost of speed. Furthermore it can guarantee that, when compared to the real target execution, the order of DSE simulation execution is not distorted because of the influence of the Logger.

Now consider the real target case. As mentioned before, there is no TimerIncrement process in this situation. Again, at a certain time the (real-time) processes are executed. Log information is accumulated in the log buffer. As soon as all processes are done, the Logger is scheduled and it will start transmitting the log data. When it is done, control is passed to the Idle task, which typically is a process doing just nothing. At the next hardware interrupt, some or all of the channels will be unblocked (whether these are timer channels or remote channels), and process execution will continue, and new log data is generated.

If, on the other hand, a hardware interrupt occurs before the Logger is finished (or even before it is started!), the Logger will not be able to transmit the complete data, as the other processes have a higher priority and will be scheduled for execution first. The log data will remain in the buffer, so it can transmitted the next time the Logger is executed. It can happen that the Logger does not get enough CPU time; in that case the log buffer will fill up and eventually be full. Depending on the implementation of the Logger, some new or old log messages now need to be discarded.

Summarizing, this process layout allows the Logger to completely execute once at every time point in the simulation case, while the same layout causes it to only run at ‘idle’ priority in the real-time case.

2.5 Variables

Although the code simulation runs on another platform than the final target, it should ideally behave the same for proper simulation and debugging. The behavior should also be the same for all possible target platform implementations as much as possible. One of the main incompatibilities of different targets is the difference in variable types. Consider for instance a type very commonly used in C++, the pointer. On a standard PC platform with a common compiler, a pointer will probably take 32 bits of storage. On an Analog Devices ADSP however, it will have a size of only 16 bits. And finally on the IA-64 architecture it would probably take 64 bits.

This problem is tackled by defining variable types that have a fixed size on all platforms. The implementation of those variables will thus be dependent on the platform used, but their use is platform independent from the applications point of view. Currently the most common variable types are defined for a number of platforms. Unfortunately, the 64-bits variables are not implemented on ADSP, as this platform does not support 64-bits arithmetic at all. All operations on those variables then need to be emulated in software, which requires quite some work.

The defined variable types also have support for logging built in. When this support is enabled, a variable of one of these types will generate a log message as soon as it is created or deleted or its value is changed. Logging is described in detail in chapter 5.

2.6 Network

The network simulators have a higher priority in the DSE simulation tree hierarchy (Figure 2-4), as in practice the transport and delivery of network packets is independent of the software that runs on the processor nodes.
Two separate networks are completely independent in reality. In the simulation this is represented by running every network in a separate process, concurrently with the other network representing processes. In most situations only one network is used, so the parallel construct of network simulators is reduced to a single network simulator process. The network simulator is treated in chapter 3.

As an interface between the network simulators and the processes on the processor nodes, *Remote Channels* are provided. Remote Channels can be used instead of local channels, between two processes. They can both be used in the simulation as on real hardware. Remote Channels and the communication protocols they support are treated in chapter 4.

2.7 Conclusions and recommendations

The designed DSE simulation environment extends the development toolchain by providing the ability to perform simulations of the complete embedded system during the implementation phase of the design trajectory. This allows the designer to investigate the consequences of design choices during this phase, instead of having to wait until the realization phase. This makes it possible to analyze the consequences of design decisions early in the design process, allowing for multiple design iterations (exploration of the design space) without high costs.

The simulation environment integrates into the existing toolchain by using the code generation functionality of 20-sim. The integration could however be improved, as it still requires quite some manual programming work to set up a simulation. As far as the generated code from 20-Sim is concerned, most manual intervention is involved in connecting the inputs and outputs of a model to the rest of the simulation. This work might be automated by extending the functionality of the *hardware connector* (Buit, 2005).

Using the gCSP design tool to design the concurrency structure and to automatically connect the processes using local or remote channels would also decrease the amount of manual work involved. Currently all network nodes, network node- and link-ID's (see also chapter 4) need to be numbered manually, which is a tedious task that can easily be (semi-)automated by using gCSP.

To extend the usability of the DSE simulation environment to situations where the assumptions about network and processor speed (see section 2.2.2) do not hold, the timing implementation of the simulator needs to be extended. The execution time of code blocks then needs to be taken into account. In order to obtain these execution times, a method should be developed to do automated timing measurements on the real target(s), as performing these measurements manually is too much a time-consuming task.

3 Network Simulator

3.1 Introduction

In order to be able to simulate distributed systems, the communication between the nodes in such a system needs to be available in a simulation environment as well. The communication is assumed to take place over one or more networks, so a network simulator component is needed.

A number of network simulators already exist. With respect to the data routed over those simulated networks and their abstraction layer, they can be sorted in two categories:

Network simulators with parameterized data flows

In these simulators, the content of the data flows is characterized by a number of parameters, such as the stochastic distribution of the packet size, the number of packets and the source and destination nodes. No real data is transmitted over the simulated network.

This type of simulator often works on a somewhat higher communication layer, like for instance the Transport Layer and may be useful to investigate for instance the effects of routing, queuing policies, flow-rate protocols and topology on network delays. The International Standard Organization's Open System Interconnect model (see the ISO/OSI documents in ICS section 35.100) contains a division of the communication model in layers. Examples of this type of simulator are ‘ns2’, a network simulator from the VINT project (VINT, 2005) and ‘Real’, a network simulator from Cornell University (Keshav, 1997).

Network simulators with real data flows

These simulators transfer real data over the simulated network. The sending network nodes will need to provide this data; the network simulator delivers it to the receiving nodes, which can then process it. Such simulators generally work at a lower network layer, like the Data Link layer. They handle media access, collision detection and retransmits. An example of this type of simulator is TrueTime (Henriksson and Cervin, 2003).

For the distributed simulation, a simulator of the second type is most useful, for two reasons.

- The first reason is that embedded controllers often do not use a standard network protocol stack like a TCP/IP stack. Instead, the software communicates directly with the network interface hardware in such cases to send and receive packets. The communication protocols are then implemented in the application software itself. In order to simulate such a situation with a Transport Layer simulator, the simulator must first be taught the behavior of the non-standard protocol used. A simulator based on the Data Link layer on the other hand will be able to simulate both default protocols and user-defined protocols, as in both cases the behavior of the Data Link layer will remain the same.

- The second reason is that it is really useful to work with real data over the network. Simulating a distributed control system differs from for instance research of large networks, where single users are often anonymous and where the data they send is not relevant. In a distributed control system, the identity and behavior of the different network nodes is generally known and the data that is sent across the network can influence the behavior of the nodes. A simulator which does not transport real data *could* be used, but then some arrangement must be made to temporarily store the transferred data, so the receiving node can pick it up again. This requires some software administration, which is not needed in case the simulator can transfer the data itself.

The network simulator part of the TrueTime simulator was chosen for the reasons mentioned above: it transfers real data and it works at the Data Link layer. Furthermore the source code of TrueTime is publicly available and licensed as freeware. TrueTime is partly written in C++, which makes it relatively easy to integrate the network simulator in the simulation environment.

TrueTime provides support for most Data Link layer types in use nowadays, such as CSMA/CD used in Ethernet, CSMA/AMP used in CAN networks and Round Robin used in Token Ring networks. A complete enumeration of all supported Data Link layer types, together with a description, can be found in section 3.4.1.

3.2 Overview of the TrueTime network simulator

The TrueTime network simulator is primarily based on *packets* and *queues*, which are represented in C++ using *objects* and *linked lists*. A packet will always belong to only one queue (an exception to this is a broadcast packet, which will be assigned to the queues of all receiving nodes).

The computers attached to the network are represented by *NWnode* objects. These objects contain the queues and information about the node. Each NWnode has its own full set of queues, unique to the network to which the node is attached. If the node is connected to multiple networks, it also owns multiple sets of queues. All data sent by a particular node to a particular network will always be added to the queue set belonging to that node and corresponding with that network.

3.2.1 Single run of the network simulator

The way a data packet travels through the network simulator is depicted in Figure 3-1, while Figure 3-2 shows a flowchart of a single run of the network simulator. These figures depict what is performed in a single run of the network simulator.

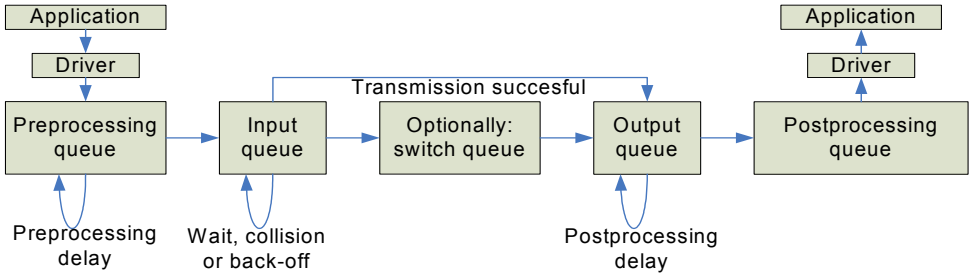

Figure 3-1 Packet flow through the network simulator

The contents of a data packet in the simulation are determined by the application that wants to transmit, just as in a real application. The application hands this data to a network interface driver. But instead of placing the packet in the transmit buffer of a network interface hardware, the packet is now placed in the *preprocessing queue* belonging to the sending node.

Packets will wait in the preprocessing queue for some period determined by the *preprocessing delay* parameter. This parameter can be used to mimic the delay between the moment the application offers a packet and the moment a first transmission could be attempted. This can incorporate both software and hardware delays, which can for instance be caused by the operating system, the network driver and the network interface hardware. After this delay the packet is placed in the *input queue*.

The input queue will hold packets until they are fully transmitted. Only the topmost packet in the input queue is transferred in the single simulation transmission. All other packets will remain untouched until they are on top (this allows for sorting the queue, see section 3.3.4). Packets will remain waiting in the input queue until the node is allowed to send. How long packets will have to wait in the input queue depends on the length of the queue and the relative priorities of other packets, but also on the network protocol used.

The conditions under which a node is allowed to start transmission depend on the chosen network type, for instance when the bus is idle (Carrier Sense Multiple Access) or when the token is acquired (Round Robin). When allowed to do so, the node starts transmitting the topmost packet from the input queue. During transmission, the packet remains on the input queue. If a collision on the bus occurs, a

back-off time is calculated. This is the time during which the node is not allowed to send. The packet remains in the input queue during back-off.

In full-duplex switched-mode networks collisions in the usual sense will not occur. However, when the switch memory is full, packets can either be dropped, or the switch can signal a collision, in which case the transmitter will retry after the back-off time. This behavior of the network switch is configurable and is explained in detail in section 3.4.8.

If transmission was successful, the packet will either be moved to the *switch queue* or the *output queue* of the receiving node, depending on the configured network type. From the switch queue a packet will be delivered to the output queue without collisions.

Packets in the output queue will remain there for a configurable period, the *postprocessing delay*. This delay is comparable to the preprocessing delay, but now for incoming packets (note that the term 'output queue' can be misleading, as the *output* of the network simulator is the *input* for the nodes). After this delay the packets are moved to the postprocessing queue, where the application can pick them up.

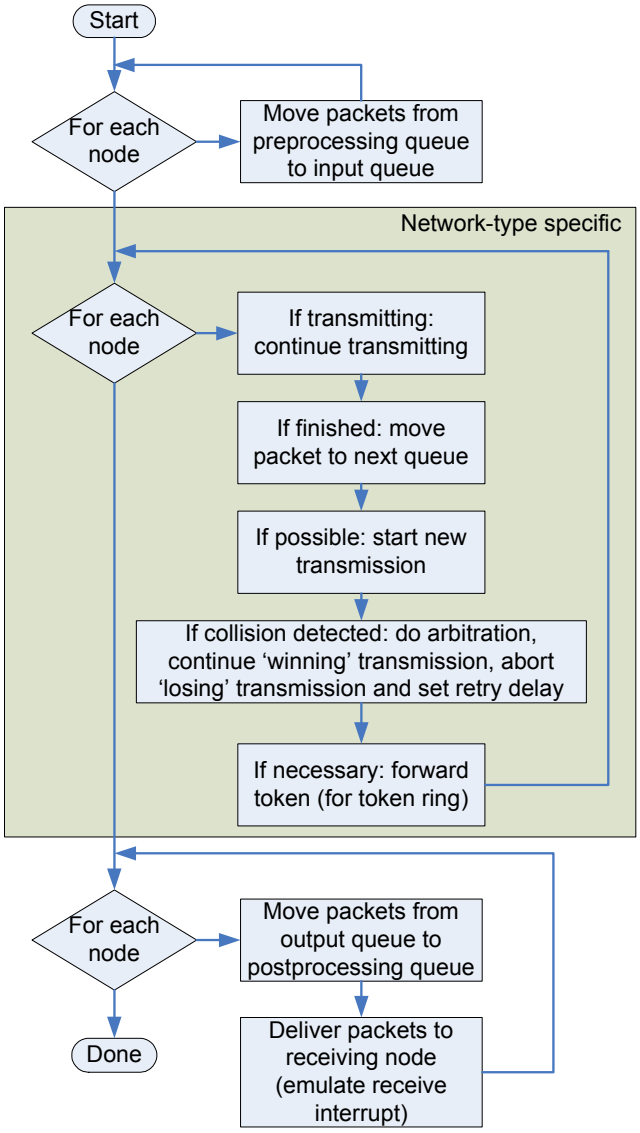

Figure 3-2 Program flow of the network simulator

The network simulator can simulate a receive interrupt, which is normally generated in hardware, by calling the *incomingPacket* function of a node. The reception interrupt service routine of the node is

now executed by the thread of the network simulator instead of a hardware interrupt thread. As the network simulator process has a higher priority than the node process, the 'interrupt' will be executed as soon as a context switch can be performed. The only difference with a real hardware interrupt is that the node's thread cannot be preempted.

3.2.2 Determining the next-run time

During the execution of a network simulator run, the simulator determines the *next-run* time. This is the nearest time point in future at which the network simulator needs to be run again. The next-run time is initially set to infinity, meaning that no next run is required yet. When during the run the simulator discovers that an additional run is needed at a specific time point, it compares this new time point with the value of the next-run time. The next-run time is only updated to the new value if this value is smaller than the previous value of the next-run time.

The need for a next run can have several reasons:

Preprocessing delay

As mentioned in section 3.2.1 packets can have a preprocessing delay. If the remaining delay of a packet in the preprocessing queue is unequal to zero, a next run is needed at the time point at which the remaining preprocessing delay reaches zero.

Data transmission time

A next run is required at the time point at which a data transmission is finished. The calculation of this time point is dependent on the configured type of network, but generally is based on the number of bytes that remain to be transmitted and the data rate available for that transmission.

Postprocessing delay

This delay is similar to the preprocessing delay, but for packets waiting in the output queue.

Back-off time after a collision

After a collision has been detected, the node or nodes that lost the arbitration must stop their transmission immediately (this is called *back-off*). They each calculate a *back-off delay*, after which the transmission may be retried. The back-off delay is stochastic, to prevent two colliding nodes from always using the same delay, as this would cause new collisions over and over again. A next run is needed when the back-off timer of some node reaches zero.

Transmission slot not fully used

In a *Time Division Multiple Access* (TDMA) network, every node is assigned a fixed amount of time every cycle, a time slot, to perform transmissions. If a node has less to send than would fit in this time slot, the network will be idle in the remaining time, as no other node is allowed to send outside its own time slot. A next run is scheduled at the time point at which a new time slot begins.

Token transmission time

This is only applicable to a *Round Robin* (RR, but better known as *Token Ring*) network. After a node has finished all transmissions it must pass the token to the next node on the network. This transmission takes some time (currently the time required to transmit a packet of the smallest size allowed on the network) and a next run is needed when this token is transferred completely.

3.3 Changes to the original TrueTime network simulator

The sources for the original TrueTime simulator were not directly usable; a number of changes were needed. These adaptations can be divided in three categories.

3.3.1 Extraction of the network simulator core to pure C++

The TrueTime simulator is meant to be used in the Simulink environment of Matlab. Therefore its sources are a combination of C++, MEX functions and Simulink blocks. Furthermore the sources provide a lot more functionality than just the network simulator.

The core of the original network simulator has been written mainly in C++, making it relatively easy to extract it from the rest of the TrueTime sources. Only the interface with the other parts of the simulator needed changes. For instance, after extraction, the stand-alone network simulator would not have access to the user-defined properties of the network anymore, as they were stored in the Simulink block objects. Instead, these settings must now be stored in the C++ object that contains the network simulator. The interface for accepting and delivering network packets needed to be changed too. A Network Device Driver has been written that provides this new interface.

3.3.2 Adapting the network simulator to fit into the DSE simulation framework

As described in section 2.2.2 the simulation is build as a PriParallel Construct, like the example shown in Figure 3-3. A simulation step starts at the top with the highest priority processes. As these processes block, the processes with lower priority are executed. When all processes are blocked, the simulation step is finished and the simulation time is incremented. This increment unblocks the processes that were blocked on a SimTimerChannel, so these processes can continue running at the new time point.

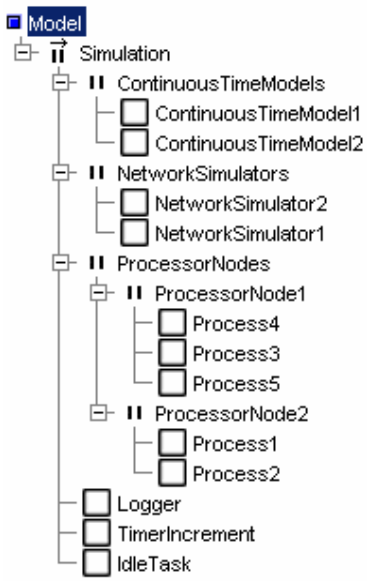

Figure 3-3 Typical simulation layout

Similarly, the network simulator performs a simulation run at a certain time point and calculates a next-run time (see section 3.2.2). When the simulation run is completed, the network simulator should block until this next-run time, otherwise it would prevent the lower-priority processes from executing at all.

The network simulator is made part of the simulation framework by converting it into a CT Process. This is achieved by making the network simulator class (called *NWsim*) a derived class from CT's Process class. The timing of the network simulator is combined with the way timing is performed in the simulation framework by the creation of a SimTimerChannel for the network simulator. The *run*-function of the NWsim Process performs a repetitive sequence of tasks, as shown in Figure 3-4.

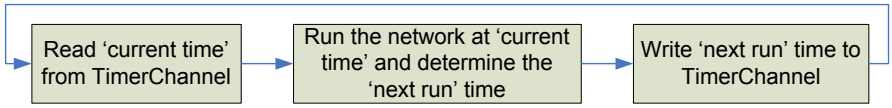

Figure 3-4 Network simulator *run*-function

At the start of the sequence the current time is determined by reading it from the SimTimerChannel. Next, a single run of the network simulator is performed. The single run of the network simulator results in a next-run time, which is written into the SimTimerChannel. The network simulator process will now block on this channel, allowing the other processes in the simulation construct to run.

When calculating the next-run time, the network simulator of course cannot account for nodes that might attempt to transmit a network packet in the near future (before the next-run time). Therefore, every time a new packet is presented to the network simulator, the timer channel of the network simulator needs to be *earlyReleased* (see section 2.3.2 about earlyReleasing SimTimerChannels). In this way, every access to the network will reactivate the NWsim process.

3.3.3 Interfacing the network simulator with the processor nodes

The original TrueTime network simulator uses just two C++ functions to interface between the processes and the network simulator (a *transmit* function and a *reception* callback function). In the simulation framework however, all communication over a network is performed via Remote Channels. A Remote Channel is a specific implementation of a CSP Channel, which can be used for communication via a network between two processes on different processors (see chapter 4).

In order to interface the Remote Channels with the network simulator, a special 'hardware specific' implementation of a Network Device Driver is used. A Network Device Driver is a driver used to interface a Remote Channel with the communication hardware and is explained in detail in chapter 4. Special about this case is that the communication hardware is replaced by the network simulator. This special case is depicted in Figure 3-5, of which the original version can be found as Figure 4-3.

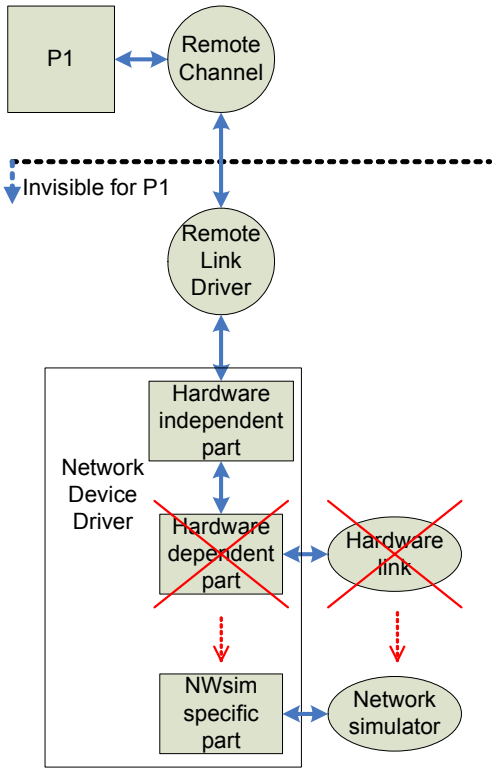

Figure 3-5 Network simulator interface

Packet representation

The Remote Channel structures use objects of type *NWpacket* to represent a single network packet. These objects provide a hardware independent interface between the generic part of the Network Device Driver and the hardware dependent driver part (see Figure 3-6). The network simulator however uses objects of a different type, *NWmsg*, to represent a packet on the network. Besides the usual data, such as the sender and recipient address and the payload data, these objects also contain data structures used by the network simulator. One of the tasks of the network simulator specific interface part (the *NWsimDeviceDriver*) is to convert between these two object types.

Node representation

The network simulator uses objects of type *NWnode* to represent the nodes on the network. To connect the Remote Channel to the network, the NWsimDeviceDriver is inherited from the NWnode class (see the class diagram in Figure 3-6).

It might seem more logical to directly make the processor nodes (*CommunicatingProcess* in the figure) a network node, instead of indirectly via the network device driver. However, if each processor node would automatically be a network node, it would not be possible to have processor nodes that are not attached to any network, nor would it be possible to attach processor nodes to multiple networks. These scenarios *are* possible with the chosen class layout, because any process is allowed to have any number of NWimDeviceDrivers, including zero.

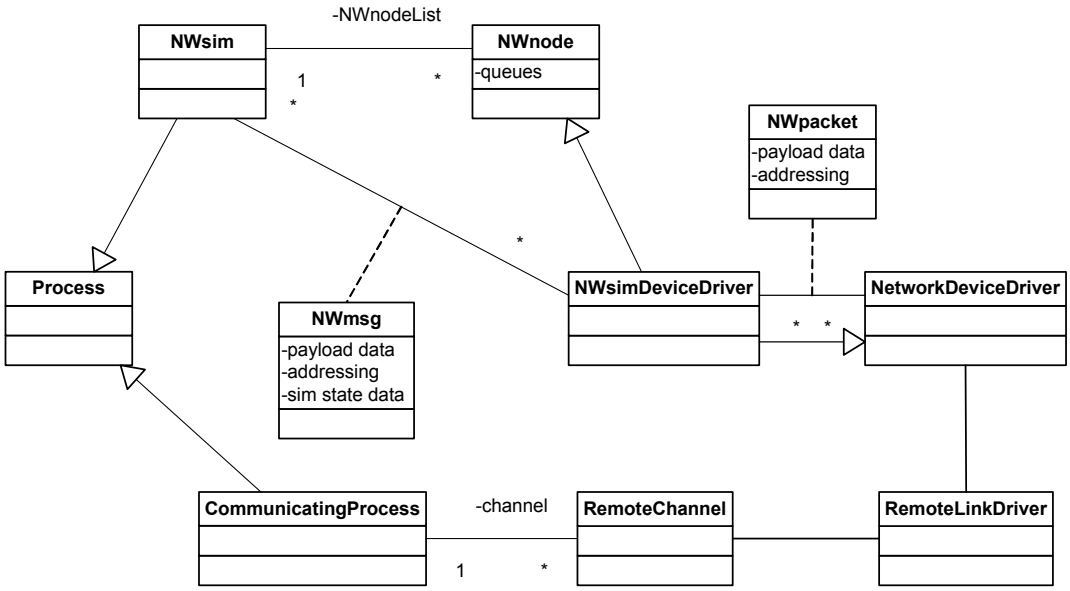

Figure 3-6 Network simulator class diagram

3.3.4 Some minor changes for improved usability

A number of changes have been made to the original network simulator, in order to improve its usability.

Node numbering

The original TrueTime network simulator stores the information about the network nodes in an array. This constrains the numbering of the network nodes: the numbering must start at zero and no numbers may be skipped. While perfectly usable in the Simulink environment, these constraints cause some problems in the CTC++ simulation environment.

The main cause of these problems is that the network simulator objects and the attached nodes are created dynamically, on program startup. The order of creation is not necessarily known (it might depend on the scheduler for instance), which would cause the nodes to be assigned a random number. With random node numbers it is not possible to specify the node to which a specific packet should be sent. This problem is solved by numbering the nodes manually, but now the user must make sure that the numbering starts at zero and does not skip some numbers.

This does not pose any direct problems, but it is not very convenient. If one would like to add or remove a node from the network, the other nodes need to be renumbered. Not only the nodes themselves need to be renumbered, also the numbers of the nodes a certain node communicates with need to be updated.

This inconvenience is solved by storing the node information in a linked list instead of in an array. The result is that arbitrary numbers can be assigned to the nodes. Nodes can now be connected or disconnected from the network without the need to renumber the remaining nodes.

Node number check

When a new node is attached to the network, it specifies its node number. The network simulator searches the list of already connected nodes for the specified node number, to prevent two nodes from having the same number. Furthermore, a node is not allowed to register with node number zero, as this number is reserved as the broadcast address. An attempt to register as node zero is detected and refused.

Input queue sorting

On some network interface devices, the transmit queue does not work as a FIFO. Instead, the device scans through all queued packets and picks the packet with the highest priority to send first. An example of this behavior can be found in the CAN bus controller of some Analog Devices' ADSP devices. To resemble this queue behavior, the input queue in the network simulator will be sorted by priority whenever a new packet is added to the queue. As an exception, the topmost packet is not touched during sorting, as it might already be in the process of being transmitted.

The priority used for sorting is taken from the priority field of the network packet object. It is the same priority that is also used for bus arbitration, if supported by the selected network type. When all packets have the same priority value, the input queue behaves like a normal FIFO queue.

3.4 Degrees of freedom

The network simulator provides a large number of configuration parameters, its degrees of freedom. These parameters are described in this section.

3.4.1 Data Link layer type

With this parameter the type of Data Link layer is selected. This setting influences the travel of the packets from input queue to output queue, as shown in Figure 3-1 and Figure 3-2. Possible settings are:

CSMA/CD

This option selects the Carrier *Sense Multiple Access with Collision Detection* Data Link layer. This layer type is used in for instance Ethernet. The communication medium is assumed to be shared amongst all nodes. This means that Ethernet systems with network hubs can be simulated with this option, but systems with network switches cannot. For (single) switched networks a separate option, SFDSE, is available.

A node is allowed to send if it does not sense a carrier signal on the network (i.e. no transmission is in progress). Because of transmission delays on the cables it is possible that a node starts transmission without sensing that another node had also just started transmitting. This is possible during the *collision window*, a fixed time interval set in the header file of the simulator (it can be adjusted, but this requires a recompile and is generally not needed. Of course, the sources can be changed to make it a configurable parameter).

When a collision is detected, all nodes that participate in the collision will abort their transmission and will calculate a retry (or *back-off*) time. This back-off time is stochastic, to prevent all nodes from retrying at the same moment, which would result in another collision. The transmission time is dependent on the size of the packet and the bitrate of the network. Completed transmissions result in the packet being moved from the input queue of the sender to the output queue of the recipient.

CSMA/AMP

The *Carrier Sense Multiple Access with Arbitration based on Message Priority* layer type is for instance used for the CAN bus. It works similar to the CSMA/CD layer type, but there is a difference in how collisions are handled.

In the CSMA/AMP protocol, every packet has a priority. This priority is used when a collision occurs. A collision will be detected by all nodes except the one that is sending the packet with the highest priority. The nodes that detect a collision will immediately stop transmission. The transmission of the packet with the highest priority will not be disturbed by the collision, so it can continue normally.

Because a collision is non-destructive, there is no need to calculate a random back-off time, as is done in the CSMA/CD protocol. Instead, the nodes that had to abort their transmission may retry as soon as possible, which means as soon as they detect that the bus is idle again. The performance of the network even does not suffer from collisions.

FDMA

The *Frequency Division Multiple Access* Data Link layer type is different from the others, in that multiple transmitters can use the medium concurrently. Each node is assigned a fixed portion of the total bandwidth of the medium, which they can use independently. This scheme can be compared with that of radio broadcasts: multiple transmission stations can send information, each at their own frequency band. The width of a frequency band determines the bitrate that can be achieved on that band. As each node transmits on its own band, collisions will not occur. The relative bandwidths assigned to the nodes are determined with the node parameter *transmit bandwidth fraction*.

TDMA

Time Division Multiple Access uses a shared medium. Each node is in turn granted a fixed period (a *time slot*) in which it is the only node allowed to send. Again, collisions cannot occur. If the currently selected node has nothing to send, the remaining time in the time slot is awaited, which means that during this time the network will be unused. On the other hand, if the slot time has elapsed while the node has not finished transmission yet, its transmission will be interrupted and the next node is assigned a time slot. The interrupted transmission can be resumed when the node is assigned a time slot again. The slot time can be configured with the corresponding parameter *slot time*, the (fixed) order in which the slots are assigned to the nodes with the parameter *node order*.

RR

The *Round Robin* Data Link layer type is comparable to TDMA, because the nodes are not allowed to send except when it's their turn. But where TDMA uses a schedule with fixed times, Round Robin uses a token packet to signal that the next node is allowed to send. A well-known example of the use of this layer type is the Token Ring bus.

The order in which the nodes are allowed to send is set with the parameter *node order*. As soon as a node has finished its transmission, it passes the token packet to the next node in the list, which is then allowed to send. A node can keep on transmitting as long as it has data available to do so. There is no time limit for the ownership of the token, so a continuously transmitting node can block the whole network.

SFDSE

As already mentioned in the description of the CSMA/CD protocol, the *Symmetric Full-Duplex Switched Ethernet* Data Link layer type is used in Ethernet networks containing switches instead of hubs. The simulator assumes one central switch, to which all nodes are directly connected. As all connections are full-duplex (allowing concurrent, bidirectional traffic) collisions are not possible.

This is the only protocol that uses the *switch queues*, as shown in Figure 3-1. Packets are first transferred from the sending node into the switch memory, then from the switch memory to the receiving node.

The behavior of the switch in situations when its memory buffer is full can be changed with the parameters *switch memory size*, *switch buffer type* and *switch memory overflow behavior*.

3.4.2 Bitrate and transmit bandwidth fraction

This parameter specifies the bitrate of the network. It translates directly into the time needed to transmit a packet of a certain size. In case of an FDMA network, this parameter specifies the bitrate that can be achieved using the *total* bandwidth of the medium. Each node can only use a fraction of this total bandwidth/bitrate. This fraction is determined by the node's parameter *transmit bandwidth fraction*.

3.4.3 Minimum frame size

On certain network types, packets are required to have a minimum size. On an Ethernet network for instance, this is required for a correct functioning of the carrier sensing mechanism. For a CAN bus, a minimum size is specified in the CAN protocol specifications. The network simulator uses this parameter to ensure that all packets meet the minimum size. This also holds for the token packets in a RR network, they are sized according to this parameter. This implies that when a value of zero is chosen for this parameter, that passing the token does not take any time (as zero bytes can be transmitted instantly).

3.4.4 Packet loss probability

This is the probability that a packet is lost before arriving at the receiver. This parameter can be used to simulate noisy connections or other stochastical processes that influence the reliability of the network. The packet loss parameter is applied at the output queue of the network simulator, independently for each packet. This implies that a broadcasted packet (a packet going from one transmitter to every other receiver) might not arrive to some of the nodes. An 'all-or-nothing' packet loss would require the packet loss calculation to be applied earlier in the network simulator (like during transmission).

3.4.5 Pre- and postprocessing delay

As shown in Figure 3-1, packets remain in the preprocessing queue for a while before being moved to the input queue. This time is set with the *preprocessing delay* parameter. It simulates the cumulative time between the moment a packet is available for transmission and the earliest moment it can actually be transmitted. This delay can for instance be caused by slow network interface hardware or a slow algorithm in a network driver.

3.4.6 Collision window

This is not really a parameter, as it has a fixed value, but by recompiling the CT library its value can be changed. The collision window indicates the maximum propagation delay on a network. This determines the time between the moment a node starts transmission and the moment all other nodes can sense the carrier of that transmission. During this period, the other nodes can also start transmission (as they sense the bus to be idle), which would cause collisions. This explains the naming of the parameter.

3.4.7 Slot time and node order

The *slot time* is only used in TDMA networks (as explained in section 3.4.1). It determines the duration of each time slot. The *node order* parameter is used both in TDMA and RR networks. It specifies the order in which nodes are granted access to the bus.

3.4.8 Switch memory size, buffer type and overflow behavior

These parameters are only used in SFDSE networks and determine the behavior of the network switch. A network switch contains a certain amount of buffer memory (sized according to the *switch buffer size* parameter) in which packets are buffered. A buffer is necessary for situations where for instance two nodes both transmit a packet to one and the same receiver node. As the link from the switch to the receiving node can only handle one packet at a time, the second packet will be queued in the switch, so it can be transmitted directly following the first packet.

The memory can be divided among the connected nodes in two ways, specified by the *buffer type* parameter. Each node can be assigned an equal portion of the total switch memory, or the total memory is shared between all connected nodes.

In the first case a queue for a single node can not be larger than the total memory size divided by the number of connected nodes. An advantage is that the memory available to a single node cannot be influenced by data traffic to other nodes. A disadvantage is that the switch memory usage is suboptimal, when only a few nodes require large queues.

In the second case a queue for a single node can grow as large as the total switch memory, leaving no room for the queues of other nodes. This means that nodes influence each other, which is a disadvantage. An advantage is that the switch memory can be optimally used.

When a memory overflow occurs, a switch can do two things. This behavior is specified by the *memory overflow behavior* parameter. The first option is to silently drop (discard) the packet that causes the overflow. The node that transmitted the packet will assume that the packet was transmitted successfully, so it will not try again. The effect is the same as for normal packet loss: higher network protocol layers have to detect the packet loss and handle it, for instance by asking for a retransmit.

The second option is to generate an error signal during the transmission of the packet that causes overflow. On an Ethernet network this can be done with the same signal that is used to indicate collisions. In this case the packet is not lost, but remains in the transmit queue of the sending node. It will keep retrying the transmission until the switch has memory available again to accept the packet.

3.5 Conclusions and recommendations

The TrueTime network simulator used in the simulation environment provides simulation capabilities on the Data Link layer level. The most common layer types are supported, and because of the packet and queue-based design, it is relatively easy to extend the simulator with other layer types.

The simulator assumes that all nodes share the same pre- and postprocessing delays. In a heterogeneous distributed system however these delays can vary from node to node. It would therefore be better to change these delay parameters from network-specific to node-specific.

Another shortcoming of the network simulator is that it lacks a maximum size for transmit and receive queues. In reality most or all network interface hardware have limited buffer memory sizes. A parameter should be added to the network simulator to specify the maximum number of packets that can be stored in the input and output queues.

Not all parameters of the network simulator can always be chosen freely. Some of them are already determined by design choices that are already made. For simulation runs with the network simulator to be as accurate as possible, values for these parameters need to be established, probably by measurements on the real target network, or by extracting them from hardware specifications.

4 Remote Channels

4.1 Introduction

All communication between CSP processes is handled via *channels*. In non-distributed applications only local channels are needed. *Local channels* (see Figure 4-1) are used for communication between processes that both run on the same computer. The implementation of local channels in the CT library uses shared memory for rendezvous and data transfer.

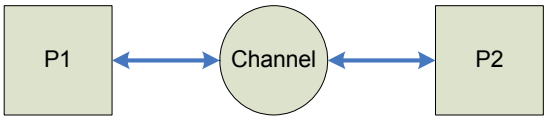

Figure 4-1 Local Channel

The use of shared memory may not be an option in case of distributed software. In most cases the distributed parts of the application are only connected via a communication link, such as a network, a fieldbus or a one-to-one connection. The use of a communication link instead of a local channel should be transparent to the processes. In the CT library originally this was solved by attaching a *linkdriver* to the channel, which is now called an *external channel* (see Figure 4-2). The interface between process and channel does not change, making the use of linkdrivers transparent. The linkdriver handles hardware access, rendezvous and handshaking.

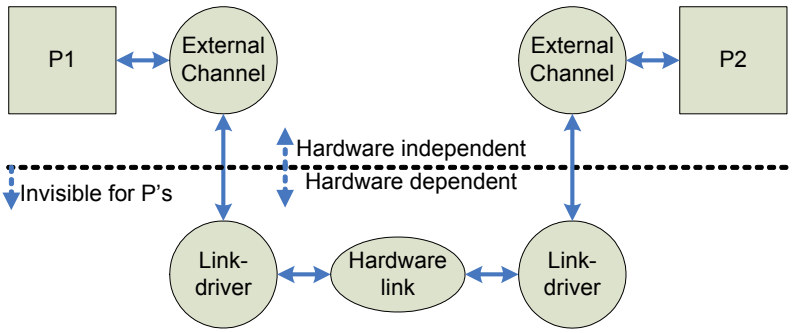

Figure 4-2 External Channel

The concept of external channels has some shortcomings, which make them rather inflexible. The main problem is the lack of *separation* between hardware-specific and hardware-independent operations. If someone wants to write a new linkdriver in order to add support for some hardware, he has to implement the rendezvous and handshaking mechanisms again too. This makes the task of adding hardware support rather tough.

Another problem is the lack of a standard way to *share hardware* between multiple channels. If multiple channels need to be routed over one network interface, which is a quite common situation for multi-node networks or fieldbuses, this functionality must be implemented in the linkdriver. This can make the way to share the interface incompatible between different hardware types, if available at all.

4.2 Remote Channel concept

In order to efficiently explore the design space for a distributed application that uses fieldbuses, one should be able to switch to different network types easily. As mentioned in the previous section, the linkdriver concept in the CT library was not very suitable for this. In (Orlic *et al.*, 2003) a new communication model was defined in which the device specific shared part of the driver (Network Device Driver) is separated from the part associated with every channel directly (Remote Link Driver). In the scope of this project, the design decisions for this communication model were explored once more, resulting in a significantly changed and improved communication model.

The structure of a single remote channel is shown in Figure 4-3. This figure shows the separation of the hardware dependent and the hardware independent parts. Only a specific part of the Network Device Driver (NDD) is hardware dependent. The rest of the structure (the rest of the NDD, the Remote Link Driver and the Remote Channel) does not depend on a specific type of hardware, which makes it possible to re-use those parts for all remote links.

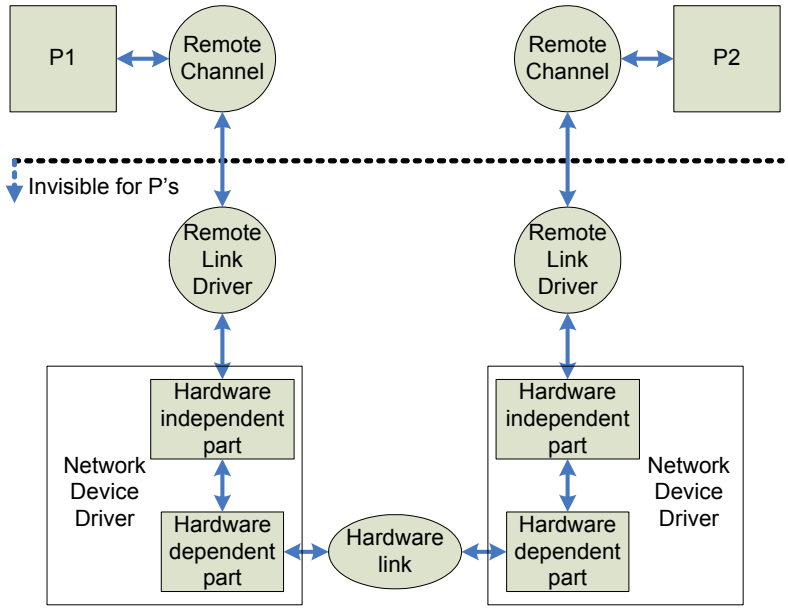

Figure 4-3 Remote Channel

4.2.1 Subdivision between RLD and NDD

The hardware independent part of the implementation is subdivided in a *Remote Link Driver* (RLD) and a *Network Device Driver* (NDD). This subdivision might seem unnecessary at first, because both parts are hardware independent, but it is there to allow multiple channels to share a single hardware interface. Sharing is completely accomplished in the hardware independent part, which means that sharing is always possible, regardless of the hardware used, and that one does not need to think about it when writing a hardware dependent part for new hardware. The hardware independent parts of the NDD, as well as the interface between RLD and NDD, are defined in the basic NDD class. For every type of network device, a new class is inherited from NDD that implements only the hardware dependent part in the device specific way.

All communication elements that are specific to one channel are located in the *Remote Link Driver*. This includes rendezvous mechanisms, handshaking, retransmission and timeout handling. Each Remote channel has its own RLD, which can be configured according to the needs of that Remote channel.

The *Network Device Driver* on the other hand contains all communication elements that are specific to one network interface device. This includes addressing, formatting of network messages (e.g. message headers or checksums), splitting data (to prevent exceeding the maximum message size) and the actual hardware access.

4.2.2 Network interface sharing and addressing

In order to share a network interface, one only needs to instantiate multiple Remote Link Drivers (one for each channel) and connect them to a single Network Device Driver. An example of this is depicted in Figure 4-4. This figure shows a network with three nodes connected to it via network interface hardware. All communication through such an interface is handled by one Network Device Driver. Multiple Remote Link Drivers can be connected to this NDD. Each RLD will only handle the communication for its own channel (shown here as open arrow ends). To allow these multiple connections, the interface between RLD and NDD is standardized (see also section 4.2.3).

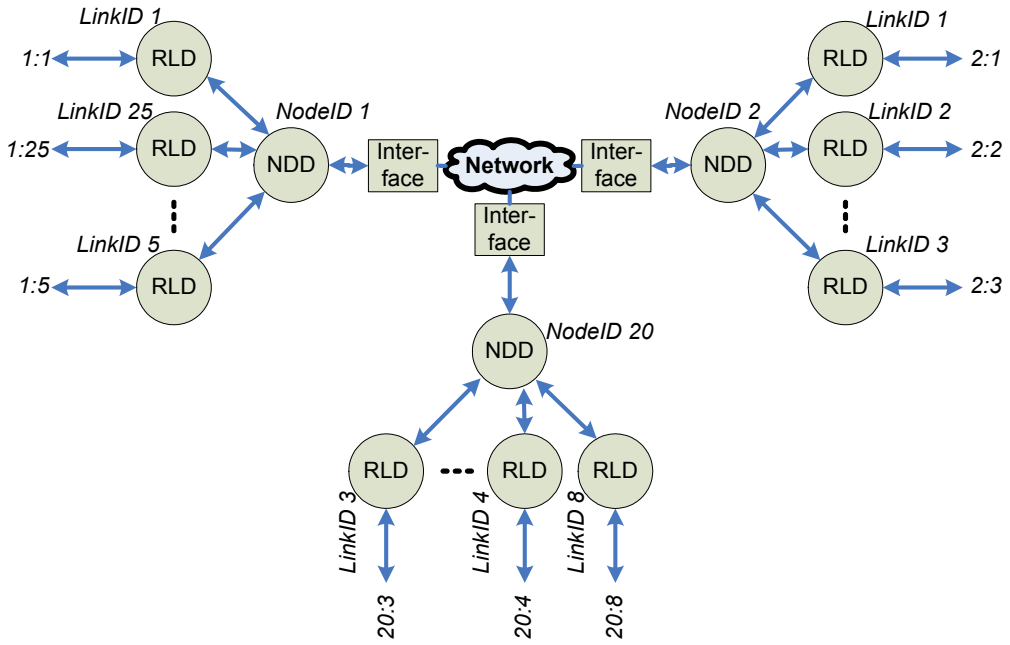

Figure 4-4 Network interface sharing and addressing

Each endpoint of a channel can be uniquely addressed by the combination of a NodeID and a LinkID. The NodeID, which must be unique across the network, is assigned to the Network Device Driver. In the example, the nodes use the numbers 1, 2 and 20. The LinkID, assigned to the Remote Link Driver, does not need to be unique across the network, but only on that specific node. In the example, the numbers 1 and 3 are used twice, but on different nodes. The node and link numbering does not need to be contiguous, allowing channels to be attached or detached from the network very flexibly or even dynamically during runtime.

The current implementation of RLD and NDD support IDs of 32 bits each, allowing for more than four million nodes on a network, each with up to four million Remote channels attached. The already implemented hardware-specific Network Device Drivers however limit these numbers to eight bit each, thereby limiting the maximum number of nodes and links to 256. The reason for this is that larger IDs would increase the overhead of the communication. This was especially a concern for the CAN bus protocol, where the maximum packet size is very limited and larger IDs would not fit in the packet headers.

4.2.3 Interface standards and data naming convention

Different (remote) channels can transfer objects of different sizes. Furthermore, the maximum data size that can be transmitted over the network can differ between network types. In order to be able to attach any Remote Link Driver to any Network Device Driver, the interface between RLD and NDD is standardized.

Figure 4-5 shows the data types involved in communication. The data that the process wants to transfer is called the Object. Due to limitations of the current implementation of the channels in the CT library a single channel can handle only one type of object. Multiple channels however are not bound to one object type; each channel can transfer its own type.

On the interface between RLD and NDD only one data type can be exchanged, namely a *Remote Data Block* (RDB, sometimes also referenced as ‘data block’ or just ‘block’ for brevity). A RemoteDataBlock consists of a data segment and some data about the block itself, like its size, a serial number and the addresses of sender and receiver. The size of the data part is fixed on the CT library level and is the same for all network types. It can be configured at compile-time, to allow optimization of the block size for the application.

The reason to apt for the fixed and uniform size of a RDB is ease of reconfiguration. A fixed size of the RDB allows any RLD to be connected to any NDD. This way, one can dynamically break the association of a RLD with its NDD, and make one with some other NDD. It is for instance possible to implement a mechanism that, in case a network link is broken, will associate the affected RLDs to some of the alternative NDDs that are also capable of delivering RDBs to the same destination node. Ideas from former research (Ferdinando, 2004) concerning fault-tolerant connections can probably be reused.

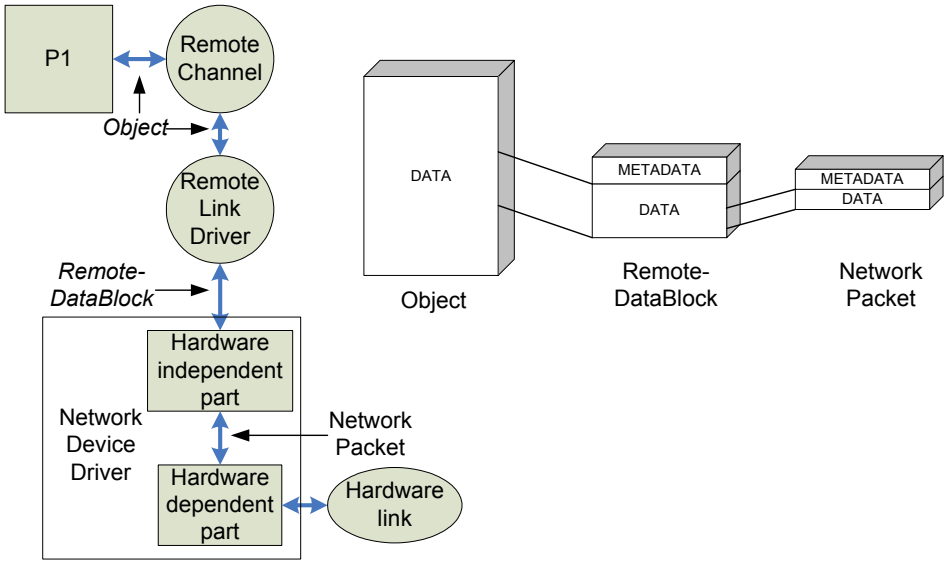

Figure 4-5 Data interface

If the object is larger than the size of the RDB's data segment, the object has to be split up into multiple blocks. This is performed by the Remote Link Driver, which also adds some numbering metadata in this process, to enable the receiving side to reconstruct the object from the blocks. If the (remainder of the) object is smaller than a data block, only part of the block is filled with data and the 'size' metadata is set accordingly. This prevents a performance penalty when small objects are transmitted using large RDB's.

All Network Device Drivers should be capable of handling RDB's of arbitrary sizes. However, only a few types of hardware links are capable of sending arbitrary long messages (e.g. an RS-232 or optical fiber link). Most hardware types impose a maximum to the message size (e.g. eight bytes for CAN or around 1.5 kilobytes for Ethernet). If the size of the data in the RDB exceeds this maximum, the Network Device Driver has to split up the RDB into multiple Network Packets. This is all performed in the hardware independent part of the NDD, leaving the hardware dependent part simple: it only has to inform the hardware independent part about the maximum packet size. Besides this, the hardware dependent part only needs to be able to transmit the contents of a Network Packet (using a single hardware packet), and to put any received message in a Network Packet and hand it to the hardware independent part of the NDD. This makes it very easy to add new hardware support to the library.

4.3 Protocols and options

As already mentioned, the Remote Link Driver is responsible for the rendezvous handshaking, retransmission and timeout handling, if needed. The current implementation of the Remote Link Driver is capable of several handshaking and retransmission protocols. Furthermore, the RLD has a number of other parameters. All choices are made during the construction of the RLD, by passing a Remote Link Config object containing all options. The options cannot be changed once the RLD object is instantiated, but multiple concurrent RLDs can use different options. Please refer to appendix A for additional information about the implementation of these protocols.

4.3.1 Unreliable, asynchronous communication

This is the simplest protocol available. It has no rendezvous behavior and it cannot guaranty that all data will arrive at the receiver. Despite the lack of these functions, it can be the best protocol to choose in some cases. Consider for instance the situation where the most current value of some variable is sent at a very high repetition rate. When a packet is lost, it might be better to just wait for the next updated value, instead of waiting for a transmission timeout followed by a retransmission of the same (old) value. However, if reliability or rendezvous behavior is required, one of the other protocols should be used.

Note that 'unreliable' does not say anything about the validity of the received data. By adding measures like a checksum (in hardware or software), one still can guarantee the correct contents of received data. The only unreliable part is *whether* transmitted data arrives at the receiver. Using this protocol, that can only be guaranteed if the underlying hardware provides this reliability to the connection.

When this protocol is used, the communication is effectively reduced to unidirectional communication. As soon as the writer becomes ready, it transmits all data, without waiting for the reader to become available. When all data is transmitted, the writer leaves the channel without waiting for any response from the reader.

The protocol can support input guards without additional communication. The use of input guards has some limitations, because since the communication itself is unreliable, so is the triggering of the guard. However, the protocol can guarantee that the guard is signaled *if* data is received.

Adding support for output guards would have been possible, using the same methods as in rendezvous protocols, but it was left out for two reasons. First, it would require bidirectional communication instead of unidirectional, adding extra requirements on the hardware link and a relatively large overhead compared to the data transfer itself. Second, output guards would have little use since the protocol does not guarantee their signaling, making them too unreliable for practical purposes.

As a limitation in the current implementation, the unreliable, asynchronous protocol can only be used if the complete object fits into one network packet. Only if packet loss can be ruled out this protocol works properly for larger objects too.

4.3.2 Rendezvous communication, object based acknowledgment

This protocol ensures that both reader and writer have arrived at the channel before either of them can leave again. The communication is reliable, but (as in the asynchronous protocol) this does not apply to the *contents* of the data blocks. It is still possible that data arrives corrupted. To prevent this, some data security measures need to be taken, for instance by using checksums or redundant data. This can be done either on a higher level, for instance by checking the complete object for data integrity, or on a lower level, by guaranteeing the data integrity of every data block or network packet. The latter could also be performed in hardware. The CAN bus for instance does not guarantee the arrival of a packet, but it does guarantee the data integrity *if* a packet arrives.

Because only one acknowledgement is used for a complete data object, the penalty for the loss of a data packet is very high: the complete object needs to be retransmitted. On the other hand, using only one acknowledgement minimizes the protocol overhead and allows for all data blocks to be sent back-to-back (i.e. by using a transmit buffer the network interface hardware can start transmission of a new block immediately after transmission of the first block is finished, without any unnecessary inter-packet delay). These properties make the protocol most suited for relatively small objects (relative to the size of a data block) and for networks with a low chance of packet loss.

As with all protocols the Remote Link Driver supports, signaling of guards is currently not implemented. However, support for both input and output guards was taken into account when

designing the protocols, and some directions are available in the source code to aid in the actual implementation of the guards.

4.3.3 Rendezvous communication, block based acknowledgment

This protocol is very much comparable to the object based rendezvous protocol. The main difference is that an acknowledgement is sent for each data block. When a packet is lost, only the corresponding data block needs to be retransmitted. The price for these small retransmissions is a larger protocol overhead. This protocol is therefore most suited for use with relatively large objects and for networks with high chances of packet loss.

In the current implementation of this protocol the writer has to wait for a data block to be acknowledged before sending the next block. The performance of this protocol could be improved by allowing the writer to send the data blocks back-to-back and wait for all acknowledgements in parallel. The first implementation ideas were based on this principle. Finally a simpler implementation was chosen as a first step, because sending the data blocks back-to-back would require a quite difficult construction for the timeout-waiting, with multiple different timeouts running in parallel.

4.3.4 Reliable, asynchronous communication (unavailable)

Looking at the above mentioned protocols, a protocol that provides reliable but asynchronous communication seems a logical next combination. Although asynchronous channels are not allowed in a pure CSP-conforming program, as all communication needs to be rendezvous-based, it can be a very usable protocol type in practice.

Unfortunately such a protocol cannot be supported by the designed structure of RLD and NDD. Because both objects are passive, the threads of the (active) sending and receiving processes are needed to perform any transmission. As for a reliable protocol bidirectional communication is needed, this implies that both the writer's and reader's threads must be available. This makes the protocol automatically a rendezvous protocol.

4.4 Implementation details

4.4.1 Dynamic resizing of position numbers

When an object is split up in blocks, or when a block is split up in packets, a position number needs to be assigned to those segments, because otherwise the object cannot be reconstructed correctly. The size of these numbers determines the maximum number of segments that can be discerned, so using position numbers with a large number of bits allows for large objects and data blocks. On the other hand a large number of bits will increase the overhead. Especially for the transmission of small objects more network packets are required than strictly necessary.

This tradeoff is prevented in the current implementation of both the Remote Link Driver and the Network Device Driver, by allowing the position numbers to resize dynamically, based on the size of the object, the (fixed) size of the Remote Data Block and the maximum size of a network packet.

As an example, consider a node on a CAN network, where the packet size is limited to eight data bytes. The dynamic resizing makes it possible to send an object of hundreds of kilobytes through one Remote channel, while an eight-byte object sent through a second Remote Channel (but using the same Network Device Driver) will fit in just one network packet. In this example, one would typically use the block based rendezvous protocol for the large object and the asynchronous protocol or object based rendezvous protocol for the small object.

4.4.2 Other configuration parameters

Besides the communication protocol, some additional parameters of the Remote Link Driver can be modified. Most of them are related to the retransmission of data blocks.

The *retransmit timeout* specifies the timeout time used at the writer's side. When this timeout expires, the corresponding block or object is assumed to be lost. As mentioned in the appendix, the timeout at the reader's side must be larger than at the writer's side. This is enforced by making the reader's timeout a multiple of the writer's timeout.

When a timeout occurs, the writer has two choices: retransmit or abort. This choice is controlled by the *retransmit count*. Retransmission can be disabled completely by setting the retransmit count to zero. Note that special care is needed in case of a transmission abort, because the reader can be left behind in any state, including an indefinitely lasting blocking state.

When block-based handshaking is used, one can select if the number of *retransmissions* may be *reset* after a data block was acknowledged. If so, transmission of the next block starts with a fresh number of allowed retransmits. If not, the specified retransmissions must be enough for the complete object, or transmission will be aborted. Enabling this option makes communication somewhat more reliable, especially when the number of blocks is very large. When this option is disabled, the maximum time the transmission may take is better controlled.

Finally, a *priority* may be assigned to every channel link. This priority is used to fill the priority field of network packets, but of course it has a meaning only on network types that support priorities. The priority with which outgoing network packets are transferred to the transmit queue of the network hardware or with which incoming packets are handled locally at any of the nodes is actually determined by the priority of the thread requesting network services. The priority of the packet itself is only used from the moment the packet is placed in the hardware's transmit buffer up to the moment the packet is delivered and handed from hardware to software.

4.5 Conclusions and recommendations

The presented concept of a separate Remote Link Driver and Network Device Driver has proven to be a flexible extension to the external channel concept. It allows multiple Remote Channels, with objects of various sizes, to be transferred across very different types of network interface hardware. A large number of parameters are adjustable, creating a large design space. The separation between hardware dependent hardware independent parts makes is relatively easy to add support for new hardware.

The assignment of an RLD to an NDD is currently a static assignment, made at design time. A method of dynamically switching these assignments should be implemented to create fault tolerant Remote Channels by use of redundant network connections. Ideas from previous research on fault-tolerant connections can be applied.

A shortcoming of the current Channel implementation in the CT library is that only one type of object can be transferred over one Channel. If several different types of objects need to be transferred from a node to another, this currently requires as many Channels as there are object types, all parallel to each other. The Channel implementation should be changed so that in such a case only one 'multi-purpose' channel is needed.

As already mentioned, when designing the Remote Link Driver support for input and output Guards was taken into account, but not implemented, as it would have been too much out of the scope of this project. Support for Guards should be implemented, to make the Remote Channel usable in for instance Alt constructs.

Improvements in the implementation of the block-based rendezvous protocol should make it possible to send all Remote Data Blocks in a continuous stream and then wait for all acknowledgements, instead of waiting for the acknowledgement after each block. This could give a significant performance improvement. It should also be investigated if it is possible to combine multiple acknowledgement packets into one packet, as this could improve the protocol's performance even more.

More protocols, such as multicasting protocols, could also be added. Addition of reliable, asynchronous protocols however requires a completely different thread architecture, in which RLDs should be active processes. Care must then be taken to synchronize the communication between RLD and user-process, possibly by using a local channel in between.

Unfortunately the node and link numbering cannot be automated, as it would then be impossible to specify the node number to which an RLD should send. It should however be possible to use a tool like gCSP to automatically generate this numbering. The different Remote Channels can then be drawn graphically between two processes and the node and link ID properties, both of the local and remote side, can then be filled in. The gCSP tool could also be used to configure the parameters of the Remote Channels, which would be more convenient than specifying them when constructing a Remote Channel.

5 Logging

5.1 Introduction

Logging can be used to keep track of the states or changes in a system, in order to monitor the functioning of the system. Although its importance is often neglected in practice, it is extremely useful in most phases of both the *development* (debugging, simulation, verification) and the *deployment* (monitoring, safeguarding) parts of the life cycle of software. In case of a system failure, logged data can provide very helpful information about the conditions that led to the failure.

In general, the software developer decides which parts of the software (so called software entities) might provide useful information, and enables those entities for logging. Besides these observed entities there are usually also one or more external observing entities that will collect the observations. The final destinations of the logged information or its derivatives are usually the screen, files and/or databases.

The process of logging can be divided in a number of logical components. Such a division is presented in section 5.2. After the logical components are defined, the design space for each component is explored in section 5.3. Of course such an exploration can never be complete, but it is a good starting point for further research. Based on this overview of the design space, section 5.4 describes how the logging is implemented in the two case studies, which are described in chapter 6. Conclusions and recommendations can be found in section 5.5.

5.2 Logging model

5.2.1 Local logging

The logging process can be divided into several activities or logical components: *observation* (or *data generation*), *storage* and *post-processing* (see Figure 5-1). The latter component includes deriving new data from the logged data, for instance mathematically or statistically, and (selectively) displaying the logged or derived data.

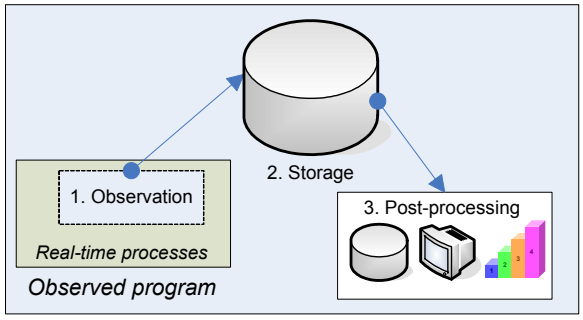

Figure 5-1 Local logging

Performing all these activities locally (i.e. inside the observed program) would have some major drawbacks:

- A crash of the observed program will destroy all locally stored log data.
- The system on which the application runs (the target) might simply not have the required facilities available, such as storage or a suitable user interface. To be able to perform the post-processing activities for such targets, one or more additional physical components need to be introduced.
- The behavior of the program can change to a large degree. The problem of measurements influencing the measured quantities is quite common and can also occur when logging, for example:
  - The processor might not have enough processing power left to handle the extra workload.
  - Writing to disk or screen can take a non-bounded amount of time, which can cause deadlines to be missed.

5.2.2 Distributed logging

Because of the issues described in the previous section, an alternative approach is assumed in the remainder of this chapter. This model assumes that at least two separate physical components are used. In this context, a physical component is a piece of software that can run independently, whether it is on a separate computer, or as a separate system level process.

In addition to the logical components *data generation*, *storage* and *post-processing*, logging now also contains a *transmission* activity. Transmission exists between any two logical components that are situated in different physical components. The transmission itself is implemented both in the sending and in the receiving physical component.

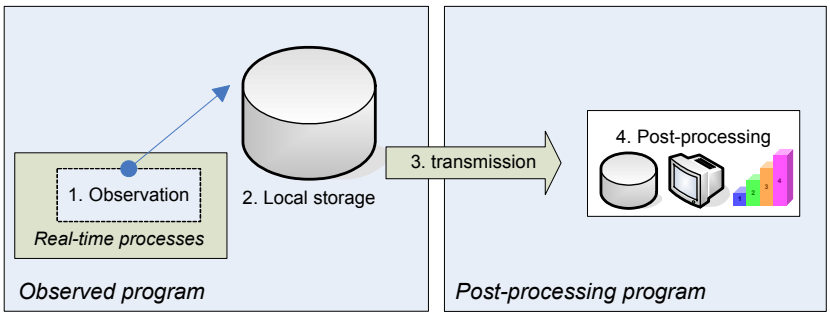

Figure 5-2 Distributed logging

The logging model can thus be divided into four logical components (see Figure 5-2): *data generation*, *storage*, *transmission* and *post-processing*. These logical components will be mapped onto the actual physical components in any way that suits the application.

5.3 Design space

Depending on the purpose of logging one can distinguish between logging performed during real-time execution on real targets and logging performed during the (non real-time) debugging of software, which can be in-target or in a simulation. To be as flexible as possible, the logging should utilize the same mechanisms in both real-time and non real-time situations. In the non real-time case, some additional mechanisms could be implemented, provided that they do not influence the program execution (with the exception of the execution speed). The next sections describe alternatives for the logging implementation in the CT library.

5.3.1 Identifying objects

In the observed program, multiple objects of the same type are capable of generating the same type of log data. The logging model needs a method to be able to distinct between those two messages. This requires some kind of *object identification*.

The easiest way to identify an object is by its memory address. Every object has an address (which is accessible via the 'this'-pointer in C/C++), so this method does not require additional storage. Pointers (or addresses) also have disadvantages: their data format depends on the computer platform used, for example they range from 16 bits on the ADSP platform up to 64 bits on the 64-bits platforms. Furthermore they can be ambiguous: if one object is deleted and a second object is created in the same memory area, the second object has the same identifier as the first one. This ambiguity can only be solved if the logger keeps track of all object creations and deletions. The risk is present that loss of these creation or deletion events leads to misinterpretation of the logged data. Finally, addresses have no sensible meaning to the user, they are just numbers. Therefore some extra measures need to be taken to make the results meaningful.

An alternative for identifying an object is to assign a unique identifier (a number, or a name) to each object when writing the program. This way the object can have sensible names, which is an advantage.

On the other hand, all objects need some extra memory to store the identifier (for instance in a symbol table), and assigning the identifiers by hand is both time consuming and error-prone.

Another alternative is to automatically assign identifiers in run-time. When an object is created, it asks for its unique number at a centralized administrative object. This alternative does not require the user to assign unique identifiers and can, depending on the type of identifier used, be platform independent. This alternative also has the same disadvantage as the first alternative, namely that the identifiers have no sensible meaning to the user.

5.3.2 Data generation

Triggering

The observation of an entity can either be *time-triggered* or *event-triggered*. When doing time-triggered logging, the application generates logging information periodically. It has the advantage of a fixed and thus predictable execution time and bandwidth, but it also means that logging actions are performed, even if the state of an observed entity is unchanged. This can be quite resource-inefficient, which is unacceptable in some situations. Furthermore, when multiple state changes occur between two time triggers, it is possible to miss one or more states of an observed entity.

Event-triggered logging on the other hand only generates logging information when an event occurs. Such an event can be almost anything, in the example of Figure 5-3 an event is generated when the value of the observed variable changes. Triggering on events ensures that no information is missed, but at the downside the event rate, and thereby the required logging bandwidth, is hard to predict.

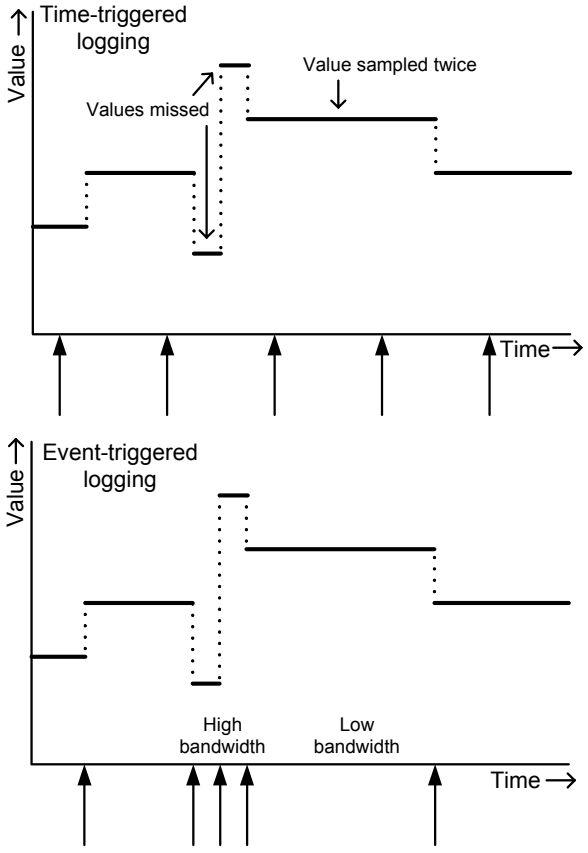

Figure 5-3 Time-triggering versus event-triggering

Initiation

The location from where the observation is initiated can be situated inside the observed entity (*autonomous observation*) or in additional entity belonging to its environment (*external observation*).

In the *autonomous observation* scenario, the observed entity will perform the observation at moments determined by the chosen triggering algorithm. The moments the resulting data will be passed to the environment are also determined by the observed entity. In this case, the communication between the observed program and its environment can be unidirectional (see Figure 5-4).

Autonomous observation does not imply that the communication between the observed entity and its environment can only be unidirectional. The environment can for instance decide to enable or disable some event types or sources, by changing the filter settings in the observed entity. Generation of the actual log data is however still initiated from within the observed entity, so the observation in this situation is still autonomous.

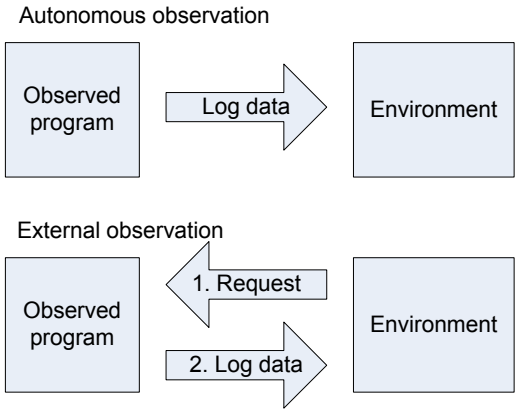

Figure 5-4 Autonomous observation (above) versus external observation (below)

With *external* (or *polling-based*) *observation*, the observed entity is polled by an external process. This external process can again base its polling intervals on the preferred triggering algorithm. Event-triggering in this case is limited, in that only events can be used that are visible to the environment. Events internal to the observed program cannot be used as trigger in this case.

When using external observation, the observed application does not have its own internal triggering mechanism; it simply waits for a poll. This can possibly decrease the resource usage inside the observed application, because it only logs when its environment (the external observer) needs to obtain certain information.

Categories of generated data

For simulations, when timely execution is not really an issue, it is possible to generate log data for virtually every event. It depends on the user's work which data types might be interesting enough to log:

- When working at the CSP abstraction level, one might be interested in *process tree hierarchies* and occurrences of *channel transactions*.
- When working with processes of a higher complexity, one might also want to know what happens inside a process. In this case logging of *variable values* is a good option. Another option is to allow *user-defined messages* to be logged, comparable to a C-programmer using 'printf'.
- From a failure analysis point of view one might be interested to track *exceptions*, *warning* and *error messages*.
- From a memory management validation and optimization point of view one is interested in tracking *stack usage* of processes and *memory (de)allocation*.
- When performing profiling one is interested in the absolute and relative (percentage of CPU time) *time spent* in various functions and processes.
- From a real-time point of view one might be interested in tracking *time constraints*, *priorities* of processes, *scheduling decisions* and the contents of the *ready queue*.

- From an allocation point of view it is interesting to track the *CPU utilization* of the nodes and the *bandwidth utilization* of networks. During the post-processing phase one could derive statistical figures from these data, which can, together with the information mentioned in the previous item, aid in the optimal allocation and scheduling of the system (Sunter, 1994).
- For users working on the CT library itself, all data coming from the part of the library they are working on might be of interest.
- In almost any case *warning* and *error messages* are worth to be logged.

Most of the above mentioned information is only useful during the development phase of an application. Real-time logging on the other hand is primarily concerned with the changes in values of variables that are important for the behavior of the overall system. In order to support post-execution failure analysis, all *error* and *warning messages* should be recorded as well.

5.3.3 Filtering

Filter location

With respect to the data generation two approaches are possible, see Figure 5-5. The first approach is to let the observed entity always generate all available data. At a later stage in the logging model the unneeded data can be filtered out. An advantage of filtering at the post-processing stage is that it is a reversible operation, because a PC is capable of storing both the unfiltered and the filtered data at the same time. One could apply a filter to the data, process the results, undo the filter and apply a different one. This way it is possible to get multiple results without having to re-run the observed process with different log filter settings.

The main disadvantage of filtering during post-processing in case of real-time logging is the large requirement on the local storage and the transmission bandwidth to be able to log all data. In many cases these requirements cannot be met so the local data storage will fill up and data will be lost.

In case of (non real-time) simulation and debugging data loss is unnecessary. Instead, the execution of the simulation can slow down in order to keep up with the transmission rate, so this approach has only the small disadvantage of a somewhat slower execution of the simulation. As the simulation clock is independent of the real clock time, this is not really a problem, only a minor inconvenience.

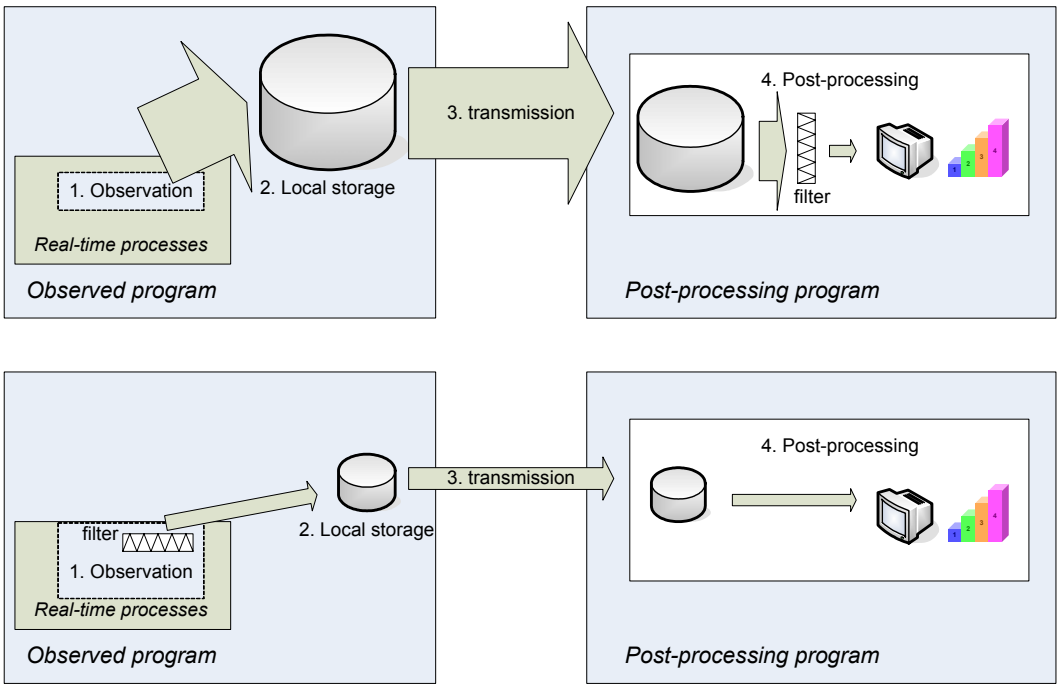

Figure 5-5 Filtering during post-processing versus filtering during data generation

The second approach is to let the observed entity only generate logging data if the user enabled that particular message type. Because of these decisions to be made, the logging code inside the observed

application will be somewhat more complex, requiring more memory or processor resources. On the other hand the logging will require much less storage space (both local and post-processing) and bandwidth, depending on the chosen filtering settings. A disadvantage of filtering during observation is that it is irreversible: once some data is filtered out, it cannot be recovered without re-running the observed program.

A combination of both approaches is also possible and will give a good trade-off between memory and bandwidth usage on one side and debugging flexibility without requiring a re-run of the application on the other side.

Filter types

Filter based on debugging mode

All unneeded logging code can be removed from the executable when not debugging to decrease the size of the code and to increase the execution speed. This can be implemented easily, for instance using conditional compiling. When the debugging code is disabled, the performance of the application can be the same as if there was no possibility for logging at all. This can be achieved by using inline functions and compiler optimization, but the results that can be achieved depend highly on the compiler used. When there is also need for logging during normal execution of the software, it is not possible to remove all logging code from the finished software.

Filter based on object

When using this filter type, the user has to select which objects will be observed, for example by passing an extra argument to the objects constructor or by calling a member function for that object. This gives a rather fine-grained control over the logging and is easy to use, but has some disadvantages.

The main disadvantage is that this filter cannot be applied to data generated from objects that are hidden inside the CT library. The user does not necessarily know about the internals of the library, so he cannot enable or disable logging for the right objects. Furthermore, each object has to keep track of whether it is enabled for logging or not, which can cause a considerable overhead for large numbers of (composite) objects. This is however not an issue for non real-time simulation.

Filter based on time

To reduce the required storage space for the logged messages, while still allowing detailed data series, one could enable logging for just a small period of time. The decision whether logging should be enabled or disabled can be centralized. The logging interval can start at a user-specified time, but it can also start when certain criteria are met. The latter function is comparable to the single-shot triggering of an oscilloscope. Of course for this filter to work some timing source needs to be provided (for instance a real-time clock or a timer), or the logging interval can be determined externally, for instance from a data processing environment.

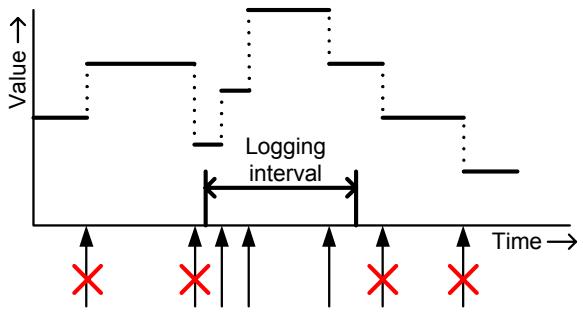

Figure 5-6 Time-based filter

Filter based on category

All observable events can be divided into several global categories. Only when the event belongs to a category that is enabled, it will be logged. A list of categories needs to be stored in the application and all events must belong to (at least) one category. An example of the usage of this filter type (together with filtering based on priority) is the implementation of the user-space logging on a typical Linux

system that uses the *syslogd* logging daemon, where all log messages are assigned to a category and have a priority or importance. Only messages of certain categories and with priorities above a certain threshold are logged.

Filter based on decimation

In signal processing applications decimation means going to a lower sample rate by taking only every n'th sample of the original. The same idea can also be applied to logging by only generating data on every n'th occurrence of that type of data (see Figure 5-7). This idea should be combined with at least one of the other criteria, to be able to set a different decimation factor for different types, categories or priorities of data. Different decimation factors are necessary, because some rare-occurring or very important data, like for instance error and warning messages, should be logged always (i.e. a decimation factor of one).

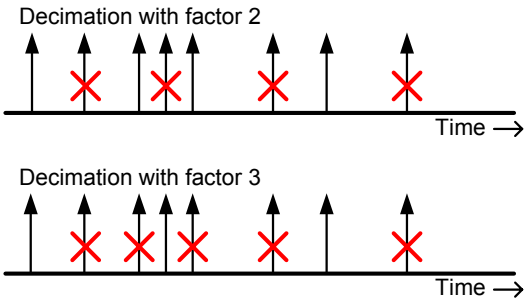

Figure 5-7 Decimation filter

Filter based on priority/importance

Log data will only be generated when its importance is equal to or higher than a user-specified level. This criterion is useful for reducing the amount of unimportant data. This criterion can be used to separate for instance informational, warning and error messages. However, it can not be used to focus on one particular problem, since events from all application and library parts, if important enough, will be logged. This logging criterion is used in for example the Linux kernel, when using the *klogd* kernel log daemon.

5.3.4 Storage

As shown in Figure 5-8 the logging model contains two locations where log data is stored. The first location is local to the observed program. This *local storage* is used to temporarily store the log data, until it can be transmitted. It acts as a buffer between the data generation component and the data transmission component. As memory is often a scarce resource on the component that contains the observed program, the local storage is often limited in size.

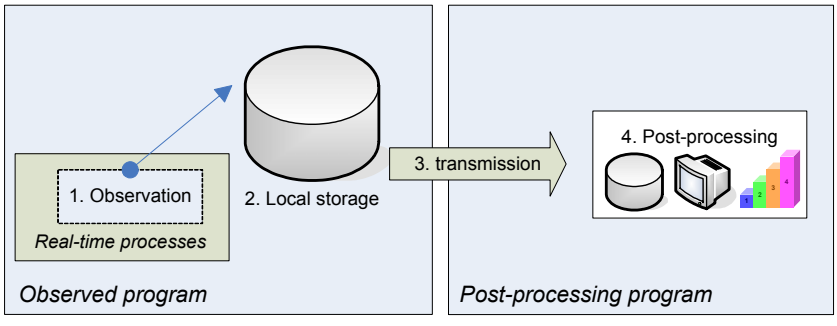

Figure 5-8 Logging model

At the post-processing side the received log data is stored again. This *post-processing storage* (for instance memory, a file or a database) has generally very relaxed size constraints (if any at all), as opposed to the local storage. In many cases post-processing storage is permanent, but it can also be temporarily if the logged data is processed immediately and does not need to be kept for later use. The remainder of this section only deals with the local storage.

Purpose of local storage

In the non real-time situation (e.g. simulation), the minimum requirements for the local storage are not that demanding. In its simplest form, its only purpose is to hold the logged data, until the transmission component can transfer the data to the post-processing component. As long as the storage is occupied, the simulation will wait (block) until the storage is vacant again.

For real-time logging the storage serves quite a different purpose. It acts as a boundary between the real-time processes, which have the highest priority, and the process that handle the transmission of the data, which has a lower priority. This boundary must prevent the real-time processes from being influenced by the transmission process. Without the local storage as a buffer this could otherwise happen, for example if the transmission process uses functions that can cause unbounded delays.

Stating that logging processes have a low priority compared to the real-time control system processes of course does not imply that logging is not important at all. In many systems logging is (almost) as important as the real-time control processes. In such cases the logging process can also be stated to be a real-time process, and care should be taken that enough time is available to perform logging. But also in these cases the control processes have a higher priority then the processes responsible for logging, as logging should not influence the behavior of the control system.

The local storage also acts as a buffer between two data rates: it is filled at the rate at which the log data is generated and is emptied at the rate of transmission. When the data generation rate is higher than the transmission rate, the buffer will fill up; when it is full data will be lost. Dependent on the implementation of the storage, the lost data can consist of the oldest or newest data, some random data or the data with the lowest priority or importance.

There is one special case of losing data: the overwrite array. Provided the number of objects to be observed is known, each object can get its own reserved position in the storage. Each update of a log value overwrites its previous value. This type of buffer is especially useful when used in combination with time-triggered or externally initiated logging, because all values are updated at once and represent a ‘snapshot’ of the complete system (compare to the method of logging used in (Buit, 2005)).

Types of storage

The way how the log data is stored is of great influence on its usefulness for real-time logging and on its relative performance. The storage types discussed in this section will be rated using the following criteria (summarized in Table 5-1):

Storage type:	Real-time	Memory efficiency	Priorities	Message types	Externally observable
One memory block, fixed message size	+	+/-	+/-	+/-	+/-
One memory block, variable message size	+	+	-	+	+/-
More memory blocks, variable message size	+	-	+	+	+/-
Linked list(s)	-	+	+	+	-
Linked list(s) with pre-allocated objects	+	-	+	+/-	-
Linked list(s) with memory pool	+	+/-	+	+	-
Overwrite array	+	+	n/a	-	+

Table 5-1 Overview of storage types

- Suited for real-time logging: not every storage type can be used for real-time logging. If for instance dynamic memory allocation is used, this can cause unbounded waits, which is undesirable for real-time logging.

- Memory efficiency: some storage types cannot use the allocated storage memory very efficiently, especially when a large variety in size of the different log message types exists.
- Suited for multiple priorities: logged data can be assigned different priorities (see section 5.3.3). In this way, when the storage is already filled with messages, it would still be able to provide additional storage capacity for higher priority messages, by discarding some messages of lower priority.
- Suited for multiple message types: different types of messages will contain different elements. The storage should allow for at least a number of different types.
- Suited for external observation: if the logging is based on external observation, it should be easy for the environment to acquire the log data, for instance by just copying a single continuous block of memory. Ideally, there should be no need for the environment to lock the storage when accessing it, as this would block the observed entity.

Continuous memory block, fixed message size

A memory block is a continuous portion of memory of a fixed size. It can be pre-allocated, requiring no additional memory allocation during the rest of program execution, which makes this type of storage suited for real-time applications. As all messages occupy a fixed size, the memory block can be seen as a linear array. This simplifies searching the storage for free locations (or for low-priority messages that can be replaced) and it prevents memory fragmentation. The effort required to find and replace a low-priority message is however dependent on the buffer size, so for large buffers this will take quite some processor time, especially when compared to the solution with *multiple continuous memory blocks*.

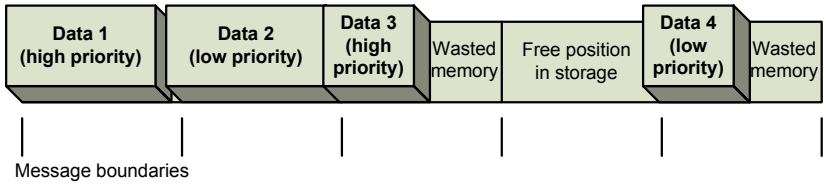

Figure 5-9 Continuous memory block with fixed message size

Allocating multiple memory blocks, but all with the same purpose, also falls under this category (compare with section *multiple continuous memory blocks*). An example of this is to allocate two memory blocks, so that one block can be filled and the other one can be transmitted at the same time, without interfering.

Because of the fixed message size, a lot of memory is wasted when messages with little data contents are stored in the buffer. A tradeoff exists between the maximum message size and the memory efficiency of the storage. Support for different message types is somewhat limited, because the fixed message size puts an upper limit on the size of the message. With some extra processing effort this limitation could however be circumvented, by allowing one message to be split up and occupy multiple positions in the storage. For external observation it is not enough to just read the complete memory block. Additional information is needed, for instance about which buffer locations are occupied or free and which block was added or removed last.

Continuous memory block, variable message size

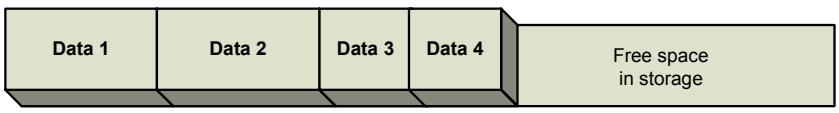

Figure 5-10 Continuous memory block with variable message size

To overcome the problem of the tradeoff between allowed message size and memory efficiency, the message size can be made variable. This way it is possible to store all types of messages, without wasting storage space when small messages are stored. Unfortunately multiple priorities cannot be supported easily anymore: when discarding a low-priority message to create space for a new higher-priority message, the emptied space may be of different size than the new message. This makes a

replacement in the storage difficult, because the new message will either not fit, or leave a gap in the storage space (this problem is known as *fragmentation*).

External observability is comparable to the memory block with fixed-sized messages: the contents of the memory block need to be read, together with extra information about the location of the data in the buffer.

Multiple continuous memory blocks, variable message size

To improve the priority support and still keep a variable message size, one can allocate multiple memory blocks. The different blocks are then used to store different priorities. This means that the storage space for each priority is independent and message replacement is not needed: the lowest priority buffer can be full, while the highest-priority buffer still contains enough space for some high-priority messages.

The disadvantage of this method is its inefficient memory usage: the memory allocated to a memory block will remain unused if no messages of the corresponding priority are generated.

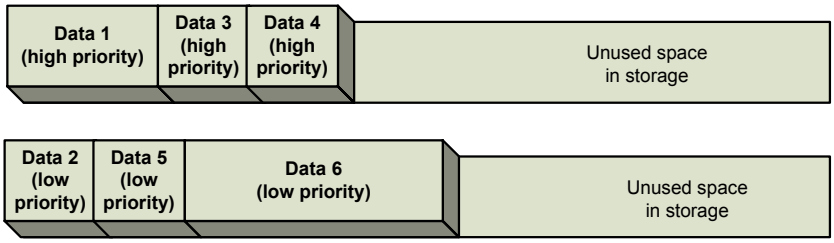

Figure 5-11 Multiple memory blocks to separate priorities

Linked list(s)

A linked list works in a different way. When a message needs to be stored, an object is allocated for that type of message. The message data is put into the object and the address of the object is added to the list. This method supports all message types.

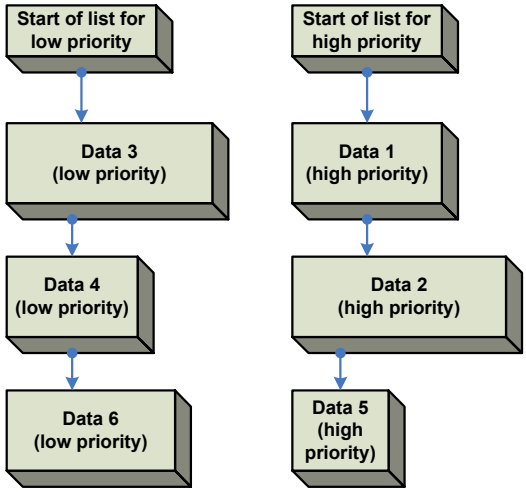

Figure 5-12 Linked lists

Because the object needs to be allocated dynamically, this storage mechanism is not suited for real-time logging. On the other hand the memory efficiency is good, because every object type only allocates the memory that is needed for that specific message type. Support for multiple priorities is good, both with a single linked list and with multiple lists. In case of a single list, the list is searched for lower priority objects, which can be removed from the list and de-allocated. As soon as enough memory is freed, the higher-priority object can be allocated and added to the list. Using multiple lists, one for every priority, the same mechanism applies, except that it is unnecessary to search the list for the occurrence of a lower-priority message object.

Linked list(s) with pre-allocated objects of fixed size

In order to make a linked list suitable for real-time, a pool can be maintained of pre-allocated message objects of a fixed size. If only one type of message object is used, this method has no advantages over the use of a single continuous memory block. However, when a number of different message objects are used (for instance one type for very short messages and one for all other messages), a better memory efficiency can be achieved, because these short messages can use a small memory block and therefore waste less space.

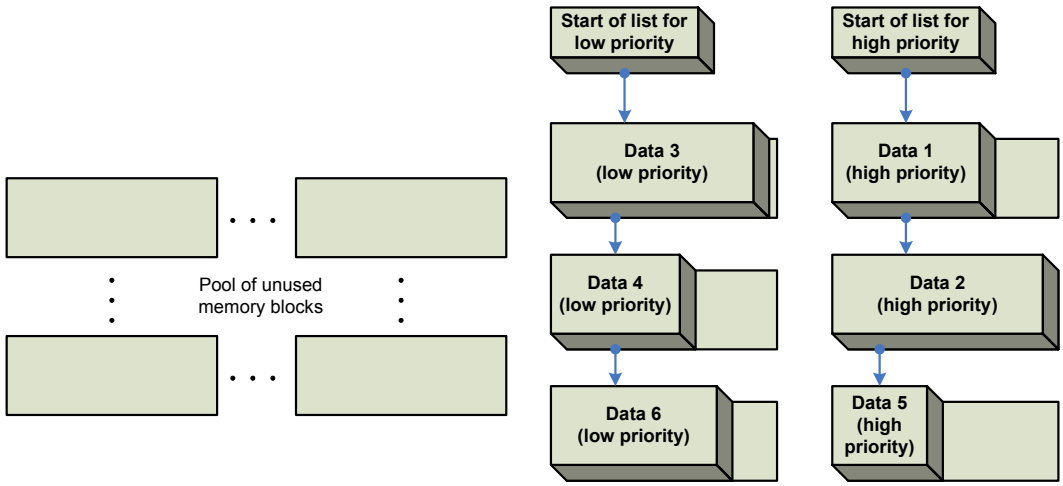

Figure 5-13 Linked list with pre-allocated objects

Linked list(s) with memory pool and objects of fixed or variable message size

Another way to make a linked list suitable for real-time is to pre-allocate and maintain a memory pool. Message objects are then dynamically allocated from this memory pool. The difference with 'normal' dynamic memory allocation that makes this type of dynamic allocation suited for real-time, is that the memory pool is completely managed by the application itself. Therefore a bounded-time allocation can be guarantied (but only if the allocation is implemented correctly). While allocating memory for a high-priority object, lower-priority objects could automatically be removed from the linked lists and de-allocated to provide space for the new object.

Because of fragmentation the memory efficiency is not optimal. Fragmentation occurs when large objects are deleted and smaller objects are allocated in that area. A gap is left which might be too small for yet another object. It can happen that enough total free space is available for a new object, but that this free space only consists of gaps that are too small for the object. These gaps can be rearranged to form one large contiguous free memory block by re-allocating (moving) other objects.

Overwrite array

An overwrite array is comparable to a memory block with variable-sized messages, except that every message that can occur has its own position in the block. This means that storage is guarantied for the allocated message types and denied for all other message types. Because of this guarantied allocation, priorities are not applicable.

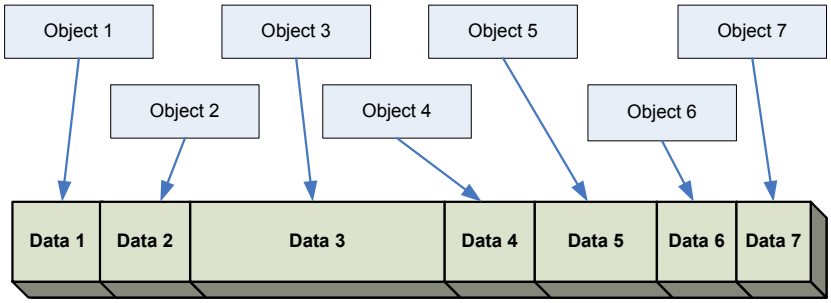

Figure 5-14 Overwrite array

The latest contents of the log messages are always available in the buffer array. This makes the overwrite array especially suited for ‘on-demand’ logging, i.e. logging where transmission of data only takes place after a request is issued to do so. This buffer type is also very suited for external observation, as the environment only needs to read (part of) the buffer. No additional information is needed at run-time for proper interpretation of the data in the buffer, as this information is already known at compile-time.

5.3.5 Transmission

The design space for the transmission component is very large and depends mainly on the used computer platform(s) and on the available hardware. Therefore only a number of general requirements can be given. These are valid for both non real-time (simulation) and real-time purposes.

Data integrity

Some provisions should be made to guarantee data integrity. This can be done by choosing a medium with built-in integrity checks, such as TCP/IP or CAN, or by adding checks, such as a CRC-value, boundary checks (for instance start and stop bits) or double transmission.

Mutual access to the storage

Because the observed component and the transmission component are separate, parallel running processes, both could access the storage at the same time. When one component has access to the storage, one must make sure that the storage is in a coherent state before allowing the next component access to it. This can be done by only allowing mutual access to the storage, using for instance mutexes, semaphores or guard processes.

Alternatively this can be done by dividing the storage in several blocks, and assigning each block to either the observing or the transmitting component. Initially, all blocks are assigned to the observing component, which can fill the blocks with data. As soon as a block is full, the observing component is disallowed all access to the block and the block is assigned to the transmission component. The transmission component can read from the block without interference with the observing component. When the block is empty it can be reassigned to the observing component.

Transmission medium

The transmission medium can be virtually anything: from a shared memory segment or a disk file to a RS-232 or TCP/IP stream. There are currently two limitations when choosing a medium. Both limitations are only applicable to real-time logging.

- First, the transmission may not cause (excessive) waiting for the real-time processes. In the current implementation of the CT library a system call from one CT-process will block all CT-processes in the same CT process tree (i.e. one system-level process). This implies that system calls which cause unbounded waits (for example a call to the ‘send’-function of a TCP/IP-socket) cannot be used.

- Second, the transmission medium must provide enough (average) bandwidth to cope with the expected data generation rate. When logging continuously, the average bandwidth must be higher than the bandwidth required for the data. In this case the storage component acts as a buffer when the instantaneous bandwidth deviates from the average bandwidth.

When logging for a limited time, the bandwidth of the medium can be less than the data generation rate. The storage will fill up at a rate which is the difference between generation and transmission rate. The transmission rate only needs to be high enough to prevent the buffer from overflowing. For low generation rates, short logging times and/or a large storage this could even mean that the storage can contain all logging data at once. In that case no transmission of logged data is needed during execution of the application, and the log buffer can be transmitted or locally processed afterwards.

5.3.6 Post-processing

The post-processing component can be even more diverse than the other components. There is only one real requirement, which is only applicable to the real-time logging situation: when the post-processing is performed on the same hardware component as where the observed entity runs, one must ensure that the post-processing does not influence the observed entity. This can be done by minimizing the CPU-usage of the post-processing or by assigning a higher priority to the activities within the observed entity.

Post-processing will typically consist of storing the data received from the observed entity, filtering of the data (see section 5.3.3) and performing some sort of analysis on the data. Storing the data first makes it possible to undo or modify the filters. A wide variety of data analysis methods can be thought of. One option is to let the post-processing software provide several different views to the user (views for e.g. scheduling, memory management, variable tracking etc). The second stage filtering can then be used to show only the log data which is relevant for a particular view.

A very convenient way of filtering is to give the user the possibility to select or deselect several logging sources from within the postprocessing environment. The data filtering within the observer can then be updated with these new selections dynamically. This allows the user to change the logging settings during the run-time of the observed entity, as opposed to a static configuration at compile-time.

5.4 Implementation decisions

In this project two cases were implemented where logging is used (see chapter 6). The logging in the first case, the PC-based simulation of a distributed control loop for linix, was written from scratch. This implementation is however not suitable for real-time logging and could therefore not be used for the second case, the distributed control loop for linix using ADSP boards. The logging in this second case is based on the logger component used before in the Mechatronics project. Both implementations have in common that they only provide the ability to log variable values, although the implementation in the first case can be easily extended with additional message types. The properties mentioned in this section are summarized in Table 5-2 and are explained below.

	1: Distributed simulation (PC-based)	2: Distributed control (ADSP-based)
Real-time	No	Yes
Filtering	In observer, based on the global debugging mode and objects	In observer, based on objects and decimation
Identification	By address and name	Using an array index
Triggering	Event-triggered	Time-triggered
Initiation	Autonomous	Autonomous
Possible data categories	All can be supported	Variable values only
Storage type	One memory block, variable message size	One memory block, fixed message size
Medium	TCP/IP stream	Unidirectional RS-232
Data integrity	Provided by TCP layer	Parity check, start-of-message identifier
Mutual storage access	Only a single thread can access the storage	Two separate buffers, flags for ‘empty’ and ‘full’
Post-processing	Data saved as CSV file	Data saved as CSV file

Table 5-2 Logging implementations summary

5.4.1 Identifying objects

The logging implementation for simulation uses a combination of identification by *address* and by *name*. In all logging messages objects are referred to by their address, because of the simple implementation, and in order to keep the message small. Additionally a name can be assigned to an object, which is more comprehensible to the user than an address. This assignment generates an event, which combines the objects address and its name. This information can later be used by the post-processing to represent the object by its name.

In the real-time case the variable values are all transmitted combined in a single message. The variables each have their own fixed position within the message, so they can be identified by their *index* in the message. The position in the message is determined by the order in which the variables are passed to the log function and is therefore determined at compile time. In order to 'decode' the contents of the log messages, the post-processing component must have knowledge of the variables that are enabled for logging, and in which order they are placed in the messages.

5.4.2 Data generation

Triggering

The logging implementation for simulation uses *event-triggered* logging, because it fits naturally in the concept of CSP programs. In those programs, processes communicate via channels on a rendezvous base. These rendezvous channel transactions can be seen as an event, and the occurrence of such an event can trigger a logging action. A change in the value of a variable can also be seen as event. In this approach all value changes are observed while no redundant data is sent.

The real-time implementation is based on the assumption that a controller model is executed with discrete time steps, and that the variable values are only of interest between two steps. Therefore after each execution step a log message can be generated containing the current value of the variables. This makes the update of the control loop variable values periodic and as a result the logging of those variables is effectively *time-triggered*. Note that in this particular situation time-triggering is equivalent to event-triggering, by defining an event 'controller calculation step finished' and only triggering on this event.

Initiation

In both implementations the logging is initiated autonomously. This causes the observed program to log all data, independent of its environment. If the communication protocol does not require any handshaking, as is the case in the real-time ADSP case, the controller can function normally even if the post-processing component is unavailable (e.g. when the serial cable is disconnected).

Categories of generated data

The logging implementation for simulation could be extended to generate all categories of data as listed in section 5.3.2. However, in the example case only four types of messages are implemented: for creation of a variable, for assigning a name to a variable, for updating the value of the variable and for deletion of a variable. The real-time logging implementation only provides support for variable values to be logged. However, some of the data mentioned in section 5.3.2, such as the stack usage, could also be represented as a variable value, thus also allowing logging for those categories.

5.4.3 Filtering

In both cases, filtering is performed during *data generation* in the observing component. The post-processing component does not perform additional filtering; it simply stores every received message.
In the simulation case, logging can be enabled or disabled *globally* (using a global 'enable debugging mode' parameter) and for *individual objects*. Messages will only be generated when both global and individual logging is enabled. The individual logging-enable flag can be changed during run-time, the global flag only at compile-time. In the real-time case, one has to specify the *individual objects* to be logged at compile-time. To decrease the bandwidth and/or storage requirements *decimation* filtering is also implemented.

5.4.4 Storage

The logging implementation for simulation uses one memory block as its storage. The memory is allocated in advance with a size large enough to accommodate the largest message. Only one message need to be stored at a time, because it will be transmitted immediately, before the simulation is resumed. The storage capacity could be increased for a possible higher communication efficiency (e.g. by bursting a number of messages at once), but for sake of a simple implementation only a single-message capacity was chosen.

The ADSP-based logging implementation uses the storage principle of a *single memory block with fixed-length messages*. Two such blocks are allocated in advance, each providing storage space for many messages. The capacity only depends on the amount of available memory. The observer starts to fill one of the blocks with messages. Once no more messages can fit in the block, it is marked 'full' and handed to the transmission component. The observer then starts filling the other block, provided it is marked 'empty'. Because all messages are of the same type, there is no notion of priorities. As long as both buffers are full, the observer will simply discard any newly generated messages.

5.4.5 Transmission

The logging implementation for simulation uses *TCP/IP sockets* as its transmission medium. These sockets guarantee data order and integrity. They are available on most computer platforms, including Windows, Linux and DOS, but unfortunately not on the ADSP platform. The sockets are very flexible and platform independent, allowing the sending and receiving side to be two processes on the same computer or on different computers. In the latter case the operating systems can be mixed, so heterogeneous simulation setups can be created. Mutual access to the storage buffer is guaranteed in the simulation case, because the observing and transmission component are executed from the same thread. Therefore concurrent access to the buffer cannot occur.

A disadvantage of the TCP/IP sockets is, that a read or write command can block for an unbounded period. This, in combination with user-level scheduling (as it exists in the CT libraries), makes the sockets unsuitable for real-time logging, if no additional measures are taken. These measures could consist of moving all socket communications into a separate system-level thread, allowing the main program to continue when the socket blocks. Of course, in this case care should be taken to prevent simultaneous access to the storage.

The real-time ADSP logging implementation uses a *unidirectional serial link* as a transmission medium. Data integrity is verified using a *parity bit* and by looking for a *start-of-message identifier*. In practice however it turned out that these measures are not sufficient and data corruption still occurs. Therefore the generated files need to be inspected by hand to remove invalid entries. This could be solved by transmitting redundant data, for instance a CRC.

As already mentioned, mutual access to the storage is implemented by passing memory blocks between the observing and transmitting component. Initially both memory blocks are marked 'empty'. The observer can only start to fill a block if it is marked 'empty'. Once started, the observer removes the 'empty' mark. When the buffer is full, the observer sets the 'full' mark. The transmission component can start transmitting the contents of the buffer, thereby removing the 'full' mark. When all messages are sent, the transmission component sets the 'empty' mark again, allowing the observer access to the buffer again.

5.4.6 Post-processing

A complete graphical user interface with data processing functions would be too much out of the scope of this project. For the real-time logging case, an already available program (called finalloggerparity) could be improved and re-used. This program listens for log data on a serial port and generates a comma-separated values (CSV) file. All logged variables are written to the same file, they can be distinguished by their column position in the file.

The post-processing program (called dsetool) in the simulation case provides more or less the same functionality. Instead of listening on a serial port, the program listens on a TCP/IP socket. Variable values are updated individually (as opposed to the real-time logging case, where the variables are all put in the same log message), so in the implementation of the dsetool it was more convenient to give each variable its own log file. By default, these files are given a name corresponding to the memory address of the variable (because that is how the variables are identified, see section 5.4.2). When a name is assigned to a variable, a message containing this assignment information is generated. As soon as the post-processing component receives this message, it also knows the name of the variable. The file will then be renamed, so it can be identified by the name of the variable.

In both the simulation and the real-time logging case, the resulting CSV-files can be imported in numerous existing computer programs, such as Microsoft Excel, Corel Quattro or 20-sim. In the latter software, the values of the logged variables can be used in new experiments, as input for a model or to compare the results of a 20-sim simulation with those of a CT-simulation or the run of a physical test setup.

5.5 Conclusions and recommendations

The current logging implementations, both for the simulation environment and for the ADSP board, provide simple logging support. However, neither of the implementations is satisfactory. The logging implementation for the simulation environment provides reliable logging of variable values and it can easily be extended to log other types of information. It is however not suited for real-time systems, as it does not provide local storage for the log events (buffering) and transmission of log data will block the whole CT program. The logging implementation for the ADSP hardware on the other hand is suited for real-time systems, but it is not extendable to also log other types of information and its communication protocol is very error-prone.

In this chapter a logging model is presented that is based on four logical blocks: the data generation, the local storage, the transmission and the post-processing. A logging implementation based on this concept can be made universal, that is: suitable for both simulation and real-time logging purposes. The current ‘simulation’ logging implementation can be used as the starting point for such a universal logging implementation.

An important thing that needs to be changed in the current implementation is the local storage component. The best alternative depends on the available memory size.

- For small memory sizes, like on the ADSP, a storage consisting of a continuous memory block will be a good choice. This storage type has a good memory efficiency, which can even be improved by using variable-sized messages (however this will decrease the support for multiple message priorities).

- For large memory sizes this solution has a disadvantage, namely that a linear search through the buffer requires quite some processor time, depending on the actual buffer size. Therefore in such situations a solution with multiple memory blocks (one for each message priority) will give better results.

Another topic that needs attention is the transmission. The current implementation uses system calls to perform the actual data transfer (for instance a *write* call on a socket file descriptor). These system calls however cause the complete CT program tree to be blocked for the duration of the system call, which can take seconds or even minutes. This is unacceptable for almost any real-time system, so a solution is necessary. Depending on the platform it is possible to use a separate system-level process or thread for all system calls, allowing the main CT process to continue. An alternative for other platforms might be to access the communication hardware directly, without using blocking system calls, as it is done on the ADSP platform. This is however not possible for all platforms, as some

operating systems prevent direct access to the hardware. In either case the transmission needs to be initiated from a separate CT Process, as was already mentioned in section 2.4 of chapter 2.

If the current ADSP logging implementation is still to be used, it is highly recommended to improve the reliability of the communication. The communication link itself, via a serial cable, is proven in practice to generate a large amount of communication errors, although RS-232 is known to be a quite reliable communication standard. The reason for the high error rate is unknown, but it could be caused by for instance slight baudrate mismatches, ground loops or electromagnetic interference. The used error detection measures already filter out a large part of the corrupted log data, but some errors remain undetected and show up in the resulting CSV file. There they need to be deleted by hand afterwards. The result of all data corruption is that a lot of log data is missing from the CSV files. The best recommendation is to find and resolve the cause of the data corruption, but if this is not possible, a better error detection (and possibly correction) mechanism needs to be implemented, in order to generate reliable log results.

This project has not focused on the post-processing of the data. An ultimate solution would be to integrate the log data will all other development tools available. For instance one should be able to combine a 20-sim simulation with a run of some processes on a hardware target all in real-time, by using real-time logging on the target and the real-time simulation option of 20-sim.

The real-time integrated logging concept could also be combined with methods to change model parameters on the target in real-time, for instance the method used by (Buit, 2005). The same methods could for example also be used to change the log filtering settings in the observer in real-time.

6 Demonstration setups

6.1 Introduction

Throughout the development of the simulation setup its functionality has been tested using a study case. This case is based on an already available setup using the *Linix* plant. In a previous project (Ferdinando, 2004) this plant was used in a setup consisting of two ADSP boards, the main focus of that project being fault tolerance based on redundant communication links between the boards.

The same setup is used again in this project, but now with only one communication link in between. The two boards form two processor nodes, both connected to the same network. Together with the plant, this distributed setup provides a test case in which most aspects of the simulation environment can be tested. As mentioned in chapter 5, both the *setup with the ADSP board*s and the *setup in the DSE simulation environment* incorporate support for logging.

Finally, a third, smaller, test case was created that can be used to measure the performance of various Remote Link Driver protocols and its parameters. A real setup with network hardware is not needed for this case, as it is designed as a simulation, using the DSE simulation environment with its network simulator. The results of this test case are discussed in section 6.6.4.

6.2 The plant: Linix

The plant used in the demonstration setup looks rather simplistic compared to all those multi-degrees-of-freedom robot constructions, as it consists only of one motor and some inertia. This simplicity however helps keeping focus on the project goals, as it does not require complex control law design.

Figure 6-1 The Linix plant

Figure 6-1 shows a picture of the plant. It consists of two axes, coupled by a set of rubber belts running over belt discs. Both axes are equipped with a rotation encoder, which can be used to measure the position and speed of both axes. A DC motor, the dark green, cylindrical shape in the picture, is connected to the right axis. The motor, together with the belt disc, adds to the total inertia of the right axis. The inertia of the left axis is comprised of the axis itself, the belt disc and a flywheel for extra inertia. On the photo this copper-colored flywheel is located behind the largest belt disc.

As mentioned before, a 20-sim model of this plant was already available. It is shown in Figure 6-2. The triangular element on the left represents the power amplifier for the motor (labeled ME in the model). In the demonstration setup a digital H-bridge amplifier is used, converting a PWM signal from the controller into a pulsating current through the motor. The average of this current is assumed to be proportional to the duty cycle of the PWM signal.

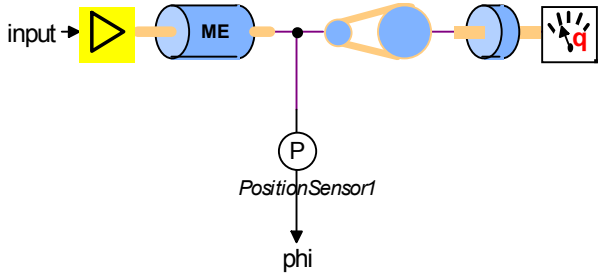

Figure 6-2 20-sim model of the plant

The figure shows that only the position sensor on the motor axis is used. The position of the load axis is not measured and is not used by the controller. Instead, the controller only controls the motor axis, and must be robust enough to accept the disturbance from the load axis. Slip of the belts would give this disturbance a different force profile. This slip is not included in the plant model, so for best comparison the motor current must be limited to prevent the belts in the real plant from slipping.

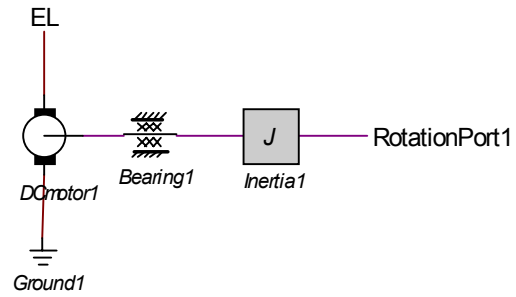

Figure 6-3 Submodel of the motor

Most of the blocks shown in Figure 6-2 contain a submodel themselves. The submodel of the motor for instance is shown in Figure 6-3. Besides the motor itself, also a bearing and an inertia element is present. This is also true for the submodel of the flywheel, as both axes run in ball bearings.

The model of the plant is converted into a software executable model, by using the *code generation* feature of 20-sim. This executable model is used in the simulation setup to simulate the behavior of the plant.

6.3 The controller

As the plant is not very complex and this project does not focus on control law design, the controller is a standard PID controller. Its model is shown in Figure 6-4. The abbreviation EIU at the input port at bottom right of the model stands for *Encoder Interface Unit*, which is the name Analog Devices gave to the circuitry used for reading the position of the axis. As the EIU port is connected to the position sensor on the motor axis, the controller always tries to drive the motor in such a way, that the position of the motor axis corresponds to the value of the setpoint input.

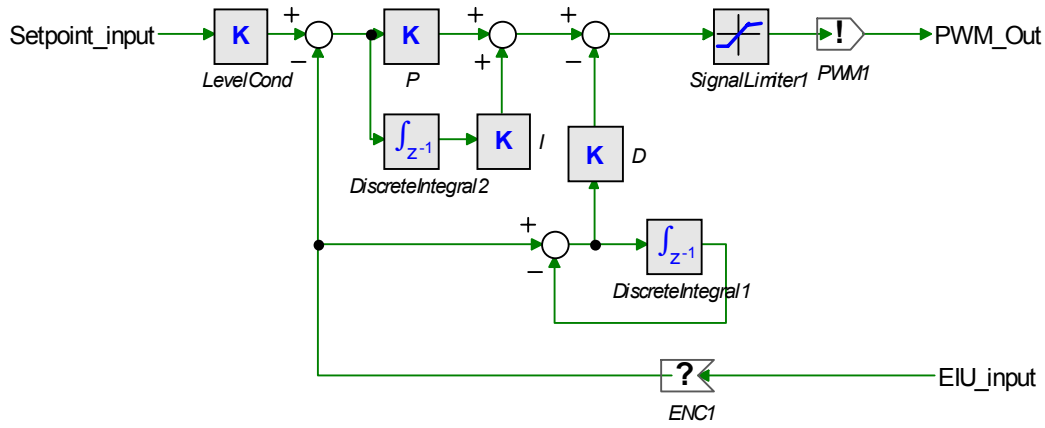

Figure 6-4 Model of the PID controller

6.4 The control loop

The plant and the controller are combined in a control loop, as shown in Figure 6-5. This model is first used to choose suitable values for the PID controller parameters. Furthermore this model is the reference for other measurements, as it depicts the 'ideal' control system situation, in which there is no network or disturbance.

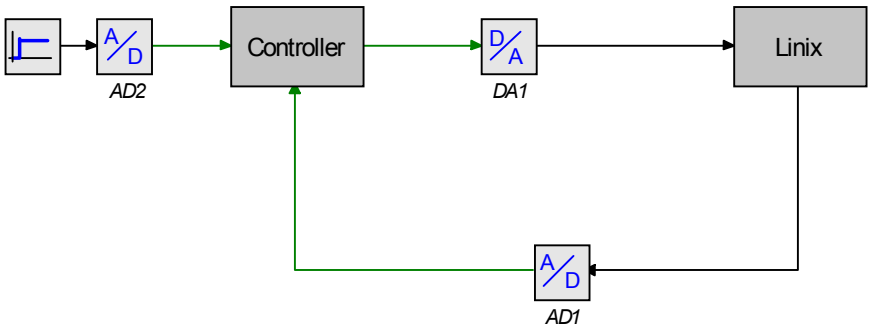

Figure 6-5 The control loop

The code generation feature of 20-sim is used to create an executable model of the controller. This generated model is used both in the real setup with the ADSP boards and in the simulation environment setup. The model is however not yet distributed.

The complete controller is located on the first processor node in the simulation and on the first ADSP board in the real setup. It is combined with a very simple *setpoint generator*, which generates a square wave with a large period, about six seconds. Now a single ADSP board can control the plant, as the first node reads the position sensor, generates a setpoint and drives the motor with its PWM output. This single-node control loop is shown in Figure 6-6, when node 1 is connected to the motor driver circuit (the arrow in the center of the schematic).

In a distributed controller application, one would normally split up the calculations of the controller, to spread the processing load over multiple nodes. However, this is quite some work and very dependent on the controller implementation. Therefore in this setup the complete executable model of the controller is left intact, and is left located on the first node. The output signal of the controller is however not only sent to the motor, but also to the second node, using a CAN fieldbus connection. The only task of the second node is to receive this information and actuate the motor according to it. This is the outer control loop shown in Figure 6-6, when the second node is connected to the motor instead of the first node.

This distributed setup is also built in the simulation environment. The process hierarchy is shown in Figure 6-7. When compared to Figure 2-4 in chapter 2, it is clear that this is a simplification of the

generic simulation hierarchy. As there is only one plant and one network simulator, these processes are not placed within their own parallel construct, but directly in the main *LinixSim* construct. Another difference is the lack of a *Logger* process in the tree. It is not present in this tree, because it is currently implemented as a passive object, not a process.

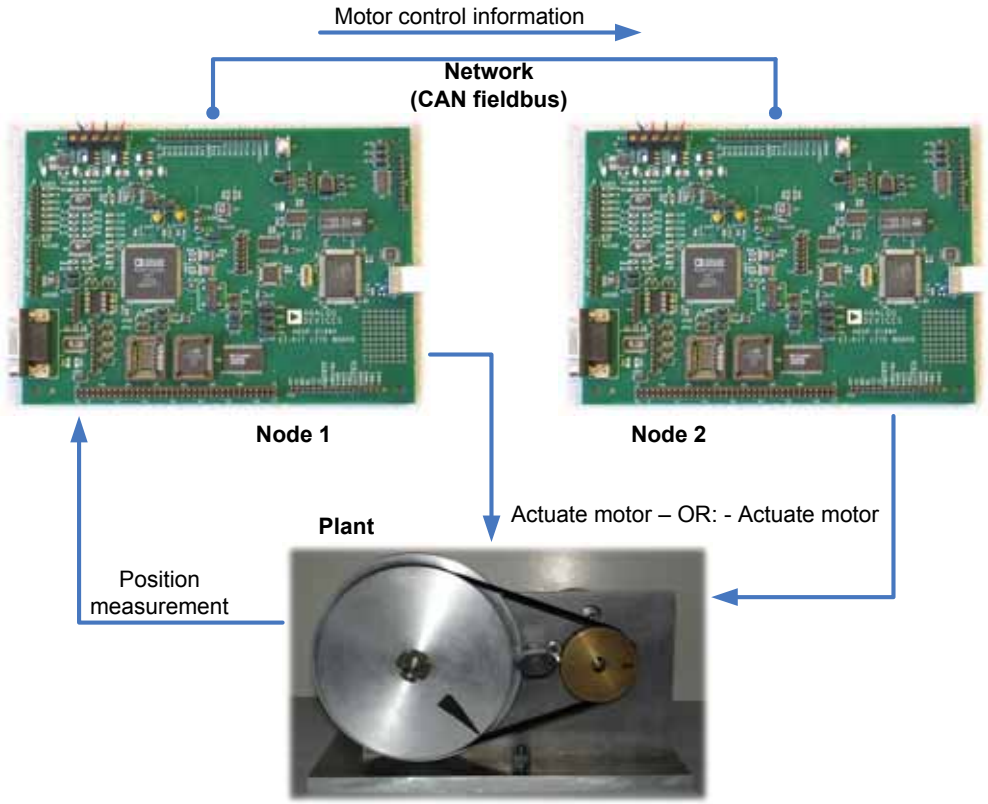

Figure 6-6 Single-node or distributed control loops

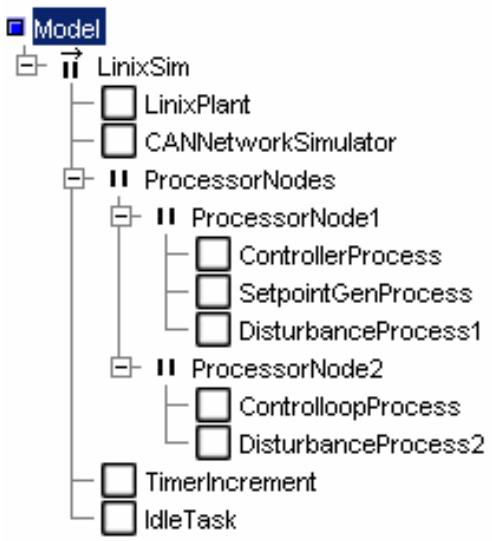

Figure 6-7 Linixsim hierarchy

In this model the control loop passes several processes. First the *ControllerProcess* reads the position of the axis from the *LinixPlant* and the desired setpoint from the *SetpointGenProcess*. It calculates the new motor steering value using the PID controller model. The resulting value is transmitted to the *ControlloopProcess* via a Remote Channel. This communication is transferred via the *CANNetworkSimulator* process. The *ControlloopProcess* receives the motor steering values and communicates these to the *LinixPlant*.

As mentioned in chapter 2, the continuous-time models (like *LinixPlant* in this case) will block on their Timer Channels after every iteration. Therefore the input and output from and to these models is done using shared memory and not via channels. If channels were used here, the model would block on the channels instead of on the Timer Channel.

Two processes are not mentioned yet: the *DisturbanceProcesses*. These are extra processes added to the simulation and they have nothing to do with the control loop itself. Instead, they can be used to generate extra traffic on the fieldbus, by just passing arbitrary information from one process to the other. For this communication again a Remote Channel is used. Therefore the disturbance processes have their own Remote Link Driver (see chapter 4), but as they use the same network interface as the control loop processes, they also share the Network Device Driver for that interface.

6.5 Logging

The design choices made for the logging in both cases are already described in section 5.4. In both cases only the first node is observed, as this node contains the controller. As mentioned in chapter 5, only variable values are logged. The interesting variables in these cases are the measured motor axis angle, the generated setpoint and the output value of the controller.

The values of these variables are stored in a *comma separated value* (CSV) file. Besides these values, the execution time of the model at the time of the event is also stored in the file. In the simulation case this is the simulation time, in the real-hardware case this is the real time measured from the start of the model execution. The CSV files can be imported in programs like Excel or Quattro for post-processing, but they can also read by 20-sim. In the latter case, the time values in the file are synchronized to the 20-sim simulation time. The values from the file can be used just like any other variable in a model. This feature can be used to create combined diagrams with both the results from the 20-sim simulation and the results from the observed setup.

6.6 Results

6.6.1 Comparison between 20-sim and the simulation environment

By performing a simulation in the simulation environment with only a single processor node, containing the controller model, and the continuous-time model of the plant, a comparison can be made between 20-sim and the DSE simulation.

As can be seen in Figure 6-8, the simulation environment yields exactly the same results as 20-sim. The figure shows the position of the motor axis, both from the 20-sim model (the crosses) and the simulation environment (the continuous line). The overlay picture is a magnification of the curves, as can be seen from the values on the vertical axis, at the location where the final position is almost reached.

The input for the model is the signal for the desired setpoint. In both cases this is a step signal, stepping from zero to one at time zero. This means the figure is actually the step response of the complete control loop. The parameters used in this example for plant and controller can be found in appendix B.

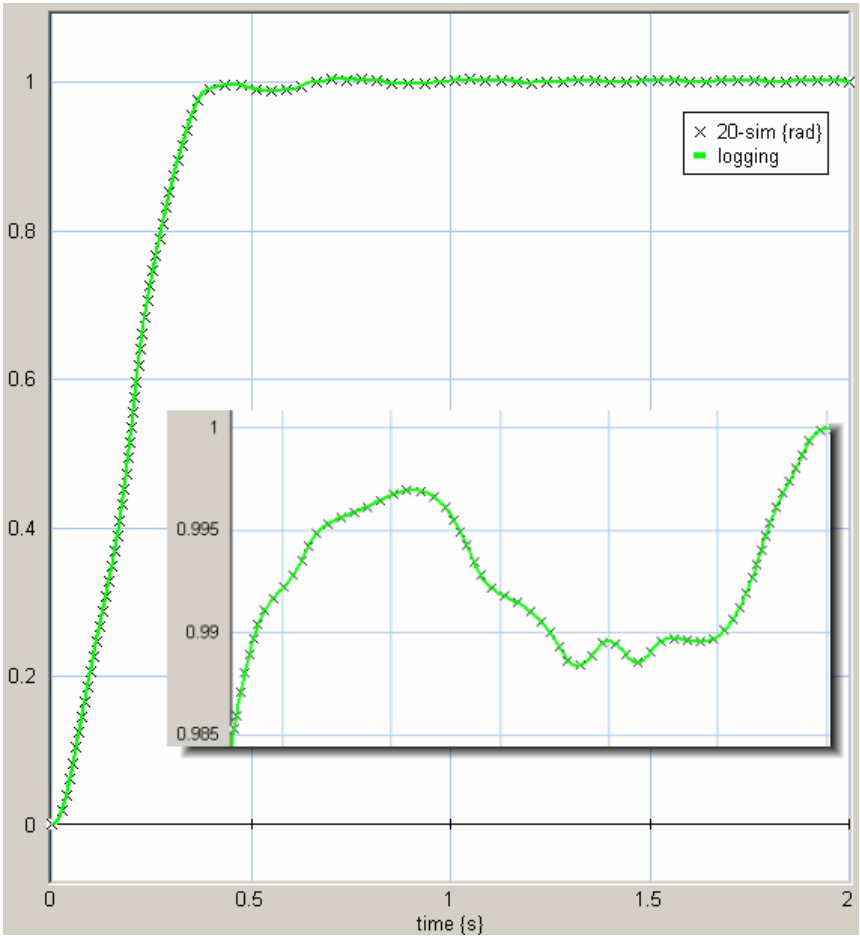

Figure 6-8 Position of motor axis in 20-sim (X) and simulation (line)

6.6.2 Validation of the 20-sim model of the plant

To be able to validate the results of the simulation, it is important to have an accurate model of the plant. Unfortunately, the available model of the Linix plant appeared to be very inaccurate. A possible cause for this is the age of Linix, which shows itself for instance in less elastic rubber belts, more slip between the belts and the belt discs and more friction in the bearings and motor.

Some effort has been made to iteratively improve the model parameters, and the results seemed very promising, as shown in Figure 6-9. The input for the model, the setpoint, is a square wave with a period and amplitude of 2 Pi. The two small signals are the motor steering values (one for the 20-sim simulation, one for the real plant), the two large signals represent the motor axis position (again one for the 20-sim simulation and one for the real plant).

As can be seen, these pairs match quite accurate, with one important difference: after about one second, the modeled plant has reached the desired position and the motor is not actuated anymore. The real plant however never reaches this final position exactly. Therefore the motor is continuously actuated, but because of stiction the plant does not move further towards its desired position. This stiction is not modeled in the 20-sim model.

The resulting model parameters however showed to be completely incorrect when different simulation circumstances were chosen, for instance when the PID parameters of the controller were changed. Therefore the remaining simulations were performed using the original model parameters.

Figure 6-9 Comparison between the adapted 20-sim model and the real setup (see text)

6.6.3 Variation of network parameters

One of the goals of the simulation environment was to be able to quickly investigate the influence of different network settings. To provide an example, some simulation runs were performed using the simulation setup shown in Figure 6-7. The overlay in Figure 6-10 is again a magnification of the part of the large figure at the location where the final position is almost reached (see the scales). A comparison was made between a simulation in 20-sim and multiple simulations using different settings of the network simulator and the disturbance processes.

As can be seen from the figure, the results from 20-sim, marked with crosses, are the same as the simulation with the network disabled (the line running straight through these crosses). For network speeds of 1 MBps and 100 kBps the control system behaves almost the same as without the network.

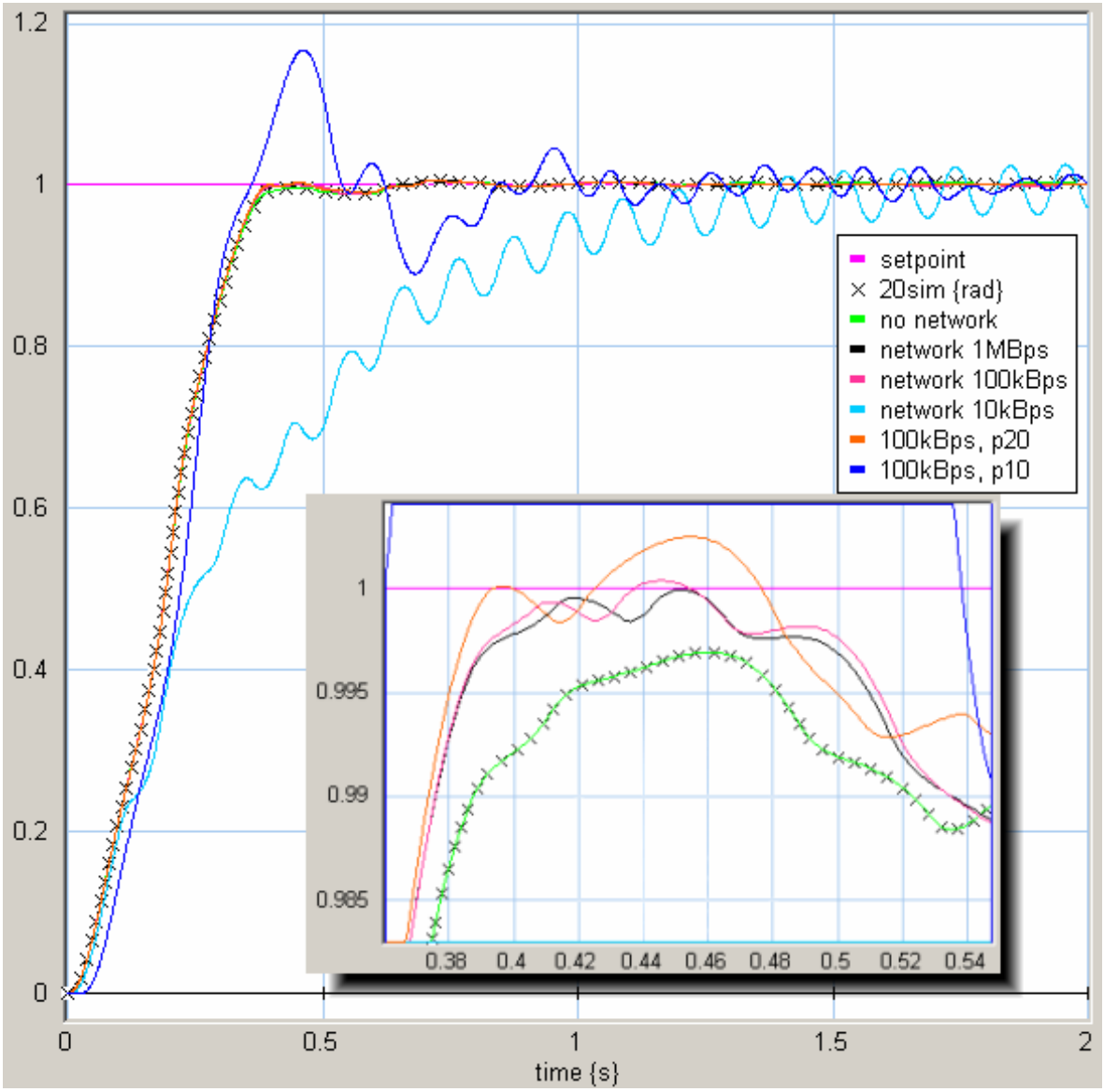

Figure 6-10 Variation of network parameters in simulation

The lines representing these two speeds are only recognizable in the overlay picture, they both touch the y=1 line at 0.54 seconds.

When the disturbance processes are enabled, they continuously send large objects over the network from one to the other. As long as these packets have a lower priority, their influence is still marginal. This situation is represented in the figure with the line that touches the y=1 line at 0.4 seconds. It shows a little overshoot, but considering the large magnification factor of the overlay picture, the effect is still negligible.

The situation becomes different when the disturbance network traffic has a higher priority than the control loop packets. The control system now has an overshoot of about 15 percent and inhibits some oscillations. These oscillations also occur when the network speed is too low, even when the disturbance traffic is disabled again. The line corresponding to this situation is the one that needs more than one second to reach the desired axis position and keeps oscillating.

A comparable variation of the network speeds was also performed on the real linix setup, see Figure 6-11. Due to program memory restrictions on the first node it was not possible to include the disturbance transmitter and receiver processes, so these measurements cannot be compared.

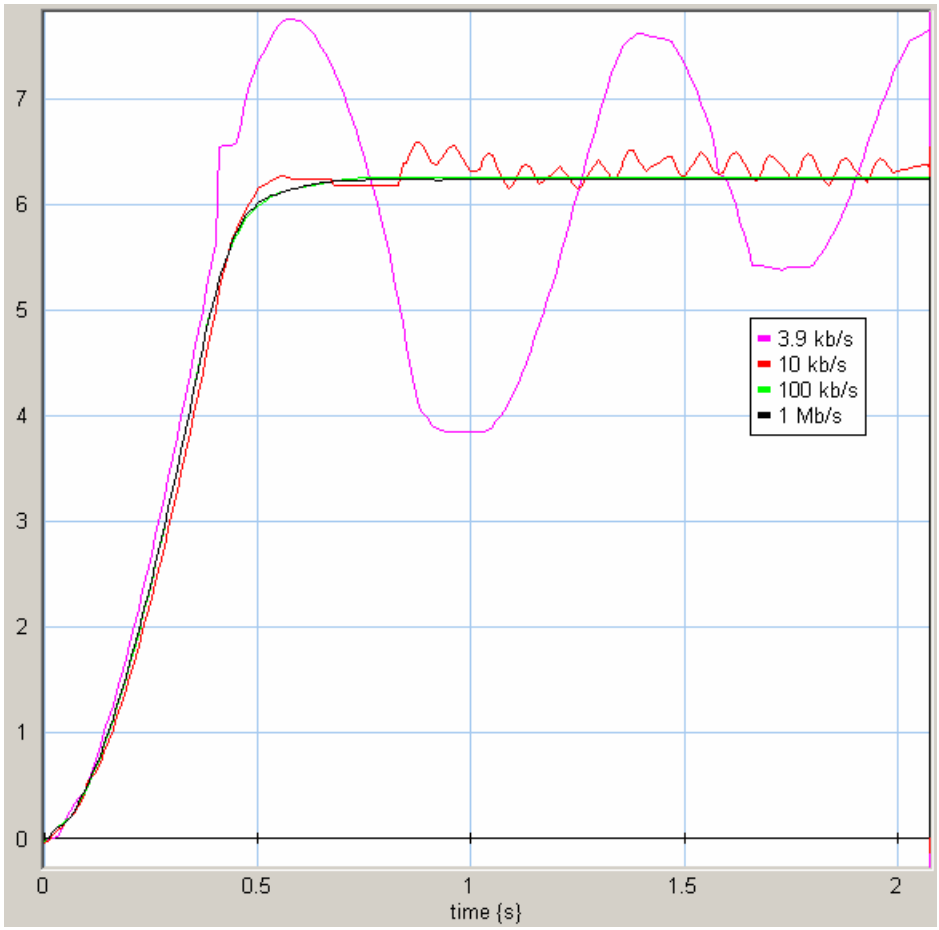

Figure 6-11 Influence of network speed on the real Linix setup

The results for highest two network speeds are similar to the results of the simulation, although due to the inaccuracies in the 20-sim model they cannot be compared directly. As can be seen in this figure, the real model does not suffer from oscillations or overshoot when the network is fast enough. It even does not seem to be influenced by the mass-spring load, formed by the belts and the axis with flywheel. The cause of this is probably the increased stiffness of the aging belts.

At lower network speeds the system exhibits some oscillations: the high-frequency oscillation occurred during the measurements at a 10 kb/s network speed, the low-frequency oscillation at the 3.9 kb/s setting. Although not the same, they were already predicted in the simulation runs (see Figure 6-10). These oscillations are not just mis-measurements of the controller, they do really occur, as they can be seen and heard. The lowest network speed setting might seem to be chosen somewhat oddly, but it is the lowest speed that can be achieved without changing the oscillator settings of the processor.

6.6.4 Comparison between both Remote Channel rendezvous protocols

It is hard to say in advance which of the implemented rendezvous protocols (see section 4.3) will deliver the best performance in a certain situation, and whether this performance is good enough or not for the intended purpose. The difficulty of prediction is mainly caused by the large number of parameters: data and packet sizes, the probability of packet loss, the network type and speed, the number of nodes on the network and their organization and so on.

The network simulator can be used to get an indication of the network performance that can be expected. In order to show how this can be done, an example was developed (see *NWsimMeasureRLD* in the CT example directory).

The example consists of a Network Simulator process, two processes depicting two nodes on the network and a process to increment the time (see Figure 6-12).

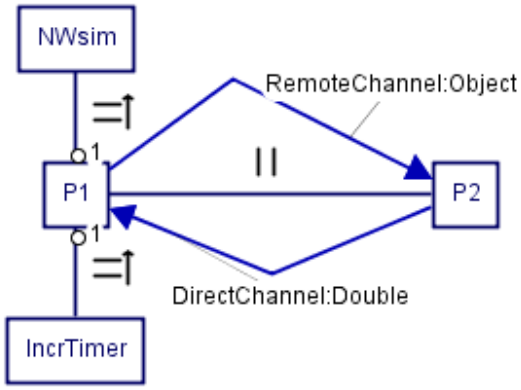

Figure 6-12 Model of the RLD measurement example

The principle of doing a measurement is very simple: process P1 first records the current time using a SimTimerChannel, then it starts the Remote Channel communication. Process P2 takes part in this communication and records the time when the communication is finished. It transmits this time to P1, using a normal rendezvous channel. P1 now knows both start time and end time, which is logged.

The simulation is a stochastical process (the packet loss in the network is a stochastical variable). To get meaningful results one can perform statistical functions on a large set of measurement results. For the next examples, the measurement was repeated 1000 times. Each measurement results in a certain communication duration. All durations are placed in a histogram.

Example one: large packet loss

In this example both rendezvous communication protocols are compared using the parameters in Table 6-1. These parameters are chosen more or less randomly.

object size:	2000 bytes
block size:	64 bytes (i.e. 32 blocks needed for one object)
max packet size:	100 bytes (larger than block size, so not a limiting factor)
network speed:	100 kbyte/sec
network type:	CSMA/AMP
pre-/postprocessing delay:	0 sec
chance of packet loss:	5%

Table 6-1 Settings used in the first example

The transmission timeout parameter is set different for the two protocols: when handshaking is based on blocks, the timeout is set somewhat larger than the time required to transmit one data block (in this example one millisecond). When the handshaking is based on complete objects, the timeout obviously needs to be chosen larger, based on the total expected transmission time (in this example 32 blocks of one millisecond give 32 milliseconds total).

The results are shown in Figure 6-13. From the figure it becomes clear that retransmission per block is the right choice for this example: more than seventy percent of the objects take between 25 and 30 milliseconds to be transferred, all objects are transferred in 40 ms or less. On the other hand, the protocol that retransmits per object performs much worse: less than twenty percent is transmitted in 50 ms or less, the rest takes much longer, one object even takes more than 180 ms.

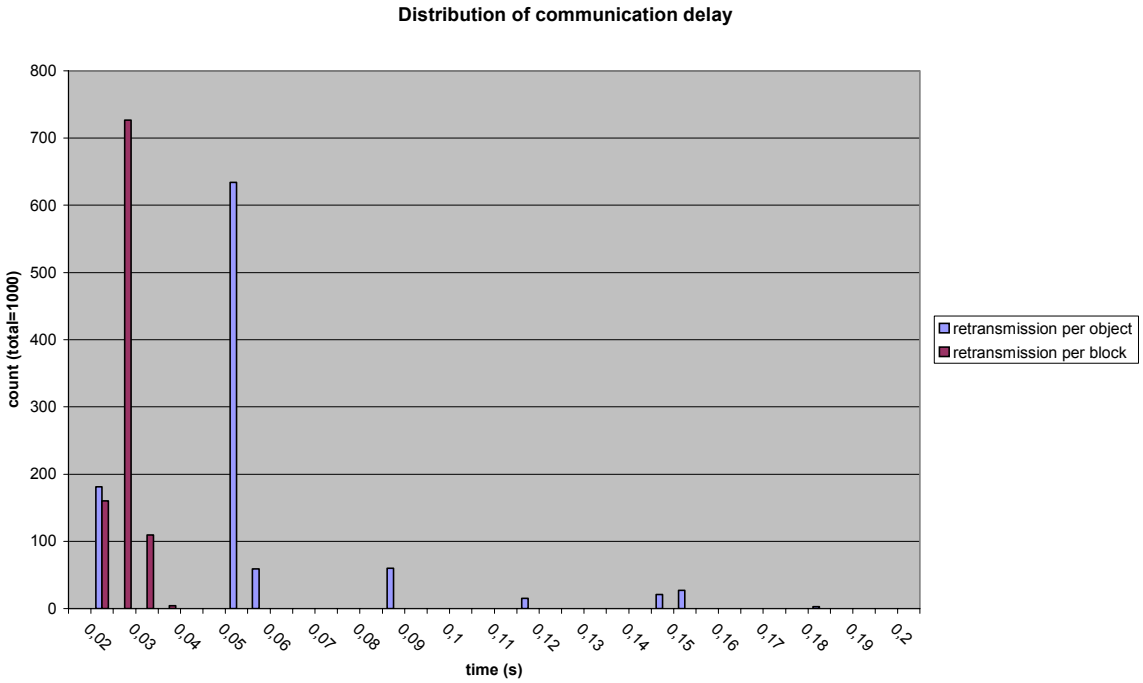

Figure 6-13 Position of motor axis in 20-sim (X) and simulation (line)

Example two: large number of small blocks

The second example uses some different parameters for packet loss chance and block size than the first, to show that handshaking per block is not always the best choice.

object size:	2000 bytes
block size:	8 bytes (i.e. 250 blocks needed for one object)
max packet size:	100 bytes (larger than block size, so not a limiting factor)
network speed:	100 kbyte/sec
network type:	CSMA/AMP
pre-/postprocessing delay:	0 sec
chance of packet loss:	0.001%

Table 6-2 Settings used in the second example

The timeouts in this situation for both protocols are chosen 0.2 milliseconds and 50 milliseconds respectively.

A low chance of packet loss, which is common on many network types, makes the time that is gained by retransmitting single blocks instead of complete objects negligible. On the other hand, the large number of blocks in this example situation causes a lot of extra traffic if every block needs to be acknowledged. As can be seen in Figure 6-14, handshaking per object is the best choice in this example situation.

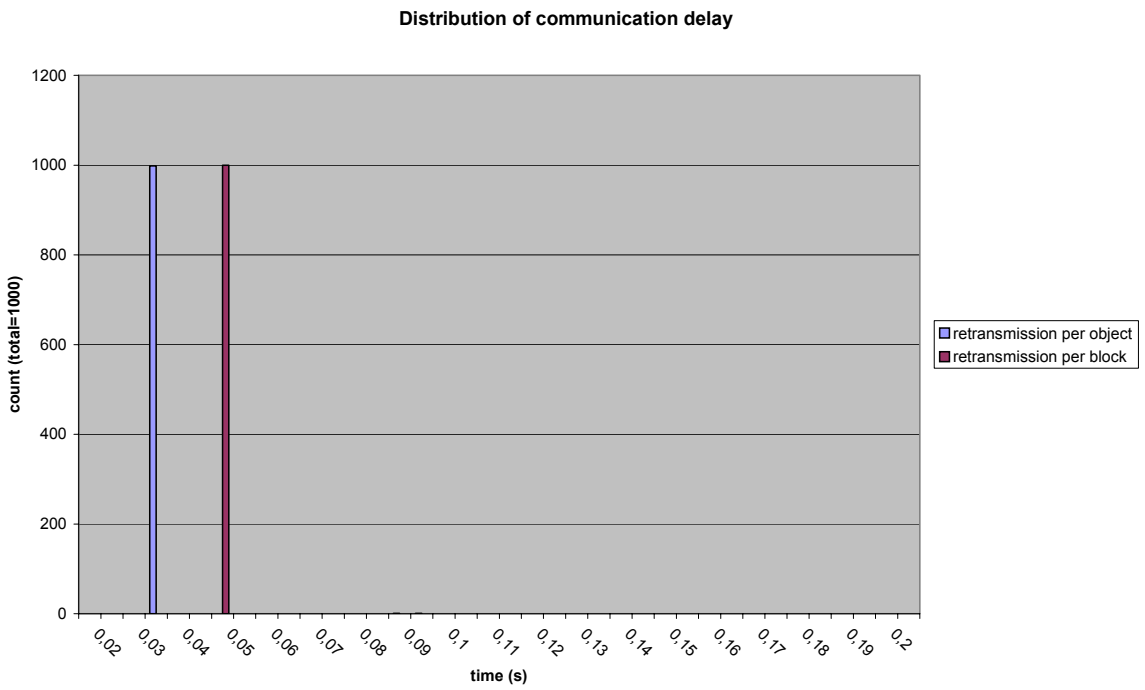

Figure 6-14 Delay distribution for example two

6.7 Conclusions and recommendations

The cases described in this chapter show that the simulation environment is very useful in an early phase of the design process. Simulations can be used to investigate the influence of network-related parameters on the behavior of the control loop.

For a correct simulation of the control system, an accurate (20-sim) model of the plant is of great advantage, but even with only a rough model the simulations can give valuable information about the functioning of the control system. The simulations can point out possible bottlenecks in the system. The example simulations for instance showed that networks with a low bitrate could cause oscillations of the plant at two different frequencies (around 1.5 Hz and 10 Hz). Although the waveforms themselves look quite different, the oscillations are indeed present for lower network speeds.

The simulation environment is also very usable to measure the behavior of network protocols under variable circumstances, as shown in the third example case. The simulation environment, with the network simulator, can therefore also be used to develop for instance new communication protocols, where simulation results can be compared to the expected (theoretical) performance.

The demonstration setup has been build using the ADSP boards, as this was the only platform on which the CT library could run *and* where both a real-time network interface and a hardware timer were available to CT. Implementing these facilities on the other platforms is recommended to make all platforms suitable for real-time distributed control. Especially useful would be a platform-independent timing implementation in the CT library. The structure of the SimTimer (see also chapter 2) could be used as a template for this, where the ‘software side’ of the timer forms a platform independent programmer API, and where the ‘hardware side’ provides an API for interfacing with the hardware.

During the implementation of the setup, it became clear that support for the ADSP platform is currently very unstable. Both the CT library and the 20-sim templates for ADSP contained a lot of bugs. Bugs that have been identified have of course been corrected, but it can be assumed that the software is still not bug free.

The example program for the first (controller) board does not fit in the program memory, unless some compiler optimization is enabled. When this option is simply enabled for all source files, the program memory usage is greatly reduced, but the resulting program just crashes. Selectively enabling or disabling the compiler optimization for certain source files results in a binary program that both fits in program memory and runs correct. This dependency on compiler optimization settings indicates that there are still bugs left in the sources. It is also recommended to keep the compiler version up to date, as (according to its website) the compiler itself also contains a lot of bugs and will under certain conditions even generate faulty code.

For development on the CT library it is recommended to have a test setup for the ADSP platform available at all times, to be able to verify the correct functioning of the library after changes have been made to the library. This way bugs can be signaled in an early stage, which makes it easier to identify and correct them. Of course the same holds for the other supported platforms.

As already mentioned the test case would not fit in program memory without some compiler optimization. The example software is not very complex; it only contains a single discrete 20-sim model and the CT library, including support for Remote Channels and the SimTimer. It can thus be expected that more complex programs will not fit anymore. A solution is needed, as larger programs are desirable. The easiest, but most expensive solution is to upgrade the ADSP hardware to a newer DSP with more program memory. A better solution would be to make a CT program run even with full compiler optimization enabled, as this saves quite some space in program memory. A third alternative is to minimize the footprint of the CT library by leaving out unnecessary library parts, an effort which is already started with the *ReducedCT* version of CT.

7 Conclusions and recommendations

7.1 Conclusions

A design trajectory commonly used in the development of an embedded control system is based on four phases. In the first two phases models are developed for the plant(s) and controllers and control laws are designed. The intermediate and final results of these phases can be verified with computer simulations using existing tools. In the final development phase, the design is realized on the real target processor(s) and plant(s). It is verified and validated by measurements on the real setup. If parts of the setup are not available yet, they can be replaced with hardware-in-the-loop simulation equivalents. The third development phase, in which the control laws and other functionality are implemented in software, did not provide any means to perform verification of the design, other than formal checking of the design.

The DSE (design space exploration) simulation environment extends the development toolchain with the possibility to perform simulations of the complete system, both plant(s) and the implemented embedded software, during the third phase of the development phase. It is particularly aimed at distributed control systems, in which the connecting fieldbus can have a large influence on the behavior of the control system. The DSE simulation environment makes it possible to signal these influences in an earlier stage of the development process, compared to not discovering problems until the final realization phase of the design process. Because the complete control system, including the plant, is simulated, the DSE simulation environment furthermore provides the ability to verify the correctness of the translation from a modeled control law to its software implementation.

The DSE simulation environment provides functionality to simulate fieldbuses and other networks. The responsible network simulation component is based on a modified version of the TrueTime network simulator. Networks are simulated on the Data Link layer level. The most commonly used Data Link layer types are already supported, and support for other types can be easily added. The network simulator is capable of simulating multiple independent networks concurrently, a feature which can be of use when adding redundant network connections to a design to improve its reliability.

The timing of the simulation is based on a platform-independent programmers interface (API) to the timer implementation. Because of this generic API, the use of the SimTimer is not limited to simulations only. It can also be used to provide timing for real-time software implementations, as shown in the demonstration case using the embedded ADSP controller boards.

Support for distributed control applications is extended with the developed Remote Channels. These channels can be used as a communication medium between processes that are distributed over multiple processors. The Remote Channels provide several communication protocols, which handle the synchronization of the processes, data transfer and addressing of the channels.

The functionality of the Remote Channels is split between a Remote Link Driver and a Network Device Driver. The Remote Link Driver is associated with an individual Remote Channel, while the Network Device Driver is associated with a network interface. The separation makes it possible to route multiple Remote Channels over any network interface, where each channel is independent and can transfer a different type of object. The Network Device Driver is divided in a hardware-dependent and a hardware-independent part. This way adding support for new hardware network interface types is very easy.

Observation of an embedded system is very important. A logging concept is proposed which can be universally used in both simulations and real-time systems. An overview of design alternatives is given for each of the components in the concept.

Three demonstration cases are presented, showing the usability of the DSE simulation environment. One of these cases is a real control system setup, based on the Linix plant and a distributed controller consisting of two ADSP processor boards. The values in the controller are logged to the PC, to provide comparison material for the simulations of the same control system.

These simulations are performed using the DSE simulation environment. The complete distributed control system is reconstructed as a simulation, using the same control law and an existing model of the plant. Due to an inaccurate model of the plant, the simulation results are not the same as the measurements from the real setup, but the same phenomena can be seen in both results. An accurate model of the plant(s) helps to get accurate results, but even a simple model with a lot of assumptions and neglected elements can give an indication of possible design problems.

A third, smaller case was designed to demonstrate another use of the DSE simulation environment. It implements statistical measurements on the data transfer times as function of the network parameters and the communication protocol used. It shows how the simulation environment can be used to verify correct behavior of protocol designs and how the performance of such a protocol can be measured.

7.2 Recommendations for further research

To create a complete DSE simulation setup, a number of things need to be done manually. A rather thorough knowledge is required about the inner workings of for instance code generated by 20-sim, the CT library, Remote Channels, the network simulator and logging. The usability of the DSE simulation environment can be improved by integrating it with the other tools used in the development process.

gCSP

A simulation setup is currently designed by creating the required components and adding them to a CT Construct by hand. The graphical design tool gCSP could be used to design the simulation graphically. It could aid in the creation of the processor nodes, the placement of the processes on the processor nodes and the addition of other necessities for a simulation, such as a network simulator, logger and SimTimer.

The gCSP tool could also be used to create the Remote Channels. Currently these are also created by hand and all endpoints need to be numbered and connected manually. This process can be simplified by drawing the Channel from a process on one processor node to a process on a different node, after which these processes are automatically connected via a Remote Channel. The node and link numbering can be automated, because all needed information about both endpoints is available.

20-sim

20-sim is used in two ways. First, it is the source of the models that can be used in the DSE simulation environment, for instance for plants and controllers. Second, it is used to read back the logged results from a simulation or real-time measurement run, in order to post-process the results, for instance by comparing them with its own simulation results.

The first part, the generation of the models, can be improved in two areas. Currently only discrete-time models and continuous-time models based on the Euler integration method are supported by the DSE simulation. Implementation of support for other integration methods in the DSE simulation environment would improve the connection between 20-sim and the DSE simulation.

The inputs and outputs of the 20-sim models are connected to the other parts of the simulation by hand, simply by looking for the input and output variables used in the generated model and writing and reading values to or from them. This method requires knowledge about the construction of the generated 20-sim code. Furthermore, it is error-prone and can break with any update of the 20-sim software. A generic interface to the inputs and outputs of the models is needed. This could be achieved by adding support to the Hardware Connector (Buit, 2005) for the DSE simulation environment.

The second part, the post-processing of the logging data, can be improved by providing a better connection between the logger and 20-sim. The post-processing component in the logging model could provide storage for the log information and an interface to 20-sim. Through this interface the log data can be exchanged as soon as it is available from the DSE simulation or the real setup being observed. The interface between the post-processing storage and 20-sim can for instance be implemented using 20-sim's ability to use DLL functions (Buit, 2005).

The logging implementation filters the log data during the data generation phase. It would be really useful to be able to influence this filter in real-time, during a run of a simulation or real-time system.

Extending functionality

The functionality of the DSE simulation environment can be extended by adding features or by improving existing parts of the simulation environment.

One of the most important assumptions is that the network speed must be slow compared to the execution time of the processes. This assumption is required, because the execution time is not taken into account by the simulation environment and can be circumvented by adding support for timed software execution, for instance by using the concept of the TrueTime kernel simulator. For this, very detailed knowledge about the execution time of the software on the target processor(s) is needed. To gather this information one could for instance do automated measurements of all blocks of code on the real target processor.

However, this is probably not an easy task; therefore it is questionable if it is worth the effort. Instead of a fully timed simulation one could measure only the delay of the most important block of code, for instance the calculation of a 20-sim model. Including this timing information might improve the accuracy of the simulation to a level that is high enough.

The logging implementation of the DSE simulation environment can be used as the starting point for a better implementation, based on the proposed logging model. It should then be usable both for DSE simulations and for observation of real-time systems on real targets. Research on a universal logging implementation is recently initiated in the context of the Boderc project (Visser *et al.*, 2004).

The logging implementation used on the ADSP demonstration setup is not flexible enough to be used as a generic logging solution. For simple observations on ADSP-based control systems it is usable, but its complex configuration and its very unreliable communication make it a rather labour-intensive logging solution.

Regarding the other components of the DSE simulation environment it might already be clear that extra features can be added to extend the usefulness of the simulation environment. The network simulator for instance can be extended to support for network types that use other Data Link layer types than the ones already supported. The network simulator can also be extended to provide extra configuration parameters. A maximum queue size for instance could simulate the behavior of communication devices with limited buffer memory. The pre- and postprocessing delays could be made node-specific, to simulate setups with heterogeneous hardware more accurate.

The Remote Channels can be extended with additional communication protocols, with support for input and output Guards or with support for communication hardware. Fault tolerance can be added to the Remote Channels by allowing them to be dynamically reassigned to different Network Device Drivers on the occurrence of a network failure.

Test cases

With respect to the CT support for multiple hardware platforms, it is recommended to continuously have a test setup available for each hardware platform. After changes are made to the CT library, the correct functioning of all test setups should be verified. This would prevent bugs from remaining undiscovered for long times, which would otherwise have made it tough to locate and correct them. Based on observations on the ADSP platform it seems that some bugs are still present in the CT

library or the 20-sim templates for this platform. A lot of bugs were already found and corrected, but software for this platform still exhibits unexpected behavior now and then, such as spontaneous resets or stalls.
The result is that the test setup for the ADSP boards could not be compiled with full optimization, resulting in large code size. With these settings, the ADSP boards can not accommodate programs much larger than the demonstration setup application. Although an effort like the *ReducedCT* library will solve the problem, it will also limit the functionality of the library.

Moving to a newer DSP platform will also solve the memory issues, as newer processors can usually accommodate larger programs. Furthermore, it allows the development software (like the compiler, linker and other tools) to be upgraded to a newer version. The current ADSP-21992 processor is not supported anymore by the latest development tools; therefore older tools, with a lot of known bugs, are still in use.

A Remote Channel details appendix

This appendix describes some details about Remote Channels. In the first three sections, the communication protocols that are implemented in the Remote Link Driver are discussed. Finally, the last section explains how to write a new driver for the Remote Channels, to extend the hardware support of the Remote Channels.

A.1 Unreliable, asynchronous communication

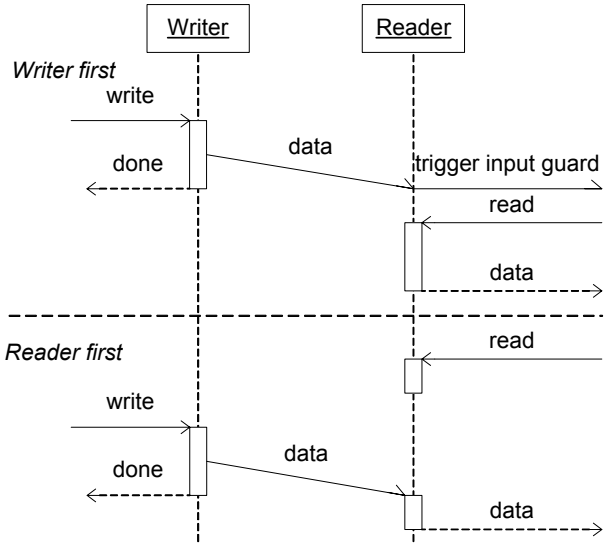

Figure A-1 Asynchronous protocol

The unreliable, asynchronous communication protocol does not provide any safeguarding against deadlock of the reader: consider the situation in Figure A-1, where the reader arrives first. The reader thread will block until the data arrives, but if that never happens the reader will block forever.

A.2 Rendezvous communication, object based

Figure A-2 shows the protocol in case the writer arrives first. The protocol is safe in case of packet loss:

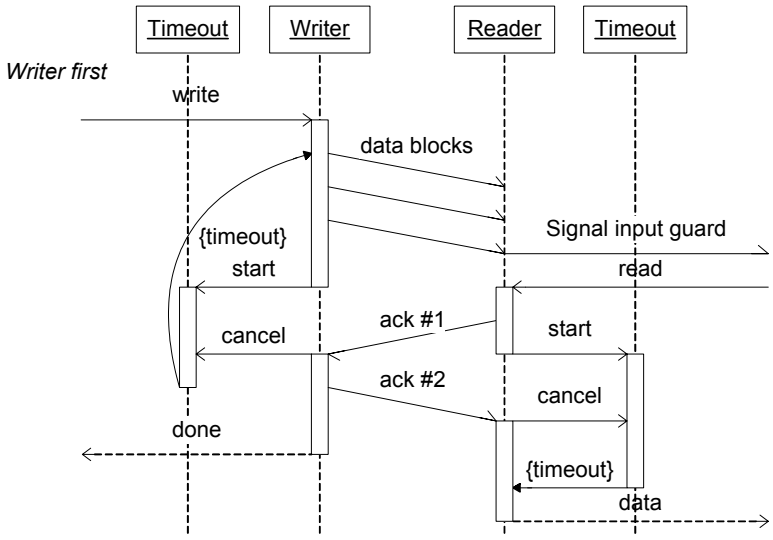

Figure A-2 Object based rendezvous protocol, writer comes first

- If one or more of the data blocks are lost: the reader stays blocked, waiting for all data to be complete. The writer will timeout while waiting for *ack #1*, after which it starts retransmitting all the data.
- If *ack #1* is lost: the writer will timeout and restart transmission of the data. The reader was blocked while waiting for *ack #2*, but as the data starts arriving again it will cancel the timeout and restart the protocol from the start (except for signaling the input guard, which is clearly not needed anymore). To make this work correctly, the timeout period of the reader must be configured to be higher than that of the writer.
- If both *ack #1* and the following retransmits of the data are lost: the writer will start retransmitting the data after the loss off *ack #1*. If this data does not arrive at the reader at all, the timeout of the reader will eventually expire. The reader will now return with success, as it correctly received all data and a rendezvous was made. The writer however will keep retransmitting data packets, without receiving an answer. When the retry limit is reached the writer will give up, indicating that there is something wrong with the network connection. This is a weak point in the implementation of this protocol: even if the network integrity is restored after the packet loss, the protocol cannot finish cleanly from the moment the reader has left. The only way to solve this is to choose a completely different threading model: if the reader's RLD has its own thread of control, it can respond to the superfluous data retransmits with some '*reader already left on success*' message.
- If *ack #2* is lost: both nodes already know that the rendezvous was successful and that the data was received correctly, so the reader may just return the data when the timeout expires. By stating this it might seem that *ack #2* is superfluous. It however has a purpose in releasing the reader immediately, instead of having to wait for the timeout to expire. Since the timeout value will always be longer than the time needed to transfer an acknowledgement packet, this will give a performance improvement.

Note that data transfer is performed in units of data blocks, the 'native' unit of the Remote Link Driver. Depending on the maximum network packet size, the Network Device Driver may decide to transmit those blocks using one or more network packets. In neither case additional handshaking is performed by the NDD. Acknowledgments are only exchanged on the RLD level after the whole object is received.

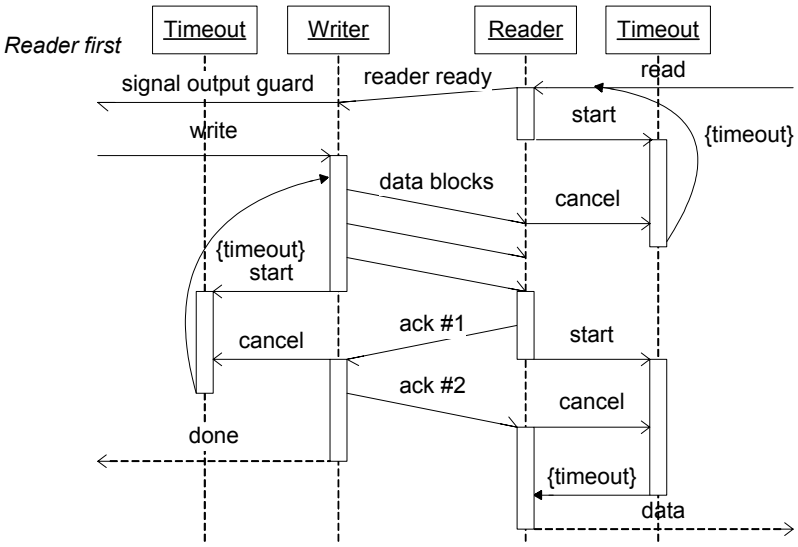

Figure A-3 Object based rendezvous protocol, reader comes first

The protocol is almost the same in case the reader arrives first, only some extra measures are added to support output guards (see Figure A-3). The reader will send a *reader ready* packet, indicating that the output guard may be signaled. If the reader receives the first data packet, it knows for sure that it is not necessary to send the *reader ready* again. In case no data is received when the timeout expires, the *reader ready* could have been lost and is therefore retransmitted. If the *reader ready* is properly

received at the writer's side, but the writer is not yet ready for rendezvous, it cannot answer the reader ready by starting the transmission. In this case the *reader ready* will also be retransmitted. As the *reader ready* is a very small packet, the actual data fields are empty, this will not pose any problems. If reader ready packets are retransmitted too often in a certain application, the timeout value can be adjusted.

A.3 Rendezvous communication, block based

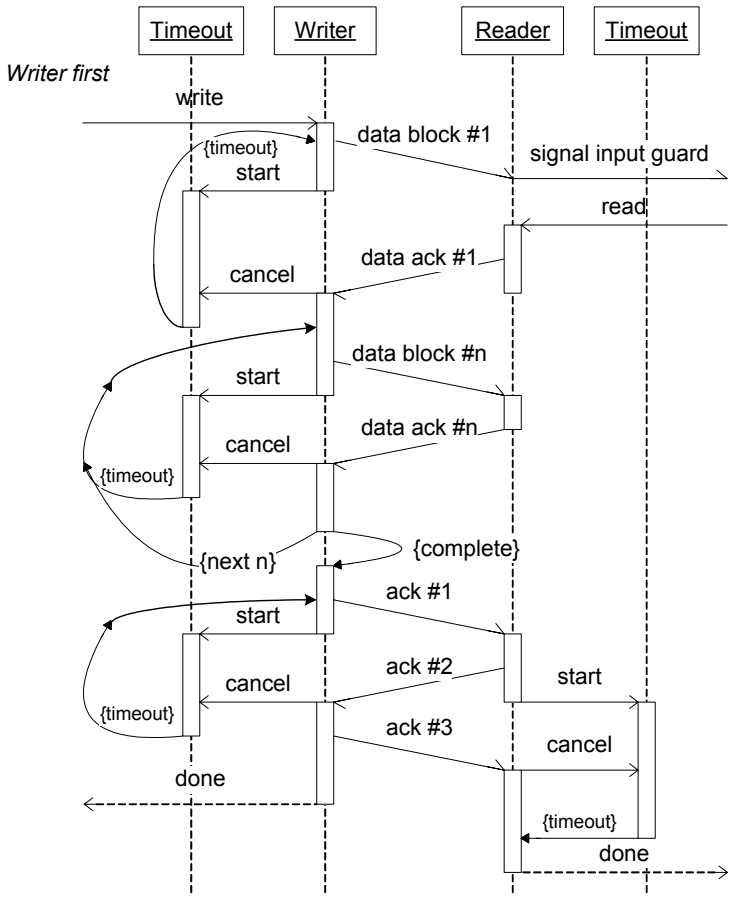

Figure A-4 Block based rendezvous protocol, writer comes first

Figure A-4 shows the protocol in case the writer arrives first. The protocol is safe in case of packet loss:

- When a data block or the corresponding acknowledgement is lost, the writer will timeout and send that block again.
- If *ack #1* is lost, the writer will timeout and retransmit it.
- As soon as *ack #1* is received, both parties already know that the data was transmitted successfully. However, at this time, the writer has no information whether the reader received *ack #1*. If the writer just had exited after sending *ack #1*, the reader could be left in a deadlock state, waiting endlessly for either additional data or an *ack #1*.
- If *ack #2* is lost, the writer will timeout first (as its timeout must be shorter than the receiver's) and the writer will retransmit *ack #1*.
- As soon as *ack #2* is received, both parties know that the rendezvous and data exchange is completed. The purpose of *ack #3* is to release the reader immediately. If *ack #3* is lost, the reader has to wait for the timeout. As this timeout is larger than that of the writer, it may assume that the writer did receive *ack #2*, and the reader may return.

Note that if the reader does not arrive in time to generate data *ack #1*, the writer's timeout will expire, causing the first data block to be sent repeatedly. This is unavoidable in the current implementation where the *writer ready* signal and the first data packet are combined, since there is no way to

distinguish between the loss of the first data block (which implies that the input guard is not signaled) or the reader being still unavailable. If resending a complete data block imposes too much overhead, the implementation can be changed to send a small *writer ready* packet after the first data packet and repeating this until a *reader ready* or a data *ack #1* is received.

For improved performance, *ack #1* could have been combined with the last data packet. There are two reasons for not doing this. The first is to keep a clear separation between the data transmission and final handshaking phases. The second, more important reason is that combining *ack #1* with the last data packet would assume that the data arrives at the reader in the same order as it was transmitted by the writer. While this is currently the case (the writer waits for an acknowledgement before sending the next data block), future optimizations of the protocol might invalidate this assumption. An obvious optimization would be to allow the writer to send multiple data blocks and wait for their acknowledgements in parallel.

Although the sequence diagrams makes a distinction between the first and all other data blocks, this distinction is not present in the implementation of the protocol. The only reason to draw the first data block separated from the rest is to visualize that the reader thread might not be available at that time already.

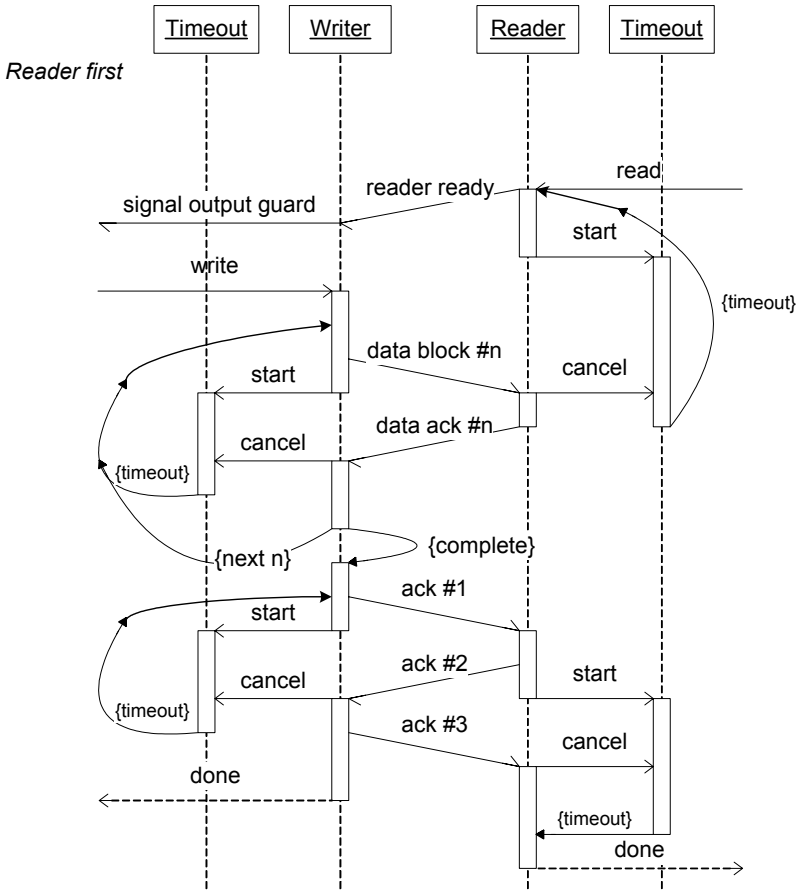

Figure A-5 Block based rendezvous protocol, reader comes first

The situation in which the reader arrives first (see Figure A-5) is handled in the same way as with the object-based protocol: the reader sends reader ready packets until it receives the first data packet from the writer. Once some data has arrived, the protocol continues in the same way as in the situation where the writer would have arrived first.

A.4 Writing a Network Device Driver

As explained in section 4.2, most of the work for a Remote Channel is already done in the Remote Link Driver and in the hardware independent part of the Network Device Driver. This makes adding support for new hardware a very easy task. This section describes how to write a new NDD, using the driver for the CAN-interface on the ADSP-21992 as an example. This driver can be found in the CT library, in the file *src/platform/adsp/ADSPCANNetworkDeviceDriver.cpp*.

A.4.1 Deriving a class

The new NDD should be derived from the NetworkDeviceDriver class, as a minimum only the constructor and the *sendPacket* function are needed:

```
class ADSPCANNetworkDeviceDriver : public NetworkDeviceDriver
{
private:
  unsigned int sendPacket(NetworkPacket * pkt);
public:
  ADSPCANNetworkDeviceDriver(unsigned int nodeID);
};
```

A.4.2 The constructor

There are two requirements for the constructor. The first is that the NodeID needs to be passed to the constructor of the NetworkDeviceDriver class; the second is that the maximum payload size (the maximum number of data bytes a network packet can carry) is set inside the constructor:

```
ADSPCANNetworkDeviceDriver::ADSPCANNetworkDeviceDriver(unsigned int nodeID):
NetworkDeviceDriver(nodeID)
{
  maximumPayloadSize = 4;
}
```

Generally the constructor will also initialize and configure the hardware, so that packets can both be sent and received.

A.4.3 The sendPacket function

This function is called by the device independent part of the Network Device Driver for each packet that needs to be transmitted. The function argument contains a *NetworkPacket* object, which contains all information needed for transmission. The function should return zero when it succeeded or any other number otherwise.

```
Unsigned int ADSPCANNetworkDeviceDriver::sendPacket(NetworkPacket * pkt)
{
  // Search for a free TX mailbox (=transmit buffer).
  // If not available, return a value other than zero.

  // Place pkt->data in the transmit buffer.
  // Use the other members of pkt to set addresses etc.
  // See include/ctcpp/linkdrivers/NetworkDeviceDriver.hpp for a list of
  // data members that must be included (not all members are required).

  // Make clear to the hardware that this transmit buffer may be transmitted

  return 0;
}
```

A.4.4 Incoming data

Incoming packets can be handled in two ways. The first and best option is to let the hardware generate an interrupt when new data is available. From the interrupt routine the data can be handed to the Network Device Driver. The second option is to make the device driver an active Process (for example running at idle time because of a low process priority), which can poll the receiver hardware for new data. However, this second option is not tested in the current CT implementation.

In both cases, the received data should be put in a NetworkPacket object, so it can be handed to the Network Device Driver:

```
void ADSPCANNetworkDeviceDriver::receiveIRQ()
{
  // allocate an object to put the data in (instead of dynamic allocation,
  // a fixed member object could also be used).
  NetworkPacket incomingPacket;

  // find the incoming mailbox that contains the new data

  // move the data from the buffer into a NetworkPacket
  // See include/ctcpp/linkdrivers/NetworkDeviceDriver.hpp for a list of
  // data members that must be included (not all members are required).
  incomingPacket.data = rxbuffer.data;  // etc.

  // hand the NetworkPacket to the NetworkDeviceDriver:
  packetReceived(&incomingPacket);
}
```

B 20-sim model appendix

The top-level model of the control loop is shown in Figure B-1. The *Sample* and *Hold* blocks isolate the discrete-time *controller* from the other, continuous-time parts of the model. The controller contains A-to-D and D-to-A converters to mimic conversion granularity, see Figure B-2.

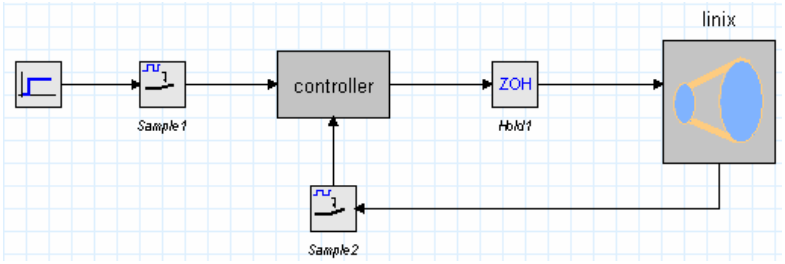

Figure B-1 The top-level control loop model

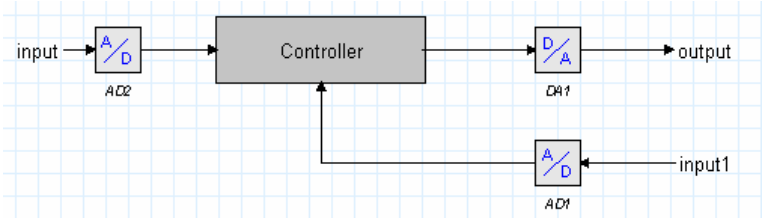

Figure B-2 The *controller* submodel

The *Controller* submodel, the actual PID controller, is shown in Figure B-3. The *ADC_in* port receives its input from the setpoint generator; the *EIU_input* is connected to the position sensor.

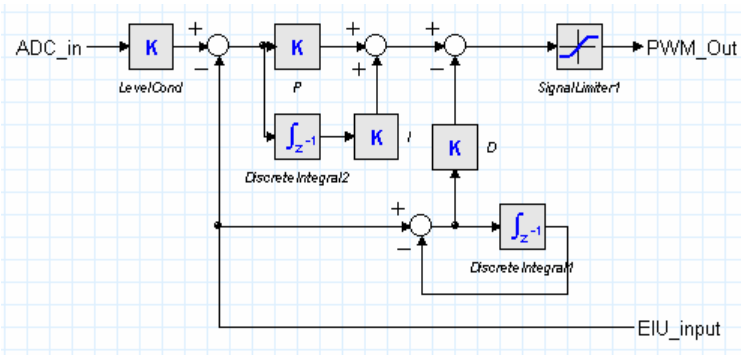

Figure B-3 The PID controller

The Linix submodel is shown in Figure B-4. The steering value from the controller is routed through an amplifier block, which represents the PWM H-bridge used in the real setup. The amplifier is modeled as an *MSf*, a modulated flow source. As the duty cycle of the PWM signal is almost linearly related to the current flowing through the motor, this is a simple but effective approximation.

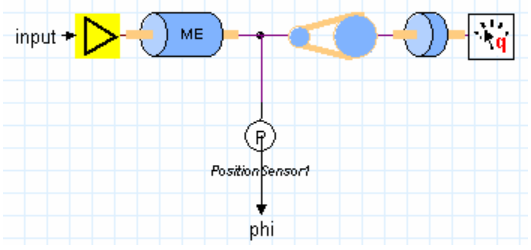

Figure B-4 The model of the Linix plant

The H-bridge used is however suspected to brake the motor during the off-periods of the PWM signal (via the *Electro Magnetic Force* .principle). If this is proven to be true, it could explain the inaccuracy

of the model, as this braking force is non-linear and related to motor speed, duty cycle of the PWM signal and electrical characteristics of for instance the power supply and transistors used.

The motor (*ME* in the Linix model) is shown in Figure B-5.

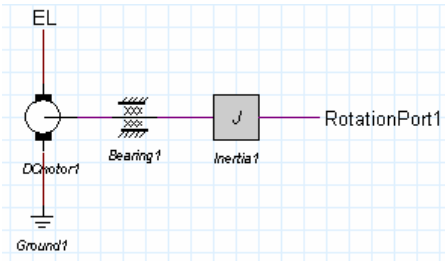

Figure B-5 Submodel of the motor

The submodel of the belt disc assembly (Figure B-6) does not contain friction elements. These are located in the motor and in the load on the second axis (Figure B-7).

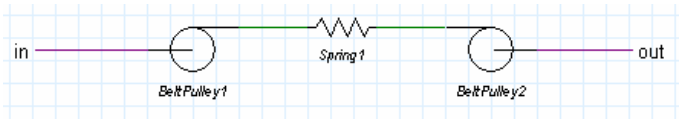

Figure B-6 The belt disc assembly

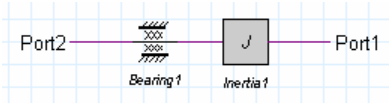

Figure B-7 The load axis

Finally, a listing of the parameters used for all elements of the model is shown in Figure B-8.

linix\Mechanical_Load1\Bearing1\d	0.1152
linix\Mechanical_Load1\Inertia1\J	0.056
linix\Axis1\Spring1\k	1.54
linix\Axis1\BeltPulley1\radius	1
linix\Axis1\BeltPulley2\radius	4
linix\DC_motor1\DCmotor1\k	0.292
linix\DC_motor1\Inertia1\J	0.00262
linix\DC_motor1\Bearing1\d	0.0001
controller\AD2\minimum	-10
controller\AD2\maximum	10
controller\AD2\bits	12
controller\AD1\minimum	-10
controller\AD1\maximum	10
controller\AD1\bits	12
controller\DA1\minimum	-10
controller\DA1\maximum	10
controller\DA1\bits	12
controller\Controller\P\K	9
controller\Controller\LevelCond\K	1
controller\Controller\SignalLimiter1\minimum	-0.6
controller\Controller\SignalLimiter1\maximum	0.6
controller\Controller\D\K	85
controller\Controller\DiscreteIntegral1\initial	0
controller\Controller\DiscreteIntegral2\initial	0
controller\Controller\I\K	1e-005

Figure B-8 Model parameters

References

Broenink, J.F. and G.H. Hilderink (2001), A structured approach to embedded control systems implementation*, in: 2001 IEEE International Conference on Control Applications,* M. W. Spong, D. Repperger and J. M. I. Zannatha (Eds.), IEEE, México City, México, pp. 761-766, ISBN: 0-7803-6735-9.

Buit, E. (2005), *PC104 stack mechatronic control platform*, MSc thesis, no 009CE2005, Control Laboratory, University of Twente, Enschede.

CLP (2002), *20-SIM,* http://www.20sim.com.

Douglass, B.P. (2003), *Real-Time Design Patterns: robuts scalabe architecture for real-time systems,* Pearson Education, Boston, ISBN: 0201699567.

Ferdinando, H. (2004), *Fault Tolerance in Real-Time Distributed Systems Using the CT Library*, MSc Thesis, no 002CE2004, Control Laboratory, University of Twente, Enschede.

Groothuis, M.A. (2004), *Distributed HIL Simulation for Boderc*, MSc Thesis, no 020CE2004, Control Laboratory, University of Twente, Enschede.

Henriksson, D. and A. Cervin (2003), *TrueTime 1.13—Reference Manual*, Technical report, no ISRN LUTFD2/TFRT--7605--SE, Department of Automatic Control, Lund Institute of Technology, Lund.

Henriksson, D., O. Redell, J. El-Khoury, M. Törngren and K.-E. Årzén (2005), *Tools for Real-Time Control Systems Co-Design - A Survey*, Internal Report, no ISSN 0280-5316, Department of Automatic Control, Lund Institute of Technology, Lund.

Hilderink, G.H., A.W.P. Bakkers and J.F. Broenink (2000), A Distributed Real-Time Java System Based on CSP*, in: The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000,* IEEE, Newport Beach, CA, pp. 400-407.

Hoare, C.A.R. (1985), *Communicating Sequential Processes,* Prentice Hall, ISBN: 0-13-153271-5 (0-13-153289-8 PBK).

Jovanovic, D.S., B. Orlic, G.K. Liet and J.F. Broenink (2004), gCSP: A Graphical Tool for Designing CSP systems*, in: Communicating Process Architectures 2004,* I. East, J. Martin, P. H. Welch, D. Duce and M. Green (Eds.), IOS press, Oxford, UK, pp. 233-251, ISBN: 1586034588.

Keshav, S. (1997), *REAL 5.0 Overview,* http://www.cs.cornell.edu/skeshav/real/overview.html.

Mocking, C. (2002), *20-sim code generation for ADSP-21990 EZ-KIT*, Individual Design Assignment, no 040CE2002.

Orlic, B. and J.F. Broenink (2003), Real-time and fault tolerance in distributed control software*, in: Communicating Process Architectures 2003,* J. F. Broenink and G. H. Hilderink (Eds.), IOS Press, Enschede, Netherlands, pp. 235-250, ISBN: 1 58603 381 6.

Orlic, B. and J.F. Broenink (2004), Redesign of the C++ Communicating Threads Library for Embedded Control Systems*, in: 5th PROGRESS Symposium on Embedded Systems,* F. Karelse (Ed.) STW, Nieuwegein, NL, pp. 141-156, ISBN: 90-73461-41-3.

Orlic, B., H. Ferdinando and J.F. Broenink (2003), CAN Fieldbus Communication in the CSP-based CT Library*, in: PROGRESS 2003, Embedded Systems Symposium,* F. Karelse (Ed.) pp. 163-171, ISBN: 90-73461-36-7.

Sunter, J.P.E. (1994), *Allocation, Scheduling and Interfacing in Real-time Parallel Control Systems,* PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, ISBN: 90-9007161-X.

VINT (2005), *The Network Simulator - ns-2,* http://www.isi.edu/nsnam/ns/.

Visser, P.M., M.A. Groothuis and J.F. Broenink (2004), Multi-Disciplinary Desing Support using Hardware-in-the-Loop Simulation*, in: 5th PROGRESS Symposium on Embedded Systems,* F. Karelse (Ed.) Nieuwegein, NL, pp. 206-213, ISBN: 90-73461-41-3.

Welch, P.H., M.D. May and P.W. Thompson (1993), *Networks, Routers and Transputers: Function, Performance and Application,* http://www.cs.ukc.ac.uk/pubs/1993/271.