University of Twente

EEMCS / Electrical Engineering
Control Engineering

Patterns for dependable and distributed embedded control

Mark Huijgen

M.Sc. Thesis

Supervisors
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
dipl. ing. D.S. Jovanovic

August 2005

Report nr. 032CE2005
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

The laboratory of Control Engineering of the University of Twente does research in the fields of embedded systems, mechatronics and robotics. One of the research projects is called Design Tools. This project focuses on the creation of a design tool called gCSP to facilitate the development of high reliable software for heterogeneous real-time control systems in a short time at a fraction of the present day costs. Key factors in this project are the safety and reliability of the designed software. The tool models software according to CSP, Communicating Sequential Processes. This is a process algebra that provides an easy way to reason about concurrent software without having to worry about multithreading, locking etc. The CT library (Communicating Threads) provides a framework on which these CSP programs can execute. Current work is on a C++ version called CTC++.

The aim of this MSc project is to create a set of CSP-compliant software design patterns that increase the safety and/or reliability of the software they are applied to. The patterns have to be implemented in the CTC++ library. To accomplish this goal three generally well known design patterns have been investigated and implemented, namely Logging and Monitoring, N-Way programming and Watchdogs.

Although CSP/CT programs can be transparently distributed, there are some difficulties in synchronously stopping distributed processes. A design pattern was created that specifically targets this problem.

The implemented patterns are demonstrated on the existing Tripod robotic setup. This setup consists of a platform connected via three arms to three linear motors. Safety and reliability of control software on such a setup are crucial factors.

The gCSP tool has support for applying the patterns to software models, facilitating easy use of them. Furthermore this project has made contributions to the CTC++ library with bug fixes and code cleanups and to the gCSP tool with bug reports and suggestions for improvements.

This project provides a starting point for distributed control on the Tripod setup by providing a distributed synchronized stop pattern and the new CSP/CT control software which is easier to make distributed then the old agent version. Finally this project presents some suggestions to further improve the implemented design patterns.

Samenvatting

De Control Engineering vakgroep van de Universiteit Twente doet onderzoek in embedded systemen, mechatronica en robotica. Een van de onderzoeksprojecten is Design Tools. De focus van dit project is de ontwikkeling van een software programma genaamd gCSP. Dit programma ondersteunt de ontwikkeling van betrouwbare software voor heterogene real-time regelsystemen door middel van een kortere ontwikkel tijd en voor een fractie van de huidige kosten. Belangrijke punten in dit project zijn de veiligheid en betrouwbaarheid van de te ontworpen software. Het programma modelleert software volgens het CSP principe: Communicating Sequential Processes. Deze procesalgebra biedt een makkelijke manier om te redeneren over parallelle software zonder op details te hoeven letten als multithreading en mutuele exclusie. De CT bibliotheek (Communicating Threads) bied een raamwerk waarop CSP programma's kunnen worden uitgevoerd. Huidig werk is aan een C++ versie genaamd CTC++.

Het doel van dit MSc project is om een set van CSP conforme, software ontwerp patronen te creëren die de veiligheid en/of betrouwbaarheid verhogen van de software waarop ze worden toegepast. Deze patronen moeten worden geïmplementeerd in de CTC++ bibliotheek. Om dit doel te bereiken zijn drie algemeen bekende ontwerp patronen onderzocht en geïmplementeerd, namelijk Logging and Monitoring, N-Way programming en Watchdogs.

Desondanks dat CSP/CT programma's gemakkelijk gedistribueerd kunnen worden, zijn er toch moeilijkheden bij het synchroon stoppen van gedistribueerde processen. Speciaal om dit probleem op te lossen is er nog een ontwerp patroon gemaakt.

De geïmplementeerde patronen worden op de bestaande Tripod opstelling gedemonstreerd. Deze opstelling bestaat uit een platform verbonden met drie armen aan drie lineaire motoren. Veiligheid en betrouwbaarheid van een dergelijke opstelling zijn van cruciaal belang.

Het gCSP programma heeft ondersteuning om de patronen toe te passen op software modellen zodat deze patronen gemakkelijk gebruikt kunnen worden. Daarnaast heeft dit project bijgedragen aan de CTC++ bibliotheek door middel van het repareren van fouten en het op schonen van de code. Tevens is een bijdrage geleverd aan de ontwikkeling van het gCSP programma door fouten en suggesties voor verbeteringen te melden.

Dit project biedt een startpunt voor gedistribueerde regeling van de Tripod opstelling door middel van het nieuwe patroon voor het synchroon stoppen van processen en de nieuwe op CSP/CT gebaseerde regel software die distributie gemakkelijker maakt dan de oude op agents gebaseerde software. Verder geeft dit project nog enkele suggesties om de ontworpen patronen te verbeteren.

Contents

1 Introduction

This project is about increasing software quality in the field of safety and reliability, the most important attributes of dependability for control systems. This has been accomplished by designing three generally applicable design patterns. These patterns are demonstrated on a robotic setup, where safety and reliability are often important factors. Robotic applications are often distributed and concurrent. A fourth design pattern is presented to solve synchronization in distributed CSP/CT software.

1.1 Background

This project is done as an MSc project at the laboratory of Control Engineering (CE) of the faculty EEMCS of the University of Twente. The main goals of this department are to investigate the applicability of modern systems and control methods to practical situations, mainly in the fields of mechatronics and robotics. In the past the department investigated the application of occam programming language and transputer hardware for real-time control systems. The approach resulted in reliable, robust and well structured software. Nowadays transputers have become obsolete, but the software design method called Communicating Sequential Processes (CSP) is now applied to modern languages like Java, C and C++.

One of the projects from the group is the Design Tools project. The focus of this project is the development of a CASE (Computer Aided Software Engineering) tool using the CT library (Communicating Threads), which allows for building high reliable software for heterogeneous real-time control systems in a short time at a fraction of the present day costs. In this project safety and reliability of the designed control software are key factors.

1.2 Software methodology

In the process of designing an embedded control system four different design phases can be distinguished (Figure 1-1).

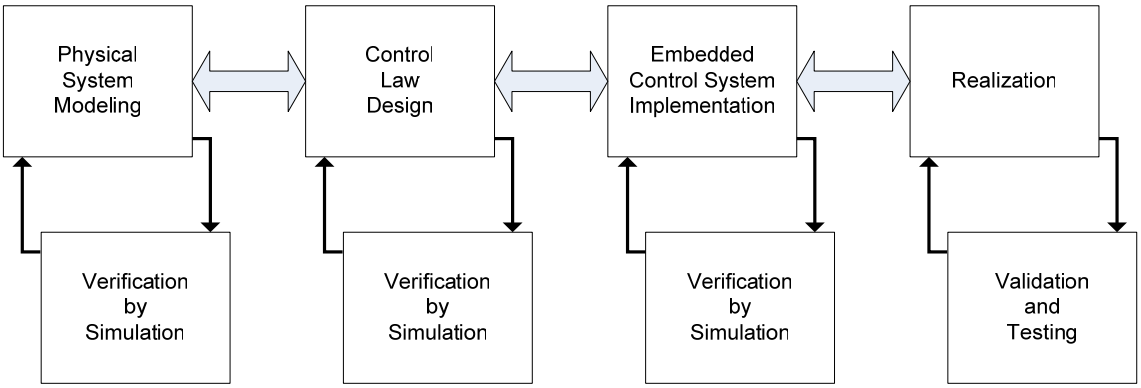

Figure 1-1 Design trajectory for embedded control systems

Support for the first two phases is covered by 20-sim. This is a modelling and simulation tool developed by Controllab Products (Controllab Products B.V., 2005), a spin-off company of the Control Engineering group of the University of Twente. It is an application consisting of several integrated modules that support modelling and design of mechatronic systems in many aspects. It has functionality for automatic code generation from (sub)models thus partially supporting the third phase. The code generation consist of filling template files with the model equations resulting in a stand alone program or a module usable in other programs. 20-sim does not model the software itself, but since software is getting ever more complex the modelling of software becomes a necessity nowadays.

The group currently works with software modelled according to the CSP model. It is a process algebra (Hoare, 1985) that can describe concurrent systems and provides a way to analyze these systems

without having to worry about low-level concepts like multithreading, locking etc. The earlier mentioned CT library (Hilderink, 2005) developed by the group implements this CSP model. Current work is on a C++ version called CTC++ (Orlic and Broenink, 2003).

To get from a 20-sim model directly to an implementation based on CSP/CTC++ is a big step, especially for complex systems. To divide this step into smaller steps a tool was created as part of the design tools project called gCSP (Jovanovic *et al.*, 2004). This tool can be used to model CSP software graphically. It provides functionality to generate machine readable CSP code (CSPm) for formal checking to do an extra verification step and code generation to generate an implementation that can run on the CTC++ library.

Control systems are generally real-time. Real-time in this context means that processing the inputs, calculating the control law and preparing the outputs must be completed within a fixed amount of time. The work has to be completed before the deadline. For software development this means for example that all memory has to be allocated before starting real-time processing.

The real world is a concurrent place. Concurrency in software introduces problems like deadlock and livelock. This makes concurrent programs harder to verify then sequential ones. The CSP/CT provides a structured way to use concurrency in software without having to worry about these problems. Control software can benefit from concurrent software because then it can better match the real world.

1.3 Safety, reliability, dependability

Dependability of a system is determined as a combination of various important system quality attributes. An overview of these attributes is given in Figure 1-2.

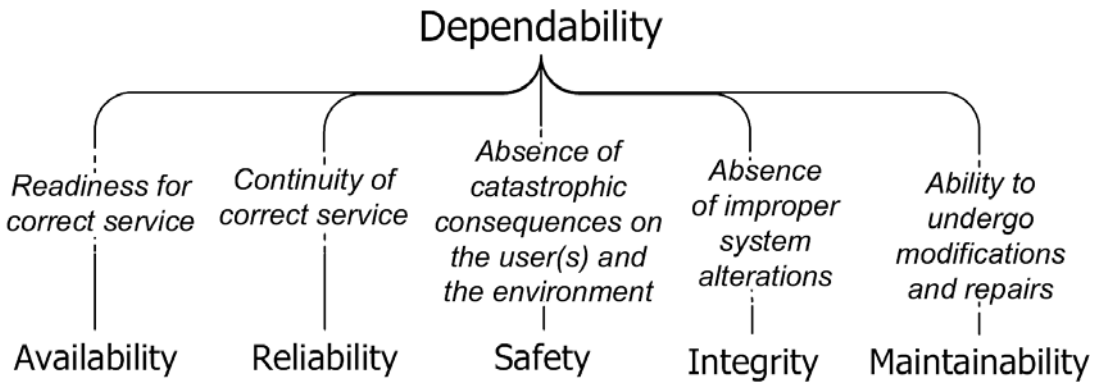

Figure 1-2 Dependability attributes according to (Avižienis *et al.*, 2004)

Two of these attributes are of main interest in this MSc project, namely safety and reliability.

Safety
Safety of a system is its freedom from those conditions that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm, (Leveson, 1995).

Reliability
Reliability of a system is taken to be a measure of the success with which a system conforms to some authoritative specification of its behaviour, (Randell *et al.*, 1978).

1.4 Objectives

The aim of this project is to create a set of CSP-compliant software design patterns that increase the safety and/or reliability of the software they are applied to. The patterns have to be implemented in the CTC++ library. To accomplish this goal three patterns were created and will be treated here shortly.

Logging and monitoring
This pattern provides a means to log data in real-time for later analysis. This pattern can for example be used for debugging during development, increasing the overall reliability of the final program. It can also be used in the final program to record important events like (imminent) failure of (parts of) the system. It can even increase the maintainability of a system. The second part of this pattern, monitoring, can be used to increase the safety of a system. A monitoring component can watch certain values in a system and take action if they get out of hand.

N-Way programming
This pattern mainly increases the reliability of a system by introducing redundancy. It uses multiple independent versions of a component with the same functionality and may do error correction. With error correction it is possible to continue working in cases of failure of some parts of the system, where without this would not be possible, thus increasing the reliability of the system. The pattern can also abort the redundant components if an unrecoverable error is detected, this can then be used to start a safety component. This can increase the safety of a system.

Watchdogs
This pattern mainly increases the safety of a system by checking if the system is alive. If a certain software component does not signal the watchdog that it is still running the watchdog can intervene. This intervention can range from just logging the event to aborting a part of a program and starting a safety component.

By nature of CSP programs designed according to this paradigm are easy to distribute: one can distribute processes as long as the communication relationships remain intact. A pattern was created to provide a means to synchronously stop multiple processes called *Synchronous stop*.

The applications of these implemented patterns will be demonstrated on the existing Tripod robotic setup. This setup consists of three vertically mounted linear motors connected to a single platform. When dealing with moving robots, safety is an important issue. Safety mechanisms are needed to prevent the robot from inadvertently harming others as well as itself. To be able to demonstrate the design patterns on Tripod its multi-agent based control software (Breemen, 2001; Eglence, 2003) has to be converted to CSP/CTC++ software.

1.5 Outline

Chapter 2 describes some of the theoretical and practical background that are important to this project. It also give a comparison of agent based software and CSP/CT software. Finally it presents the Tripod setup.

Chapter 3 presents the pattern for distributed synchronized stop demonstrated on an example. Second it describes the new CSP based control software for the Tripod setup and does some suggestions for distributed control of this setup.

Chapter 4 treats the three design patterns for increased safety and reliability: Logging and Monitoring, N-Way programming and last Watchdogs. All patterns are demonstrated with example programs.

Chapter 5 discusses the application of the three design patterns on the Tripod setup in the same order as chapter 4 did. It shows how the patterns were applied and what the results are.

Chapter 6 contains the conclusions and recommendations of this project.

This report often refers to a CDROM (see also Appendix C). It contains all the gCSP models and code for the examples discussed in this report. Furthermore it contains the code of the new Tripod program and the gCSP model of this program.

2 Background and Tripod setup

This chapter presents the theoretical and practical background of this project. First an introduction to gCSP is given, a graphical modelling language for CSP programs. The second section discusses the similarities and differences between agent based software and CSP/CT based software. Last a description of the Tripod robotic setup is given.

2.1 Introduction into CSP/CT and gCSP

This section introduces the concepts of CSP/CT and how they are modelled by the gCSP tool. It starts by introducing the concept of processes and how they can be put in a composition. Then the primitive processes that are available in the CT library will be treated. How processes communicate is explained in the fourth subsection. Finally parenthesizing in graphical models is introduced.

2.1.1 Processes

A process is an independent entity that performs a specific task. It does not know of the existence of other peer processes, but it can be a collection of other processes (children). It can only communicate with other processes or the environment via channels. In gCSP a process is represented by a rectangle with a name in it like Process1 and Process2 in Figure 2-1. The figure also shows an arrow between the two representing a channel used for communication between Process1 and Process2, channels will be explained in subsection 2.1.4. The straight line is the compositional relation between the processes, in this case it means that the processes execute in parallel, compositional relations will be explained in the next subsection.

Figure 2-1 gCSP diagram with two communicating processes running in parallel

2.1.2 Compositions

Figure 2-1 shows a relation with the parallel operator, meaning that the two processes execute in parallel. In total there are seven compositional relations that can exist between processes. Table 2-1 shows all operators. The watchdog operator is a contribution of this project, see section 4.3

Table 2-1 The seven compositional relations in CSP/CT

→	Sequential		
‖	Parallel	□	Alternative
‖	PriParallel	□	PriAlternative
△	Exception	□	Watchdog

Sequential

This operator says that the processes in the relation will be executed sequentially in the order the arrow indicates, in the case of Figure 2-2 first Process1 will execute and when that one is finished Process2 will execute. After the last process finishes the construct is also finished.

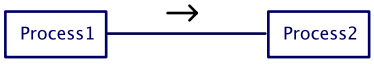

Figure 2-2 Sequential composition

(Pri)Parallel

This operator says that the processes are executed in parallel (concurrently). The parallel version does not have priorities and gives all processes in the construct an equal chance of executing. The prioritized version gives a higher priority to the process it is pointing to; in the case of Figure 2-3 Process1 has a higher priority then Process2. The construct finishes if all the processes in it have finished.

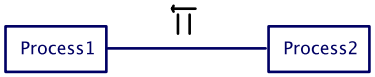

Figure 2-3 PriParallel composition

(Pri)Alternative

This operator actually works on guards in stead of processes. However each guard has a process associated with it. Alternative is about making a choice, the construct chooses the first guard that is ready and runs the associated process. After the process finishes the construct finishes also. If there are multiple guards ready at the same time only one is chosen. All guards have an equal chance unless the prioritized version of the operator is used. The prioritized operator chooses the guard with the highest priority indicated in the same way as with PriParallel. Figure 2-4 shows two guards with processes in an alternative construct. The guards are indicated by ChannelIn[], in this case they will become ready if the associated channel is ready for communication (e.g. when a process on the other end of the channel wants to write to it).

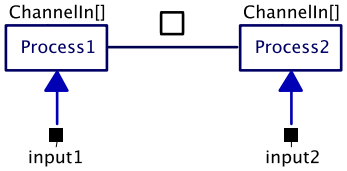

Figure 2-4 Alternative composition

A guard can also work on an output channel; it will become ready when the other side wants to read from that channel. Third there is the SKIP guard, which is always ready. Guards also can have a condition between the brackets like [variable >= 1], the guard can only be ready when this condition is true.

Exception

The exception operator makes it possible for a process to handle exceptions thrown in another process. For example if Process1 (Figure 2-5) throws an exception it will abort and the exception can be handled by Process2. Process2 is drawn as an ellipse despite the fact that it is a normal process just like Process1. Exception handlers are drawn differently so a user can clearly see what part of the program is for exception handling.

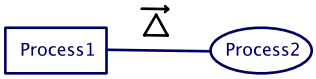

Figure 2-5 Exception composition

Consider the diagram in Figure 2-6. It shows two relations: first Process1 and Process2 in a parallel relation; second (indicated by the bubble with index 1) the parallel in an exception relation with Process3. If Process2 throws an exception, this exception will not abort the parallel construct, only Process2. The parallel construct keeps running until Process1 also finishes (either normally or also throwing some exception). The exception thrown by Process2 will only be caught by Process3 after the parallel construct has finished.

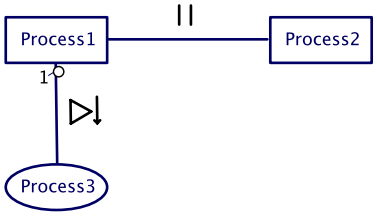

Figure 2-6 Parallel composition inside an exception composition

More information on exception handling in CT can be found in (Engelen, 2004).

Watchdog

The watchdog operator (Figure 2-7) is a contribution of this project and is explained in detail in section 4.3.

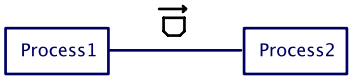

Figure 2-7 Watchdog composition

2.1.3 Primitive processes

This subsection treats the primitive (processes) that are available in gCSP. Described are repetitions, codeblocks, readers and writers, barrier with syncer processes, linkdrivers and last watchdog operations.

Repetition

A repetition primitive can conditionally restart the process (or construct) it is coupled to. An example is given in Figure 2-8, where Process1 is restarted until the condition '!stop' becomes false. The relation between a repetition and the process it works on is always sequential. The direction of the sequential operator determines if the condition is a pre-condition (while[condition] do {…}, Figure 2-8) or a post condition (do {…} while[condition]).

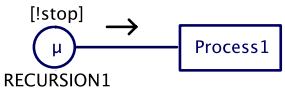

Figure 2-8 Repetition construct

Readers, Writers and Codeblocks

A reader is a primitive process that reads from a channel and puts the result in a local variable. The writer process does the opposite; it takes the content of a local variable and writes it into a channel. After their action is performed they finish. Readers/writers can be made aware of their local variable by means of a varchannel. The example in Figure 2-9 first reads from the input channel. Then the codeblock does some processing on the variable read and places the result in the output variable. The writer then writes the result into the output channel.

A codeblock is a primitive process that can contain no other processes. They are used for low level data processing and can access local variables in a process. Codeblocks are made aware of those variables by means of varchannels (see Figure 2-9), those special channels will be treated in the next subsection). Codeblocks cannot read from normal channels.

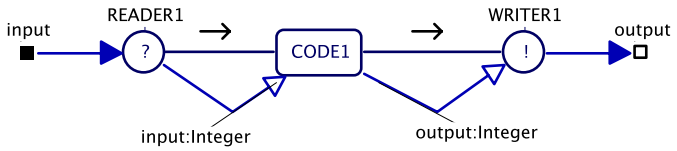

Figure 2-9 Reader to codeblock to writer

Barrier with syncer processes

A barrier is a method to synchronize multiple independent processes. A barrier synchronizes syncer processes. When a syncer executes it becomes ready and it will suspend until the barrier becomes ready, the barrier becomes ready when all its associated syncers are ready. All syncers then resume execution. An example is shown in Figure 2-10, where to parallel branches synchronize after reading from their channels (channels are not shown here). The barrier is represented by a line with on both sides a diamond shape, a syncer process is represented by a circle with a * in it.

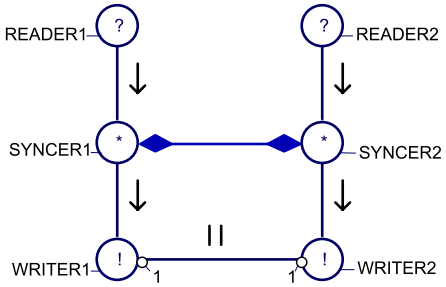

Figure 2-10 Two parallel branches synchronized with a barrier

Linkdrivers

Linkdrivers can provide an interface between a CSP/CT program and the environment, for example hardware (Hilderink, 2005). An output link driver is modelled as a reader that reads a value from a channel and converts it to a form acceptable for the environment. Since the reader link driver reads from a channel, although it models an output port from the software, it is depicted as an inverted reader icon. Figure 2-11 shows both a writer linkdriver (from which Process1 reads) and a reader linkdriver (to which Process1 writes).

Figure 2-11 Linkdriver writer and reader

Watchdog operations

Watchdog operations are depicted by shield symbols; there are three of them (Figure 2-12). Every shield has a letter inside. S is for starting a watchdog, H is for hitting the watchdog and R is for stopping (removing) the watchdog. The operations can be put in compositional relations just like other processes. The watchdog is explained in more detail in section 4.3.

Figure 2-12 Watchdog operations

2.1.4 Channels

In CSP processes communicate via channels. A channel conveys a message from one side to the other. In CSP this can be bidirectional communication, but in CSP/CT the channels are unidirectional. By default the communication is done synchronously, two processes rendezvous before the message is passed. Channels are indicated in gCSP by arrows with a closed arrowhead (see Figure 2-1). Since message passing is synchronous a channel between two sequential processes or between two processes in the same alternative is a design error.

A channel can be replaced by a probe channel. Probe channels help to facilitate the logging system contributed by this project and are discussed in more detail in subsection 4.1.1. Graphically they differ from the normal channel in thickness and colour.

The second form of channels in CSP/CT is called a varchannel and its abstraction in CSP is variable. Communication in a varchannel happens asynchronously. A varchannel can be seen as a shared variable between processes. These channels should not be used between parallel processes because race conditions can occur. They are useful for example to pass a value between a reader and a codeblock as happens in Figure 2-9, they are indicated with an open arrowhead.

Channels can be poisoned, also called rejected. At the time a channel is poisoned an exception is stored inside and when a process attempts to read from or write into this channel it will throw the stored exception.

2.1.5 Parenthesizing

Consider the gCSP diagram in Figure 2-13. This model has compositional ambiguity. Without compositional grouping it would not be clear which execution behaviour is desired. For example it could be that (5 -> 1) || 2 || (3 -> 4) but also (5 -> (1 || 2)) || (3 ->4).

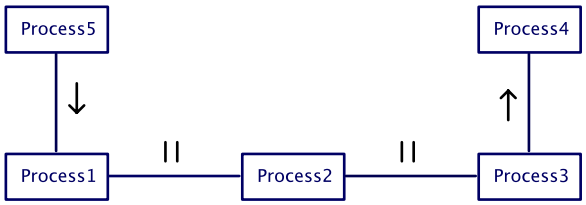

Figure 2-13 Compositional ambiguity

In the diagrams the parenthesizing is done with bubbles with a numeral index (Figure 2-14). The image also has boxes drawn to complement these bubbles. This also explains how the index number comes to be: it is the number of boxes that cross on that side of the relation. In this diagram boxes make it easier to understand the parenthesis, but in more complex diagrams they tend to clobber the diagram. Figure 2-14 shows (5 -> (1 || 2)) || (3 ->4).

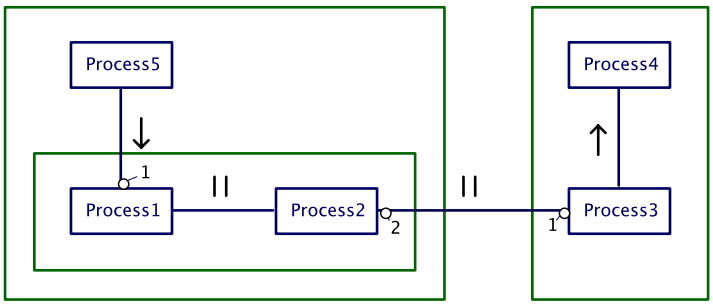

Figure 2-14 Ambiguity solved by parenthesizing

2.2 Conversion from agent based control software to CSP software

To demonstrate the patterns on the Tripod setup, first the existing Tripod software needed to be converted to run on the CTC++ library in which the patterns will be implemented. The existing software for Tripod is agent based (Breemen, 2001), (Eglence, 2003). This section describes the main differences between agent based software and CSP based software with respect to structure and execution framework. It also describes the steps necessary to convert agent based software to a CSP/CTC++ based implementation.

2.2.1 Comparison

Agent based software consists of one or more agents. Each agent can handle a specific task. Agents can be grouped with a coordination object, see Figure 2-15. This object controls what agent(s) of the group can be active at a given time. CSP based software consists of processes. Processes can be grouped in a hierarchy in which they can run in parallel, in sequence or alternative to each other. The relation between agents and processes and the relation between coordination objects and the CSP hierarchy will be treated in this subsection.

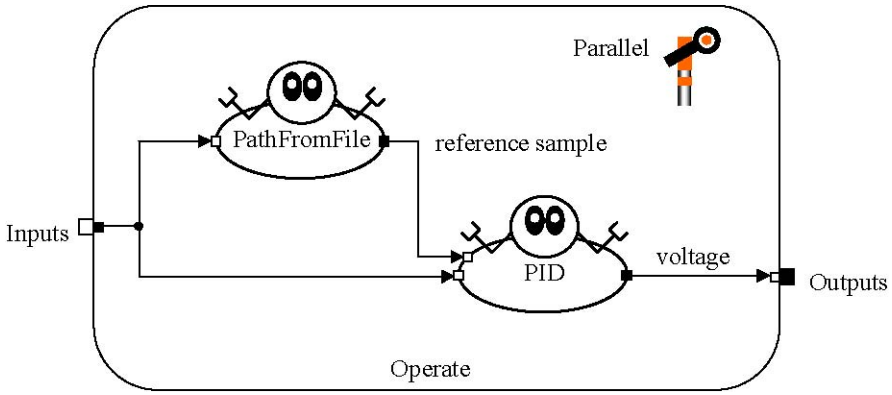

Figure 2-15 Two agents in a Parallel coordination object

Execution

The main difference in execution between agents and processes is the fact that agent based software calls agents each period to perform their job and CSP/CT processes are started once and continue to do their job until the process itself stops.

The agent framework checks every period if an agent wants to be active. Depending on the coordination object associated with the agent it is then decided if the agent actually is allowed to run this period. If so the agent is called to perform its job for that period. Execution of agents is controlled centrally by the framework.

CSP has no real notion of time. Processes are started by the framework, but will only stop if they decide to do so. The framework itself cannot just stop processes; execution is controlled decentrally by the processes themselves (more on this in the next subsection about communication).

The second important difference is that the agent framework is running in a single thread, this in contrast to the CSP model where each process could be a separate thread and really execute concurrently.

Communication

Another key difference is in the communication between agents versus the communication between processes. In the agent framework communication is done via ports. Upon creation of an output port a local variable is coupled to this port. Writing to the port is performed by just modifying this variable. Reading from a port however is done via the port itself with the method `port->getValue()`. Important here is that no explicit synchronization occurs between agents, but in the agent framework

this is not a problem since it is single threaded. This is the main advantage of the agent framework, it is light weight, but also its disadvantage, no real concurrency.

In CSP communication between processes is performed via channels. In CSP writing and reading is done via the channel itself. The default channel behaviour is to let the processes rendezvous before exchanging data. This means if a process wants to write to a channel it will suspend until there is also a process on the other side of the channel that wants to read from it. This is how control of execution is decentralized in CSP.

Implementation

The main implementation difference between an agent and a process is the number of operations it has. Operations as in the sense of member functions of a C++ class. A process has only one public function: `run()`. The agent equivalent to this is `calculate()` (or `sense()` in case of a sensor agent or `actuate()` in case of an actuator agent, see Table 2-2). Consider an agent/a process that implements a simple controller. Calling the calculate function will result in one step, and calling the run function will result in running the controller until it determines to stop. Agents also have got the following extra functions with respect to processes: `start()`, `stop()`, `initialize()`, `finalize()` and `activation()`. When converting agent software to CSP/CTC++ software, start and stop can be integrated into the constructor and destructor of the process. The initialize and finalize function can be integrated at the start and end of the run function. Since the activation function describes when an agent wants to be active it can be used by the new process to determine when to stop and/or could be used to enable/disable the guard if the process is used in an alternative construct. It can be concluded that the CSP/CT framework in this respect is simpler to understand and use.

Table 2-2 Agent types vs. CSP

Agent types	CSP
Controller agent	Process
Sensor agent	(Input) Linkdriver
Actuator agent	(Output) Linkdriver

2.2.2 Conversion

Knowing the main differences between agents and processes, it is possible to convert agent-based software to CSP based software. This subsection presents a number of steps that can be used as a guideline to accomplish this conversion. Since only one agent program was converted, this guide may not cover all possible agent constructions.

The easiest way is to start with a new .cpp and .h file for each agent that will be converted to a process. Create a class derived from process and create a constructor and an empty run function. Start with converting all elementary agents (agents that don't contain other agents) to processes.

- Copy initialization code from agent constructor to constructor of process
- Copy code from `start()` function to process constructor
- Copy code from `stop()` function to process destructor
- Copy code from agent destructor to process destructor
- Create a variable and channel for each port.
- Copy code from `initialize()`, `calculate()`, `finalize()` into `run()`.
- Add around the code copied from `calculate()` a `while(condition)` construct, since timing is now implicit via channels. The expression for 'condition' depends on the desired behaviours of the agent, most importantly when to stop. The condition has to be determined by studying the code of the agent.
- Find where port variables are assigned a value and add a `channel->write(&variablename)` directly thereafter.
- Find where port variables are read in the code and add a `channel->read(&variablename)` before that.

Now convert sensor and actuator agents to linkdrivers so they can be triggered on external events, e.g. a timer. The same steps as with an elementary agent should be taken, except that `calculate()` is called `sense()` in case of an sensor agent and `actuate()` in case of an actuator agent.

In the case of a timed input linkdriver, a read call will suspend the calling process until the linkdriver unlocks it, for example on a timer interrupt. However caution is advised when a process has to read multiple timed linkdrivers that have the same period. This needs to be done with a number of reader processes in a parallel construct. In this way all readers will be unlocked at the same time by their respective linkdrivers. If you would read in sequence, the read call on the first channel will suspend the process and on the next timer interrupt it will resume again. Now the second read takes place, which will again block. Now the reading takes two periods! (Or more of course, depending on the number of sequential reads.)

The last step in the conversion is mapping agent coordination objects to CSP/CT compositional relations. The different types of coordination objects as well as the different CSP compositions are in Table 2-2. For each coordination object its equivalent CSP composition will be presented.

Table 2-3 Coordination objects vs. compositions

Coordination objects	CSP compositions
Independent Parallel Cooperative Master-slave Cooperative Fuzzy addition	(Pri)Parallel
Competitive Fixed priority	(Pri)Alternative
Competitive Sequential Competitive Cyclic	Sequential

Independent Parallel
This means that agents work independently in parallel to each other. This maps to parallel interleaving processes in CSP/CT.

Cooperative Master-Slave
This means that when the master agent is active, the slave agent(s) is(are) also active. In most situations this directly maps to parallel processes in CSP when there is direct communication between the processes. In this way the master-slave principle will be taken care of by the rendezvous of the channels.

Cooperative Fuzzy addition
This coordination object cannot be mapped directly; an extra process is needed to do the fuzzy addition. The output is a function of the outputs of all the agents taking part. The coordination object can also handle the situation when not all agents are active, so it only takes the active agents into account. With CSP this exact behaviour is somewhat difficult due to the rendezvous behaviour of channels.

Competitive Fixed priority
The agent with the highest priority that also wants to be active may run this period (and the others not). This maps to pri-alternative processes in CSP/CT. There is a difference however, with competitive fixed priority each period the framework chooses which agent is run. So if an agent with a higher priority also wants to be active that one is chosen. With CSP/CT the chosen process will keep on running until it decides to stop, also if a higher priority process in the pri-alternative wants to be active.

Competitive Sequential
The agents are activated in sequence. This maps to sequential processes in CSP/CT.

Competitive Cyclic
The behaviour of cyclic coordination is the same as sequential, with the added rule that when the last agent ends, the first one is activated again. This maps to sequential processes in CSP/CT with repetition.

2.3 Tripod setup

Tripod is a simplified version of a Stewart platform (hexapod). It is constructed by Imotec B.V. Whereas a Stewart platform has six degrees of freedom, the platform of Tripod only has three, namely translation along the x-, y- and z-axis. By construction the platform is always aligned in the x-y plane. The three arms connected to the platform can all be moved individually along the z-axis by three linear motors. A photo and schematic drawing of the Tripod setup are shown in Figure 2-16 and Figure 2-17 respectively.

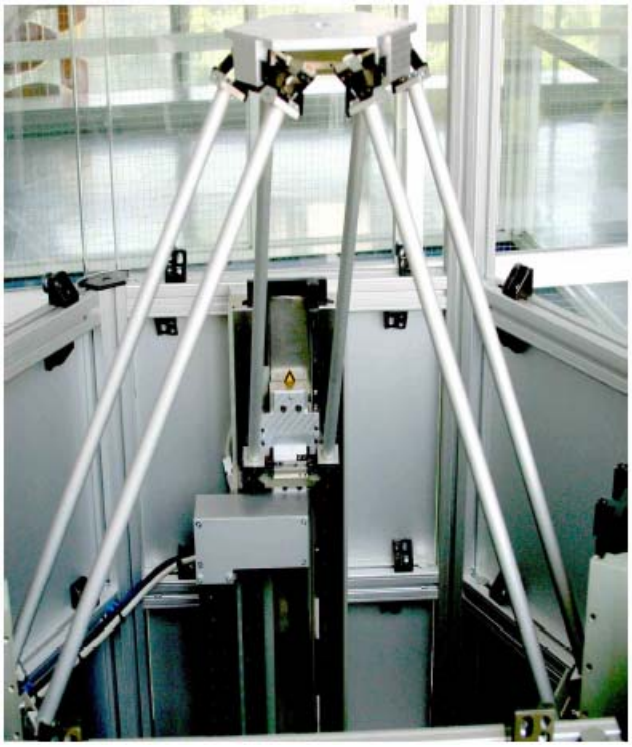

Figure 2-16 Photo of Tripod setup

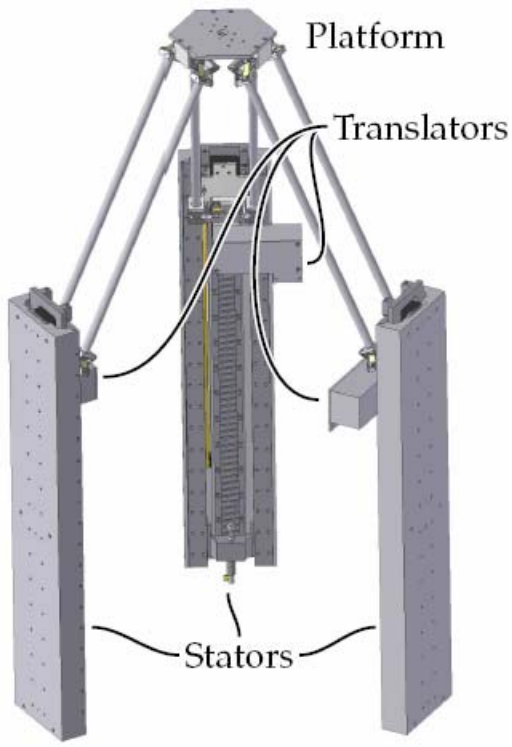

Figure 2-17 Schematic view of Tripod setup

An arm actually consists of two arms in a parallelogram construction together with the platform and a translator (linear motor). A detail of the connection to the translator is shown in Figure 2-18. The joints connecting the arms to the translator only allows rotation in the φ and θ direction. The connection of the arms to the platform is similar.

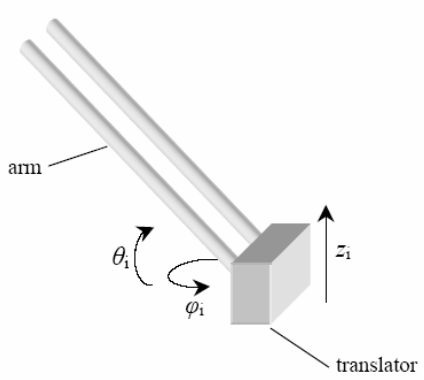

Figure 2-18 Connection of arm to translator

The position of the linear motors is measured with a linear quadrature encoder. Besides the usual quadrature signal and index signal, this encoder is also equipped with end switches at both ends and a signal that indicates if the motor is at the lower or upper half of its stroke. This last signal can be used to check if the motor is below the index marker on the encoder before calibration is started.

The safe working area of the platform is a cylinder shape with a radius of 200 mm and a height of 250 mm. If the platform exceeds this area, the leaf springs in the joints or even the joints themselves will be damaged. In (Eglence, 2003) the following formulas are given to approximate the distance from the center (p_{appr}) and the height (z_{appr}) of the platform, these should stay within respectively the radius and height of the safe working area. In the second formula l is the length of an arm which measures 541 mm and r is the radius in which the three motors are positioned which amounts to 300 mm.

$$p_{appr} = \frac{0.246}{r}\sqrt{z_1^{\,2} - z_1 z_2 - z_1 z_3 + z_2^{\,2} - z_2 z_3 + z_3^{\,2}}$$

$$z_{appr} = \frac{z_1 + z_2 + z_3}{3} + 1.355(l^2 - r^2 - p_{appr}^2) - \sqrt{l^2 - r^2} + 0.149$$

Because these formulas are approximations the safe limits are somewhat lower then 200 mm for the radius and 250 mm for the height. According to (Eglence, 2003) the safe values are 170 mm for the radius and 234 mm for the height. With these limits the platform can never move beyond the safe working area.

Information on how linear motors work can be found in Appendix A. Information on the computer hardware included in the Tripod setup can be found in Appendix B. Finally some other interesting technical specifications of the Tripod:

Max. stroke linear motors	520 mm
Max. speed	1.5 m/s
Max. acceleration	30 m/s^2
Max. payload	5 kg

3 CSP/CT control software and distribution

This chapter discusses the new Tripod CSP/CT control software for Tripod. During the development problems were encountered with synchronization. The solution for this problem is treated first in this chapter. Then a description of the new Tripod software and finally some suggestions for distributed control on Tripod.

3.1 Non blocking synchronized stop in CSP/CT

There are different types of synchronization possible between processes. One of them is barrier synchronization between N processes. When a process reaches the barrier it will wait until all other processes have also arrived. At that moment data may be exchanged and all processes will continue execution. This form of synchronization has one problem however: processes will block. This blocking is unwanted in the case of those processes being controllers. What one wants is to keep controlling. Only when all controllers are ready to stop, they stop together.

3.1.1 Solution: SyncStop

The solution to the blocking problem is given in Figure 3-1. Two sequences of processes need to terminate synchronously when both sequences are running the last process. None of the processes SequenceX_2 may block on this synchronization. The solution is to put the last process of each sequence in an alternative with a syncer process, with all syncer processes linked to the same barrier. A repetition that may be inside the last process has to be placed outside of the alternative; otherwise the SequenceX_2 will run indefinitely because the alternative can never select the syncer process.

As soon as all sequences (in this case two) are running their final alternative construct, the guard on the barrier becomes ready. This will run the barrier synchronization and some code to modify the repetition condition to break the loop. It this case the sequence ends, but it is possible that each sequence continues with another process. This pattern works also if the two sequences run on separate systems.

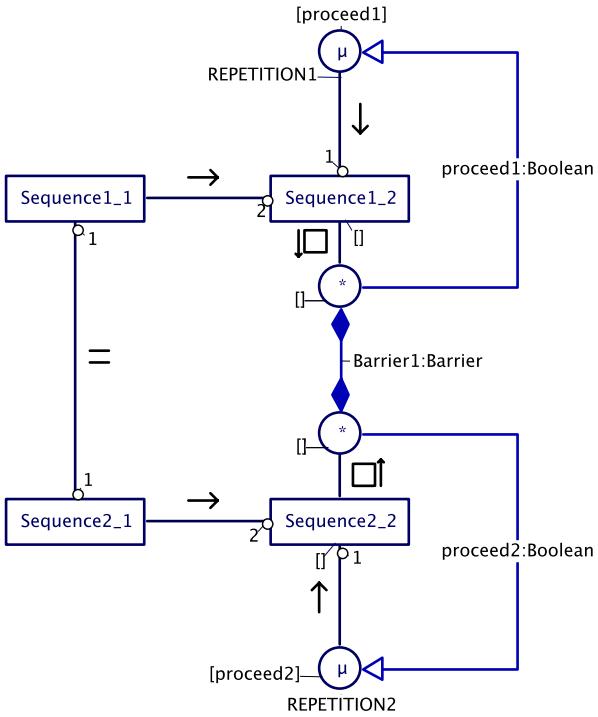

Figure 3-1 Non blocking synchronized stop

3.1.2 Example program

At the time of writing the library does not yet support syncers and barriers. Here a design pattern is provided that achieves the same functionality with basic communication primitives like readers, writers and channels. This example contains three sequences in parallel called worker_1 to worker_3 (Figure 3-2). The workers together are called a crew. The other figure (Figure 3-3) shows one sequence (one worker). A sequence consists of two processes: job and jobhold. Input represents the data processed by either job or jobhold.

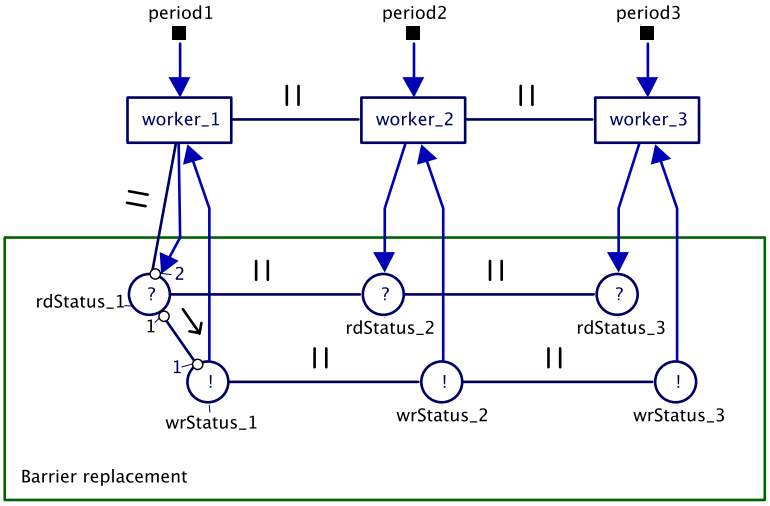

Figure 3-2 Barrier replacement

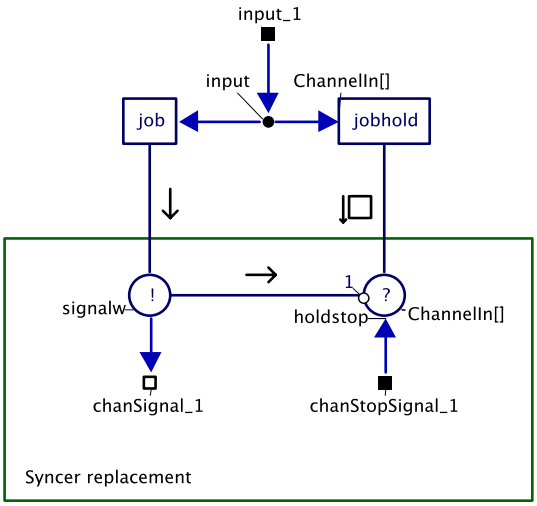

Figure 3-3 Syncer replacement

The barrier in this example works as follows:

- Once a Job process terminates the ‘syncer’ signals the ‘barrier’ that it is ready to rendezvous and then goes into the alternative with jobhold. The repetition is omitted in the diagram.
- Once all three workers have signalled the ‘barrier’ the parallel with the readers ends (Figure 3-2). In sequence a parallel with writers is started that will communicate back to the syncers.
- All three alternatives will now run the reader part of their ‘syncer’ and all sequences terminate.

The whole example contains two crews with each three workers that run in sequence after another. The complete code and model for this example can be found in “FinalExamples/Example SyncStop” on the CDROM.

3.2 CSP/CT based control software for Tripod

The agent software that is used as a basis for the new CSP/CT version was created by M. Eglence for his MSc project (Eglence, 2003). The code for the controllers and hardware drivers are reused in this project.

An overview of the top level of the program is shown in Figure 3-4. The square at the top contains the linkdrivers that communicate with the hardware (or with the 20-sim simulation model). They are read by the processes below them. The Commutator process interfaces again with the hardware or simulation model. Lastly the hardware or simulation model closes the loop.

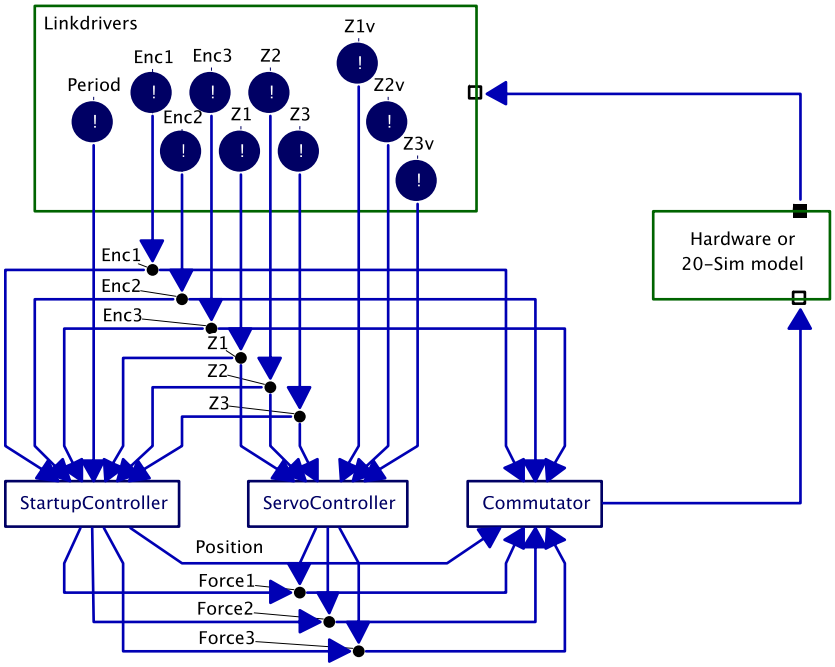

Figure 3-4 Top level of Tripod control software

The program can be compiled in two versions, one that is targeted for controlling the real hardware setup and a version that uses generated C-code from a 20-sim model. The first version only works for the DOS operating system, since this is the only operating system for which the CSP/CT library provides strict timing. The simulation version can be compiled for the DOS and for the Linux operating system.

First the linkdrivers will be treated. Then the startup phase of the Tripod, controlled by StartupController. The third subsection will treat the servo control phase, controlled by the ServoController process.

3.2.1 Linkdrivers

Linkdrivers can provide an interface between a CSP/CT program and the environment. In case of the Tripod program there are four linkdrivers that provide communication with the outside world. They are divided by functionality as is shown in Table 3-1.

Table 3-1 Linkdrivers in the Tripod control software

Linkdriver	Functionality
DigInput	Provide the status of the start and stop button plus the 'vrijgave' (enabled) status of the robot. Secondly it provides the status of the endswitches and 'at lower/upper half' / 'at upper half' information for all motors.
DigOutput	Provides the ability to enable / disable the motor amplifiers
Encoder	Provides three raw encoder values, three absolute positions and three first order derivatives of the position (speed).
Commutator/DAOutput	Reads three forces and applies them to the three axes.

For the Tripod program two versions of the linkdrivers were created: one to interface with the real hardware and one to interface with the 20-sim model of the hardware – by just replacing the linkdrivers the program can use a simulation model of the robot.

Hardware version

The hardware versions of the linkdrivers serve as an interface to the Tripod hardware. The Tripod hardware is connected to the controlling computer via three interface cards: one encoder card that is connected to the measurement rulers, one digital I/O card used for buttons and a D/A converter to steer the amplifiers (see Appendix B for more information about the computer hardware).

Simulation version

The simulation versions of the linkdrivers serve as an interface to a 20-sim model (Figure 3-5). This 20-sim model models the mechanical hardware. It takes three forces as inputs and provides the three motor positions as outputs. The same model was also used by Eglence for his MSc project (Eglence, 2003). It is a simple model, for example the influence of gravity is omitted. The accuracy is also not good enough to design a controller for example. The model is however accurate enough to check if the new Tripod program behaves as expected. Important to check are the start and stop time of all processes with respect to each other and to check if the program doesn't behave erratically. Because of the inaccuracy the model is unstable with the controller parameters that are used for the hardware version. All K parameters of the PID controllers in the program are adjusted when compiling the simulation version.

From this 20-sim model C-code was generated. This code is then used by the simulation versions of the linkdrivers. The Encoder linkdriver takes the position from the model and the Commutator writes the forces to the model and calls the XXCalculateSubmodel() function to update the model.
The 20-sim model and experiment are included with the Tripod software on the CDROM ("CSPTrip/zzzman.(em|exp)").

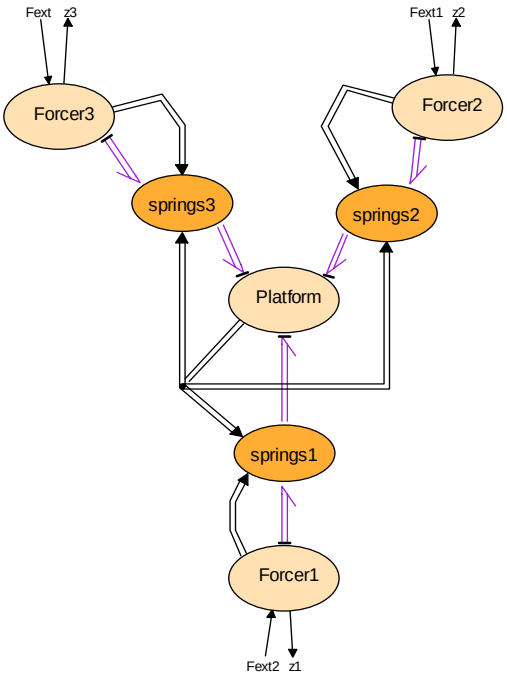

Figure 3-5 20-sim model of the mechanical hardware of the Tripod setup

3.2.2 Startup phase

The StartupController takes care of preparing the Tripod for servo control. This mainly consists of aligning the linear motors and calibrating the encoders. For working principles and the need for aligning the linear motors, see Appendix B. During this process the platform will be raised and then lowered to a position that is a couple of centimetres above the lowest possible position. This process consists of four distinct phases, handled by four different processes: ChkStart, Align, Calibrate, GoHomePID (Figure 3-6).

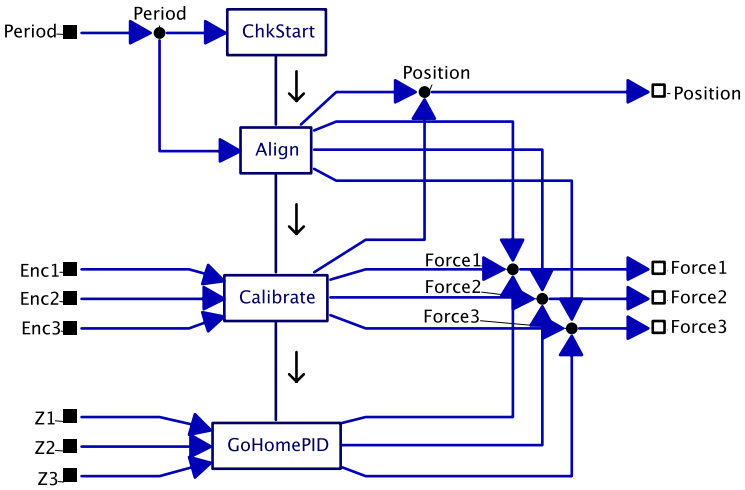

Figure 3-6 StartupController process

ChkStart

This process checks if the Tripod system is enabled. This is done by pressing the ‘vrijgave’ button on the control panel. Pressing the emergency stop button, or a reboot of the control pc, will reset this state. It will wait until the system is enabled. When the system is enabled the process checks if the translators are at the lower half and if the end switches are activated. The motors have to be at the

lower half and the switches have to be active, since all the motors should be at the lower end of their respective strokes. If everything checks out, the next process will run.

Align
This process aligns all motors one pitch above their lower end position. One pitch is the distance between two coils in the linear motor (see Appendix A). This is performed in three sub phases. First the force on all motors will be increased, keeping the position sent to the Commutator zero. This will make each motor move to a detent position, unless it is at the end of its stroke. Hereafter the position sent to the Commutator is increased while holding the force constant. This will move the motors up one pitch. This ensures the motors are no longer at the end of their strokes. The third phase will increase the force again to prepare for the next phase. Now the encoder counters are reset to the middle of their range.

Calibrate
This process will locate the index pulse on the encoders. It first sets the hardware linkdriver Encoder in calibration mode. Reading from the encoder now results in the contents of the index latch of the counters. Then it will increase the position sent to the Commutator, the force will remain the same value set by the Align process. All motors should now move up. When the position has been increased enough to make sure all motors have moved past the index pulse, a check is done to see if this is indeed the case. Then Encoder linkdriver is put in normal mode again and the final phase starts.

GoHomePID
This is the first closed loop process. It will steer all motors back from the final calibration position to a couple of centimetres above the lower end of stroke. When all motors have reached this position, the Encoder linkdriver is told to make the current position the absolute position 0. The Tripod hardware is now ready for control by the ServoController process.

3.2.3 Servo control phase
Servo control is handled by the ServoController process. During servo control the control software can be in several modes. Each of these modes has its own control function within the program, for example one mode holds the platform at the zero position and another mode controls the robot along a given path. A mode can be activated by the ModeSwitcher process and only one mode can be active at a given time. Once a mode has been activated, it can only be deactivated by itself, but the ModeSwitcher has the possibility to negotiate deactivation of certain modes via channels. The internals of the ServoController process are shown in Figure 3-7. The channels to/from the ModeSwitcher process will be treated in the paragraph on mode transitions at the end of this subsection.

Each mode when activated reads the current position of the robot via the position channels Z1 – Z3. An exception to this is the braking phase in the GoPositionAll mode, during braking the speed channels Z1v –Z3v are read. Then the mode does the necessary calculations (the control law) and writes the resulting values into the output channels Force1 – Force3. These forces are then read by the GravityCompensator process. This process adds a constant force to each output to compensate for the gravity. However the distribution of the weight of the platform on the motors depends on the position of the platform. Eglence already suggested in (Eglence, 2003) to take the position into account, however the calculations for this are not trivial. Finally the compensated outputs are written to the Commutator process.

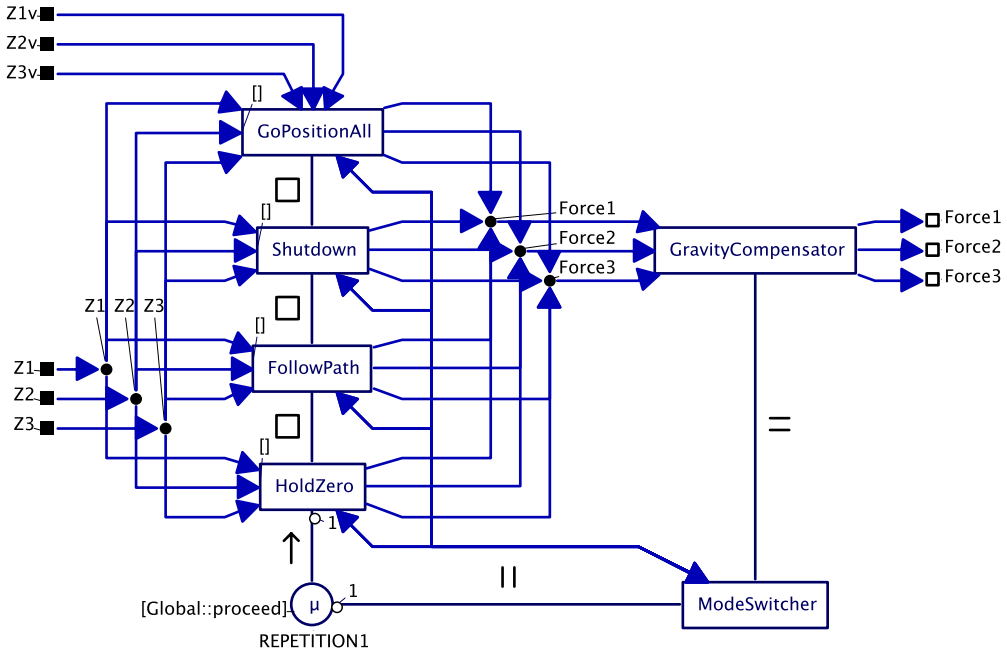

Figure 3-7 ServoController process

GoPositionAll mode

This mode can always be activated; it has no constraints on the current position or speed of the robot. For each motor the speed is first reduced to zero and then it is returned to the zero position. When all motors have reached the zero position the mode is deactivated again. The GoPositionAll process that implements this mode contains a GoPosition process for each motor. Each GoPosition process consists of three processes: Brake, GoPos and Hold. Because the initial position of each motor can be different, the motors will reach the zero position at different moments in time. To synchronize each GoPosition the SyncStop pattern is used (section 3.1). The gCSP diagram for GoPositionAll is shown in Figure 3-8. This diagram makes use of the barrier to represent the synchronization. As soon as the barrier becomes ready (all three sequences have arrived at the alternative HoldN/Syncer) the repetition of all three alternatives will end. The implementation uses the SyncStop pattern of section 3.1.

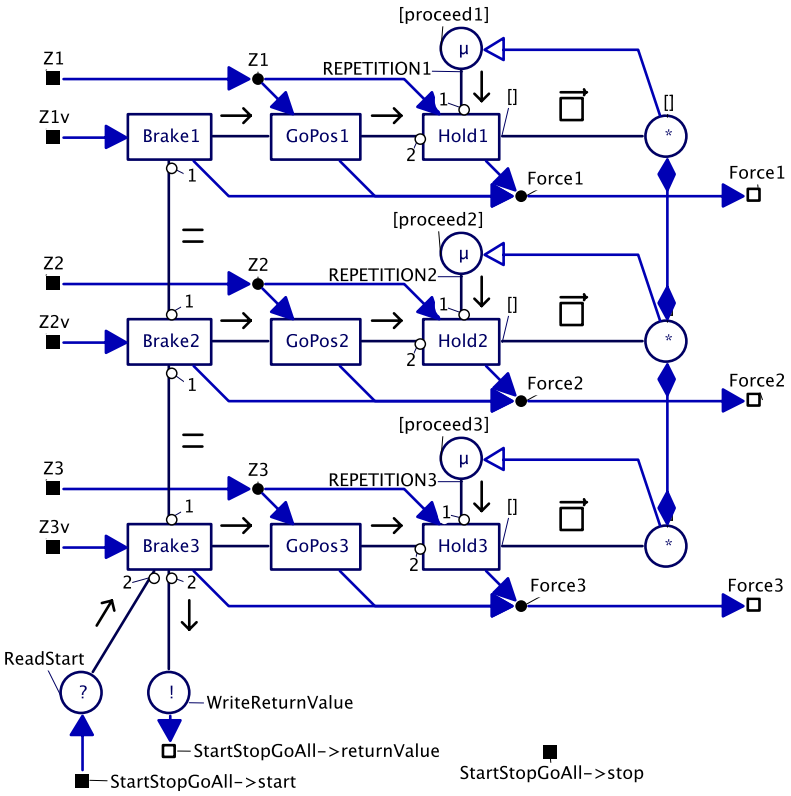

Figure 3-8 GoPositionAll process

Shutdown mode

This mode may only be activated when the robot is at position zero with speed zero. The purpose of this process is to lower the motors to just above the lower end stops before turning off the amplifiers. When this mode is finished the program should end (or restart). The process consists of only three PID controllers in a parallel relation; therefore no diagram is shown here.

FollowPath mode

This mode may be activated at any position with current speed zero. The purpose of the mode is to control the robot along a path that is read from a file. It starts by moving the robot to the first position in the file. When the start position is reached, it will control the robot along the path. This path has to be read when the program starts because reading a file cannot be done under DOS in real-time when a controller is running. The mode takes a set point from the file on every period. The mode finishes when the path is complete. The ModeSwitcher can ask this process to stop earlier by writing to the stop channel on the bottom right in Figure 3-9, if this happens the robot isn't necessarily at the zero position with speed zero, so the GoPositionAll mode should be activated.

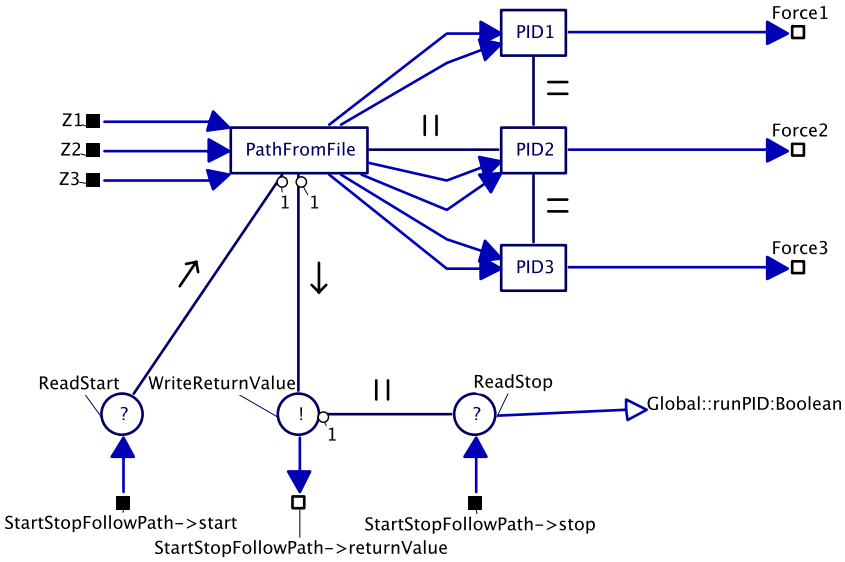

Figure 3-9 FollowPath process

HoldZero mode

The sole purpose of this mode is to hold the platform at zero when no other mode controls the robot. The position and speed should be zero before entering this mode. The mode becomes active after receiving a value through the start channel (StartStopHoldZero->start on the top left in Figure 3-10) from the ModeSwitcher. It sets some initial values in the Init code block and starts an alternative in repetition. In the alternative are three controllers (one for each motor) and the reader of the stop channel. The repetition continues until a value is received on this channel. When this has happened, the mode will end by writing a return value into the returnValue channel on the bottom right of Figure 3-10.

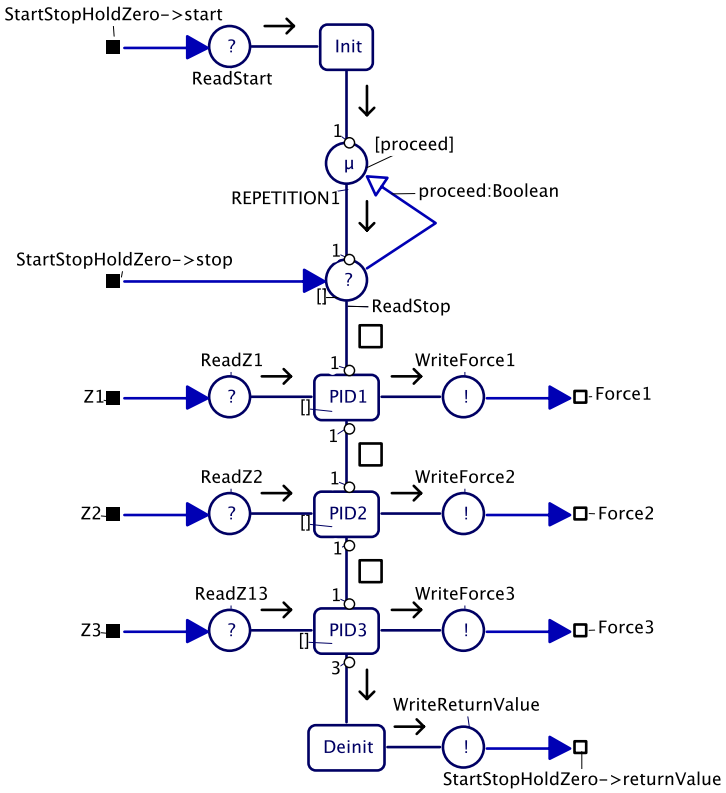

Figure 3-10 HoldZero process

ModeSwitcher process

As stated earlier the ModeSwitcher process determines which mode becomes active. Which mode is chosen depends on external inputs (like buttons and keyboard) and internal feedback (return value from the mode that ended). All possible mode transitions are indicated in Figure 3-11.

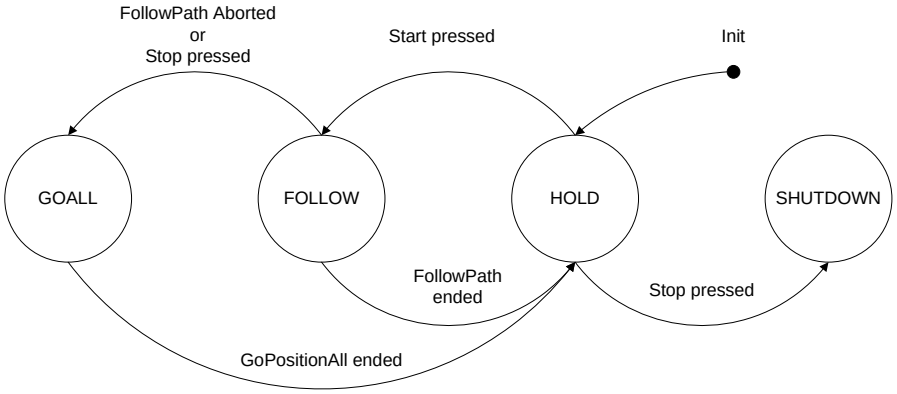

Figure 3-11 Mode transitions in the servo control phase

The ModeSwitcher is connected to each mode process with a so called StartStopChannel. This 'channel' actually is a combination of three channels, a start channel, a stop channel and a returnValue channel. The start and stop channel are from the ModeSwitcher to the mode and the returnValue channel is from the mode process back to the ModeSwitcher. To start a mode the ModeSwitcher writes to the start channel belonging to that mode. Some modes can be asked to stop; this is done by writing to the stop channel. A mode writes a return value back to the ModeSwitcher through the returnValue channel when it finishes. This return value can influence what mode is to be activated next. For example when the Follow mode ends prematurely due to an exception, or when it is asked to stop before it is done, the return value will indicate this. The ModeSwitcher now knows that the GoAll mode should be activated.

3.3 Possibilities for distributed control on Tripod

A possible distributed control system can be done as follows with four nodes:

- Hardware driving node with safety checks.
- Control node(s), implementing the actual controllers.
- Setpoint/Path generation node.
- (G)UI node.

This setup is depicted in Figure 3-12.

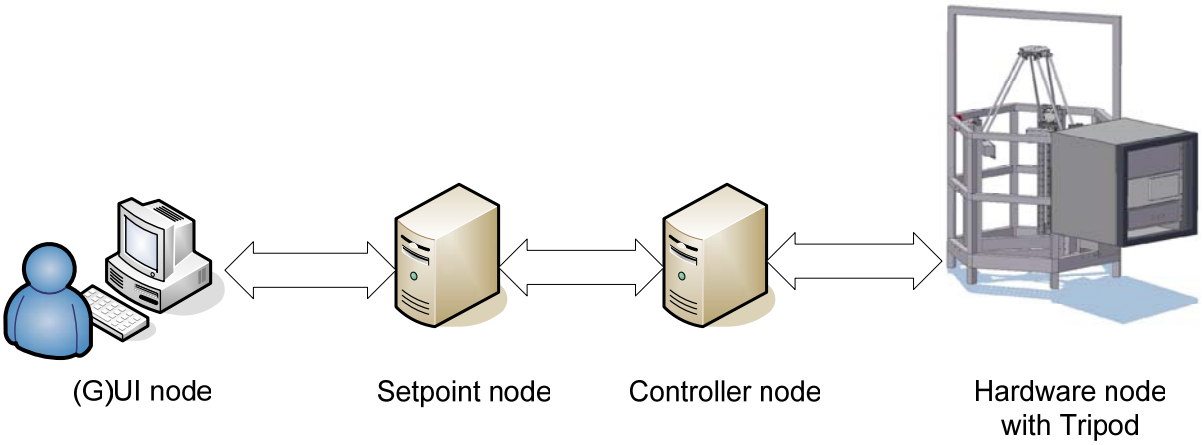

Figure 3-12 Possible distributed setup for Tripod

The hardware driver node provides position/speed feedback outputs to and accepts force as input from the controller nodes. It can also do safety checks, like checking endswitches, if the robot is in its workable region, checking if the current speed/acceleration doesn't cross the safe limit. It is possible for this node to intervene by switching to an internal safety controller if those limits are crossed. This allows also for safe testing of new controllers in the controller node. The hardware driver node should also do the startup phase and hold the platform until a control node is available to take over control.

The control node accepts setpoints from a setpoint node and feedback from the hardware driver node. It contains the control law to calculate the force and provides it back to the hardware driver node.

The setpoint generation node provides setpoints to the controller nodes. The setpoints may be loaded from a file, or generated from data received from the (G)UI node. This last feature makes it possible to manually control the robot with for example joysticks or keyboard.

The (G)UI node provides a user interface. The UI can be used to start/stop the system. It can provide a facility to store data that other nodes have logged. Last it can provide data to the setpoint generator if the robot is controlled manually with for example joysticks or keyboard.

4 Design patterns for safety and reliability

This chapter presents three design patterns that can help to increase the safety and reliability of CSP/CT software. First the Logging and Monitoring pattern is discussed; it provides a means to log data for on- or offline analysis and via monitoring the ability to intervene if wrong data is detected. The second pattern is N-Way programming. This pattern achieves extra reliability through redundancy by making it possible to run multiple versions of a process in parallel and comparing their outputs. The last pattern that is discussed is the Watchdog pattern. This pattern is mainly intended to increase the safety of a system by intervening if (a part of) a program locks up.

All patterns are designed to be used in a real-time program; all actions take a fixed or a bounded amount of time. They all have as requirement that little changes to the processes in a CSP/CT program are required to make use of them.

4.1 Logging and Monitoring

Logging is useful for a number of purposes, for example debugging during development, logs can be used for maintenance (to prevent or solve problems), or in the case of a more serious event it can help in legal issues like liability. In general: a logging component stores data for later analysis, but the stored data can also be used for on-line presentation and adaptation.

While logging is passive, in the way that it does not react to anything it receives and only stores data, monitoring is an active pattern. A monitoring component can react to data it receives, for example if it detects that the data is wrong. Action can then be taken to correct the problem.

In this section a framework is presented that provides logging and monitoring within the CSP/CT architecture. Logging and monitoring (L/M for short) is divided into two classes of components that are interchangeable. This means in a place where a logging component is used no changes are necessary to use a monitoring.

A requirement for use of these L/M components is that little changes are needed in a CSP program to make use of it. The internals of processes should not have to be changed at all. By examining data that flows through channels it is possible to observe the behaviour of a CSP program. To observe this data in channels a derived class of Channel named ProbeChannel was made. When a normal Channel is replaced by a ProbeChannel, a copy of all data that is written into it will be send to an L/M component.

4.1.1 ProbeChannels

Probe channels are the same as normal channels with the added functionality of logging. All values written into the probe channel are send to a L/M component before the normal implementation of the `channel->write()` is called. By default this L/M component is the global logging component `gLog` (to be treated in subsection 4.1.3). Because it is written to the component first, a monitoring component can check if the value is ok, if not it can intervene before the wrong value is really written into the channel. This intervention will be described in the subsection about monitoring components (4.1.4).

Probe channels take one extra argument in their constructor, a string that contains their name. A string is an array of characters. This name is registered with the global logging component upon construction of the probe channel. All values are written to the L/M component together with the ID received when registering this name.

A normal channel can be replaced with a probe channel by replacing the constructor in the new statement where the channel is created. An example for a channel of type int named 'somechannel':

```
Channel<int> * somechannel = new Channel<int>;
```

If this code is modified like this

```
Channel<int> * somechannel = new ProbeChannel<int>("MyNameInTheLog");
```

then the channel will be created as a probe channel with name 'MyNameInTheLog'. The L/M component of the probe channel can be changed by calling the following member function on the probe channel: `ProbeChannel::setMonitor(myNewLMComponent)`.

Probe channels can be poisoned by the L/M component after a value is written to this component. This functionality is used by monitoring components which will be discussed later in this chapter. A current limitation of the probe channels is that values are all converted to double before they are sent to the L/M component. This limitation originates from the logging component that can only store doubles and symbols (for strings) in its buffer. It is possible with some coding effort to extend the functionality to include also all other basic types.

It is not required to use probe channels for logging. In any part of the code it is possible to write something directly to an L/M compone nt, as long as it is registered first. This has the advantage that more sophisticated functions of the components become available. For example the default logging component can also cope with predefined strings. This feature can be used to record certain events, for example a part in a program that measures the system load can write a message to the logging component when the system is overloaded. Or a message from a watchdog component that times out. This functionality should not be used to examine the dataflow of a program, for this only probe channels should be used. More information on this direct use of a logging component can be found in subsection 4.1.4.

4.1.2 Framework for Logging/Monitoring components

Before a client can use an L/M component, it first has to register itself once with the global logging component `gLog`. This is done by calling the function `gLog->getID("MyNameInTheLog")`. It then receives back an ID. It can use this ID to write messages to any L/M component. Names do not have to be unique to the program. Each registration will return a unique ID.

As pointed out earlier, L/M components have in common that data can be written to them. In the framework this common functionality is described in the base class of L/M: MonitorLoggerBase. All L/M components derive from this class; see also the diagram in Figure 4-1. In this diagram class RTLog is a default implementation of a LoggingComponent.

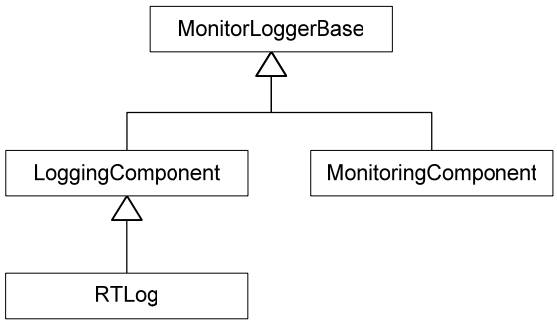

Figure 4-1 Inheritance relations of L/M components

This base class defines the three write functions for writing data to a component, one for writing values, one for writing symbols (text strings, this will be explained in subsection 4.1.3), and one to write a LogMessage. The last function is used by an L/M component to write data that it has received and stored earlier to another L/M component. The next part of this subsection will discuss how data is stored.

Storing data

When data is received by an L/M component it is stored with some extra information: when the data was written to the component, by whom it was written and what type of data was written. All this extra information is stored together with the data itself inside a structure called LogMessage.

```
struct LogMessage {
      timestamp_t time;       ///< When it was sent
      LogClient* logID;       ///< Whom it is from
      MonLogMsgIDs msgID;     ///< What type of data is stored
      LogMessageData data;    ///< The actual data
};
```

To keep the system to store multiple LogMessages simple, the LogMessage structure has a fixed size. This makes it possible to store LogMessages in an array and when this array is full to start overwriting messages from the start of the array. This is also called a circular buffer. The LogMessageData type is a union that can contain all possible data types. In the current implementation this can be a double or a symbol number for a string. The size of the union is the size of the biggest object it can contain; in this case it is 8 bytes for a double. It is possible to extend this union with support for integers and other types. It then becomes possible to write other types than double to an L/M component if the proper write functions are also implemented.

4.1.3 Logging components

A logging component logs all data it receives from its clients. Clients can be probe channels but also code that writes directly to the component. The data is stored in a circular buffer of a user specified size. Since the buffer is circular, old values will be overwritten when the buffer is full. In this way only the last 'size of buffer' values are stored. Storing data in this buffer is real-time and can be done from anywhere in the code. Real-time means in this case that it takes a fixed amount of time, the write call to the component can never block. All other actions with respect to the logging component, such as the extra features that will be discussed later, are not real-time. The default implementation that is discussed here is called `RTLog`, a real-time logging component.

Besides support for logging numerical values, the `RTLog` logging component has support for logging predefined text strings. These strings have to be made available to the logging component before use. All strings are stored under a numeric ID in a symbol table. Storing the strings in the symbol table is not real-time. The logging of strings is done by writing the clients ID together with the symbol of the string (ID of the string) to the log via `writeSymbol(logID, symbol)`. When the log is analyzed later on, the strings can be found again via the stored symbol. The logging of strings is real-time.

Other (non real-time) features of RTLog include resizing the log without loosing data (as long as the new log is bigger, otherwise only the newest values that fit will remain), printing the log on the screen and writing the log to a file. These log files are stored in a CSV (comma separated values) format. This format is readable by spreadsheet programs and after conversion also by 20-sim (see section 5.1). An example log file with a header, two logged values and one logged symbol follows:

```
"timestamp","name","value"
1342,"PID",0.01
1562,"PID","Hello world"
1774,"PID",235.81
```

A default instance of RTLog is globally available in the library via the `gLog` pointer. This log is also the default target for new probe channels. A couple of examples are now given how to use the default component without probe channels.

An example with only logs values:

```
// Register with gLog
logID = gLog->getID("MyName");
// Store symbols

// Write some value
gLog->write(logID, 1.23);
// Some other code that takes some time ...
// Write again some value
gLog->write(logID, 3.21);
```

An example with strings stored as symbols:

```
// Register with gLog
logID = gLog->getID("MyName");
// Store symbols
logID->symbolTable[0] = "Message 0, hello";
logID->symbolTable[1] = "Message 1, world!!";
// Write symbol of first string to the log
gLog->writeSymbol(logID, 0);
```

To make it easier to use these symbol numbers, one can for example store them in a enumeration type. The code then becomes like this:

```
// Register with gLog
logID = gLog->getID("MyName");
// Store symbols
enum myLogMessages { MSG_HELLO, MSG_WORLD };
logID->symbolTable[MSG_HELLO] = "Message 0, hello";
logID->symbolTable[MSG_WORLD] = "Message 1, world!!";
// Write symbol of first string to the log
gLog->writeSymbol(logID, MSG_HELLO);
```

Private log

The global log will automatically overwrite values when the buffer is full. This is not always desired, for example if there are some critical messages that need to be logged and that may not be overwritten. Therefore it is possible to create private instances of RTLog with their own buffer. Private in this context means that this log can only be used by code that knows of the new instance. A private log can be created with `myLog = new RTLog(size)`. To make a probe channel use this new log one can use `ProbeChannel::setMonitor(myLog)`. These private log's can be printed on the screen or to a file in the same way as the global log (e.g. `writeLogToScreen()` and `writeLogToFile("filename")`).

4.1.4 Monitoring components

Monitoring extends logging with checking the values written and possibly taking action if something is not in order by commenting in the log or by poisoning the client with an exception. The algorithm that does the checking is often specific for the context it is used in so there is no general implementation for it. It is possible to use multiple inputs (from multiple clients) in the algorithm.

The easiest way to use monitoring in a program is by using probe channels. This requires no changes inside the processes that will be monitored. The monitoring component can use the `setMonitor(this)` operation on a probe channel to make the probe channel write all values to it in stead of the probe channels default target (the global log). The monitoring component can decide during each write call if the data is ok by returning 0, or if the data is not ok returning an exception that will poison the probe channel. The processes connected to the probe channel will then throw the exception that the monitoring component returned to the probe channel. By throwing the exception they abort and an exception handler is started. This exception handler can then bring the system in a safe state to either stop or try again depending on the situation.

4.1.5 Function reference

Table 4-1 Functions related to ProbeChannel

Function	Functionality
`ProbeChannel(const char *name)`	Creates a new ProbeChannel with the supplied name.
`void setMonitor( MonitorLoggerBase * logTarget)`	Changes the target L/M component of the ProbeChannel to logTarget.

Table 4-2 Functions common to both logging and monitoring components

Function	Functionality
`ExceptionSet * write( LogClient* logID, double logData)`	Log one value (double) in the log under the given ID.
`void writesymbol( LogClient* logID, int symbol)`	Log a predefined string under the given ID. The string was stored under the given symbol in the symbol table.
`void write(const LogMessage *message)`	Log an earlier logged message. Using this function keeps the stored timestamp of the message. This is useful if an L/M component needs to forward an already logged message.

Table 4-3 Functions related to RTLog component

Function	Functionality
`RTLog(unsigned int size)`	Creates a new log with given number of entries.
`unsigned int getLogSize()`	Returns the maximum number of entries of the log.
`bool resize(unsigned int size)`	Resizes the log to the new size while maintaining the stored date. If size is smaller then current size only the last size entries will be kept.
`void writeLogToScreen()`	Prints the log to the screen.
`void writeLogToFile( const char *filename)`	Stores the log in a file with the supplied name.
`LogClient * getID( const std::string &clientName)`	Register with the logging facility with the supplied name. Returns a log ID to use for logging.

4.1.6 Example program using Logging/Monitoring pattern

To clarify the L/M pattern an example will now be discussed. The example program consists of a multiplier called Multiplier that multiplies values it receives from its two inputs and writes the result in an output channel. The channels in and out of Multiplier are probe channels that are connected to a monitoring component designed to detect sign errors in the output with respect to the inputs. A defect in Multiplier is introduced so that it will produce the wrong sign when the absolute result is bigger then 10. A gCSP diagram of the program is shown in Figure 4-2. This diagram contains the Multiplier process in an exception construct. The monitoring component is not shown here, but monitors all three channels.

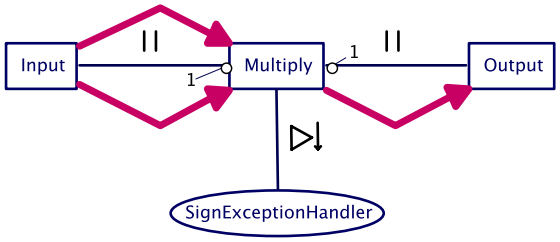

Figure 4-2 Top level of monitored multiplier

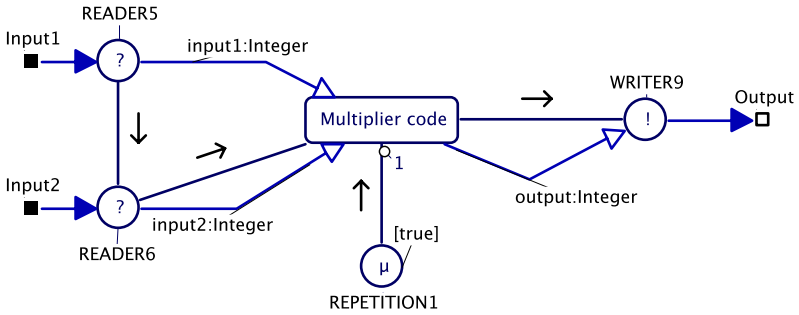

Figure 4-3 Internals of process Multiply

The contents of the multiplier process are shown in Figure 4-3. The steps this multiplier should perform are:

1) Read two input values, one from each input channel;
2) Multiply these two values;
3) Write the result into the output channel;
4) Go to step 1.

When a value is written into an input channel it is first written to the Monitor by the probe channel. Since the monitor cannot make a decision only on input values, the return value of writing to the Monitor will be 0. The probe channel then knows everything is ok, and then writes the value to the multiplier process. When the multiplier has two input values it performs the multiplication and write the result in the output channel. This probe channel also first writes this value to the monitor. The monitor can now make a decision based on the three values it has received, if the sign is correct it returns again 0 to the probe channel and the result is written into the real output channel. But when the sign is incorrect the monitor will return an ExceptionSet with a multiplier sign exception. The probe channel then poison itself with the ExceptionSet received from the monitor and the multiplier process will abort and jump to its exception handler SignExceptionHandler.

The monitor stored all values it received in LogMessage structures with timestamps etc. In case the monitor detects an error, the values resulting in this error (the two input and the output values) are written to the global log together with an error message “Sign error detected”. This logging is of course optional.

To show what information is stored in the log for this example, the contents of the log from a run are given below. During the run of the example a number of multiplications were done. The multiplication 3 x -4 should result in -12, but due to the defect the multiplier gives 12 as result. In this version of the program all values were written to the global log by the monitor, also when no error was detected.

```
"Time","Name","Value"
2691903980602,"ExampleMonitoredMultiplier","Starting Example Monitored
Multiplier"
2691918772814,"Input 1",-2.000000
2691918772843,"Input 2",-2.000000
2691918772845,"Output",4.000000
2691921672314,"Input 1",-3.000000
```

```
2691921672320,"Input 2",2.000000
2691921672322,"Output",-6.000000
2691923564635,"Input 1",1.000000
2691923564643,"Input 2",2.000000
2691923564644,"Output",2.000000
2691925913708,"Input 1",1.000000
2691925913713,"Input 2",8.000000
2691925913728,"Output",8.000000
2691930319961,"Input 1",-4.000000
2691930319968,"Input 2",2.000000
2691930319969,"Output",-8.000000
2691932003627,"Input 1",1.000000
2691932003633,"Input 2",1.000000
2691932003635,"Output",1.000000
2691933740091,"MultiplierMonitor","Sign error detected"
2691933740038,"Input 1",3.000000
2691933740044,"Input 2",-4.000000
2691933740046,"Output",12.000000
2691933740283,"ExampleMonitoredMultiplier","Example Monitored Multiplier
Ended"
```

The complete code for this model can be found in “FinalExamples/Example Monitored Multiplier” on the CDROM.

4.2 N-Way programming

N-Way programming (NWP), also called N-version programming, was first introduced in (Avižienis, 1977) as follows:

> “N-version programming is defined as the independent generation of N >= 2 functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution, during which comparison vectors (“c-vectors”) are generated by the programs at certain points. The program state variables that are to be included in each c-vector and the cross-check points (“cc-points”) at which the c-vectors are to be generated are specified along with the initial specification.

The concept of NWP consists of three elements. The first element is creating an initial specification for NWP which states that the programs are developed independent from each other and ensures that they are functionally equivalent. The second element is the N-Way software which can execute concurrently and has cross-check points at which comparison vectors can be compared. The last element is an execution environment that supports execution of N-Way software. The environment also provides mechanisms to do the checking at the given cross-check points.

The whole idea behind NWP is to provide a means to achieve fault tolerance in software, especially against development faults in software. Since all versions of an N-Way software program are developed independent from each other, it is less likely that the same programming errors are made. With each extra version of a program this chance is even further reduced.

4.2.1 How to implement N-Way programming in CSP/CT

To support N-Way programmed versions of processes in CSP/CT a small framework is needed to run these versions in parallel and to do the cross-check of their outputs. A framework that satisfies these requirements is given in the gCSP diagram in Figure 4-4. A process in a CT program can be replaced with this framework. Inside the framework are N versions of the process with the same functionality as the original, but developed independently from each other. The outputs of the different versions are fed to the process DecisionMaker that does the cross-checking. The algorithm for this cross-checking comes from the initial specification of a program. Finally there is a Delta process that makes sure every version gets a copy of the input data.

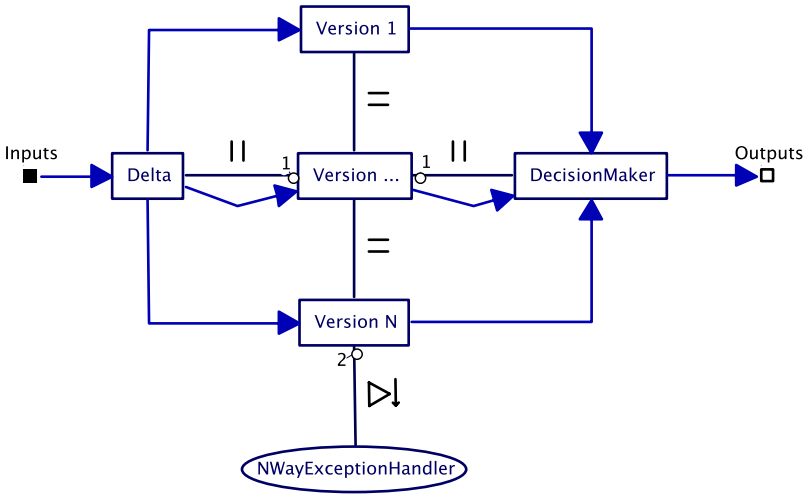

Figure 4-4 Implementation of N-Way programming in CSP/CT

Depending on how many versions there are, a few options are available to react on errors. When only two versions are available it is possible to detect an error when one of the versions is faulty (or both, if the errors made are not identical). It's not possible to detect which version was faulty, only that there is a difference between the two. With the help of exceptions in CSP this error can be thrown upwards in the program to start a recovery mechanism.

When there are more versions it is possible to do error correction when a fault occurs by using majority vote. Depending on the application it might be better not to do corrections but always throw an exception in case an error is detected. The more versions there are, the more reliable the N-Way system becomes.

4.2.2 Example program using the N-Way pattern

This example will demonstrate a three-way programmed process in CSP/CT. The process that will be three-way programmed is the same multiplier as in subsection 4.1.6, with the difference that the channels are now normal channels and no ProbeChannels. The multiplier process is implemented three times with different code in the multiplier code block. The cross-check algorithm is to compare the results of all versions, and in this case these have to be equal.

Figure 4-5 shows three multipliers combined in an N-Way framework. Process Multiply (at the bottom) has a correct implementation. Process Multiplyf (top) has limiters on the inputs limiting input values to +/-10. Process Multiplyf2 limits input values to +/- 20. For example the outputs from top to bottom with the multiplication 5 x 12 are respectively 50, 60 and 60. The top process Multiplyf gives the wrong answer. In case of 21 x 2 the answers are 20, 40, 42, now Multiplyf and Multiplyf2 both give wrong answers. Last there are process Input, which distributes values from input1 and input2 to all the multipliers, and process Decision, which implements the cross-checking of the outputs.

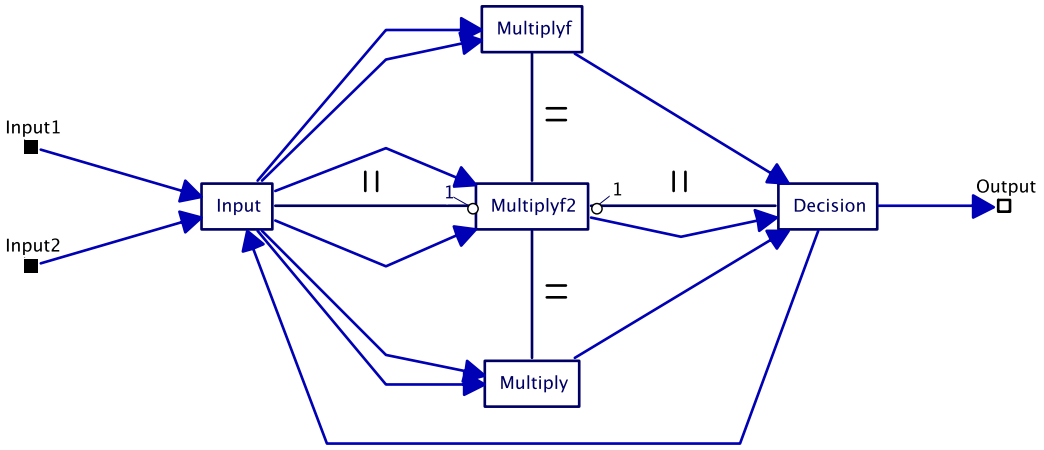

Figure 4-5 Three versions of the multiplier in the N-Way framework

The internals of the Decision process are shown in Figure 4-6. The Decision process takes the three results of the multipliers and compares them in the block DecisionMaker. There are now three possibilities:

1) All outputs are the same, then this value is also written to the Output channel of Decision;
2) One value deviates (for example in the case of 12 * 5), but since there is a majority agreement, the output will follow the majority. It is possible in this case to write a message of the failure to a log as described in subsection 4.1.3;
3) All three values are different; in this case the correct answer is unknown. It is possible to throw an exception to start an exception handler to put the system in a safe state. More on exception handling in the N-Way pattern in the next subsection.

The channel back from Decision to Input is used in this example to be able to stop the framework in a nice way and in the next subsection it is necessary to be able to do correct exception handling.

The complete code for this model can be found in “FinalExamples/Example NWay multiplier” on the CDROM.

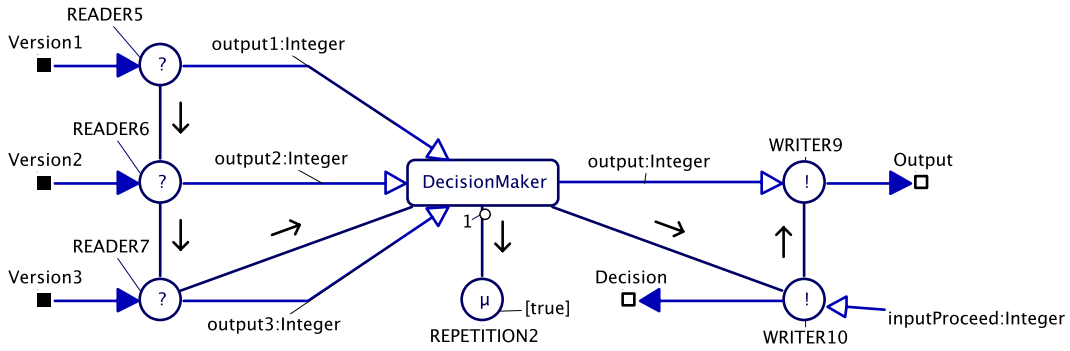

Figure 4-6 Internals of process Decision

4.2.3 N-Way exception handling

In the case the Decision process cannot decide what the correct value is, it is possible to throw an exception so an exception handler can take action. In Figure 4-7 the three-way multiplier (NWayMultiplier process) is shown together with this exception handler. But before this exception handler can be started, all processes in the framework need to stop. This termination of all processes is initiated by the exception handler of Decision.

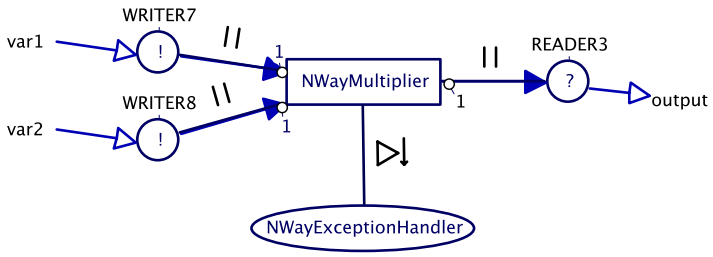

Figure 4-7 Top level of N-Way programmed multiplier with exception handling

When Decision throws the exception, the Decision process aborts and its exception handler DecisionExceptionHandler is started (see Figure 4-8). This exception handler poisons the channel that goes back to the Input process. Since process Input is blocking on this channel (before starting the next cycle) it immediately throws this exception. The exception handler InputExceptionHandler now poisons all its output channels to the multiplier processes. Since these processes are blocking on these channels they also throw an exception. The MultiplierExceptionHandler handles these exceptions and now the entire framework is aborted.

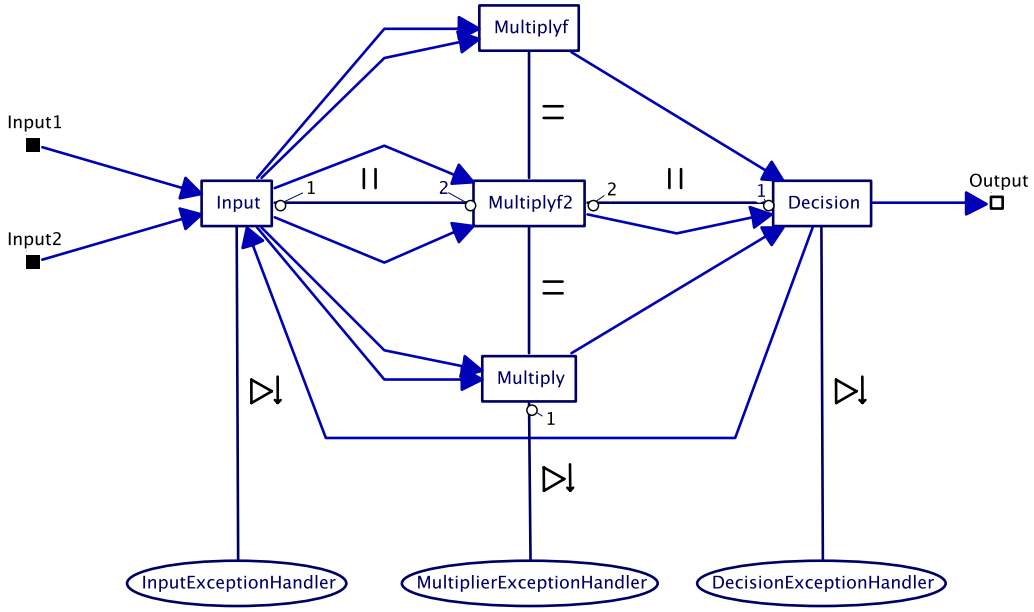

Figure 4-8 The N-Way framework with exception handlers

Together these are a lot of exceptions. The outside handler, NWayExceptionHandler in Figure 4-7, only needs one exception. Therefore both InputExceptionHandler and MultiplierExceptionHandler mark the exceptions they catch as handled (the exceptions from poisoned channels). This leaves the exception thrown by Decision. DecisionExceptionHandler does not mark this exception as handled, and it is rethrown higher up in the hierarchy of processes to NWayExceptionHandler. In case of this multiplier this exception handler will stop the program, but for more complex systems some form of graceful degradation can be started to put the system in a safe state.

The complete code for the model with exceptions can be found in “FinalExamples/Example NWay multiplier with exceptions” on the CDROM.

4.3 Watchdogs

This section treats watchdogs. Watchdogs can help as a last resort when a program is not running as it is supposed to. This is especially true when a program is livelocked, deadlocked or when the system on which the program runs has too much work to do (overload). These situations can happen for example when a process goes into an infinite loop, or when a channel between two processes is (physically) separated with as a result that these processes will block on that channel. Normally if a process does not notify the watchdog anymore the watchdog will trigger a reset of the system. However in a controlled system this is not always desired behaviour, it would be better for example to go to a safe state first and then reset the system. The pattern presented here will do just that. First classic watchdogs will be treated. These watchdogs need to be triggered after they are started to prevent timeout. Secondly a more general type of watchdog is treated. When it is started it periodically measures the load of a system and can intervene when it becomes too high.

4.3.1 How to implement watchdogs in CSP/CT

Every CSP process can create its own watchdog. It can set the timeout and start it. From that point on it should notify the watchdog before the timeout elapses to restart this timeout. We'll call this notifying 'hitting' the watchdog. It is also necessary to be able to stop a watchdog, for example if a process wants to terminate and there is no need for the watchdog anymore.

The timeouts have to be based on interrupts from a hardware timer. This way it is ensured that the watchdogs will always run independent of the CT scheduling. If they would be implemented as CSP processes for example, it would not be possible to catch an infinite loop inside some other process. An infinite loop inside a CSP process means that there won't be any context switches, so the watchdog process would never run.

In case a watchdog timeout runs out, the CSP/CT program is pre-empted by the timer to run the watchdog code. Recovery from a watchdog timeout is hard, because it is not possible to determine the process that is at the base of the failure. It is very well possible that when some process locks up in an infinite loop, the watchdog of another process triggers first. What can be done however is starting a recovery process and aborting the normal processes of the program. This recovery process can then put the system in a safe state; however it does not know the state of the system when it is started. It may be necessary to reinitialize some hardware to make sure it is in a sane state. After the system is put into a safe state it is possible to restart the normal processes, either by restarting them (call the `run()` function of the process again), the entire program (reload and restart the executable), or by resetting the whole system.

Another problem arises when there are multiple watchdogs. If there is a problem it is likely that more than one watchdog will timeout. Therefore all watchdogs belonging to the same watchdog construct will be cancelled on the first watchdog timeout. This prevents starting the safety process multiple times for one problem.

A graphical representation of the internals of the watchdog construct is depicted in Figure 4-9. The construct consists of the normal processes and the safety processes. On the left are three watchdogs belonging to this construct. The watchsdogs are shown here for the overview of the watchdog construct; normally they would be inside the normal processes. The SafetyProcesses process is only started after a timeout of one of the watchdogs. When a watchdog timeout has occurred, the channel from WD_timeout will be unlocked so the reader WaitForWDTimeout can run. After this the SafetyProcesses can run.

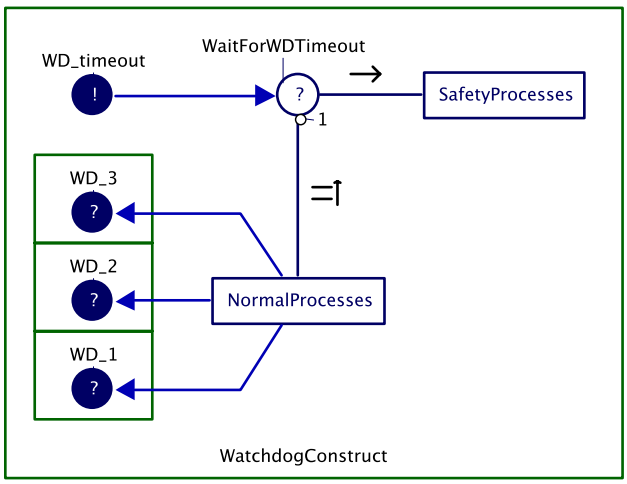

Figure 4-9 Internal workings of a watchdog construct

To make sure that the NormalProcesses is not scheduled anymore the best solution would be to remove them from all scheduling queues and to let it appear to be finished for the PriParallel inside the watchdog construct. In this way after the safety processes are finished the watchdog construct itself will finish. The parent process of the watchdog construct can then decide what to do next, as discussed earlier the possibilities range from restarting the construct to resetting the whole system.

A small code example demonstrates how a watchdog construct can be created.

```
WatchdogConstruct * wdc = new WatchdogConstruct;
Process *safety = new SafetyProcess;
Process *normal = new NormalProcess(wdc);
wdc->create(normal, safety);
wdc->run();
```

On creation of a watchdog one can specify the desired timeout period in hours and microseconds. This division into two parameters is compatible with current the timer interface of the CSP/CT library. The third parameter specifies in which mode the new watchdog is created. Watchdogs can be created in two different modes: periodic (true) or one-shot (false). When the watchdog is periodic it automatically restarts the countdown after it is hit by the process that is guarded. In one-shot mode it stops the watchdog after a hit from the process. The fourth parameter is used to specify the watchdog construct this new watchdog should belong to. That is why a pointer of the watchdog construct is also given to the NormalProcess: to pass it along to the watchdog(s) it creates. An overview of the member functions from the Watchdog class is given in Table 4-4.

Table 4-4 Functions related to watchdogs

Function	Functionality
`Watchdog(unsigned int hours, unsigned int useconds, bool periodic, WatchdogConstruct *wdc)`	Creates a new watchdog with the given timeout. It is possible to specify whether it should be in periodic or one-shot mode. The last argument is to specify the WatchdogConstruct the new watchdog belongs to.
`void setPeriodic()`	Changes the mode to periodic.
`void setOneShot()`	Changes the mode to one-shot.
`void setTimeout(unsigned int hours, unsigned int useconds)`	Changes the timeout. The new timeout is used after the next `hit()` call or when `start()` is called.
`void start()`	Starts the watchdog. The `hit()` function has to be called before the timeout expires to prevent watchdog intervention.
`void hit()`	Resets the countdown of the timeout and in periodic mode also restarts the countdown.
`void stop()`	Stops the watchdog. The current timeout is cancelled.

4.3.2 Example program using Watchdog pattern

This example will demonstrate the functionality of the watchdog pattern. The CSP diagram of the top level can be found in Figure 4-10. The example consists of three producer processes. Each of these producer processes has a watchdog. The producers are connected via channels to one consumer process. This consumer process continuously reads from all three the channels. The producers and consumer run in parallel. This parallel construct runs as the normal process inside the watchdog construct. At the bottom the safety process can be found, in this case it only prints that there was a watchdog timeout and then ends.

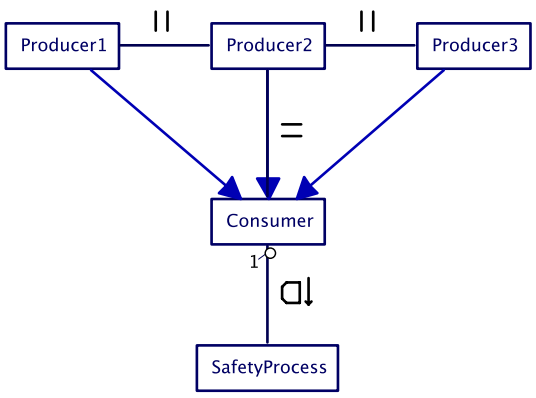

Figure 4-10 Top level of watchdog example program

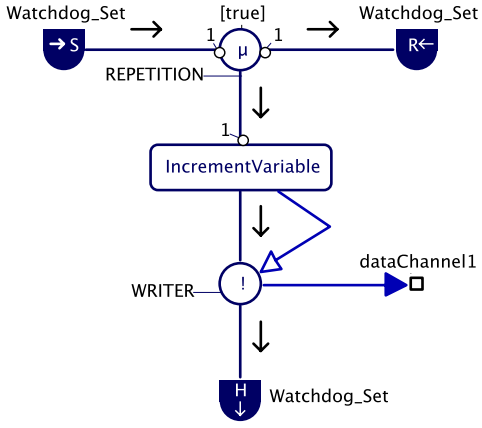

Figure 4-11 Internals of a producer process

The second figure (Figure 4-11) shows the internals of a producer process. It starts with starting its watchdog and then entering its main loop that consists of incrementing a variable, writing it to the consumer and lastly hitting the watchdog. Since the loop is `while(true)` it will never end (at least not normally), because of this the watchdog stopping can be omitted, but it is shown here nevertheless for completion.

To get this example to generate a watchdog timeout, the consumer process stops reading from one of the three channels after a certain amount of time. The watchdog from the producer process on that particular channel times out because the producer is blocked on the channel. Because of the timeout

the normal processes will be stopped by the watchdog construct and the safety process will be activated. When the safety process finishes, the entire construct is finished and the program will end.

The complete code for the model can be found in “FinalExamples/Example Watchdogs” on the CDROM.

4.3.3 Load measurement of CSP/CT program

Assessing the real-time behaviour of a CSP/CT program is important for control applications. To be able to determine the load of a CSP/CT program on a CPU helps to asses this behaviour. The load can be determined by measuring how long the program is idle (t_{idle}) within a certain period T. The load L is then given by the following formula:

$$L = \frac{T - t_{idle}}{T}$$

With the current CSP/CT library it is possible to determine t_{idle} by measuring how long the idle thread runs. This thread only runs when all other threads in a program are blocked.

Load measurement can be performed by an instance of the class `LoadMeasurement`. When enabled in the library, support for measuring the load of a CSP program via the idle thread method is automatically included in the Dispatcher of the library, but it has to be enabled by the user at run time. The instance can be accessed through the Dispatcher in the following way:

```
Dispatcher::instance()->idlePTload
```

And to enable call:

```
Dispatcher::instance()->idlePTload->setEnabled(true);
```

The default measurement period of this load measurement is 200 miliseconds. This can be changed however at will with the appropriate function call (see Table 4-5).

Table 4-5 Functions related to LoadMeasurement

Function	Functionality
`LoadMeasurement(const char *name, unsigned int period)`	Creates a new LoadMeasurement object with the given name (for logging) and given measurement period in microseconds. By default new instances have their measurement disabled.
`void setPeriod(unsigned int period)`	Changes the measurement period to the new period given in microseconds. Minimum is 1 ms.
`float getLoadTime()`	Returns the last measured load value.
`void setOverload(float overload)`	Sets the overload level. If the measured load rises above this value the LoadMeasurement triggers. Default is 1.0 which effectively disables overload check.
`void setEnable(bool enable)`	Enables or disables measuring. Default is disabled.
`void setFullLogging(bool on)`	Enables logging of every measured value to the global log in stead of only overloads. Default is only overloads.
`startOfRun`	Start measuring (automatic for load measurement via idle thread method).
`endOfRun`	Stop measuring (automatic for load measurement via idle thread method).

Configuring

The different aspects of a `LoadMeasurement` object can be configured with the member functions that are given in Table 4-5.

Example log

Below a piece of the global log is shown from a program with load measurement. The margin for overload was set to 0.8 and the measurement period was 10 ms. Full logging has been turned on, so every 10 ms the load was stored in the log. After a while the load rose above 0.80 and at that point overload messages were stored.

```
"timestamp","name","value"
1342,"SystemLoad",0.01
1352,"SystemLoad",0.03
1362,"SystemLoad",0.21
1372,"SystemLoad",0.05
1382,"SystemLoad",0.14
1392,"SystemLoad",0.35
1402,"SystemLoad",0.65
1412,"SystemLoad","Overload"
1412,"SystemLoad",0.81
1422,"SystemLoad","Overload"
1422,"SystemLoad",0.84
1432,"SystemLoad",0.80
1442,"SystemLoad",0.61
1452,"SystemLoad",0.21
```

An example program that enables load measuring with full logging can be found on the CDROM in the folder “FinalExamples/Example LoadMeasurement”.

Measuring other loads in a program

Load measurement is not limited to measure the load of a whole program. It can also be used in a more generic way to measure ‘a load’ in a program. LoadMeasurement provides two functions to accomplish this `LoadMeasurement::startOfRun()` and `LoadMeasurement::endOfRun()`. For example if one wishes to measure the load of a certain calculation, one would call `startOfRun()` just before the calculation starts, and `endOfRun()` just after the calculation is finished. A modified version of the previous example that uses this form can be found in “FinalExamples/Example LoadMeasurement 2”.

5 Case study: Patterns on Tripod

The patterns presented in chapter 4 have been applied to the new Tripod control software. This chapter discusses how they were applied and presents the results.

5.1 Logging

In the Tripod program there are a number of channels that are interesting to log for later analysis or for online monitoring. The most important ones are the position channels (Z1 – Z3). Secondly also the speed channels (Z1v – Z3v) and the force channels (Force1 – Force3 that go to the Commutator process) are logged. All these channels can be found in Figure 3-4. The force channels have been changed to ProbeChannels to accomplish this logging. For the position and speed channels ProbeChannels are not an option because they are actually linkdrivers with partially the same interface as channels but not the same implementation. Therefore the logging has been integrated directly into the linkdriver with the method described in section 4.1.3. Besides logging data from channels, logging is also added when important processes are starting or stopping and when other important events occur such as switching mode or when the Tripod platform moves outside its safe working area.

5.1.1 Data logging

Section 4.1.3 states that with some effort the logged values can be read by 20-sim. The result produced by the logging component is a file with 3 columns for all the data logged in the format ‘time;name;value’. The format 20-sim likes is ‘time value value value …’ so a time with a value for each input separated by a space or a TAB character. However in the Tripod software not all logging clients (‘logging inputs’) give a value each period, some only log once every couple of seconds, and others like position, speed and force every millisecond. Also the clients do not produce values at exactly the same time. If the data would be stored in a 20-sim like format, then for each new logged value all other values have to be printed again (to be more precise: interpolated), since the new timestamp won’t exactly match with the previous one. This is because it is not possible to skip a value for a certain column in the 20-sim format. A file in the 20-sim format with three value columns will be two times the size of RTLog format and this factor grows with the number of clients.

This problem can be solved by converting the log file to multiple files that can be read by 20-sim. A conversion program that accomplishes this is included in the Tripod source tree.

The logging functionality is used together with the conversion program and 20-sim to generate the plots in Figure 5-1 and Figure 5-2. The first plot is made from data of the real setup. The vertical position of all three motors is plotted against time. The second plot shows the difference in position between the real setup and the simulated setup.

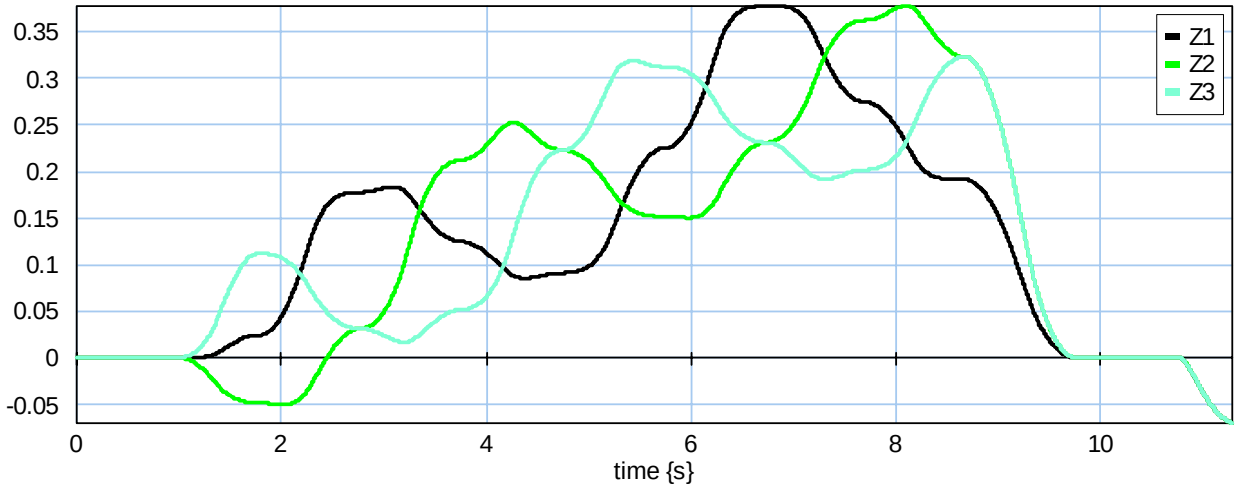

Figure 5-1 Plot of the three motor positions in meters against time

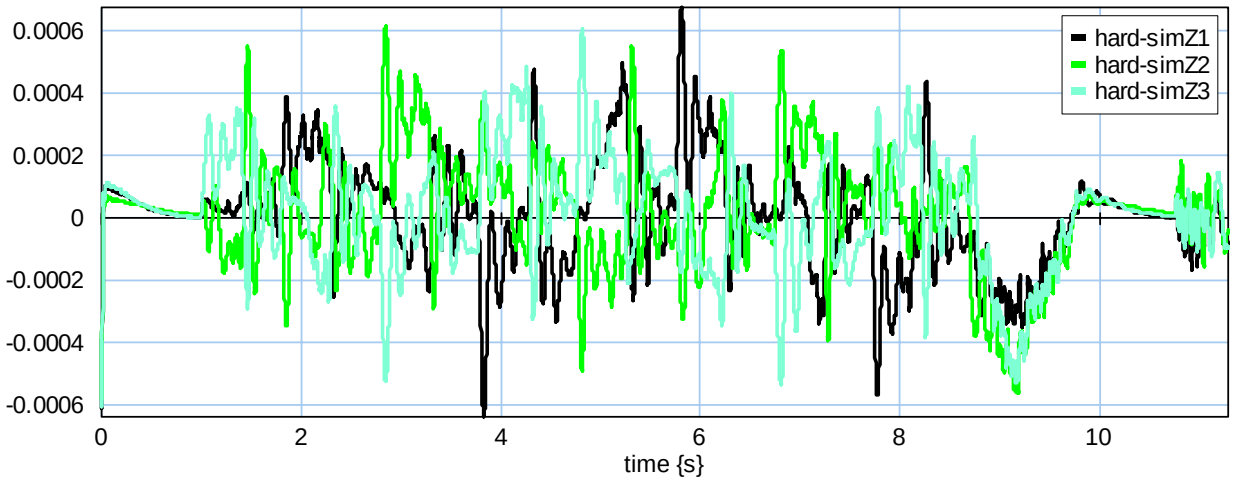

Figure 5-2 Plot of the difference in meters between the real setup and the simulated setup against time

5.1.2 Other logging

Besides the continuous logging of position, speed and force, events like starting and stopping of important processes are also logged. The ModeSwitcher process logs also to which mode it has switched. A typical start of the log for Tripod looks like this:

```
525480061,"WorkingArea","Registered"
525671911,"Commutator","Starting"
525671914,"TrackingErrorGuard_1","Started"
525671916,"TrackingErrorGuard_2","Started"
525671917,"TrackingErrorGuard_3","Started"
552626778,"ServoController/ModeSwitcher","Starting"
552626779,"ServoController/ModeSwitcher","SwitchToHoldZero"
552626981,"ServoController/HoldZeroPID","Starting"
552627063,"ServoController/GravityCompensator","Starting"
552627127,"Encoder_Z1",-0.000392
552627127,"Encoder_Z2",-0.000373
```

Note that after the ModeSwitcher logged “SwitchToHoldZero” that indeed the HoldZeroPID process started. The last two lines shown here are the first position values logged. Next another part of the log is shown in which the ModeSwitcher switches to the servo control mode named FollowPath. One can see that after the switch log message, process HoldZeroPID stops and the construct FollowPath starts.

```
553626547,"Encoder_Z3v",0.000000
553626653,"Force_1",37.190289
553626656,"ServoController/ModeSwitcher","SwitchToFollowPath"
553626657,"Force_2",38.080298
553626658,"ServoController/HoldZeroPID","Stopped"
553626820,"Force_3",35.738433
553627274,"ServoController/FollowPath/PathFromFile","Starting"
553627462,"ServoController/FollowPath/PID_1","Starting"
553627466,"ServoController/FollowPath/PID_2","Starting"
553627467,"ServoController/FollowPath/PID_3","Starting"
553627548,"Encoder_Z1",0.000001
553627548,"Encoder_Z2",0.000007
```

This starting and stopping of process is especially useful in debugging to see if some processes fail to start/stop. More events are logged by for example monitors in the program. These will be treated in the next section.

5.2 Monitoring

It is important that the Tripod remains in its safe working area (section 2.3). This is something that can be under supervision of a monitoring component connected to the three position channels from the Encoder. Each loop it can calculate if the radius and height of the platform are within the safe limits.

The formulas necessary for this calculation are given in section 2.3. Below a part of the log is shown were the calculated Z position of the platform was too high, although only barely as shown by the message following the error (safe value is smaller than 0.234).

```
559499022,"Encoder_Z1",0.377681
559499022,"Encoder_Z2",0.236473
559499022,"Encoder_Z3",0.226276
559499023,"WorkingArea","Error zcalc, outside safe position"
559499023,"WorkingArea",0.234001
```

This error was found with the same path as in Figure 5-1. The path file that described this path was found together with the agent version of the software on the Tripod system. When the radius and height are plotted together with the two limits the result is de plot in Figure 5-3. The maximum height is at 8.16 seconds and is 0.2367m, 2.7mm above the limit.

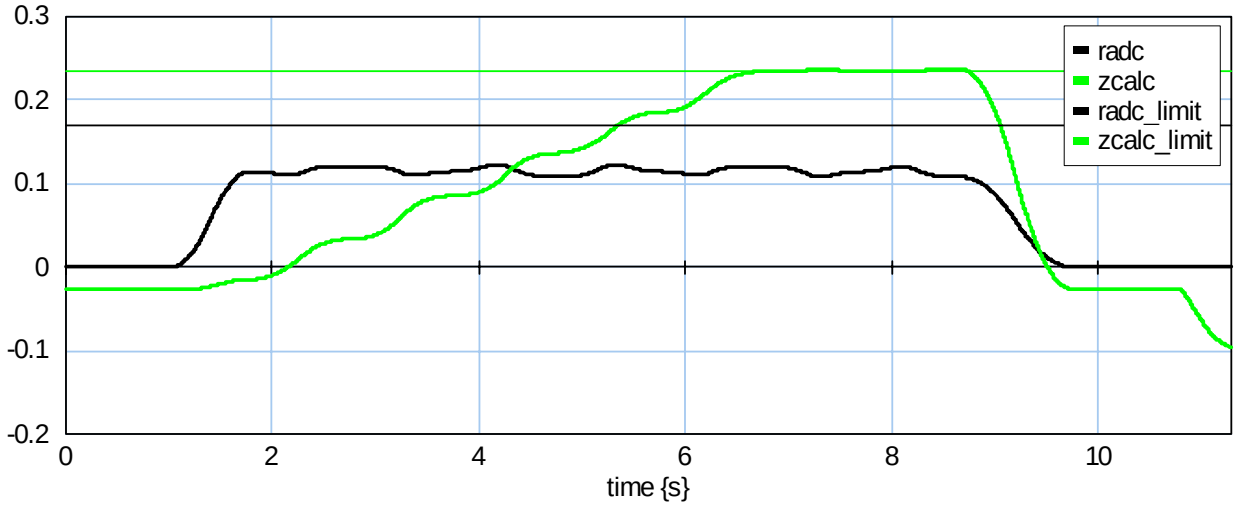

Figure 5-3 Plot of the platforms approximated distance from the center (radc) and height (zcalc) in meters against time

Currently the only action taken by this monitor when the platform is out of range is writing a message in the log. A simple action to prevent serious damage is disabling the amplifiers. A more complex one is poisoning the position channels. However since these are not real channels (but linkdrivers) support has to be added for poisoning them first.

5.3 N-Way

The process that will be N-Way programmed is the PID process. This process contains the controller which is used in the normal operation mode of the Tripod. N will be three in this case. Because designing three controllers independent from each other is beyond this project, three copies were made of the already existing one. Cross checking will be done by comparing the output values. Since the versions are exact copies, the output values should be an exact match, which alleviates the cross-check algorithm in the Decision process of the N-Way framework. To demonstrate the functionality of the pattern defects have been introduced in some copies, like inverting the output after a certain number of calculations.

The N-Way framework can throw exceptions in the case it cannot decide the correct value. To cope with these exceptions, exception handlers needed to be added to the FollowPath mode. The modified version of the FollowPath process is shown in Figure 5-4. Each framework (there are three, one for each axis) gets an exception handler that will catch a possible NWayException. If such an exception is caught, the handler will poison both input channels that come from PathFromFile and rethrow the NWayException that will be caught higher up. When PathFromFile tries to write to one of those two channels it will also throw an exception so PathFromFile also needs an exception handler. This

handler will poison all the output channels towards the N-Way PID controllers, so that in turn the other controllers also abort. The last exception handler named FollowPathExc is there to catch the N-Way exception that was thrown by the N-Way framework that detected the problem. This exception handler will set the return value of the FollowPath mode to a non-zero value, this makes that the ModeSwitcher will switch to the GoAll mode to bring the Tripod back to the zero position.

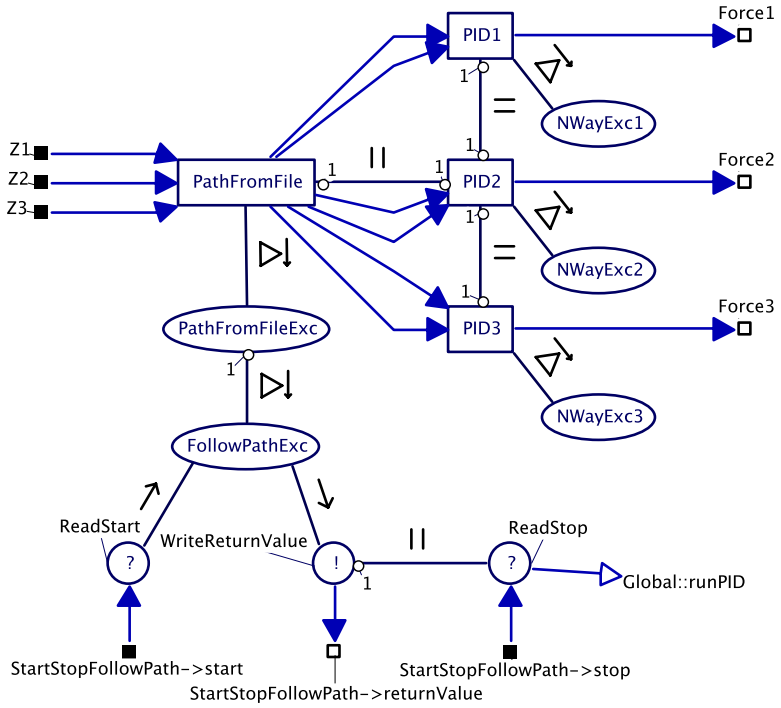

Figure 5-4 Modified FollowPath process to support exceptions of N-Way programmed PID processes

Of the three versions two are faulty. However the faults don't occur at the same time. First only one version behaves faulty: version two will invert its output. The Decision process in the N-Way framework detects and corrects this. The operation of the Tripod is not interrupted by the error in version two. The Decision process will place an entry in the log that an error was corrected; an example of such an entry is given here. The first entry reports the error and the following three are the results from all three versions.

```
47095402,"ServoController/FollowPath/NWayController_3/Decision","Corrected
error in version 2"
47095403,"ServoController/FollowPath/NWayController_3/Decision",0.999709
47095403,"ServoController/FollowPath/NWayController_3/Decision",-0.999709
47095403,"ServoController/FollowPath/NWayController_3/Decision",0.999709
```

At some later time the third version also starts making errors, it doubles it output. Now the three values are all different and the N-Way framework will throw an exception. The result is that all processes abort like described earlier and that the next mode is GoPositionAll.

```
49096321,"ServoController/FollowPath/NWayController_3/Decision","Fatal
error"
49096321,"ServoController/FollowPath/NWayController_3/Decision",-1.162940
49096321,"ServoController/FollowPath/NWayController_3/Decision",1.162940
49096322,"ServoController/FollowPath/NWayController_3/Decision",-3.488821
49096378,"ServoController/FollowPath/NWayController_3/Decision","Aborted"
49096680,"ServoController/FollowPath/NWayController_3/Input","Aborted"
49097319,"ServoController/FollowPath/NWayController_3","Aborted"
49097418,"ServoController/FollowPath/PathFromFile","Aborted"
49097441,"ServoController/FollowPath/NWayController_1","Aborted"
49097492,"ServoController/FollowPath/NWayController_2","Aborted"
49097547,"ServoController/ModeSwitcher","SwitchToGoAll"
49097560,"ServoController/GoPositionAll","Starting"
```

In Figure 5-5 a plot is shown with the three motor positions over the course of time. Version two fails in this plot from 4.076 seconds on. Only when version three also fails at 6.076 seconds the FollowPath mode is aborted and the motors are returned to the zero position by GoPositionAll. This plot is from a simulation version because the GoPositionAll mode has not been tested on the real setup due to time constraints. After GoPositionAll the HoldZero mode is activated, and a little later, the Shutdown mode is activated.

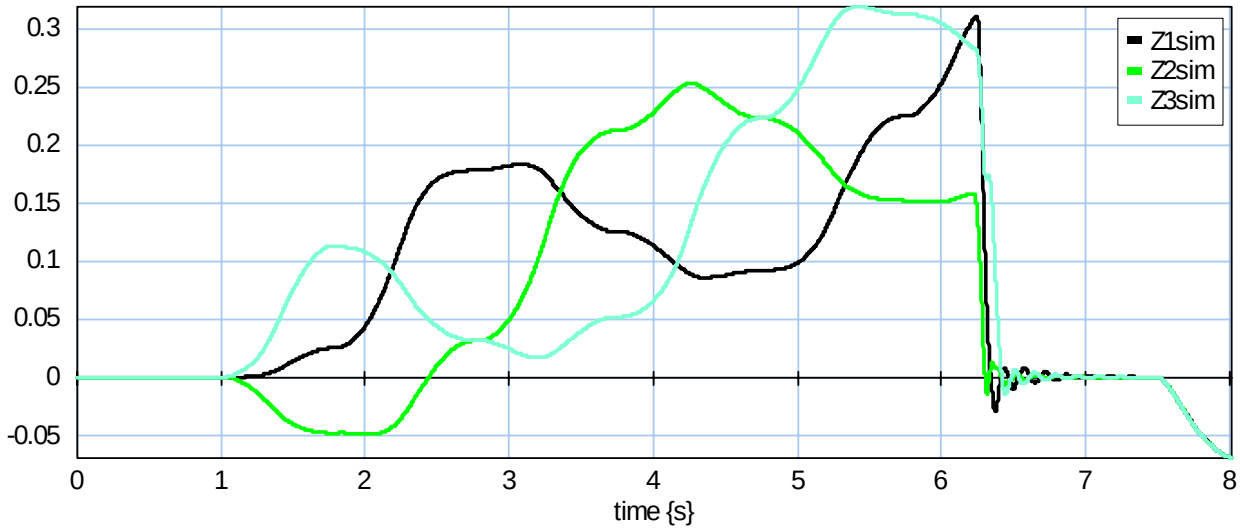

Figure 5-5 Plot of the three motor positions against time with the N-Way controllers for the simulated setup.

5.4 Watchdog

To guard the Tripod software with watchdogs, they can be put in the Commutator process. This process is chosen because in this process it can guard the entire chain of processes from the first processes that reads from the input linkdrivers to the last process, the Commutator itself. If some process in the chain locks up or for some reason takes too long, the watchdog in the Commutator will not be hit in time and will timeout. The watchdog can then start a safety mechanism to deal with the situation.

Figure 5-6 shows the top level of the Tripod software, now extended with the WatchdogSafetyProcess that will run when a watchdog timeout has occurred within Commutator. To be able to control the Tripod in the safety process, this process also needs to be able to read from the linkdrivers; however it was chosen not to draw these channels in the current model to be able to keep a clear picture. Figure 5-7 shows the internals of the Commutator process. It consists of three parallel branches, one for each axis of the Tripod. Each branch is protected with its own watchdog which will be hit every loop, unless the branch remains blocked on the reader.

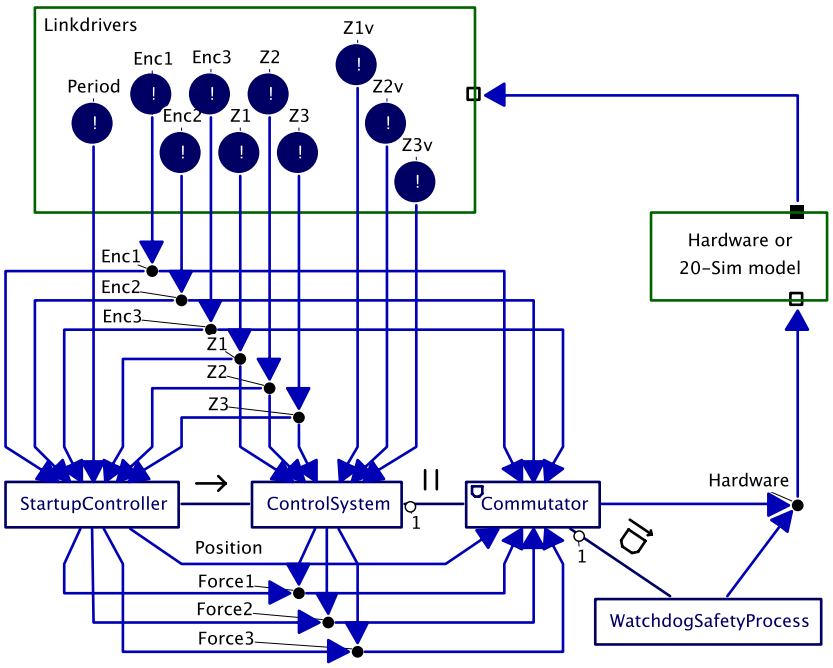

Figure 5-6 Modified top level of Tripod software to support watchdogs

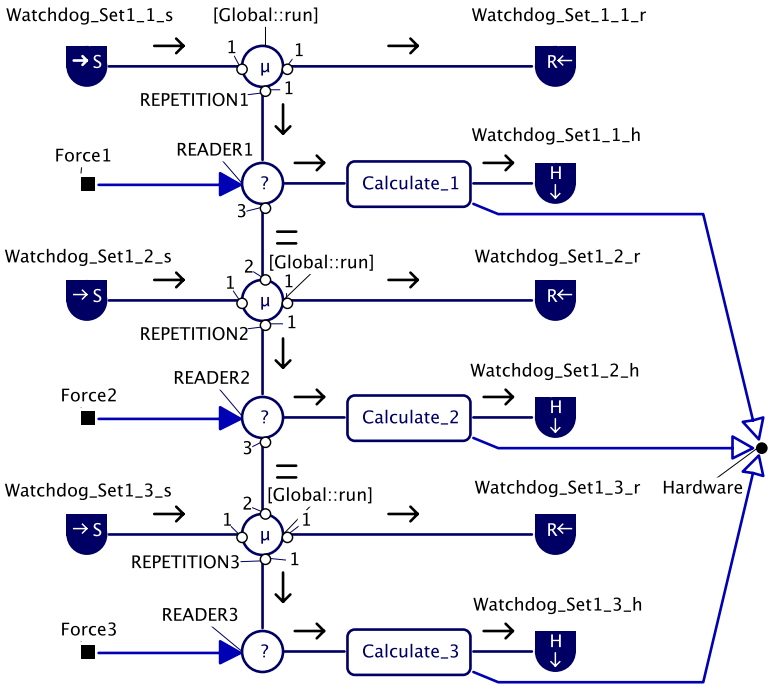

Figure 5-7 Internals of process Commutator with watchdogs

Unfortunately support for aborting processes, necessary for correct watchdog construct functionality, could not be implemented on time in the library. Therefore no tests could be done with the simulation or real setup.

6 Conclusion and recommendations

6.1 Conclusions

The patterns described in this report make concurrent (control) software based on CSP/CT more dependable by increasing their safety and reliability. They are suitable for use in CSP/CT software in all fields of research of the CE department: embedded, robotic and mechatronic control systems.

The first pattern, Logging and Monitoring, supports many aspects of the CSP/CT software. The logging pattern is useful for example during the development phase of a program for debugging and for maintenance it can contain information on which parts of a system need service. The monitoring pattern helps to increase the safety of a system with the possibility to intervene if wrong values are detected. With probe channels this complete pattern is easy and fast to use. Support for probe channels has already been integrated into the gCSP tool and support for monitoring components is underway.

The second pattern, N-Way programming, mainly provides increased reliability through redundancy. In terms of implementation costs it is fairly expensive because multiple versions of a process have to be developed independent from each other. Depending on the complexity of the cross-check algorithm it is easy to implement in CSP/CT. The gCSP tool supports automatic generation of the N-Way framework in a model via the context menu on a process in the C-Tree.

The third pattern, Watchdogs, increase the safety of a system by intervening if (a part of) a system becomes unresponsive, and misses a deadline. The watchdog pattern can then activate a safety mechanism to bring the system back into a safe state. Watchdogs should be considered a last resort mechanism for the software to recover from serious faults. The gCSP tool supports the modelling of watchdogs and the accompanying constructs, but is incomplete to generate code for them. For example there is no support yet to specify the size of the timeout.

For distributed systems based on CSP/CT software the distributed synchronized stop pattern is very useful. Together with the CSP/CT based Tripod software a start is made on distributed control on Tripod.

Agents and processes, the agent framework and CSP/CT have similarities. Both agents and processes are independent entities that are related to each other by communication and compositional relations. The agent framework is more light weight, but the CSP/CT implementation addresses concurrency of control systems more directly and is easier to distribute.

An early version of the new control software had a bug that resulted in the setup damaging itself. This proves that software has to be tested before being put to use, especially on setups that have the potential to damage themselves. This event led to the implementation of the linkdrivers that simulated the real mechanical hardware of the Tripod. This HIL (hardware in the loop) (Isermann *et al.*, 1998) approach proved to be very effective for testing later versions of the program without risking damage to the real setup. During the work on the Tripod setup it was also discovered that there were multiple wiring faults, which have all been corrected.

The patterns are not the only improvements made to CTC++ library. Numerous minor bugs were fixed and code was cleaned up. The new Tripod control software is the first large project built on the redesigned library with complex compositional relations. This resulted in the discovery of a number of less obvious bugs. Finally the examples created to illustrate the patterns can be used as examples for other users of the library, complete with gCSP models.

During the work for this project the gCSP tool was extensively used for modelling software. This resulted in quite a number of (suggestions for) improvements. The gCSP tool is now more mature and easier to use.

6.2 Recommendations

The Logging and Monitoring pattern can be improved on multiple fronts. A fairly simple improvement that just takes time to implement is the support for more message types. The current implementation only supports doubles and predefined text messages. This should be extended with support for all standard data types like booleans, characters and integers (32 and 64 bits). Another improvement that can be made is the introduction of priorities to messages. Messages stored in the buffer can only be overwritten by messages of an equal or higher priority. These priorities can also be used in a component that sends logged messages to another program, possibly on another computer system. Finally the registration of clients should be done in a separate central component in stead of the default logging component RTLog.

The presented N-Way programming pattern should be tried with really independently designed versions and a more complex cross-check algorithm. Possibly this can be done as part of a practicum for students learning gCSP or CSP/CT. Each group can design one version of a process doing some complex function. All versions are then made available to all groups. Each group can then implement the N-Way programming pattern with a number of versions and implement a cross-check algorithm for them.

The watchdog pattern can be extended with soft-deadlines. When a soft-deadline is crossed another (less drastic) action can be taken like writing a warning message to the log.

Investigate distributed control on Tripod further. This project provides the building blocks for distributed control of the Tripod setup: a distributed synchronization pattern called SyncStop (subsection 3.1.1) and all functionality in the Tripod software is divided over multiple processes that can easily be distributed.

Development on DOS, the current OS on the Tripod system, is far from optimal for the following reasons: no network support besides copying files, no support for running multiple programs with different priorities and a lot of outdated tools. The main obstacle to change the OS for example to Linux with RTAI (real-time extension for Linux) is that the CTC++ library lacks support for timing on this platform. Especially for control applications it is important to have good timing. The CTC++ library is missing a good cross platform timer. The Tripod software itself is fairly easy to port, only the hardware linkdrivers have to be converted, the rest of the program is portable across all platforms the CTC++ library runs on.

A lot of the code in the new Tripod program can be automatically generated by gCSP tool. Unfortunately it was not mature enough for this project to be used for that purpose. In the near future the tool should be able to generate all code required for the Tripod program. Because of the level of complexity in the Tripod program, this can help to further improve the code generation of the tool.

Appendix A Working principle and drive of linear motor

Adapted from M. Eglence MS.c. thesis (Eglence, 2003)

A.1 Working principle of linear motor

The brushless permanent-magnet motor configuration consists of a base plate (stator), covered with permanent magnets, and a sliding part (translator) that holds the electric coils and their iron cores, see Figure A-1. By applying a three-phase current to the coils, a sequence of attracting and repelling forces between the poles and the permanent magnets will be generated. An incremental linear encoder measures the position of the translator.

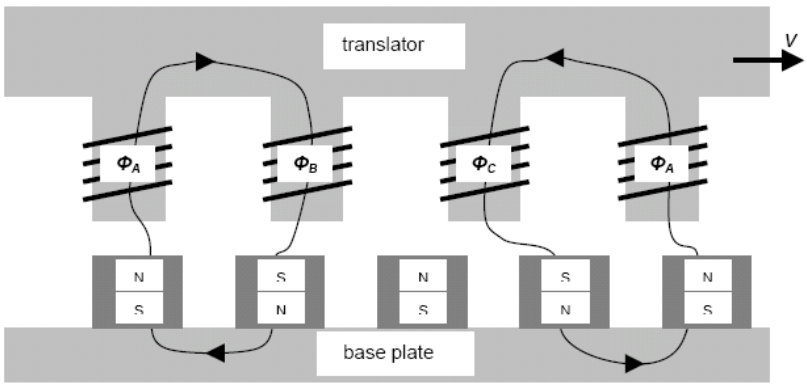

Figure A-1 Working principle of the linear motor. The indicated lines are the flux lines of the permanent magnets.

To increase the efficiency of the motor, iron cores are placed in the coils. These iron cores have a magnetic interaction with the permanent magnets on the base plate. This interaction, regardless of whether there is a current in the coils or not, results in a force that tries to move the translator into stable detent positions. This phenomenon is known as cogging. Because of the spatial distribution of the magnets with a period of 12 [mm] a stable detent position is also found every 12 [mm].

A.2 Drive of linear motor

To obtain the right sequence of the attracting and repelling forces for the movement commutation is needed. There are several commutation methods, like trapezoidal and sine wave. The best performance is achieved with sine wave commutation, which is also the most computationally intensive. The currents trough the individual coils depend on the required force for the movement and the relative position of the coil with respect to the magnets. In Figure A-2 the principle of commutation is explained. In the figure only one coil is shown. Because of the three-phase configuration of the coils the alternating current through the other coils are 120° shifted. The colour of the coil gives the amount of current and the N or S symbols gives the polarity of the field, which depends on the sign of the current. Darker colours represent larger currents. The direction of the movement can be changed by applying a phase shift of 180° to the current. Because of the sine function this is the same as changing the sign of the reference current.

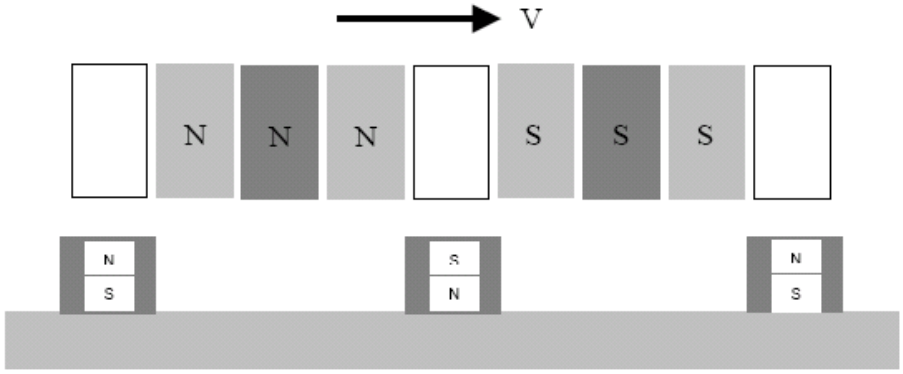

Figure A-2 Principle of commutation. The darkness of the coil represents the amount of current through it.

The commutation can be written out in formulas:

$$I_1 = I_{ref} \sin(Pos \frac{2\pi}{0.024})$$

$$I_2 = I_{ref} \sin(Pos \frac{2\pi}{0.024} - \frac{2\pi}{3})$$

$$I_3 = I_{ref} \sin(Pos \frac{2\pi}{0.024} - \frac{4\pi}{3})$$

With *Pos* the absolute position of the translators, which starts at zero exactly when coil number 1 is above a North pole magnet on the base plate. There are amplifiers on the market that can determine the relative position of the coils with respect to the magnets by measuring the magnetic field. This is done by Hall sensors in the translator. The Tecnotion linear motor that is used for the manipulator doesn't have any Hall sensors. This means that before the commutation can start, a procedure has to be executed to bring the translator exactly with coil number one above a North pole. Which North pole doesn't matter, as long it is a North pole on the base plate. This procedure is called aligning. A feasible aligning method that can be performed with the used amplifiers is the so called "wake and shake" method. This goes as follows. Coil number one is fed with 0.5 I_{nom} and coil number two with –0.25 I_{nom}. In a three phase system the three currents add up to a sum of zero, therefore a current of –0.25 I_{nom} is also fed to coil number three by the amplifier. Coil number one has become a "strong" South pole and coil number two and three "weak" North poles. This causes a force that drives the translator exactly above the nearest North pole with coil number one. By holding on the currents for about half a second, the translator is aligned. With this position known the encoder can be reset and the commutation can start. The maximum displacement during aligning is 12 [mm].

Appendix B Tripod computer hardware

Computer interface cards

Advantech PCL833 encoder counter interface card (ISA) (http://www.advantech.com.tw)
The PCL-833 encoder card from Advantech reads in the 3 linear encoders of the motors. It has 3 independent 24-bit up/down counters (cascadable to 48-bit) and 5 digital inputs with 2500VDC isolation. The encoders on the motors are from the manufacturer Numerik Jena and have a grating period of 20 [µm]. After the quadrature and the 5X interpolation factor this results in a resolution of 1.0 [µm].

- Three encoder channels used for three encoders

ICP P16R16 Digital I/O card (PCI) (http://www.icpdas.com/products/card/pci-p16r16.htm)
The digital I/O card is from ICP DAS. It has 16 optically isolated inputs and 16 relay outputs. The input voltage is 5-24 volt AC or DC.

- Digital Inputs (16)
  - 'Vrijgave' status (1)
  - Start button (1)
  - Stop button (1)
  - Amplifier and motor status, e.g. overheating (12)
  - Unused (1)
- Digital Outputs (16)
  - Enable/Disable Amplifiers (3)
  - Safety circuit (1)
  - Unused (12)

PIO-DA8 DAQ card (PCI) (http://www.icpdas.com/products/card/pio-da16.htm)
The DAC card is an 8 channel 14-bit analogue output board from ICP DAS. Only 6 of the 8 channels are used. It has output modes of ±10 Volt and ±5 Volt. For the manipulator the ±5 Volt mode is used. This results in a maximum output current for the amplifier of ±10 Ampere, which is feasible for the motors. It also has 16 digital inputs and 16 digital outputs (TTL compatible).

- D/A channels (8)
  - Linear motors (6)
  - Unused (2)
- Digital Inputs (16)
  - Endswitches (3)
  - AtLowerHalf signal from encoder (3)
  - Unused (10)
- Digital Outputs (no connection from computer to Tripod cabinet) (16)
  - Unused (16)

Control PC

Rocky-4783EV-r1.2 motherboard
P4 1.3 GHz processor with Hyper Threading support
1 GB RAM
1 Advantech PCL833 encoder counter interface card (ISA)
1 ICP P16R16 Digital I/O card (PCI)
1 PIO-DA8 DAQ card (PCI)
1 network interface card
2 serial ports
1 parallel port (no connector though)
1 USB 2.0 port (probably more ports on motherboard)
1 1394 Firewire port
1 VGA
1 floppy drive
40 GB HD

Second PC

The second PC contains the following hardware:
Rocky-3705EV-R2 motherboard
Celeron 733 MHz
128 MB RAM
2 network interface cards
2 serial ports
1 parallel port
1 VGA
1 floppy drive
1 CD-ROM drive
40 GB HD

The second pc is not in use at the moment.

The whole setup is powered via one 230V AC connection. This connection is buffered by an UPS of 700 VA. Sometimes the UPS cannot handle the load, this happens for example during the aligning phase of the program. This shouldn't be too much of a problem, but this has to be taken into account when more power consuming appliances are connected to the UPS.

Appendix C CDROM contents

This appendix gives an overview of the contents per directory that can be found on the CDROM. The latest version of this overview can be found in the README.txt that is also on the CDROM.

CSPTrip/

20-sim model for the simulation model of the Tripod hardware
gCSP model of the Tripod control software
Source code for the CSP/CT version of the Tripod control software

LogConverter/

Contains the software to convert RTLog's CSV format to a 20-sim readable format

FinalExamples/

Contains the source code and gCSP models for all the examples in this report in the following subdirectories:

Example LoadMeasurement/

Example LoadMeasurement 2/

Example Monitored Multiplier/

Example NWay multiplier/

Example NWay multiplier with exceptions/

Example NWay multiplier with exceptions and logging/

Example ProbeChannel/

Example SyncStop/

Example Watchdogs/

Report/

Contains this report in PDF and doc format

Images/

All images used in this report as separate files

Presentation/

PowerPoint presentation

Tools/

gCSP/

Latest version of the gCSP tool used to create all models

Compiler/

Cross compiler to compile dos executables under Linux

Literature

Avižienis, A. (1977), Fault-tolerant computing - progress, problems, and prospects, *Proc. IFIP Congress*, pp. 405-420.

Avižienis, A., J.-C. Laprie, B. Randell and C. Landwehr (2004), Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Trans. on Dependable and Secure Computing*, **1,** (1), pp. 11-33.

Breemen, A.J.N.v. (2001), *Agent-Based Multi-Controller Systems*, PhD thesis, University of Twente, NL, ISBN.

Controllab Products B.V. (2005), http://www.20sim.com/, pp.

Eglence, M. (2003), *Design and Realization of a Safe Control System for a Parallel Manipulator*, MSc thesis, University of Twente, ISBN.

Engelen, T.v. (2004), *CTC++ enhancements towards fault tolerance and RTAI*, no 022CE2004, MSc thesis, University of Twente, Enschede.

Hilderink, G.H. (2005), *Managing Complexity of Control Software through Concurrency*, PhD thesis, PhD thesis, University of Twente, NL, ISBN.

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

Isermann, R., J. Schaffnit and S. Sinsel (1998), Hardware-in-the-loop simulation for the design and testing of engine control systems, *Control Engineering Practice*, **7,** (5), pp. 643-653.

Jovanovic, D.S., B. Orlic, G.K. Liet and J.F. Broenink (2004), gCSP: A Graphical Tool for Designing CSP Systems, *Proc. Communicating Process Architectures 2004*, 5-8 Sep 2004, Oxford, UK, I. East, J. Martin, P. Welch, D. Duce and M. Green (Eds.), pp. 233-251.

Leveson, N.G. (1995), *Safeware: System Safety and Computers*, Reading, MA: Addison-Wesley.

Orlic, B. and J.F. Broenink (2003), Real-time and fault tolerance in distributed control software, *Proc. Communicating Process Architectures CPA 2003*, 7 - 10 September 2003, Enschede, NL, J. F. Broenink and G. H. Hilderink (Eds.), pp. 235-250, ISBN: 1 58603 381 6.

Randell, B., P.A. Lee and P.C. Treleaven (1978), Reliablity Issues in Computing System Design, *ACM Computing Surveys*, **10,** (2), pp. 123-164.