Managing Complexity of Control Software through Concurrency

ISBN 90-365-2204-8

Printed by Ipskamp Enschede, the Netherlands

Cover design: Gerald Hilderink
Cover art: Leny Bilderbeek-Wilens

MANAGING COMPLEXITY OF CONTROL SOFTWARE THROUGH CONCURRENCY

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 19 mei 2005 om 15.00 uur

door

Gerald Henk Hilderink

geboren op 23 september 1968
te Haaksbergen

Dit proefschrift is goedgekeurd door

Prof.dr.ir. J. van Amerongen	promotor
Dr.ir. J.F. Broenink	assistent-promotor

Contents

Voorwoord

In traditionele methoden, zoals ik die gedurende mijn opleidingen kreeg voorgeschoteld, heb ik tekortkomingen en valkuilen ervaren bij het ontwikkelen van software voor ingebedde systemen. Bepaalde ontwikkelingen op het gebied van software ontwerpen en programmeren maakten de dingen complexer in plaats van juist eenvoudiger. Door het bestuderen van de programmeertaal occam en de achterliggende theoretische concepten, heb ik ontdekt dat er wel een paradigma bestaat dat software simpeler maakt in plaats van complexer. De clou zat hem in een elegante benaderen van 'concurrency'.

Ik had het geluk dat mijn ideeën, in het kader van mijn afstudeeronderzoek, resulteerden in dit promotieonderzoek bij de Leerstoel Regeltechniek van de Faculteit Elektrotechniek, Wiskunde en Informatica (EWI) aan de Universiteit Twente. Door dit onderzoek heb ik geleerd om software voor ingebedde systemen op een adequate en systematische manier aan te pakken, d.w.z. dat concurrency de complexiteit in de hand houdt. Door deelname aan conferenties en reacties op eerdere publicaties heb ik kunnen vaststellen dat mijn bevindingen ook ervaren en gedeeld worden door anderen.

Graag wil ik Job van Amerongen, Jan Broenink en André Bakkers bedanken dat zij dit onderzoek mogelijk hebben gemaakt. Dusko Jovanovic, Bojan Orlic en Peter Visser wil ik bedanken voor alle suggesties en discussies die ik met hen gehad heb. Natuurlijk ben ik ook mijn dank verschuldigd aan die talloze studenten die hun bijdrage hebben geleverd aan dit onderzoek.

Mijn stiefvader Jan Bilderbeek en mijn moeder wil ik van harte bedanken voor hun steun en adviezen, en mijn moeder vooral voor haar creativiteit gelegd in de omslag van mijn proefschrift. En dan nu wil ik mijn lieve José bedanken voor haar geduld, toeverlaat en liefde gedurende dit karwei. Dankjewel lieve mensen!

Gerald Hilderink

Enschede, 16 april 2005.

Summary

In this thesis, we are concerned with the development of concurrent software for embedded systems. The emphasis is on the development of control software.

Embedded systems are concurrent systems whereby hardware and software communicate with the concurrent world. Concurrency is essential, which cannot be ignored. It requires a proper handling to avoid pathological problems (e.g. deadlock and livelock) and performance penalties (e.g. starvation and priority conflicts). Multithreading, as such, leads to sources of complexity in concurrent software. This complexity is considered frightening, because it complicates the software designs and the resulting code. Moreover, this paradigm complicates the understanding of the behaviour of concurrent software.

A paradigm with a precise understanding of concurrency is essential. In this thesis, a methodology is proposed that comprises a paradigm of fundamental aspects of concurrency. These fundamental aspects are derived from the *Communicating Sequential Processes* (CSP) theory. CSP is a theory of programming that is developed by Hoare, Brookes, and Roscoe. CSP comprises *fundamental concepts* useful for precisely describing and studying concurrent systems. These concepts are based on processes and events. Processes and events are abstract entities, more abstract than objects. Processes and events are essential in describing and reasoning about the real-time behaviour of process architectures. A process architecture describes a (sub-) program as a composition of communicating processes.

The proposed methodology brings a subset of CSP to practice in order to specify, design, and implement process architectures. The CSP concepts bring forth a glue-logic between these phases in the development trajectory. Furthermore, these concepts offer technical solutions, which

have been enhanced with notion of priorities, exception handling, and timing. The precise semantics of the concepts and their restrictions provide the guidelines to create reliable and robust concurrent software. The abstraction and separation of well-defined concerns contribute to managing complexity in concurrent software.

The proposed methodology defines the following ingredients:

1. A *graphical modelling language* defines graphical notations and rules that are derived from CSP. The graphical modelling language is used for specifying, designing, and graphically programming process architectures. This results in *CSP diagrams*.

2. An *object model* implements the CSP concepts by means of object-oriented techniques. This model can be implemented in object-oriented programming languages. This results in the *CSP libraries* for Java, C (in object-oriented style) and C++.

The graphical modelling language and the object model go together. CSP diagrams are used to describe and to analyse process architectures. A CSP library is used to implement process architectures in an object-oriented programming language. This methodology uses process-oriented and object-oriented techniques, and hides thread-oriented techniques.

The proposed methodology is applied to control applications on embedded computer systems.

Samenvatting

In dit proefschrift houden we ons bezig met de ontwikkeling van concurrent software voor ingebedde systemen. De nadruk ligt op de ontwikkeling van regelsoftware.

Ingebedde systemen zijn concurrent systemen, waarbij hardware en software communiceren met de concurrent wereld. Concurrency is wezenlijk en kan niet worden genegeerd. Het vereist een goede behandeling die pathologische problemen (zoals deadlock en livelock) en prestatieproblemen (zoals starvation en prioriteitconflicten) dienen te voorkomen. Multihreading, als zodanig, leidt tot een bron van complexiteit in concurrent software. Deze complexiteit wordt als afschrikwekkend ervaren, want het bemoeilijkt de softwareontwerpen en de resulterende code. Bovendien bemoeilijkt dit paradigma het begrijpen van het gedrag van concurrent software.

Een paradigma met een precieze kennis van concurrency is essentieel. In dit proefschrift wordt een methodologie voorgesteld dat een paradigma van fundamentele concurrency aspecten behelst. Deze fundamentele aspecten zijn ontleend aan de *Communicating Sequential Processes* (CSP) theorie. CSP is een theorie over programmeren die is ontwikkeld door Hoare, Brookes en Roscoe. CSP behelst *fundamentele concepten* die geschikt zijn voor het nauwkeurig bestuderen van concurrent systemen. Deze concepten zijn gebaseerd op processen en events. Processen en events zijn abstracte entiteiten, abstracter dan objecten. Processen en events zijn noodzakelijk voor het beschrijven van en het redeneren over het real-time gedrag van procesarchitecturen. Een procesarchitectuur beschrijft een (deel-) programma als een samenstelling van communicerende processen.

De voorgestelde methodologie brengt een deelverzameling van de CSP theorie naar de praktijk voor het specificeren, ontwerpen en

implementeren van proces architecturen. Deze CSP concepten leiden tot een perfecte passing van deze fasen in het ontwikkeltraject. Bovendien bieden deze concepten technische oplossingen, welke zijn aangevuld met prioriteiten, foutenafhandeling en notie van tijd. De precieze betekenissen van de concepten en hun beperkingen zorgen voor richtlijnen om betrouwbare en robuuste ingebedde regelsoftware te ontwikkelen. De abstractie en de scheiding van goed gedefinieerde belangen dragen bij tot het beheersen van complexiteit in concurrent software.

De voorgestelde methodologie definieert de volgende ingrediënten:

1. Een *grafische modelleringstaal* definieert grafische notaties en regels die afgeleid zijn van CSP. De grafische modelleringstaal wordt gebruikt voor het specificeren, ontwerpen en grafisch programmeren van procesarchitecturen. Dit resulteert in zogenaamde *CSP diagrammen*.

2. Een *object model* implementeert de CSP concepten door middel van object-georiënteerde technieken. Dit model kan vervolgens worden geïmplementeerd in object-georiënteerde programmeertalen. Dit resulteert in *CSP bibliotheken* voor Java, C (in object-georiënteerde stijl) en C++.

De grafische modelleringstaal en het object model sluiten op elkaar aan. CSP diagrammen worden gebruikt om procesarchitecturen te beschrijven en te analyseren. Een CSP bibliotheek wordt gebruikt om procesarchitecturen te implementeren in een object-georiënteerde programmeertaal. Deze methodologie maakt gebruik van proces-georiënteerde en object-georiënteerde technieken en verbergt thread-georiënteerde technieken.

De voorgestelde methodologie is toegepast op regelapplicaties voor ingebedde computer systemen.

CHAPTER 1

Introduction

1.1 Concurrency and complexities in embedded control software

In this thesis, we are concerned with the development of embedded control software for mechatronic systems. Mechatronics deals with controlled mechanical systems that are designed as a whole. Mechatronic design is the integrated design of a mechanical system and its embedded control system. A mechatronic design approach leads to more flexible and cost effective machines with better performance (van Amerongen, 2003). Examples of mechatronic systems are robots, production machines, modern cars, airplanes, CD- and DVD-players, etc. The controller part of a mechatronic system is mostly realized in software as an embedded control system. Embedded control systems require safety, reliability, robustness, and the guarantee that their processes meet their deadlines for a safe and reliable operation of the entire system. Those systems are hard real-time and inherently concurrent since they interact and respond in time to a concurrent world.

The total behaviour of a mechatronic system is described by its physical-system dynamics, control laws, and the characteristics of the software and hardware. These artefacts are inherently concurrent in which all components participate and aggregate. Concurrency naturally fulfils a glue-logic between the different artefacts and components in software

and hardware. It offers the means to integrate components and take away discontinuities between them. In fact, concurrency offers the tools to manage complexities.

Embedded control software is considered both sophisticated and complex. It has to deal with several sources of complexities in software engineering, such as:

- multithreading,
- interrupt handling,
- exception handling,
- inter-processor communication,
- priority scheduling and preemption,
- precise timing and sampling,
- reactivity and responsiveness,
- safe-guarding and fault-tolerance.

These sources of complexities concern concurrency in both software and hardware. Software design and programming tools, languages, and methods have to deal with these sources of complexities. Commonly, ad-hoc solutions are offered that deal with these issues. Ad-hoc solutions are individual solutions meant for one thing only. It is up to the user to integrate these ad-hoc solutions into a concurrent framework. This process is being complicated when a paradigm of loosely coupled concepts are used. These ad-hoc solutions require a common level of abstraction in order to understand the separate concerns as a whole and in a systematic way.

Software design and programming tools, languages, and methods must capture a good understanding of concurrency. Concurrency should be driven by coherent and formal concepts and not by ad-hoc solutions. Therefore, the foundation that underlies these tools, languages, and methods must contain coherent concepts that integrate the previously mentioned sources of complexities and abstract away from ad-hoc solutions.

The semantics of these concepts must be preserved during the development of embedded systems. In this thesis, a methodology is proposed that captures a good understanding of concurrency as a whole, in a natural and systematic way. A keystone of this methodology is *Communicating Sequential Processes* (CSP) (Hoare, 1985; Roscoe, 1998). CSP is a theory embracing fundamental concepts describing and understanding concurrency in a formal and systematic way. The foundation that underlies the proposed methodology is suitable for developing embedded software, in particular, embedded control software.

1.2 Scope of subject

1.2.1 Embedded control

The following definitions are frequently used in this thesis, which are derived from common definitions found in literature and on the internet.

An *embedded system* is a combination of computer hardware and software that is embedded in a larger system, hidden from the end-user. A common characteristic of an embedded system is that it is programmed to perform a set of functions that minimizes end-user or operator intervention; thus an embedded system automates a product.

A *real-time system* is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced.

An *embedded control system* is an embedded real-time system with the task of controlling a physical process. An embedded control system consists of one ore more control loops (i.e. controller processes) interacting with a physical process through sensors, actuators and its input/output interface. A blueprint of an embedded control system, as part of a mechatronic system, is depicted in Figure 1-1.

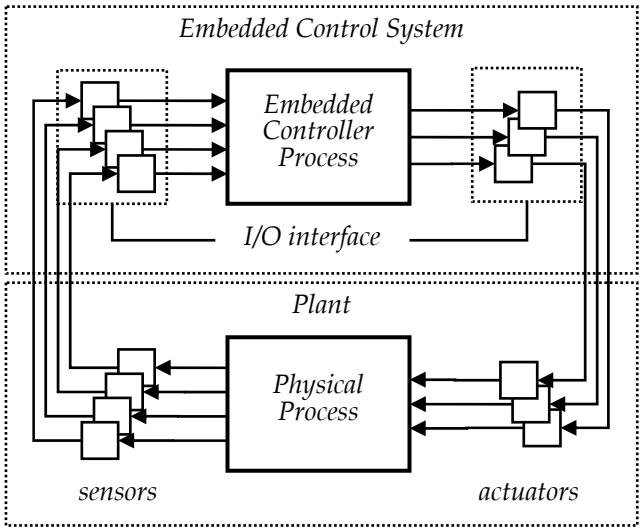


Figure 1-1 *Blueprint of a mechatronic system.*

A *sensor* is a device that responds to a physical stimulus (heat, light, pressure, motion, flow, and so on), and produces a measurable corresponding electrical signal. This electrical signal is translated by the computer hardware into a digital value that is consumed by the control software.

An *actuator* is a device which performs a physical action to an electrical stimulus generated by the computer hardware.

The electrical circuits that perform signal conversion between the computer, sensors, and actuators are called the *input/output interface* (I/O interface) of the embedded computer system. The arrows between the embedded control system and the plant are usually electrical signals. The arrows between the embedded control system and its I/O interface are digital signals and the arrows between the plant and the sensors or actuators are physical stimuli.

The controller process interacts with one or more physical processes. The behaviour of this control process is composed by at least two parallel processes that engage in communication at discrete moments in time. Conceptually, a controller process is event-driven, which may engage in respectively periodical communication events, sporadic communication events, or perhaps a mix of both. The communication events represent

the completion of signal conversion (e.g. sampling and actuation), based on fixed or variable time intervals. In this thesis, hard real-time control systems are considered for which every missed deadline is an error. Therefore, the communication events must happen within a deadline or at precise moments in time. The real-time behaviour of an embedded control system is observable by tracing the events. These traces are suitable to guarantee the proper functioning of the system.

Embedded control systems are often part of larger heterogeneous systems connected by their I/O interfaces. These systems vary from single *central processor unit* (CPU) systems that are compact and constrained by a limited amount of resources (e.g. memory and CPU speed), to single CPU systems with ample system resources, or to multiple CPU systems distributed over a plant. Considering the variety of concerns in hardware and software and the variety of ad-hoc solutions to these concerns, it is not surprising that embedded software engineering is often considered complex and specialized, even without considering the difficulties of control law design. A programming model is required which deals with this complexity in an elegant way. Such a programming model is usually based on a *real-time kernel* (RTK) technology that provides an abstract layer of services, which controls the hardware and schedules multiple threads of control. This layer comprehends an application programming interface (API) with desirably low overheads. A RTK can exist as a microkernel real-time operating system (RTOS) (e.g. QNX (1998), CMX-RTX (1998) or µC/OS (1998)) or as part of a larger real-time operating system (e.g. Real-Time Linux (FSMLabs, 2002; RTAI, 2002), VxWorks (WindRiver, 2002), and OS9 (RadiSys, 2002)). These RTKs provide loosely coupled imperative primitives (e.g. thread control and synchronization, signal handling, and timed interrupt handling) which widen the number of features but complicate writing reliable concurrent software at the same time. No RTK is the same and their semantics and behaviour are not uniform for real-time systems. One may realize that this programming paradigm is meant to instruct the processor, but this paradigm is too low level and too detailed for the human understanding of the structure and behaviour of control software. Furthermore, the API incorporates ad-hoc solutions to pathological problems, such as deadlock and livelock, and priority

inversion. This is done by using respectively asynchronous communication and priority inheritance techniques to cure illnesses in the implementation rather than providing guidelines or rules to prevent them in the first place.

1.2.2 Computer-aided design tools

Computer-aided design (CAD) tools for developing control software, such as the Real Time Interface (RTI) from dSPACE (2002) and the Real Time Labview Module (RTLM) from National Instruments (2004), are world-widely known and used for creating control software. These tools deliver a powerful graphical development environment for signal acquisition, measurement analysis, data presentation, and model-based control system design. Essentially, they give the flexibility of a programming language without the complexity of traditional development tools and give all-in-one solutions for dedicated hardware. Their success is due to the automation of code generation and system monitoring, which negates the need for the control engineer to code. The tools are guided with documentation and are easy of use.

Under the hood of RTI and RTLM, these tools hide a rigid software framework that is performed by the embedded computer system. The core of the generated code is basically a simulator that comprises a sequential state machine, which is timed on constant timing intervals. This sequencing makes model checking for tracing pathological problems unnecessary, but the sequential framework becomes complicated when concurrency is unavoidable. Concurrency involves performing distributed tasks in parallel, triggering tasks at a specified time interval, and assigning equal or different priorities to each parallel task. This is specified in a process diagram, apart from block diagrams that describe the functionality of the controller. A process diagram specifies the tasks (specified by block diagrams) to be performed in parallel or on (timed) interrupts. Once the block diagrams and process diagram have been completed, the controller is up and running on the processor board by a few user actions.

Despite the flexibility and user-friendliness of these CAD tools, the concurrency paradigm that is used in the translation from a process diagram to its implementation is based on RTOS or RTK primitives. The semantics of these primitives may not be uniform and these primitives complicate the code. The automation from design to code, results in a one-way transformation disallowing round-trip engineering. Consequently, customizing the framework for an initially unsupported computer target is a costly task that is dedicated for specialists.

A concurrency paradigm should have been used that scales well with complexity, is highly portable, and that is uniform. CSP offers a concurrency paradigm that fulfils these requirements. CSP provides a good foundation for CAD tools to develop control software. Examples of which can be found in THESIS (1993) and Ptolemy II (2003). These methods describe two different control software design strategies. These strategies benefit from using CSP for building concurrent control software in an elegant way. THESIS and Ptolemy use simplified CSP constructs, which are restricted for a broader use.

1.2.3 Multithreading

Concurrent software involves multithreading. Multithreading is the ability to have more than one task occurring in a program (Lewis and Berg, 1996; Silberschatz and Galvin, 1994). A *thread* is a set of statements or coherent functions that execute sequentially at the same priority. Multithreading improves the utilization of a single or multiple CPUs, whereby the program can continue performing those tasks that are not waiting for an event to happen. This improves the throughput and responsiveness of the program.

A scheduler slices the main thread for each available CPU into multiple sub-threads (semi parallel). A thread is scheduled on a CPU and comprises a program context; i.e. the program counter, general registers, stack pointer, and the stack. At any one time only a single thread can be executed on a CPU. A multiprocessor system with n CPUs can therefore

execute *n* threads simultaneously (truly parallel). The entirety of a (sub-) thread is also known as a task.

Synchronization is required between multiple threads upon a shared resource. Two important synchronization primitives are semaphores (Dijkstra, 1968a) and monitors (Brinch-Hansen, 1973; Hoare, 1974). Both synchronization primitives offer various kinds of mutual exclusion constructs (or critical regions), where each thread may enter the construct (or region) one at the time. Hoare (1974) described a fair monitor as a concept for operating systems based on Dijkstra's semaphores. Derived monitor implementations are found in modern operating systems and in programming languages like Java (Arnold et al., 2000) and C# (Microsoft, 2003).

These synchronization constructs may depend on global conditions among different synchronization constructs. However, these synchronization constructs cause a few problems:

- The synchronization primitives intertwine with objects, which complicate the implementation of the objects. While the software grows, its complexity may not linearly scale with the growth. Consequently, the understanding and the verification of its correctness will become error-prone and hard to grasp.
- In case the synchronized resource is an object that mostly is accessed by a single thread, the synchronization construct decreases the overall performance.

Operating systems offer higher-level constructs that encapsulate semaphore and monitor constructs. Examples of such higher-level constructs are signal-handling, mailboxes, input/output-streams, and barriers (joins). The higher-level constructs make concurrent software less error-prone and they are applied when multiple threads are certainly involved. We observe these higher-level constructs as ad-hoc solutions to individual problems without collaborating to coherent concepts and without a formal mathematical foundation.

Hoare advocates reasoning about concurrency with processes and events, rather than with threads and monitors (Hoare, 1985). Monitors

are too complicated to understand the behaviour of a complex concurrent program. Instead, Hoare developed CSP as a formal foundation for describing concurrent systems. CSP is also multithreaded but it surpasses threads, semaphores, and monitors by defining channels and other related fundamental primitives. CSP provides all the syntactic and semantic information for describing and understanding concurrency based on fundamental and compositional semantics. CSP is further discussed in Section 1.4.3.

1.2.4 Occam and Transputer

The transputer and occam technology provided a simple and elegant platform for building sophisticated, reliable and robust control systems. The parallel programming language occam (Inmos, 1988) is an implementation of a subset contained in CSP. Furthermore, occam is a highly secure programming language, which detects hazardous or errors-prone concurrency constructs at compile-time. Transputers are microprocessors that are designed to execute occam programs most efficiently. Transputers are equipped with four links. Transputers are building-blocks in homogeneous multiprocessor systems based on distributed memory. A *link* is a peer-to-peer connection between transputers that provides external channel communication between processes distributed on a network of transputers.

The manufacturing of transputers ceased around 1996. Ivimey-Cook (1999) notes that the Inmos transputer was more than a family of processor chips, it was a concept, a new way of looking at system design problems. In many ways, that concept lives on in the hardware design houses of today, using macro cells and programmable logic. STMicroelectronics continues with the transputers core of the T414, in a low-cost chip called the ST-20, which is no longer referred to as a transputer. The ST-20 is nowadays sold as the STi5518 CPU which can be found in many TV set-top boxes and satellite receivers. Transputer links are found in other products. DEC's Alpha processor 21364 uses transputer-class links for building multiprocessor configurations (DEC,

2003). Transputer-class links are used for SpaceWire networks for the space industry (4Links, 2003; SpaceWire, 2003).

Since transputers, as such, have become obsolete, the programming language occam evolves slowly and is still supported by a small community in the world. Translators exist for porting programs that are written in occam to processors other than transputers. Two translators are available: Kent Retargetable Occam Compiler KROC (KROC, 1999) and the Southampton Portable occam Compiler SPoC (SPoC, 1998). KROC translates transputer code that is produced by the occam compiler to native code for a target processor and SPoC translates transputer code to portable C code. Recently, the occam compiler source code has been released. An updated occam compiler supporting an updated occam programming language may be expected in the future. Despite these efforts, the future of occam is very uncertain. Occam suffers from not being a popular programming language due to the fact that the concepts behind occam are not well known by most software developers and perhaps the syntax is not favoured among programmers. The programming languages Ada (Barnes, 1988), Limbo (Stanley-Marbell, 2003), and Handel-C (Page, 2001) have similar roots. Similar to the afore-mentioned reasons, Ada and Limbo also suffer from lack of worldwide acceptance. However, Handel-C is C-alike and grows in popularity for programming field programmable gate arrays (FPGA). On the other hand, the look-a-like syntaxes of C (Kernigham and Ritchie, 1988), C++ (Stroustrup, 2000) and Java are preferred and widely accepted. One can imagine that the C, C++ and Java community could benefit from CSP by providing a CSP library for these programming languages. Furthermore, such a CSP library should be suitable for heterogenous multiprocessor systems based on shared or distributed memory

1.2.5 THESIS

A sound and intuitive foundation for the realization of control software and hardware was described by Wijbrans (1993). This resulted in the *Twente Hierarchical Embedded Systems Implementation by Simulation* (THESIS) method, which investigated the use of parallel processing and

structured design methods for embedded control system realization. This method was aimed at filling the gap between the derivation of the control algorithms and the controller realization by recommending a strategy that guides the engineer during the design process, provides a formalism for the description of the controller, and suggests support tools that aids the engineer during the design process.

A control application is inherently data-flow oriented. Therefore, THESIS was naturally based on a channel-based methodology in order to guarantee consistencies and filling the gap between the different stages in the controller design and the final code. Wijbrans chose a software design tool based on the structured analysis and structured design (SA/SD) method of Hatley and Pirbhai (1987). The implementation and realization of the design is based on the parallel programming language occam and transputer hardware. Due to the CSP concepts that come with occam and transputers, this method resulted in reliable, robust, and well-structured real-time control software for various mechatronic systems at the laboratory of Control Engineering. THESIS was applied to several industrial applications (Wijbrans, 1993). The technical abstraction that comes with occam and transputers reduced complexities in implementing controller software. This increased the development speed compared to imperative programming, which uses sequential programming languages, like C or C++, with multithreading primitives. This is an important lesson we learn from occam and transputers.

The Hatley and Pirbhai method, occam, and transputers are considered outdated, taking into account the present trends in technology. These should be replaced by technology that keeps THESIS up-to-date.

1.3 Aim of research

Considering the scope of subject, the aim of this research is formulated as follows:

> *The aim of this research is developing a CSP-based methodology for building embedded real-time software for heterogeneous embedded control systems.*

This aim is multilateral because concurrency concerns all phases of software engineering. For each phase a sub-aim can be formulated:

- In the *specification phase* this research aims at identifying and specifying concurrency as part of the requirements.
- In the *design phase* this research aims at designing solutions to the problems, while maintaining concurrency and dealing with complexity reduction and absorption.
- In the *implementation phase* this research aims at the development of an object-oriented concurrent framework that protects the engineer from needing exclusive skills on programming threads.
- In the *realization phase* this research aims at porting the implementation to hardware so that the hardware is efficiently used and satisfies the required performance.
- The concurrent software should be systematically tested to determine whether or not the software satisfies the required specification.

The methodology should deal with common sources of complexities in programming concurrent software, such as multithreading, interrupt handling, exception handling, inter-processor communication, priority scheduling, reactivity, responsiveness, etc. These technical issues should be elevated to a high level of abstraction in order to simplify the design of the application and to simplify the required mindset of the engineer. The resulting concurrent software should be similar to that obtained by an experienced software engineer. A structured approach using sound

and proven concepts is required, which deals with these technical issues without introducing surprises, discontinuities, or non-scalable complexities. Such a structured approach should stretch over all phases of software engineering. The approach should provide an architectural view that binds all phases in the development process and can be implemented when completeness is achieved. The continuity and consistency between the different phases of software engineering should be guaranteed, which allows for rapid prototyping and round-trip engineering. The proposed methodology should provide the technical "how to's" for building concurrent software.

The proposed methodology is guided by the following goals:

- to make things reasonably safe but not too restrictive,
- to make compromises so as not to introduce unreasonable inefficiencies,
- applicable for real-time and embedded applications, in particular for control applications,
- and portable among different platforms.

It is expected, due to experiences with occam, that the CSP concepts will result in tools that obtain control software at a fast pace in development and at a moderate cost. The programming languages C, C++, and Java are of interest for coding control software since these programming languages are used and supported by the vast majority of embedded software engineering companies. The proposed methodology will be applied to several embedded control systems. During this research 20-sim (2003) is used for controller design and automatic code generation of the control laws.

1.4 Research approach

1.4.1 Complexity

The growth of software, in terms of size and the number of features it should perform, increases the complexity of software. The software engineer must deal with this complexity. Software design tools, modelling and programming languages, and software design methods are slowly evolving in order to simplify this task. However, the lack of common concepts that stretches over these tools, languages, and methods cause discontinuities between them. These discontinuities are hurdles in the software design trajectory. The solution to this problem is to eliminate discontinuities between the different models and phases in the software design trajectory. In order to understand what is required, we elaborate on complexity in this section.

Complexity is a conception that is related to the human intuition. One person may find something complex to understand whereas someone else may find it simple to understand. In order to understand this phenomenon of complexity, complexity is defined as follows:

Definition (complexity): *Complexity* is the amount of thought it takes a person to grasp a problem and/or to develop a solution to that problem.

The amount of thought depends on many factors which are human related, i.e. previous knowledge and the ability of complexity reduction (simplification through abstraction, generalization, or mental images) and complexity absorption (speed and capacity of remembering, followed by reconstruction). Complexity is something that is cognitive or subjective and can be different for each individual person or common to a group of persons who share similar skills. Complexity can be measured by comparison between two or more alternatives. Quantities have been proposed to measure complexity and capture all our intuitive ideas about what is meant by complexity and by its opposite, simplicity. Complexity is often measured by time measures or information measures (Gell-Mann, 1995). Time measures express how much time or steps it takes to

grasp the problem or to finish a computation. Information measures express the length of the shortest message conveying certain information. Complexity measures are context-dependent.

In order to manage complexity, a common context-related language is required, which the human mind can easily grasp with assumed previous knowledge and understanding. The language should advocate complexity reduction and complexity absorption, which eventually results in reasonably low complexity measures. Time measures and information measures are reduced by concurrency. In other words, concurrency manages complexity! Therefore, the language should incorporate concurrency. Most tools, languages, and methods lack a good understanding of concurrency and not surprisingly they fail to describe a concurrent system with low complexity measures. The UML is a good example of a common language that suffers from discontinuities between different diagrams (or views), due to a poor concurrency model. Its concurrency concepts do not stretch over the multiple views and this makes concurrent software complex and error-prone rather than simpler and safe. Apparently, a wrong concurrency paradigm has been used in the UML.

A mathematical foundation may contribute to a quality of understanding and reasoning about the behaviour of concurrent software. Often the results of mathematics can be summarized to formal and abstract concepts, implemented as practical constructs, guidelines, and rules. This is similar for CSP and its underlying theory. CSP comprehends a mathematical foundation, whereby simplification is achieved due to abstraction and a separation of well-defined concerns.

1.4.2 Concurrency

We live in a concurrent world where multiple tasks exist at the same time. These tasks are carried out in parallel, in sequence, or by some choice, and possibly communicate with each other. As in real life, clarity is obtained through concurrency. If one had to describe the behaviour of our environment as a strictly sequential model (or as one task) then this

would be too complex. Concurrent tasks exist at the same time, and they can be observed individually or in composition of other tasks. The existence of multiple tasks at the same time does not imply that these tasks are in parallel. Some tasks may be in parallel, some are waiting for another task to complete, or tasks are alternatively performed due to certain conditions. Parallelism implies that when a task has to wait for an event to happen, another task can continue. This most likely increases the overall throughput and responsiveness of an application.

Although the high performance and the simplicity of a computer is attributed to concurrent hardware, concurrency in software is often thought to be an advanced topic that is much harder than serial computing.

The term *concurrent* is often used as a synonym for *parallel*. These terms have something in common but their nature has different semantics. Essentially, concurrency comprehends more than just parallelism. For example, consider a simple system of two communicating computers in parallel. The parallelism is simply and solely not sufficient for understanding the behaviour of the system. More interesting still, is the understanding of the total behaviour of the system as an aggregation of sequential parts on each of the two computers, which are executed in parallel and that synchronize on communication. The elements sequential, parallel, synchronization, and communication are subject to concurrency in the system. Such a system is known as a *concurrent system*, where there is more than one process existing at a time, whose component processes interact with each other by communication. Concurrency accommodates common goals, whereas parallelism accommodates two or more independent goals with respect to performance requirements. The distinction between concurrency and parallelism helps a great deal in separating concerns in embedded system engineering. In this thesis, the term *concurrent system* is used when the parallel system is viewed as a whole and the term *parallel system* is used when the distribution of individual computers in the system is considered.

Concurrency is defined as follows:

Definition (concurrency): *Concurrency* is an abstraction of behaviour, where the system is viewed as a set of parallel, sequential, and alternative processes that interact with each other by communication.

In sequential programs, parallelism is replaced by sequential patterns of code which sequence is valid in parallel form. In a sequential language, communication is a combination of actions on shared variables or shared objects, whose actions are streamlined by a single sequential flow of control. In this sense, concurrency in sequential programs is done implicitly. Roscoe (1998) points out that this happens *too* implicitly in sequential programming languages. This becomes a major disadvantage when using a sequential programming language for creating concurrent (multithreaded) programs. Roscoe brings to mind that "*this effect also shows up when it comes to mathematical reasoning about system behaviour: when it is not made explicit in a program's semantics when it receives communications, one has to allow for the effects of any communication at any time.*". A paradigm that comes from true parallel systems (e.g. computer hardware, electronic components, etc.) benefits from concurrency and makes systems simpler. This illustrates that concurrency is too powerful and, indeed, too simple an idea to be set aside. With a better handle, it can simplify both the design and the implementation of most complex systems, as well as boost performance.

Concurrency should provide:

- a powerful tool for *simplifying* the description of systems;
- natural separation of concerns at the highest level of abstraction in terms of processes and their interrelationships;
- performance that spins out from the above, but is not the primary focus;
- a model that is mathematically, clean springs no engineering surprises, and scales well with system complexity.

The observable and fundamental entities of concurrent systems are events. An *event* is an occurrence in time and space, which involves two

or more processes that engage in the event. One process cannot engage in an event on its own. During the event, particular indivisible actions are performed. These actions are strictly concurrency related, such as, data transfer, synchronization, and thread scheduling. An event represents the completion or successful termination of its actions, which only occurs when all associated processes rendezvous with each other. An event is *not* an object, it is not simply a method call on an object, and it is not an expression that becomes true. An event can only occur on rendezvous between two or more processes. The term *process* will be explained in Section 2.3.1.

In this thesis, we distinguish between communication events, termination events, timeout events, and exception events. A *communication event* is the occurrence of two processes engaged in communication over a channel or barrier. A *termination event* is the transition from one process to a subsequent process. A *timeout event* is a rejection of communication when it is not ready before a specified time. An *exception event* is an internal event of a process that stops the process from making progress. Communication events and termination events are the primary events from which timeout events and exception events are derived.

1.4.3 Communicating Sequential Processes

CSP is a theory of programming and a notation for describing concurrent systems whose component processes interact with each other by communication (Hoare, 1985; Roscoe, 1998). Its concepts are based on mathematics and compositional semantics. In CSP, one can specify requirements precisely and prove that they are satisfied by our implementations. CSP is about 10 years newer than object-orientation and 5 years newer than monitors. The theory has evolved over time and its concepts are timeless (Hoare, 1978; Hoare, 1985; Roscoe, 1998; Schneider, 2000).

CSP deals with processes, networks of processes and various forms of synchronization and communication between them. A network of

processes is also a process and so CSP naturally accommodates layered (or nested) structures, i.e. networks of networks. A CSP process isolates data and operations from other processes. Its behaviour is completely described by the way it communicates with its external environment via *channels*. A channel performs a barrier synchronization between two processes. A *barrier* is a rendezvous point on which two or more associated processes are blocked until all processes reach the rendezvous point. A barrier can be represented by a single channel between two processes, by a bundle of channels between two or more processes, or by a parallel construct that terminates when all participating processes have terminated. Channel and barrier communication are observable as communication events.

Processes are components that are complete and have no complex dependencies on other components. The definition of a process comprises syntactic and semantic information on how the process interacts with its environment. This information is entirely specified through a defined interface—its abstraction—consisting of various synchronization primitives as defined in CSP. The synchronization primitives encapsulate the principles of multithreading and brings about channels, barriers, and binary operators. These binary operators are represented as compositional constructs.

CSP is founded on what is called *compositional semantics*. CSP offers three distinct ways of describing the meaning of a program, namely operational, denotational, and algebraic semantics. The operational semantics interprets programs as state diagrams. The denotational semantics maps a language into an abstract model in such a way that the value (in the model) of any component is determinable directly from the values of its immediate sub-components. These values are based on traces, failures, and divergences (Roscoe, 1998). Algebraic semantics are defined by a set of algebraic laws. Each semantic complements each other. The mathematics are left to the CSP books (Roscoe, 1998; Schneider, 2000) and model checking tools (FDR, 2004; ProBE, 2003), but its assets should be brought to practical use for the user to describe the compositional semantics of the software suitable for formal analysis.

1.4.4 Strategy

A concept is an abstract principle with a well-defined semantics. A concept is less sensitive to changes than its implementation. After all, good concepts live longer than their implementations. Concepts are the building-blocks of a methodology. Therefore, the quest of this research is to develop an implementation using modern technology that replaces occam and transputers while maintaining the invaluable CSP concepts. The implementation of the CSP concepts is the foundation for the proposed methodology.

The aim of this thesis is accomplished by realizing the following goals:

1. A CSP-based graphical modelling language should be defined for specifying and designing control architectures using graphical notations.

2. A object model should be developed that implements the CSP concepts using object-oriented techniques. The object model should be abstract, but eventually the model must be implemented in the programming languages C, C++, and Java. This should result in three CSP libraries for C, C++ and Java. Currently, the Java run-time environment takes significant run-time overhead, making it unsuitable for embedded real-time systems. Therefore, the CSP libraries for C and C++ are meant to boost performance on embedded systems.

3. Demonstrate the proposed methodology on several control systems at the laboratory of Control Engineering.

The object model is prototyped in Java and partly documented using UML diagrams. Java was used for the following reasons:

- Java is popular and world-widely supported,
- Java is better than C++ (i.e. simpler, safer, C look-a-like),
- Java was meant for embedded systems,
- Java is object-oriented and multithreading is supported within the language and the Java run-time system.

The prototype in Java was meant for educational purposes and from which C and C++ versions should be derived. A version in C and C++ is required in order to boost performance. UML diagrams are used to document the object model.

Valuable aspects of component-, process-, and object-oriented technologies should be incorporated in the object model. A relationship with other methodologies, being suitable for creating control software, should be maintained. Methodologies of interest are UML (1998), ROOM (aka RT-UML) (Selic et al., 1994), Octopus (UML version) (Awad et al., 2002), Ptolemy II (Ptolemy, 2003), and structured methods (Hatley and Pribhai, 1987; Yourdon, 1989), and THESIS (Wijbrans, 1993).

1.5 Overview of thesis

The realization of the postulated aims is described in the following chapters:

Chapter 2

A structured approach to embedded control systems implementation is discussed. This chapter emphasizes the importance of processes and events during the development of control models and control software.

Chapter 3

A graphical modelling language for specifying and designing process architectures is described. The graphical notations are derived from CSP which allow the user to convert data-flow oriented control models into executable models, called CSP diagrams.

Chapter 4

A CSP library for Java is described. This library renders the application programming interface (API) of the object model in Java. Also, the library stands model for implementations in other object-oriented programming languages. CSP diagrams can be straightforwardly implemented with this CSP library to Java. The implementation of the API is not described in this chapter.

Chapter 5

Real-time behaviour is important and usually priorities are the solution to allow real-time processes to meet their deadlines. A notion of priorities for CSP-based software is described. Priorities are supported by CSP diagrams (discussed in Chapter 3) and are implemented in the CSP libraries for Java and C++ (discussed in Chapter 4 and 6).

Chapter 6

The CSP diagrams and the CSP library for C++ have been applied to several embedded control systems. Several applications illustrate how CSP diagrams and the CSP library for C++ contribute to the development of reliable and robust control software.

Chapter 7

The results of this research are reviewed. Conclusions and recommendations are subject of this chapter.

CHAPTER 2

A Structured Approach to Embedded Control Systems Implementation

2.1 Introduction

Control systems are concurrent systems, which involve processes deployed in hardware and in software. These processes perform tasks at periodic intervals (e.g. sampling, actuation, and data processing) or at sporadic stimuli from the environment (e.g. mode-switching, safe-guarding). These processes must guarantee real-time constraints; e.g. reactivity, responsiveness, and deadlines. The control software integrates concurrency related concerns, such as multithreading, interrupt handling, exception handling, timing, and scheduling. These concerns propagate through the design and implementation of control applications for which an appropriate understanding of concurrency is crucial to their development.

The above mentioned concurrency related concerns can complicate the implementation of the execution framework when these concerns are treated as ad-hoc solutions. The transformation of the controller design to its implementation is usually automated, which hides the complicated code framework from the user. Consequently, the user has control over the objects in the design, but the user has restricted control over the execution framework and its performance. This automation is not a problem unless its restrictions become a burden. This can cause a serious

gap between a controller design and its implementation on dedicated target platforms. A structures approach is required that is based on sound and formal concepts that enable the integration of concurrency related concerns without surprises.

The user is primarily focussed on processes during the design of a control system, and secondarily on objects. A control system is a process of one or more control loops. Each control loop is a process that is performed partly in software and partly in hardware. The control engineer is concerned with a hierarchy of processes during specification, design, implementation, and verification by simulation of control system. It are those processes that describe the behaviour of the system. Processes and objects go hand in hand, whereby objects implement processes. Therefore, a control system is not solely described in terms of objects, as some object-oriented engineering approaches do suggest. The THESIS method (Wijbrans, 1993) showed that process-orientation offers a structured approach to control system design and implementation. Although THESIS is more than 10 years old, most of its propositions are still applicable to the methodology presented here.

This chapter emphasizes the importance of process identification, process architecture design, and process analysis for the development of control systems. This process-orientation constitutes to a structured approach, which involves the different disciplines in control system development. The process architecture design elevates the previously mentioned concurrency related concerns to a higher-level of abstraction that is in control of the user. The structured approach, presented here, follows the chain of thoughts behind the THESIS method.

The conceptual design of control software for mechatronic systems is discussed in Section 2.2. Several disciplines in the control system design trajectory are distinguished. The emphasis is on identifying processes that are subject to refinement. In Section 2.3, the importance of processes and process architectures in the development of control software are discussed. The THESIS method is briefly described in Section 2.4. Conclusions to this chapter are described in Section 2.5.

2.2 Conceptual design for controller software of mechatronic systems

2.2.1 Multidisciplinary design approach

In a mechatronic systems' approach, the dynamic properties of the total system play a central role. The controller being designed is in general a dynamic process, which is connected to dynamic processes that model the physical system to be controlled. These processes are the building-blocks that manifest a process architecture that describes the entire behaviour of the system. Therefore, a process architecture encompasses a multidisciplinary design approach and may involve a team of people. Specific design methods and design tools are required for processes in specific domains, which are tailored to the type of system for which they are intended.

A typical process architecture of a control application starts with three kinds of processes; controller processes, physical processes, and supplemental processes. Figure 2-1 illustrates a process architecture of processes (rectangles) and their communicational relationships (arrows).

Figure 2-1 *Context diagram (part 1 of 2) of a control system showing communication relationships between the controller, physical, and supplemental processes.*

The *controller processes* are comprised of a variety of separate controllers. The *physical processes* separate behaviours of a real plant to be controlled. The *supplemental processes* are responsible for other things, such as, maintenance, data analysis, diagnostics, repository, or user monitoring and commands. Note that a *control loop* is described by a *control process*, which involves a controller process and one or more physical processes.

Figure 2-2 illustrates another view of the process architecture, with the same processes, but with compositional relationships (connections). This figure specifies that all three kinds of processes run in parallel and, together with Figure 2-1, it specifies that these processes must synchronize on communication.

Figure 2-2 *Context diagram (part 2 of 2) of a control system showing parallel relationships between the controller, physical, and supplemental processes.*

Both Figure 2-1 and Figure 2-2 show a top-level design, also called a *context diagram*, representing the fundamental processes to be designed within a particular context of the system. The graphical notation used, along with their semantics, is explained in Chapter 3. This context diagram portrays a *CSP diagram*, providing uniform connections between the concurrent processes. Also, these connections depict the relationships between the different disciplines. The context diagram is a joint point of departure for the different design disciplines. It is possible that, for every control loop, a separate context diagram can be designed and combined together.

A process architecture acts as an intermediate in a multidisciplinary design approach, which

- captures concurrency in the system,
- endures the stages in the design process,
- supports stepwise refinements.

Concurrency is an inherent part of control systems and establishes a clear separation of concerns within the stages of control system design, process architecture design, and the underlying stepwise refinement. The

stages of control system design are discussed in Section 2.2.2. Stepwise refinement is addressed in Section 2.2.3.

2.2.2 Control system design trajectory

The control system design trajectory is partitioned in the following four stages (Broenink and Hilderink, 2001):

- *Physical-System Modelling*—The dynamic behaviour of the system is object–orientedly modelled, using a port-based approach (e.g. bond graphs) as a main modelling paradigm.
- *Control law Design*—Using the model acquired in the previous step or a simplified version of it, control laws are designed.
- *Embedded Control System Implementation*—Transforming the control laws into efficient concurrent algorithms (i.e. computer code) is guided via a stepwise refinement process.
- *Realization*—The realization of the control system is a sequence of refinements that deals with the limitations, technical issues, and the behaviour of an embedded computer system.

An overview of these stages is depicted in Figure 2-3.

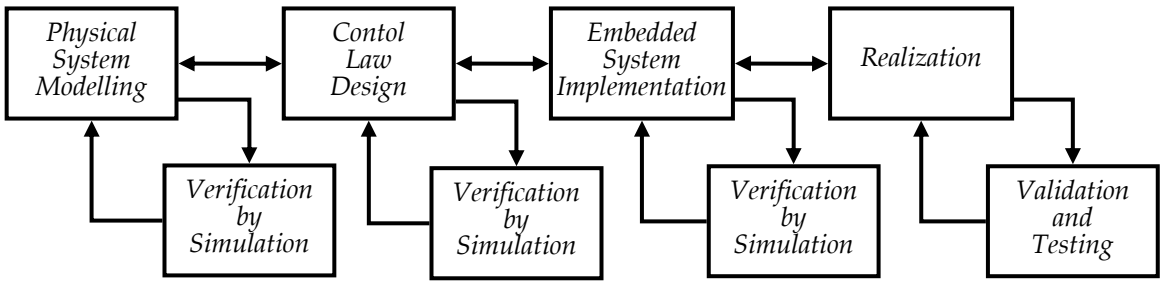


Figure 2-3 *Control system design stages*.

After each step, the results are verified by simulation, or validated by experiments, on the real embedded computer system. Verification determines if "the product was built right" and validation determines if "the right product was built". Verification is a process of testing,

inspecting, comparing, and analyzing, which determines whether or not the product or model of a given stage, fully implements the specified requirements of that stage. Validation is a verification process, which comprehends the evaluation of any misfits between the system requirements and the system, in the real-world. Validation demonstrates the usefulness of the product, i.e., the embedded system. Validation usually takes place at the end of a development trajectory of a prototype or the final product and looks at the complete system as opposed to verification. Complete validation requires the validation of the requirements, in order to determine whether or not the right product was built; validation of the validation. Verification and validation are essential processes in stepwise refinement. A positive result of verification or validation is a permit to go to the next stage in the development process; otherwise further refinement is required.

The previously mentioned stages will be detailed in the following sub-sections. It may then become clear that the notion of processes is essential to each of these stages.

Physical-system modelling

The physical system, which is to be controlled, is preferably modelled in a *process-oriented* way; since a physical system exists of 'real-world' processes, in which 'real-world' objects participate. The purpose of physical-system modelling is to create a *competent* model of the system under study. A competent model is a sufficiently detailed or qualified model of the physical system that captures the relevant dynamic behaviour of that system. It can serve as a kind of physical system replacement.

Competent models are created for at least three goals, namely:

1. understanding the dynamics of the physical system,
2. structuring the functional and non-functional requirements,
3. deriving control laws.

Hierarchically structuring models is necessary since models are of non–trivial complexity. This also implies that the encapsulation of model details should be provided at an appropriate degree, so that extensibility, maintainability, and reusability can be achieved. Commonly, port-based elements and block diagram elements with parameters that are directly related to the physical properties of the system are combined to form one model, which serves all three goals. In this, a model is composed of sub-models. Each sub-model manifests a separate concern and is encapsulated by well-defined interfaces. The connections between these interfaces are signals or ports that exchange energy (bilateral or power conjugate pairs of variables). The sub-model formulae are written in a declarative style, i.e. as equations in the mathematical sense and not as assignment statements. For an introduction in physical modelling, refer to van Amerongen and Breedveld (2003).

Control law design

The control laws are subject to implementation in software. Often, a simplified and linearized version of the physical-system models is used for deriving control laws. The interdependencies between the physical system design and the control law design make the modelling process iterative. This puts an extra demand on model extendibility and maintenance.

The following, rather common, procedure of control law design is phrased:

- *Generate competent model(s)*
  A competent model represents the physical system to be controlled. The competent model (either reduced automatically by linearization and/or order reduction, or diminished by hand) serves as a substitute for the physical system when the control laws are designed. It may be necessary to have more than one simplified model to cover the whole workspace of the control system.
- *Verify competent model(s)*
  A competent model is simulated to check whether or not the

model satisfies its goals. The model should be verified (i.e. its simplification should sufficiently reflect the dynamics of the system) and validated (i.e. compared with measurements on the real system).

- *Derive the control law(s)*
  Using standard procedures, a control law is now designed using the model(s) acquired in the previous stage. It is also possible to derive a set of control laws, each having its own operating domain (state invariants). This can make each individual control law simpler or give it a better performance. Additionally, switching from one control law to the other must be designed. It may be required that switching behaves smoothly from one control law to the other: i.e. bumples transfer (Hilhorst et al., 1994; van Breemen, 2001).
- *Verify the control law(s)*
  Construct a test bed in which the control law is connected to the model. Verify the control laws by performing simulations. Run experiments in such a way that the demands on the controller performance can be checked. Arriving at this stage, the control laws, together with the model, can be used in the process of embedded system implementation.

This procedure results in a process architecture as discussed in Section 2.2.1. The controller processes can be comprised of loop control-, sequence control-, or supervisory control processes (Wijbrans, 1993). *Loop control* performs digital control algorithms. *Sequence control* guides sequences of operations, based on logical actions in time. *Supervisory control* contains optimization algorithms (e.g. adaptive or self-learning) or expert systems (e.g. knowledge-based) that generate (optimal) input signals for the control loops or adapt parameters of the control algorithms. The physical processes describe the behaviours of the mechatronic system or the plant to be controlled. The supplemental processes may provide user interaction, which can have influence on the behaviour (or mode of operation) of the control process.

In this thesis, the design and simulation tool 20-sim is used for designing and simulating physical-system models and for deriving control laws.

Embedded system implementation

The entire process architecture is subject to implementation in software, hardware, or in both. Physical processes are already part of the hardware, e.g. the mechanics. Supplemental processes may also exist in hardware or in software. The controller processes are subject to implementation in software. Broenink and Hilderink (2001) proposed a procedure to structure the implementation process for the controller part. The procedure is divided into four concerns:

- *Integrate control laws*
  The controller processes are the central and embedded part of the entire system. After the control laws have been designed and verified by simulation, they need to be implemented on the embedded control computer. Control laws for different situations are combined with sequence or supervisory control processes. The computation of the algorithms is influenced by the resolution and truncation of values or mathematical functions. The errors caused by numerical integration methods are also taken into account. The sensors and actuators are assumed to be *ideal*. The traces of events are also considered *ideal*; i.e. no event will be refused.
- *Capture technology–independent functionality*
  Facilities for safety, maintenance processing, data repository, and user-interaction functionality are added. These supplemental processes should be independent of the behaviour of the control processes. Note that supplemental processes consume processor time, so these processes may have influence on the overall performance of the system. Furthermore, the execution framework of the control software should be independent of the underlying operating system.
- *Capture technology–dependent functionality*
  The specification is augmented with the *non–idealness* of sensors, actuators, and events. The operation of sensing and actuating is no longer considered *ideal* or *faultless*. Characteristics of the input and output devices are added to the description, e.g. delays, quantization, and discretisation on analogue-to-digital

and digital-to-analogue conversions. The algorithms are no longer *faultless* and should be protected by integrity constraints to prevent or to handle illegal states. Hardware is drift-sensitive in terms of aging, temperature, and wear-out. Furthermore, the environment in which unforeseen errors can occur is not ideal, especially when the non-idealness of events means that events may not occur due to defects in the system. Escape routines are required to take appropriate actions, such as error-recovery or graceful termination.

- *Capture timing characteristics*
  Timing is completely event related. The events on which the control laws are performed are periodic or sporadic and the computation is associated with completion-time. Therefore, controller processes are usually hard real-time processes whose outputs must be computed before the next sampling interval; i.e. missing any deadlines will result in an error. Scheduling techniques and/or algorithm optimization techniques may be used to obtain a viable performance. The notion of priority is a solution to deal with the limited processor time in order to guarantee that processes will meet their deadlines. Priorities and buffering techniques decouple critical processes from (non-real-time) support processes. This also deals with communication latencies between processes.

In each of these concerns, a good understanding of concurrency is required. A change in the process architecture means a change in the implementation and visa versa. A one-to-one mapping between the process architecture and its implementation, while maintaining the semantics of concurrency, makes the development trajectory predictable.

Realization

After the process architecture and the control algorithms have been coded by the previous implementation phase, one can work towards realization on the target computer and physical device.

A stepwise approach is advocated, whereby a *real* embedded system is divided into four main parts:

- *Embedded computing*—This manifests the embedded computer system which consists of *all* computing functionality that performs the control algorithms.
- *I/O interfacing*—Input-output (I/O) interface boards connect the plant to the computer system. Specific operating system resources, namely *device drivers*, are required for the program to address these I/O interface boards. In case interface boards are equipped with one or more processors, the control algorithms may be distributed over the interface boards.
- *User interfacing*—The user interface connects the computer to the human world. It may be required to monitor internal values (signals and parameters) of the controllers or to command the controller. This feature is often used for validating the systems behaviour, maintenance, collecting data, or for external mode-switching by the operator.
- *The plant*—The plant is the mechatronic system with actuators and sensors connected to the embedded control system.

The process architecture, starting from a context diagram as shown in Figure 2-1, takes these four parts into account during the design of the control system. The controller processes are mapped on the embedded computing part. The supplemental processes provide the user interfacing. The communication relationships implement the I/O interfacing. The physical processes are part of the plant.

The implementation on the *real* embedded system proceeds with precaution and safety, so that the variables do not get out of bound, or the system cannot hurt anyone or damage itself. Similarly, some parts of the plant that are not yet available can be simulated.

Before the final realization, a method of testing the implementation of embedded control systems can be carried out using a *hardware-in-the-loop simulator*. Hardware-in-the-loop simulation (HIL simulation or HILS) is characterized by the operation of real components in connection with

real-time simulated components (Isermann et al., 1999). The input and output of the real-time simulated processes show the same time-dependent values as the real physical processes (Isermann et al., 1999; Sanvido and Cechticky, 2002). The idea of HILS is to replace the real physical process with a simulated physical process. The embedded control system communicates with the simulated physical process, as if it were communicating with the real physical process. The simulated physical process performs the dynamic model, which represents a part of the real plant. The mathematical model can be altered to induce faults, which the controller process on the embedded control system must anticipate. Any incorrect behaviour of the controller process cannot cause damage and this improves the understanding of the behaviour under different working loads and conditions. In Isermann (1999), two additional variants of HILS were mentioned, which are

- simulated controller processes and real physical processes (called control prototyping),
- simulated controller processes and simulated physical processes.

These approaches may be required if the computer hardware and/or the plant is not available, or the examination on behaviour before HILS is considered.

HILS is a verification process that tests the completeness of the controller processes and eventually ensures that the system is correct, even in dangerous situations. The following two steps are missing in the verification process which should be taken into account.

1. The physical-system model should be validated to ensure that the model does represent the real system, even in critical situations.

2. The HIL simulation should be validated to ensure that the right conditions are tested and no conditions are forgotten.

A diagnostic process can probe the responses from the controller processes and the physical processes. The diagnostic process aims at detecting correct/incorrect responses from the embedded control system.

Fault generation and diagnostics can be automated by tools and the use of a fault specification language (Sanvido and Cechticky, 2002). The diagnostics of the embedded control system is orthogonal and should not alter the design of the controller processes or the (simulated) physical processes.

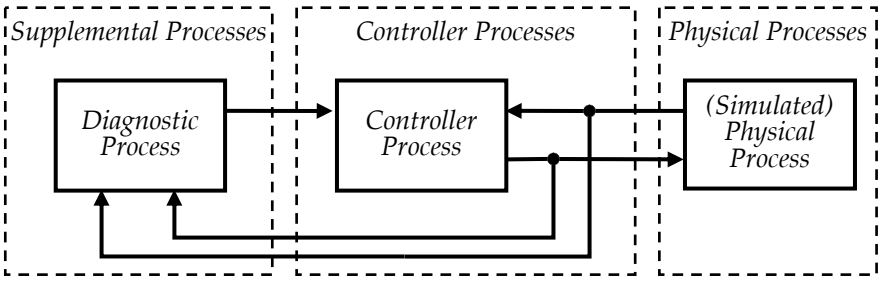


Figure 2-4 *Context diagram refined with a diagnostic process (communication view).*

In Figure 2-4, a refinement of the context diagram in Figure 2-1 is shown that is suitable for HIL simulation. The graphical notation is explained in Chapter 3. Here, a diagnostic process observes the communication between a controller process and a (simulated) physical process. If the diagnostic process detects a fault, it can ask the controller process to correct the error or ask it to stop. In order to complete the process architecture, a compositional diagram must specify that these processes are performed in parallel, as in Figure 2-2. This is explained in Chapter 3.

In the proposed methodology, any process in the process architecture can be replaced by a real process or a simulated process. The hardware interface of the embedded control system is between the controller processes and the real or simulated physical and supplemental processes. The arrows are channels that synchronize the processes on communication and these channels encapsulate the hardware interfaces from the processes. In the context diagram, channels are responsible for sampling and actuation. The channel model abstracts away from the real or simulated devices and are observed as (periodical or sporadic) communication events at this stage of design. This channel model simplifies the development of the process architecture. The channel model is based on the CSP channel communication model and therefore

the semantics of the process architecture are uniform. The controller processes automatically block on these channels for the next sampling interval. Once these channels become ready (e.g. after the device has performed the sampling or actuation on a timed interrupt), the processes continue and will be blocked until these channels become ready on the next interval. These timed channels perform sampling and actuation at precise intervals (jitter-free), whereas the wakeup of processes does not have to be precise. Processes must make sure that their outputs are available before their deadline, i.e. before the next interval. A CSP diagram enables the user to understand the concurrent behaviour under timing constraints, early in the control system design trajectory.

2.2.3 Stepwise refinement

Stepwise refinement is all about *improving* specifications. Refinement is the process of adding information, whereby the requirements, design, and implementation become more precise. This usually results in deterministic structures of information, shaped in executable or simulateable models. Intermediate or final models must be verified for correctness and consistency between refinements or development stages, before carrying on. The growth of information should be managed by appropriate hierarchical structures (abstraction) and proper separation of concerns.

The process of improvement involves the removal of uncertainties or non-determinism by functional decomposition. Sequencing activities is often applied to remove non-determinism. Sequential constructs, which service non-deterministic (e.g. simultaneous or alternative) inputs and outputs, are usually complex. Instead, abstraction should capture non-determinism, which should maintain in the trajectory of refinement.

At the implementation phase, the right choices must be made to deal with non-deterministic behaviours using appropriate deterministic constructs. Such deterministic constructs are defined by CSP and are discussed in Chapter 3 and 4. CSP comprises simple and busy-polling-free constructs that boost the reactiveness and responsiveness of the

program. The CSP paradigm helps a great deal in detecting and solving pitfalls in the process of refinement. The solutions are orthogonal to the design of the process architecture and can be applied in a later stage of the design trajectory. For example,

- Parallel activities can be systematically sequenced for boosting performance in circumstances where context-switching does not contribute to better performance.
- Event handling can be prioritized so that urgent events are handled first.

The real world is concurrent and not entirely described in terms of deterministic structures and behaviours. For example, the occurrences of sporadic or simultaneous events are non-deterministic in relation to time. Non-determinism is a natural phenomenon in which behaviour can be a requirement on its own. For example, buttons on a system can be pressed by a user in any order or simultaneously, which should not cause an error in the program. This is similar to a control system that has to handle multiple control loops at multiple frequencies. With the right abstraction, like using the CSP paradigm, concurrent behaviours can be described and maintained in the process of stepwise refinement. In other words, identifying deterministic and non-deterministic behaviours in a system is fundamental to stepwise refinement. Stepwise refinement should start with process-orientation, which provides the fundamentals for capturing the desired behaviours of the system. At the design stage, one should tend to stick to a more abstract, if necessarily non-deterministic, definition of processes. The deadlock and livelock issues will usually be addressed at this point. In this way, one can build robust programs for which deadlock-freedom cannot be compromised by implementation decisions at a later stage (Martin and Jassim, 1997).

Furthermore, the process model and channel model in CSP allow for prototyping on the basis of partial products. Consequently, stepwise refinement is a predictable process without engineering surprises. Each step in the refinement undergoes process analysis where something complex is studied and examined by separating it into more simple processes, as opposed to synthesis.

2.3 Processes are in control

2.3.1 Processes

The term 'process' is frequently used throughout this thesis. Several different kind of processes are mentioned, such as controller process, physical process, supplemental process, software process, hardware process, compositional process, and more. These are processes in different contexts with different objectives to achieve and different implementations. However, they share the term 'process', which means that they must have something in common.

In each different semantic of the word 'process', as can be found in Webster's dictionary, a common property is that a process describes some *progress whereby something is to be done with a certain goal to achieve*. Another property of a process is that *it may or may not react on certain events*. The behaviour of a process describes the evolution of a system in time and, in particular, how it interacts with its sub-systems and the environment. Furthermore, a process is described by simpler processes.

A generalized definition of the term 'process' is as follows:

Definition (process): A *process* is an independent self-contained entity that performs one or more task, within its private workspace, to achieve a joint goal, and during its progress it may or may not interact (engage in events) with its environment by means of communication.

This definition applies to the variety of processes, as mentioned in this thesis, for which they can be viewed at the same level of abstraction. The objective and behaviour of a process becomes clear when viewed in a concurrent system where processes are viewed in relation to other processes. See Section 1.4.2. Once a process is created it exists. The relationships between processes determine when a process starts or when it has to wait for an event to happen. Once a process has been started, it is in progress (even when it waits for events) until it terminates. A precise semantics of processes and their interrelationships are elaborated on in Chapter 3.

2.3.2 Identification of processes

In the context of software engineering, process identification is a term for identifying tasks, being relevant to the system, with joint goals, and operating in the same domain. This results in well-defined boundaries that encapsulate the workspaces of the joint tasks. Each boundary specifies a process. Each process has a unique identity. The identities must prevail in all stages of the development trajectory; i.e. from specification to its realization. Process identification is a hierarchical and decompositional process, for which processes are described by simpler processes. This results in a process architecture, which encompasses all relevant functional aspects of the system at different levels of detail. A process architecture is multidisciplinary. Typical domains in embedded systems are software, computer hardware, mechanics, analogue electrical elements, and FPGA's.

The identification of processes starts with extracting canonical tasks that are most obvious to be performed by the system. A *task* is a collaboration of activities which aims to achieve a common goal within a (yet) undefined boundary. A task is underpinned by a task description in the solution domain of the application that is under construction. Those tasks that are identified as building-blocks become processes in the process architecture. The identification of processes is a stepwise refinement trajectory until all the decisions and responsibilities that are taken by the application are captured. The identities of processes are distinguished by their existence and not by their descriptive properties. Process identification is intuitive and it comes in various flavours, such as agents, actors, capsules, active objects, etc. In this thesis we will abstract away from these flavours. Instead, we will use the term 'process' as defined in Section 2.3.1.

A task that fulfils one of the following observable patterns can be identified as a process.

- the activities of a task operate on the *same private data structure*,
- a task that *reacts on events* (event-handling),
- a task has a particular *deadline* before completion,

- a task has a different *priority* than another task,
- a task can be performed *in any order* with respect to another task,
- a task *communicates* with a task in a different domain,
- a task plays a *unique role*,
- or a task is *mobile* or *autonomous*.

Identifying processes can be performed in a systematic way by means of task-based reasoning, communication-based reasoning, and compositional-based reasoning. The latter two approaches identify the interrelationships between tasks, which eventually results in process identification.

- *Task-based reasoning* concerns the operational and logical activities in the software system. This approach is an inside-to-outside view of tasks, with the initial focus on the inside of a task towards its boundary. Tasks with well-defined boundaries are likely to be identified as processes. The boundary of a process marks the goal and responsibilities of the inside tasks, separated from other processes.
- *Communication-based reasoning* concerns the message-flow between independent tasks. This message-flow oriented approach is an outside-to-inside view of tasks. The message-flow influences the behaviour of tasks without changing their goal and responsibilities. This identifies two separate processes, one at the sending-end and one at the receiving-end of the message-flow. The objects that are required to establish the message-flow are identified as channels or barriers. Identifying channels or barriers will identify the boundaries of tasks and thus this will eventually identify processes at each end of the communication interrelationship (message-flow). Channels and barriers are explained in Chapter 3.
- *Compositional-based reasoning* concerns the control-flow between independent tasks. The control-flow can be composed by sequential control-flows, parallel control-flows, and choices between multiple control-flows. Tasks that spawn from other tasks (forking), tasks that join to a single task, or decisions made

by the tasks to perform sub-tasks, are symptoms of process identification, based on their compositional relationships. Every exception handling construct identifies two processes, namely the task that can be in exception and the exception handling task.

Process identification and process analysis are essential and an ongoing discipline in the stepwise refinement of control system design. This is essential not only for managing complexities but also to be able to understand its implementation and realization.

2.3.3 Process Analysis

Process analysis is a study or examination of the behaviour of an entire process and is divided into sub-processes. It investigates these sub-processes and their interrelationships to create a picture of its behaviour as a whole. In other words, process analysis partitions the system behaviour into deployable and concurrent processes. This can be based on particular approaches where processes are given the role of agents, actors, capsules, or active objects.

Process analysis manifests in:

- *Requirement analysis* is the process of extracting the desired requirements and structuring them into a comprehensive form. The requirement analysis is, in its very essence, a functional description of the system and relies heavily on task decomposition. The structural units of this decomposition are behaviours, functions, and tasks. These structural elements are arranged into *system models*. The system model along with the requirements model forms the system specification. System models should meet the functional and non-functional requirements at the same time (Douglass, 1999). Stepwise refinement should obey the functional and non-functional requirements of the system. *Functional requirements* are the expectations of system behaviour as viewed from outside the system. These requirements outline the transformation

behaviour from input to the output of the system. *Non-functional requirements*, also known as *quality of service* (*QoS*) *requirements*, specify the necessary performance, reliability, robustness, accessibility, usability, and safety of the functional requirements. Non-functional requirements concern the environment in which a system evolves. Non-functional requirements manifest important design and implementation decisions.

- *System analysis* partitions the system models and requirements into mechanical, electronic, and control components using structural elements. In control system design, structured elements are ideal physical model (IPM) elements, bond graph elements, and block diagram elements. The system requirements specify the constraints within these domains.
- *Object analysis* identifies the structural units of object decompositions (classes and objects), identifies the organizational elements (packages and components), and the relations among these elements. Objects allocate the functional structure of methods and data that implement the processes.

These analysis processes can be carried out in any interleaving order and do not manifest a waterfall approach. Process analysis has a global character applied to all stages in the development trajectory and therefore prohibits discontinuities between the stages. Hence, the absence of discontinuities determines the quality of analysis.

2.3.4 Process Architecture Design

The processes and their interrelationships that must be performed by the system are the building-blocks that manifest a process architecture. A process architecture describes the entire behaviour of a concurrent system. A process architecture is a structured computational model that is executable or simulatable.

A good architecture comprehends the essential and behavioural framework on which all other aspects of the system depend. Its structure connects the identified processes in the system, regardless if these

processes reside in software or in hardware. Furthermore, a good architecture simplifies the construction of the initial system and readily accommodates changes forced by a steady stream of new requirements.

A process architecture abstracts away from objects. Objects structure data and code while processes structure behaviour. Objects and processes are related in the sense that processes can be implemented with objects and objects are part of a process. Unlike objects, processes embrace observable properties of a concurrent program, such as reactivity, timeliness, responsiveness, priorities, and performance. These properties are essential for creating real-time applications.

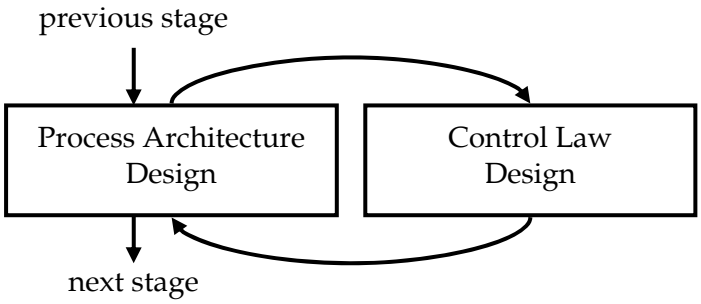


Figure 2-5 *Interleaving between process architecture design and control law design (state diagram).*

Process architecture design is fundamental to control engineering. It is always part of the mindset of the control engineer. The control system design trajectory is a continuous interaction between process architecture design and control law design. The start-point and end-point take place at the design of the process architecture. See Figure 2-5. The design trajectory starts with an initial process architecture, i.e. its context diagram.

The user starts with identifying the control loops and identifies the associated controller processes and physical processes. This may also include supplemental processes. After the context is created each process must be further refined by sub process architectures or detailed by a specific design method. In this latter case the control laws are designed for controller processes and port-based models are designed for describing the physical processes. The verification of the process

architecture is the last step in this design trajectory. Verification of the process architecture also implies verification of the integrity of processes since the total behaviour should be satisfied before continuing to the next stage. The next stage is usually the implementation of the process architecture.

The implementation of a process architecture results in an execution framework. Process architecture design is essential in control system design and should be available to the user who is responsible for the resulting execution framework. For example, the tool suite dSPACE (2002) includes process architecture design using process diagrams, so that the user can specify the concurrent processes that must be ported to the concurrent target system. The user has some control over the execution framework. However, the solutions are quite practical with no formalism. On the other hand, the modelling and simulation tool 20-sim does not include process architecture design and the execution framework is entirely hidden by the simulator or hidden by the code-templates when code generation is used. Consequently, the user has too little influence on the execution framework. Nevertheless, 20-sim is meant as a modelling and simulation tool with the capability of code-generating control laws to C. The resulting C-code is commonly suitable for third-party frameworks.

The semantics of process architectures should be based on a mathematical formalism that allows model checking and formal process analysis. A precise understanding of concurrency is important and therefore CSP is used for this purpose. In this thesis, a process architecture is represented as a CSP diagram which graphical notations implement the CSP concepts as described in Chapter 3. In Chapter 6, 20-sim is used together with CSP diagrams for designing and implementing control software.

2.4 The THESIS method

THESIS prototypes a design method that aims specifically at the realization of embedded control systems, in order to handle the inherent

complexity of these systems. The main objective of THESIS is to support the engineer in eliminating design and coding errors and to reduce development costs. This is accomplished by the following three mainstreams:

- *Mathematical formalism*
  Formal handling of concurrent and real-time behaviour using a mathematical formalism is important for proving that the behaviour of the software satisfies the requirements. The formalism should be operational in such a way that the specification itself is executable. Wijbrans recommended the theory of CSP, because this formalism can be used both for designing and analyzing concurrent systems.
- *Data-flow modelling*
  THESIS describes a data-flow-oriented graphical specification method based on a Structured Analysis and Structured Design (SA/SD) methodology for embedded real-time systems. Wijbrans recommended the Hatley and Pirbhai method adapted with rigorous syntax and semantics definition based on CSP. The information hiding and encapsulation are taken from object-oriented design and the dynamic behaviour is defined in a CSP-like fashion. This method is used as the intermediate step in the refinement trajectory between control system design and its implementation.
- *Transputers and the programming language occam*
  Wijbrans prototyped a stepwise refinement procedure that transforms the control algorithm into efficient (concurrent) occam code on transputers. The semantics and design rules that Wijbrans imposes to the Hatley and Pirbhai specification method and on entry design tools allows a straightforward implementation to occam and transputers.

What is special about these streams is that they form a coherent system, which bridges the gap between controller design and its realization in an efficient way.

The implementation of these mainstreams is outdated for the following reasons:

- Transputers have become obsolete.
- Occam's future is uncertain and other programming languages (e.g. C, C++ and Java) are used by the majority in the embedded software industry.
- The Hatley and Pribhai method does not really comply with the current state of object-oriented technology and CSP technology. Object-oriented software design methods and tools dominate the market.

Although THESIS does not longer keep up with new developments, its philosophy behind bridging the gap between controller design and its realization is continued by this research. In this research, different roads have been travelled to obtain innovation and enhancements that could bring THESIS up-to-date. This research comprehends the following significant changes:

- Transputers are replaced by different types of processors and computer hardware, which are linked together on different kinds of networks (e.g. TCP, CAN bus, transputer-links).
- The occam programming language is replaced by CSP libraries for the programming languages C, C++ and Java.
- The Hatley and Pirbhai modelling language, as suggested by Wijbrans, is replaced by CSP diagrams.
- The control engineering tool CAMAS has been replaced by its successor, namely 20-sim. We continue to use the control design concepts that is now implemented by 20-sim.

These replacements enhance THESIS with technology that is closer related to the CSP concepts and object-oriented concepts. Each replacement fits the chain of thought behind THESIS. In the following chapters, this proposed methodology is presented as a standalone solution, independent of THESIS.

2.5 Conclusions

The development of control systems is guided by concurrency. Existing and main stream used control engineering tools do not allow the user to explicitly specify the desired concurrency. Instead, much as possible is automated and invisible to the user, as if the user should not much be concerned about concurrency issues. However, the user is concerned about concurrency, since the user is concerned about the performance and customization of the system. This approach prevents further refinement at the design level to make the design behave more efficient on a particular embedded computer system. Ultimately, the user has little control over the resulting code framework and its performance. Such a framework is often too implicit and limited to a restricted class of embedded systems. The proposed methodology aims at eliminating the gap between control system design and its realization. This is based on the sound and formal foundation of CSP concepts, represented by the channel model and process model. Their abstraction and well-defined semantics provide the guidelines to specify concurrency and to be able to manage complexities during system design. Therefore, the notion of processes is essential for control system design, due to the fact that processes are adequate entities for deploying and observing behaviour in the system.

In this chapter, the importance of process identification, process architectures design, and process analysis were discussed. Process identification is multidisciplinary. The design of process architectures, based on CSP concepts, allows one to specify the desired concurrency and event-driven behaviour in control applications. This implies that a process architecture integrates different engineering and modelling disciplines, each dedicated to specific processes. This results in a concurrent framework of concepts, which has the ability to eliminate discontinuities (read: gap) between the required disciplines during the specification, design and implementation of the control system.

The use of CSP diagrams provides a powerful tool for capturing a variety of concurrency related issues and giving them a formal semantics. Concurrency issues are: multithreading, interrupt handling, exception

handling, inter-processor communication, deadline guarantees (priority scheduling), precise timing (sampling and actuation), reactivity and responsiveness, safe-guarding and fault-tolerance. These issues are integral part of CSP diagrams, which enable precise specification, model checking, and process analysis on these issues. Its abstraction simplifies reasoning about the overall behaviour. Consequently, design failures are found early in the development phase.

After the context is determined and described by a context-diagram (i.e. top CSP diagram), the stages physical-system modelling and control law design are performed for the sub-processes. These stages are interleaved until both are satisfied. After the completeness of the process architecture is achieved, the design is implemented and tested. For systematic testing a hardware-in-the-loop simulation can be used. Eventually, the embedded control system is realized. The process architecture specifies how the system must behave. This pre-knowledge is a measure for success and likely saves time since engineering surprises are avoided and the outcome is predictable to a large extent.

CHAPTER 3

Graphical Modelling Language for Specifying Concurrency based on CSP

3.1 Introduction

In this chapter, a graphical modelling language for specifying process architectures is defined. The graphical modelling language is an improved version of the one that was published in Hilderink (2002). A design tool that supports this language is in development (Jovanovic et al., 2004). This design tool was not yet mature enough to be useful. Instead, a simple drawing tool was used.

The designs that are modelled with this graphical modelling language are called *CSP diagrams*. CSP diagrams describe the blue-print of systems on which concurrent hardware and software aspects in embedded system engineering fall back. The graphical modelling language is defined such that it allows specifying, designing, and programming concurrent frameworks using a simple graphical notation. The language abstracts away from hardware and software implementations at the specification and design phases. On the other hand, the graphical notation can be straightforwardly translated to hardware and software implementations. This abstraction and refinement prevents a gap between design and implementation. This is discussed in Chapter 6.

The graphical modelling language merges process-orientated and object-orientated technologies. The graphical notation and their semantics are

founded on CSP concepts. These CSP concepts are a subset of the CSP theory, which are practical and relevant for describing the behaviour of concurrent systems. In this graphical modelling language, enhancements have been incorporated, such as exception handling, priorities, timing, and imperative facilities. These enhancements are essential for designing real-time process architectures.

The semantics of the graphical notation and its association with CSP are defined. The affiliation of the language with CSP enables formal analysis of process architectures. Systematic techniques and rules are defined which are part of the language and provide guidance for detecting and reasoning about compositional conflicts (i.e. errors in design), deadlocks (i.e. errors at run-time), and priority inversion problems (e.g. performance burden) at a high level of abstraction and early in the development trajectory.

The distinction between processes and objects separates different concerns, such that, reasoning about behaviour and structure becomes well-defined. The differences between processes and objects are discussed in Section 3.2. Section 3.3 gives an overview of the relationships in CSP diagrams. These relationships are described by two processes and their interrelationships. Interrelationships are described in Section 3.4. The relationships are distinguished between communication and compositional relationships. Communication relationships are discussed in Section 3.5. Compositional relationships are discussed in Section 3.6. Hierarchies in process architectures are addressed in Section 3.7. Analysis techniques are defined in Section 3.8. These analysis techniques support the user in funding conflicts in the design. The design freedom that is incorporated in the proposed graphical modelling language is discussed in Section 3.9. The process of refinement and verification is elaborated on in Section 3.10. The conclusions to this chapter are given in Section 3.11.

3.2 Processes and objects

The notion of processes is inevitable in order to let object-orientation succeed in concurrent systems. The notion of processes is explained in this section with regards to objects.

A definition of processes was given in Section 2.3.1. Objects can be defined as follows:

Definition (object): An *object* is a conceptual, visual, or real entity (or thing) with crisp boundaries and meaning that encapsulates attributes (data), behaviour (operations or methods), state (memory), identity, and responsibility.

The definitions of a process and an object seem to have lots in common, but they are different with respect to their semantics, abstraction, and their interrelationships. In object-orientated methods, the term process is often associated with the transformation of data values (Rumbaugh et al., 1991). The CSP theory illustrates that this is not the whole truth of processes. The distinction and the close relationship between processes and objects can be illustrated by the following example.

> *Imagine a person drives a car. The car and the driver are objects in the sense that they represent real things. But what about changing gear, accelerating, or braking? These are processes, not objects, in which the car and driver participate.*

The car and driver are "real world" objects from a physical perspective, but they are individual and self containing "real world" processes in the process of driving. Every method in an object specifies a fragment of behaviour and applies when the methods are invoked. This behaviour has a meaning only when the method is processed, i.e. when the object is part of a process. Objects may exist at the same time, but there is no interrelationship which tells that they operate in parallel. Parallelism concerns processes, not objects. Furthermore, an object cannot be considered to be real-time. An object is said to be real-time only when it participates in a real-time process. Therefore, one can put a "real-time" tag on a process but not on an object.

In object-oriented programming languages, objects commonly implement the code structure and data structure of a program. Concurrency is not dealt with at all. The inclusion of multithreading makes the object-oriented paradigm unnecessarily complicated, since threads are not object-oriented. Obviously, threads are process-oriented. The notion of processes, as for example those defined in CSP, provides concurrency at an appropriate level of abstraction that scales well with complexity.

Processes and objects play different roles with different abstractions in concurrent systems. Processes are concerned with concurrency and behavioural structures, whereas objects are concerned with implementation structures. In this thesis, a process can play the role of a CSP process or a process instance. This is elaborated on in Section 3.5.4 and 4.3. Objects implement process instances, and process instances implement CSP processes. Each of these roles comprises different interfaces and different concerns, whereby CSP processes are more abstract than objects.

Processes and objects are both message-passing driven with different intentions. Objects can directly invoke services on other objects; the invokee always follows the invoker. This type of message-passing is synchronized on a sequential basis, namely *run-to-completion*. The next service is performed after the previous one has completed. Processes cannot directly invoke services on other processes. Processes can only invoke services within themselves. They can be influenced by communication with other processes using intermediate objects, i.e. channels or barriers. Message-passing is synchronized on the basis of *run-to-rendezvous*.

Processes and objects are distinct by their different kinds of relationships. The relationships between processes are classified by communication relationships and compositional relationships. The communication relationships specify producer/consumer, client/server, or communication-peers scenarios. The compositional relationships specify parallel, sequential, alternative, or exceptional dependencies. The relationships between objects are classified by generalization (inheritance), association (invoker/invoke scenario), dependencies, and aggregation.

Furthermore, the interface of a process is different from the interface of an object. In this proposed methodology, a process has two separate interfaces, namely the *process communication interface* and the *process instance interface*. The process communication interface is used during the execution of the process and the process instance interface is used before or after the execution of the process. Both interfaces are safely synchronized (thread-safe) and they cannot be used at the same time. These interfaces are discussed in Section 3.5.4. An object has a single interface that specifies a set of methods and variables, which the object offers. These methods and variables are exposed to concurrency. Thread synchronization constructs are required when objects are exclusively shared by multiple threads. The thread synchronization constructs are integral part of objects and must be explicitly implemented by their methods. Processes are synchronized by constructs that implement the communication relationships and the compositional relationships. See Chapters 4 and 6

The proposed graphical modelling language captures this notion of processes, which allows describing the (real-time) behaviour of concurrent systems. At a lower level of abstraction, objects are required to implement processes.

3.3 The CSP diagram

A CSP diagram is a graph of processes and their interrelationships, which models a concurrent (sub-) system. CSP diagrams are used for specifying, designing, and programming process architectures. The graphical notations and their semantics are derived from CSP. The interrelationships are displayed as lines. Processes are displayed as circles, bubbles, or rectangles.

Two processes and their interrelationship form a relationship. Some relationships are communication-oriented and some are composition-oriented. Each orientation can be viewed as

- a *communication diagram* representing communicational relationships; i.e. data-flow between processes,
- a *composition diagram* representing compositional relationships; i.e. control-flow between processes,
- a *hybrid diagram* representing both communicational and compositional relationships.

An overview of these relationships, which are defined by the graphical modelling language, is given in Figure 3-1.

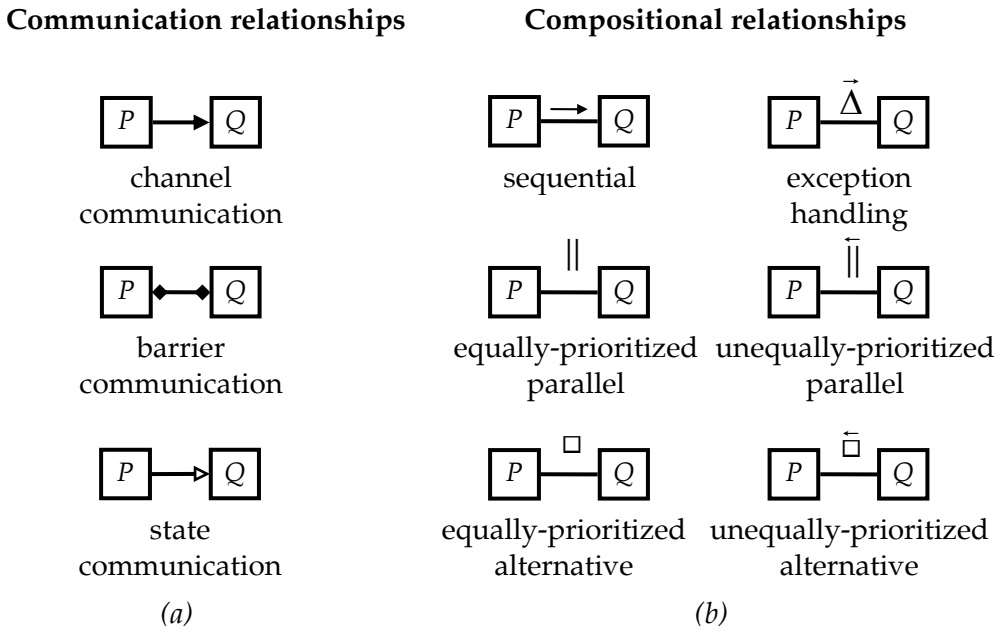


Figure 3-1 *Overview of CSP relationships: (a) communication relationships, (b) compositional relationships.*

The symbols at the ends of the communication interrelationships specify the type of communication and the roles the processes play at the point of communication. The symbols on top of the compositional interrelationships are operators that specify different fundamental behaviours in which the associated processes participate. Two processes are related to each other by one compositional interrelationship and zero

or more communication interrelationships. Thus, CSP diagrams comprehend both data-flow and control-flow modelling.

CSP diagrams can be used together with object-oriented methods (Selic et al., 1994; UML, 1998) and structured methods (Hatley and Pribhai, 1987; Ward and Mellor, 1985; Yourdon and Constantine, 1979). Other modelling languages, such as the UML, are recommended for describing the structural and functional aspects. The graphical elements can be stereotyped in the UML, which enables the integration with the UML. Consequently, one can omit the concurrency model of the UML and replace it with CSP diagrams.

CSP diagrams are computational models that are executable in the sense that they contain information suitable for simulation, code generation, or model checking. A CSP diagram can be viewed as a kind of state diagram. CSP diagrams scale well with complexity. Composing processes (states of execution) in CSP diagrams scales linearly with adding or removing processes or interrelationships. Therefore, refinement does not cause state explosions.

Priorities are essential in making decisions, which improves the performance of process architectures in systems with shared resources; e.g. a single processor or a communication channel for which interleaving is imposed. The CSP theory abstracts away from priorities, but it allows us to add priorities by way of refinement. The graphical modelling language includes equally-prioritized and unequally-prioritized operators. These operators include fairness and unfairness iu the process architecture. These operators apply to true-parallelism (i.e. processes distributed on multiple processors) and semi-parallelism (i.e. processes on a single processor).

The graphical modelling language inherently supports techniques for checking the correctness of CSP diagrams. With these model checking techniques, the user is able to determine and to reason about failures, mismatches with the user's mindset, deadlocks, and priority inversion problems in a design. Model checking can be automated by the design tool or performed by other formal model checking tools that are available. Feedback from these model checking tools can support the

user in understanding and solving errors or conflicts in the design of the process architecture. Furthermore, the user should be aware of the semantics of the notations, but the user does not have to be aware of the underlying mathematics.

3.4 Interrelationships

The simplest interrelationship is *concurrency*, whereby processes exist at the same time and possibly communicate with each other. Processes execute and synchronize in several ways, which can be expressed by interrelationships between the processes. The interrelationship is depicted as a labelled line (or edge) between two processes, as shown in Figure 3-2. The figure shows a relationship of two processes connected by an interrelationship on which both processes depend. Each interrelationship represents a synchronization construct in the process architecture.

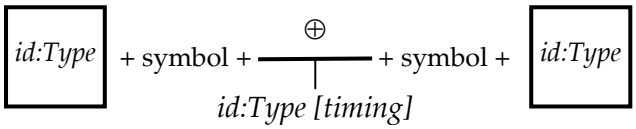


Figure 3-2 *Interrelationship between processes.*

Special symbols can be attached to the ends of the line to indicate a directional interrelationship between processes. It is important to note that the line should be seen as distinct from these symbols. The line itself is undirected because events are symmetric (Roscoe, 1998). The line renders an event in which both processes engage. Depending on the kind of interrelationship, the associated event can be a communication event, termination event, exception event, or a timeout event. The symbols are a gloss on this. They indicate the polarization of message-passing or they assist in composing processes.

A relationship is identified with *id* and is related to a specific *Type*. These attributes are combined with ":" as one label, which is refered to as an

identifier label. The *id* attribute specifies a unique name that refers to the declared relationship or process. The *operator* and *identifier* labels are floating labels and are attached with a thin line to the centre of the interrelationship. Like processes, these labels can be moved in the diagram. The symbols, operator, identifier, and type, specify the *type of interrelationship* between two processes.

The *timing* attribute is optional (see brackets) for communication relationships. This attribute can be used to specify timed communication events. Timed communication events offer the ability to specify the real-time behaviour of the process architecture. The timing attribute can contain an exception parameter, which will be thrown by the interrelationships to the associated processes when the specified deadline was not met. The timing attribute is not defined for compositional relationships. This option is reserved for future use.

The symbol ⊕ represents an *operator*, which is strictly used in compositional relationships. An overview of the possible operators was shown in Figure 3-1b. The operator is depicted on top of the compositional interrelationship line and the identifier label is usually depicted below the line. Communication interrelations have no *operator* and it is replaced by the identifier label located on top of the line.

The *Type* attribute of the interrelationship is one of the following types:

- primitive data type,
- class name for an object type,
- service interface type,
- reserved class name representing barrier synchronization,
- reserved class name representing a compositional process.

The first three types are used in communication relationships and the last two types are used in compositional relationships. Primitive data types and object types address data channels. Service interface types address call channels. The reserved class names are omitted from the identifier label, because these are redundant or derivable from the operator or symbols.

In Figure 3-1 and in other examples, the identifier labels on compositional relationships are hidden. This means that they are anonymous or simply invisible. Anonymous relationships are useful for sketching a compositional diagram without worrying about the exact names. Identifier labels can be specified and visualized by the user until later in the design trajectory. Identifiers that do not directly concern the user can be automatically generated by the design tool. These identifiers remain hidden. The naming convention in CSP diagrams is the same convention as used in Java.

Each process has a unique identifier *id* with a (non-unique) *Type* depicted in the rectangle. The *Type* attribute is the process class name from which the process is instantiated. A process class can be used to instantiate multiple processes with the same behaviour. In this thesis, several examples use capital letters *P*, *Q*, *R*, *S*, *T*, and *U* to identify distinct processes. These are equivalent to respectively *p:P*, *q:Q*, *r:R*, *s:S*, *t:T*, and *u:U*. Label *P* is a short notation for *p:P*. These abbreviations are used to simplify the related algebraic expressions.

3.5 Communication relationships

Two communicating processes participate in a communication relationship. A communication relationship is defined as follows:

Definition (communication relationship): A *communication relationship* is a labelled and directed relationship, which represents message-passing between a sender process and a receiver process.

Three classes of communication interrelationships are specified, namely

- channel communication,
- barrier communication,
- state communication.

Channel and barrier communication perform message-passing between executing processes. State communication performs message-passing

between non-executing child processes and their parent process via state variables.

3.5.1 Channel communication

Message-passing between processes via channels is symbolized by a solid arrow, as depicted in Figure 3-3.

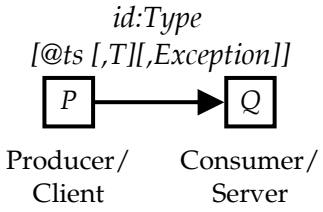


Figure 3-3 *Channel communication relationship.*

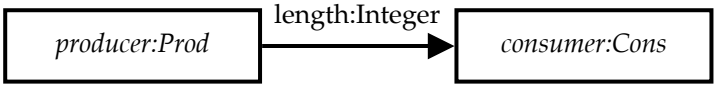
The arrow head symbol '►' is attached to the receiver of a message and the tail of the arrow is attached to the sender of the message. This does not necessarily imply that data is moving strictly from tail to head at communication. Data may very well be returned at the end of the invocation. Furthermore, the communication relationship specifies that communication can take place between two processes (i.e. two-way) whereas it does not specify exactly when communication takes place. The actual channel invocations are specified by a few primitive communication processes, as described in Section 3.6.8. Both participating processes must rendezvous, whereby the processes are willing to engage in the communication event. In some circumstances, buffered channels can be used to solve particular performance issues. These issues are discussed in Section 3.8.5.

The identifier *id* is the name of the channel and *Type* expresses the type of message passing. The timing parameters *ts*, *T*, and *Exception* are optional. The occurrence of communication events can be set to a specified moment in time or at periodical moments in time; i.e. respectively @*ts*, and @*ts*,*T*. The parameters *ts* and *T* are in microseconds, where *ts* is the absolute start time and *T* is a periodical interval relative to *ts*. A channel with timing parameters is called a *timed channel*. In case an *Exception*

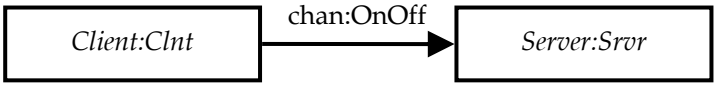
parameter is specified, an instance of *Exception* is thrown to the related processes when one or both processes are ready *after* the specified time *ts* or *ts*+*n*.*T* with

$$\left\{ n \middle| n \in \mathbb{N}^+ \wedge n = \left\lceil \frac{t - ts}{T} \right\rceil \right\}$$

The arrow designates the role of the processes in the relationship. A process at the tail of an arrow can play the role of a producer or client. A process at the head of an arrow can play the role of a consumer or server. Thus, channel communication specifies producer/consumer or client/server relationships.



(a) Producer/Consumer data channel communication



(b) Client/Server call channel communication

Figure 3-4 *Channel communication scenarios.*

Producer/Consumer relationship (data channels)

A producer/consumer relationship is based on *data channels* between a producer and consumer process. The *Type* of the data channel specifies either an object or a primitive data type that can be passed from producer to consumer. Data channels are unidirectional, i.e. data can be passed in the direction of the arrow.

An example of a producer/consumer relationship is shown in Figure 3-4a. In this example, *length:Integer* specifies a channel name *length* that can pass objects of type *Integer*. Or identifier label *length:int* specifies a channel that can pass integer data primitives of type *int*.

A data channel can be one of two possibilities, namely:

- unbuffered channel (rendezvous),
- buffered channel (fifo, super-sampling, sub-sampling, etc.).

Unbuffered channels usually provide an optimal reactive behaviour, as outlined by the process architecture. Buffered channels can improve the performance of a process architecture in circumstances where unbuffered channels cannot sufficiently decouple multiple frequencies. This is discussed in Section 3.8.5.

Client/Server relationship (call channels)

A client/server relationship is based on *call channels* between a client and a server process. *Type* indicates a service type, which specifies a set of methods.

The client process can request a method that is a member of the service type of the call channel. The requested method can only be accepted by server processes that implement the service type. When a method is accepted it is performed by the server process. A method can be provided arguments and a return value. Therefore, call channels can be bidirectional, i.e. data can be passed in both directions of the arrow. The client process can send data as arguments along the call to the server process, and on the completion of the method the resulting data can be sent back to client process.

In Figure 3-4b, the service type `OnOff` specifies the methods `on()` and `off()`. *Srvr* must implement these methods and it must be willing to accept any calls.

Consistency between a service type, server, and client can be checked. A server that implements service type S may accept certain requests A in S, thus $A \subseteq S$. The client process may request a set of calls, denoted by C, on a call channel with service type S. It is important that C is a subset of A otherwise certain calls will never be accepted, thus $C \subseteq A$; otherwise

$C \subseteq S - A$ and this would result in a deadlock or livelock. This design error is called a *service conflict*.

Identifier Labels

Identifier labels on communication interrelationships are floating labels, which are connected to arrows in various ways. When the arrow is connected between processes, the identifier label is connected with a thin line to the centre of the arrow. On the other hand, an identifier label can be used to create a joint for shared communication interrelationships or to relay communication interrelationships.

Figure 3-5 shows three different ways in which processes can be connected via channels. These three representations do not change the semantics of channel communication. Figure 3-5a shows the identifier label on top of a channel. Figure 3-5b shows that the label can be used as a joint between two parts. Labels can be duplicated and each duplicate refers to the same instance. Figure 3-5c illustrates two processes separated from each other via duplicated labels. This notation is similar as for barriers (Section 3.5.2) and state variables (Section 3.5.3).

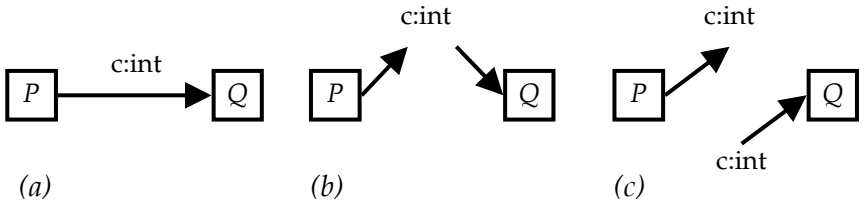


Figure 3-5 *Three ways to connect processes via a channel:*
(a) directly connected,
(b) via a label,
(c) via label duplicates.

Shared channel communication

Channels can be shared between two or more processes. The arrow may consist of branches of multiple tails and/or multiple arrow heads. This is

depicted as a fish bone. Four different channel configurations are supported, as shown in Figure 3-6.

The configuration displayed in Figure 3-6a depicts channel communication between two processes, as previously discussed. The configurations in Figure 3-6b and 3-6d specify a choice of service between multiple writers. The configurations in Figure 3-6c and 3-6d specify a choice of delivering messages between multiple readers. The choice is non-deterministic in relation to time.

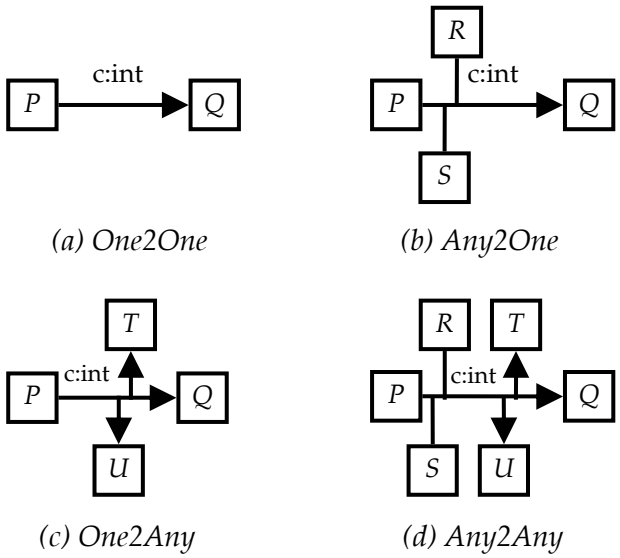


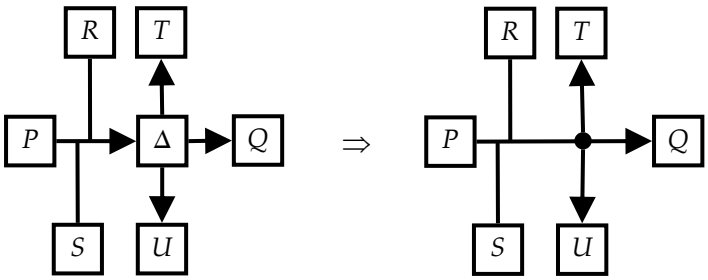
Figure 3-6 *Channel configurations.*

On a shared channel, the service or delivery between multiple readers and writers can be uncertain or unfair. For example, in practice and in a worst case scenario, a reader may read all the time and other readers have no chance to get a message. This is known as *starvation*. A fair queuing policy can prevent starvation between the alternating processes. The graphical modelling language has the taxonomy for detecting and reasoning about this pathological problem. The implementation of shared channels is concerned with the fairness and the performance of the program. This is elaborated on in Chapters 4 and 5.

The configurations shown in Figure 3-6b, 3-6c and 3-6d virtually swap to the configuration in Figure 3-6a. Shared channels do not broadcast

messages. In case messages must be broadcasted over many channels, a *delta process* must be used. A delta process reads from an input channel and then outputs the data on multiple output channels in parallel. Delta processes are only useful with data channels. Delta processes can be implemented in two ways: as an explicit or as an implicit delta process.

Figure 3-7a illustrates the use of an explicit delta process.



(a) (b)

Figure 3-7 *Shared data channel and broadcasting messages, (a) broadcasting with an explicit delta process, (b) broadcasting with an implicit delta process.*

An explicit delta process is part of the compositional relationships. This implies that the delta process can only terminate when it is told to terminate. This requires a poison token, an additional channel, or poisoning channels in order to stop the delta process. A poison token is a special token that is send via the input channel. Once the token is received, the delta process will terminate. An additional stop channel is also a solution. Both solutions lead to *graceful termination* (Welch, 1989). Poisoning channels is a technique that allows channels to throw exceptions to the associated processes when they are willing to communicate on a poisoned channel. The exception handling must gracefully terminate the delta process.

A simplified solution is the implicit delta process. In Figure 3-7b, an implicit delta process is depicted as a black circle from which broadcasting spawns. This black circle has already been shown in the context diagram in Figure 2-4. This delta process is not part of the compositional relationships of the processes that participate in the

communication relationship. Therefore, an implicit delta process is not explicitly part of the compositional relationships and no explicit measures are required to stop the delta process. The resulting framework can take care of starting and terminating implicit delta processes.

Unconditional or conditional channel communication

The previous communication relationships express *unconditional communications*, i.e. if both participating processes are ready for communication then both are committed to communication, for which they engage in a communication event and withdrawing is impossible. *Conditional communication* is a circumstance whereby the readiness of the channel is required as a condition. The readiness is true when *at least one* process is willing to communicate over the channel. A process at the conditional end of the channel may commit in communication when the other side is willing to communicate or it may withdraw when this condition is not met. Conditional communication is specified by the alternative relationship. See Section 3.6.4.

3.5.2 Barrier communication

Another communication relationship is the barrier synchronization, in which two or more processes participate at the same time. A fixed number of processes are required to synchronize their execution at some point before proceeding. A barrier is depicted as a fishbone of arrows between two or more processes (i.e. multi-way). Each end of the arrow is symbolized with a diamond '♦' symbol; shaped by concatenation of ◄ and ►. A barrier has only one type, namely *Barrier*. This type is reserved for the barrier relationship and is omitted from the identifier label. See Figure 3-8.

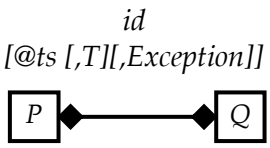


Figure 3-8 *Barrier synchronization relationship.*

The timing parameters *ts*, *T* and *Exception* are optional and are used in the same way as timed channels, as discussed in Section 3.5.1. A barrier with timing parameters is called a *timed barrier*.

An example of six processes that synchronize on a single barrier is depicted in Figure 3-9.

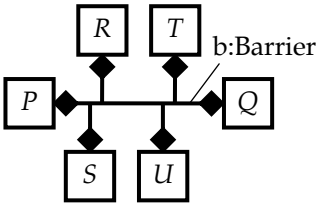


Figure 3-9 *Barrier synchronization.*

When all processes reach the barrier synchronization, the barrier construct can communicate information between the processes in a unidirectional or bidirectional way. All processes continue after communication is performed.

A barrier synchronization pattern can be described in terms of a protocol of channel communications (Roscoe, 1987). The protocol is described as a network of parallel processes that communicate with each other via channels. Therefore, a barrier can encapsulate a sub-diagram, whereby its completion is observed as a communication event.

3.5.3 State Communication

State communication is used to initialize state values and to pass state values to other processes instances before or after they are executing.

State communication is formulated as follows:

Definition (state communication): *State communication* is the ability of the parent process to communicate states between itself and its child process instances.

State communication requires state variables to communicate the postconditional state of one process instance to the preconditional state of another process instance. Note that the term 'process instance' is used. A state variable is defined as follows:

Definition (state variable): *State variables* are variables that manifest a precisely measurable property or a deterministic attribute, which characterizes the state of a process, independent of how the process was brought to that state.

Preconditional and postconditional states are formed by state variables. A preconditional state must be true or valid before the process executes and the postconditional state must be true or valid after the process has terminated. In case the preconditional or postconditional states are invalid, the results of the process can be unpredictable or wrong.

State communication is only allowed before or after the execution of child processes. State communication involves a state handling method that can initialize or return state variables. State communication does not describe concurrency aspects; *it must not be used for process communication*. Furthermore, state communication does not establish communication events, since it relates processes to state variables and not processes directly to each other. Note that parallel processes must exchange data using channels or barriers, and not via state variables.

A parent process has hold of the process instances of its child processes. The parent process can update the preconditional state of its child processes according to the process instance interface. State variables that are not specified by the process instance interface cannot be updated by the parent process. See Section 3.5.4. Public state variables can be updated by the parent process before the compositional relationship (to which the process belongs) is executed. Public state variables can be

retrieved from the child process after the compositional relationship has terminated.

Simultaneous updating of the same state variable is forbidden and prohibited by safety rules. Safety rules are described in Section 3.8.1. These rules declare that state variables are implicitly synchronized by the sequential constructs in the process architecture. This is similar to using variables in a sequential programming language.

The initialization of a process is depicted with an open arrow (i.e. open arrow head) towards the process. See Figure 3-10a. In this example, the state variable is identified as *id:Type*. The content of the state variable is passed to *Q* right before the relationship (to which *Q* belongs) is executed. The *Type* attribute is a primitive data type or an object type (class).

The state update of the parent process is depicted with an open arrow between a child process and a local variable. This arrow is directed from the child process towards the local variable. See Figure 3-10b. In this example, the variable *id* is updated immediately after *P* has terminated.

A state variable of a child process can be used to initiate another child process. See Figure 3-10c. This example is a simplified version of Figure 3-10a and 3-10b together. It depends on the compositional interrelationship between *P* and *Q* whether or not

- *P* updates *Q* via *id* (in sequence)
- or *Q* is updated by *id* before *P* updates *id* (in reverse sequence).

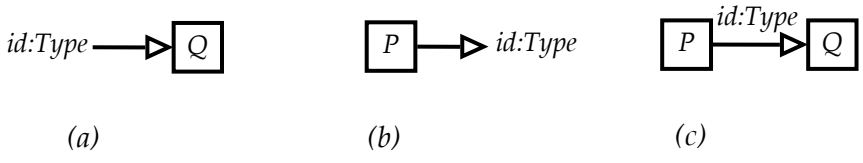


Figure 3-10 *State initialization:*
(a) child process initialization,
(b) parent process state update,
(c) state passing between child processes.

Figure 3-11 shows two processes performing the processes *f* and *g*. This example shows state dependencies between *f* and *g*.

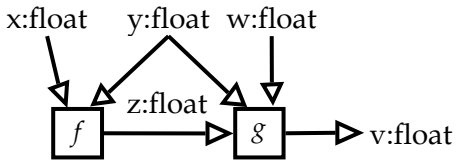


Figure 3-11 *Example of state initialization.*

This example depicts the following equations:

$$z = f(x, y)$$
$$v = g(y, z, w)$$

A cyclic dependency, as in Figure 3-12, does not cause an algebraic loop. Be aware that the states can be updated with a shift in time. This is determined by the compositional relationships.

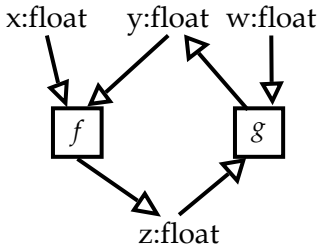


Figure 3-12 *Example of algebraic loop.*

This example depicts the following equations:

$$z = f(x, y)$$
$$y = g(z, w)$$

Possible solutions are,

$$\left(z_i = f(x_i, y_{i-1})\right); \left(y_i = g(z_i, w_i)\right) \quad or$$
$$\left(y_i = g(z_{i-1}, w_i)\right); \left(z_i = f(x_i, y_i)\right) \quad or$$
$$\left(z_i = f(x_i, y_{i-1})\right) \| \left(y_i = g(z_{i-1}, w_i)\right)$$

The indexing of the state variables denotes a sequence between the current (*i*) and the previous (*i*-1) values. This sequence prevents race hazards on state variables. Rules for safely sharing state variables are described in Section 3.8.1.

A state variable can be initialized with a start value. Figure 3-13 illustrates this in three different ways. In the examples, the state variable *x* is allocated in the parent process of *P*. *P* is initiated by *x* right before executing P and *x* is updated right after *P* has terminated. Figure 3-13a shows a state variable *x* being initiated with value 0.0 when the process is constructed. The state variable is not re-initiated on the next run of *P*. It remembers the previous value. Instead of using two separate open arrows between a process and a state variable one can use a single bidirectional open arrow. See Figure 3-13b. Figure 3-13b illustrates something similar, whereby *x* is initiated with the value of another variable x_0. This notation is useful when x_0 is documented as the (constant) initial value of the process and *x* is a variable in the workspace. In case a variable needs to be initiated each time before the next run, the parent process *Q* must initiate the state variable *x* of its child process *P*. This is illustrated in Figure 3-13c.

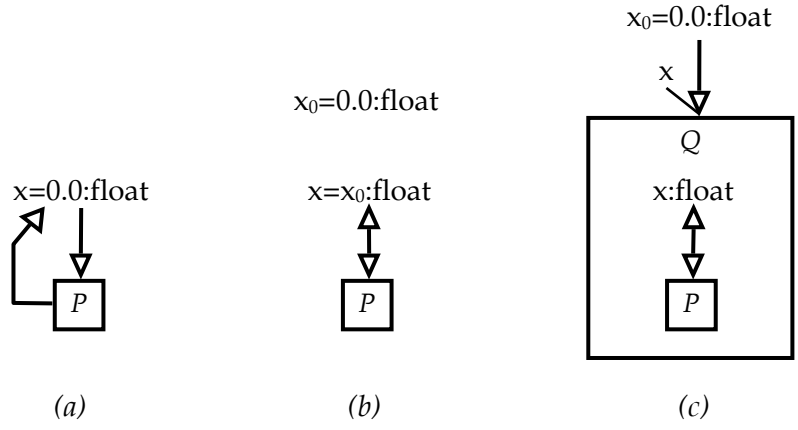


Figure 3-13 *Initiating state variables:*
(a) initiate x once with value 0.0,
(b) initiate x once via state variable x_0,
(c) initiate x before every run of process Q.

One can easily distinguish between channel communication and state communication. For example, Figure 3-14 shows a process that is connected with solid arrows and open arrows.

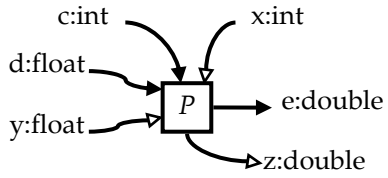


Figure 3-14 *Example of mixed channels and variables.*

In this example, the process communicates with other processes via the solid arrows. State communication is performed by the open arrows and they are performed right before *P* is executed and right after *P* has terminated. Obviously, the labels *c*, *d* and *e* belong to the process communication interface and the labels *x*, *y*, and *z* belong to the process instance interface. Solid arrows and open arrows cannot be connected. Connection is only possible via primitive communication processes, which are discussed in Section 3.6.8.

3.5.4 Process interface

The process interface consists of two separate interfaces, namely the *process communication interface* and the *process instance interface*. The process communication interface comprises a set of public *channel-ends* and *barrier-ends*. The process communication interface can specify a protocol of communication via these ends, which describes its behaviour. The process instance interface defines a set of public *state variable-ends* that are responsible for initializing the preconditional state or for retrieving the postconditional state of the process instance. These channel-ends, barrier-ends, and state variable-ends are called the *ports* of the process interface.

The process communication interface is used during the execution of the process and the process instance interface is used before or after the execution of the process. They cannot be used at the same time.

Each pair of ports that is supposed to be connected must be compatible. Incompatible ports cannot be connected. For example, a channel-output of message type Integer cannot communicate with a channel-input of message type Float. Also, a producer process cannot communicate with a server process, because the producer process requires a data channel and the server process requires a call channel. Furthermore, state variables can be connected to other state variables of the same type.

Floating identifier labels that are specified as ports in the process instance interface and the process communicating interface must be used to connect the process with the outside world. These ports can be depicted as identifier labels on the edge of the process in the upper CSP diagram. This was already shown in Figure 3-13c. The identifier labels are called *port labels*. A port label and the corresponding floating identifier label must have the same name and of the same port type. Port labels are annotated by a thin line to the end of a communication interrelationship and the entry/exit point of the associate process. Edge labels are essential for the user to know to which port or state variable the interrelationship is connected. Examples of data channel, call channel, barrier, and state communication are given in Figure 3-15. These annotations can be visualized or hidden at wish.

In Figure 3-15a, the *producer* process performs an output on data channel *length* via port *size*. The *consumer* process performs an input on port *len*. These outputs and inputs are discussed in Section 3.6.8. In Figure 3-15b, the *client* process requests a call on the call channel *chan* via port *out*. The server process accepts the call via port *in*. In Figure 3-15c, the processes *one* and *two* synchronize on the barrier-ports *bar* and *join*. They are linked together via *barrier*. In Figure 3-15d, process *one* provides data for initializing process *two*. The date is passed from *y* to *z* via *x*.

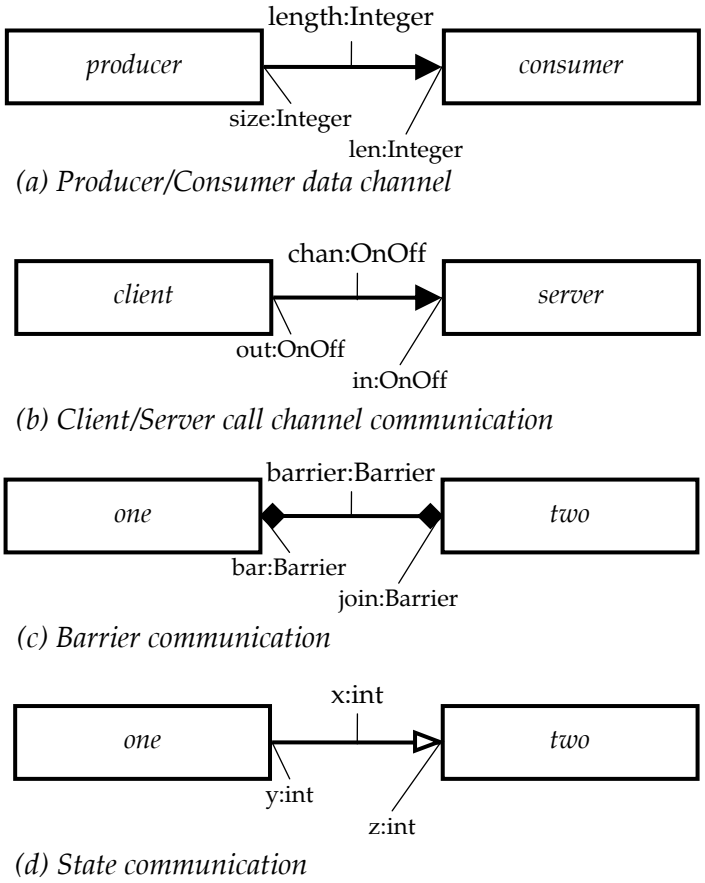
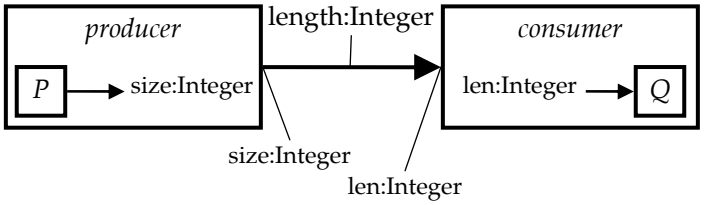


Figure 3-15 *Process interfaces shown by label annotations.*

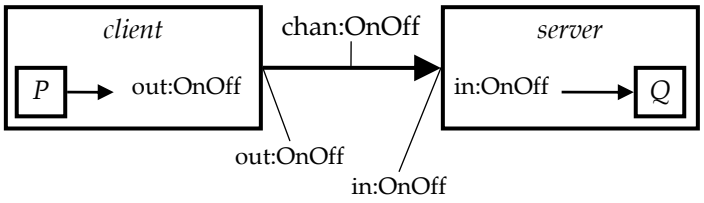
Figure 3-16 illustrates examples of processes *P* and *Q* that are connected respectively via floating identifier labels and their ports. The floating identifier labels and port labels establish a connection in a hierarchical architecture.

Roscoe described that: "A CSP process is completely described by the way it can communicate with its external environment. In constructing a process we first have to decide on an *alphabet* of communication events – the set of all events that the process (and any other related process) might use. The choice of this alphabet is perhaps the most important modeling decision that is made when we are trying to represent a real system in CSP.". The process communication interfaces represent the alphabets of the processes. Therefore, the design of a communication diagram is most important. The communication diagram will determine the alphabets (or process communication interfaces). The compositional relationships will

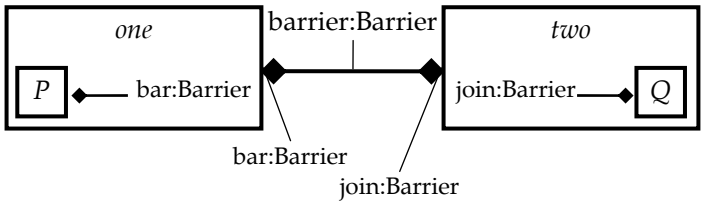
combine these alphabets to new alphabets. These alphabets should be consistent with the specification. These alphabets are required for describing the appropriate parallel processes. See Section 3.6.3.



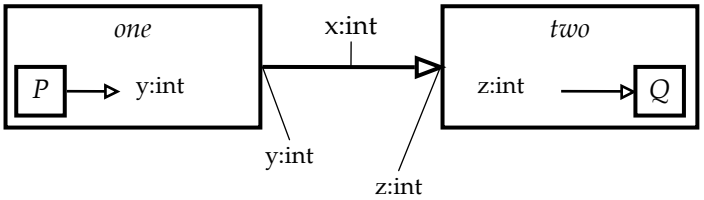
(a) Producer/Consumer data channel communication



(b) Client/Server call channel communication



(c) Barrier communication



(d) State communication

Figure 3-16 *Examples of edge and floating labels.*

Channel and barrier ports make up the alphabet of communication events, not the state communication ports.

3.6 Compositional relationships

Compositional relationships are a kind of relationships between processes that are useful for describing the execution order of communicating processes. Compositional relationships are defined as:

Definition (compositional relationship): A *compositional relationship* is a labelled relationship between two processes whereby the label is a binary operator that expresses their compositional behaviour.

Figure 3-17 shows two processes in relation to each other. This represents a composition of two processes, which semantics are described by the operator $\oplus$ on top of the connection.

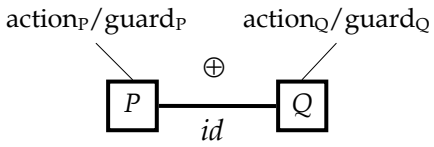


Figure 3-17 *Compositional relationship.*

Action bodies (e.g. $action_P$ and $action_Q$) are optional and these can be specified next to the process when at some point in the process architecture a state change is required. The state change is part of a state machine specification (automaton) that is local to the parent process. Guard bodies (e.g. $guard_P$ and $guard_Q$) are required when the processes are related to a choice. Action and guard bodies are discussed in Section 3.6.1.

The operator $\oplus$ can be one of the following sets:

- $\oplus \in \{\rightarrow, \leftarrow, ||, \overleftarrow{||}, \overrightarrow{||}, \overrightarrow{\Delta}, \overleftarrow{\Delta}\}$ when no guard bodies are specified,
- $\oplus \in \{\square, \overleftarrow{\square}, \overrightarrow{\square}\}$ otherwise.

The operators $\{\square, \overleftarrow{\square}, \overrightarrow{\square}\}$ require the specification of guards on all participating processes. For the other operators, guards have no effect. The operators with a small arrow on top of the symbol are directed operators, whilst the remainders are undirected operators.

The identifier label *id* attributes a unique name of the construct. Compositional relationships are typed by their operators. For example, operator symbol || is equal to *id:Parallel*, whereby *Parallel* is a reserved class name for the parallel operator. Other reserved class names are *Sequential*, *Alternative*, *PriParallel*, *PriAlternative*, and *ExceptionCatch*. The *Type* attribute is redundant and is therefore omitted. In Chapter 4, it will be shown that these types represent special processes or constructs in Java.

Each compositional relationship is explained in Section 3.6.2 till Section 3.6.5. The identifier labels are anonymous and omitted in the next sections.

3.6.1 Automaton

The behaviour of a process architecture depends on state changes in time. State changes can be specified by state communication or using action bodies. Guard bodies are used to take decisions so that the process architecture behaves in one way or the other. An action body or a guard body can be specified next to a process connected with a thin line to show its association with the process. This is illustrated in Figure 3-17. Only one action body or guard body can be specified per process. The action and guard bodies are not an integral part of the associated process but they are part of a local state machine or automaton; e.g. to control repetitions (Section 3.6.4) and conditional communication (Section 3.6.6).

Action bodies contain operations that change variables in the process; i.e. in the parent process of the process to which the action body is associated with. An action body will be executed right before the process will be executed. For example, Figure 3-17 specifies the algebraic expression $(action_P ; P) \oplus (action_Q ; Q)$. The scope of variables is determined by the parent process in which they are declared. The updating and reading of state variables by action bodies follow certain rules. These rules are similar for state communication in Section 3.8.1. Instead of open arrows, the action bodies use assignment statements.

The following two methods illustrate two ways to increment a variable that is used to create an imperative construct, which is part of a state machine. An imperative construct can be created by an action body and a floating variable, or it can be created by state communication. See respectively Figure 3-18a and 3-18b. Combinations of the two methods are also possible.

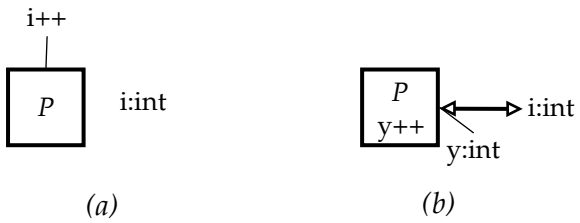


Figure 3-18 *Two examples of incrementing a variable:*
(a) incrementing by an action body,
(b) incrementing by a process.

Here, the increment statements `i++` and `y++` (in *P*) are short notations for `i=i+1` and `y=y+1`. These increment statements require bidirectional state communication. In Figure 3-18a, the variable *i* will be incremented right before *P* is executed. In Figure 3-18b, the variable *i* will be incremented right after *P* has terminated.

3.6.2 Sequential relationships

A sequential relationship (SEQ) between processes *P* and *Q* is denoted by the label '$\rightarrow$'. This sequential composition is written as $P \rightarrow Q$. This has strong similarities with the CSP single action transition $P \xrightarrow{\surd} Q$. This process will behave as *Q* if *P* has successfully terminated ($\surd$-event) otherwise this process behaves as *P*. These semantics are relaxed by saying *P* is executed before *Q*, which provide more design freedom. This relationship (being a process) terminates when *P* and *Q* successfully terminate. The sequential relationship is depicted in Figure 3-19.

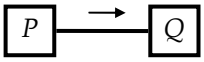


Figure 3-19 *Sequential relationship.*

A sequential relationship is written as $(P,Q,\rightarrow)$. A sequential relationship of more than two processes can be represented in the same way, for example $(P,Q,R,S,\rightarrow)$. See Figure 3-20a. Relationship $(P,Q,R,S,\rightarrow)$ also represents other partial relationships, such as, $(P,Q,\ \rightarrow)$, $(P,R,\rightarrow)$, $(P,S,\rightarrow)$, $(Q,R,\rightarrow)$, $(Q,S,\rightarrow)$, $(R,S,\rightarrow)$, $(P,Q,R,\rightarrow)$, $(P,Q,S,\rightarrow)$, $(P,R,S,\rightarrow)$ and $(Q,R,S,\rightarrow)$. Only tuples can be depicted in a CSP diagram. See Figure 3-20b. Thus, Figure 3-20a implies 3-20b.

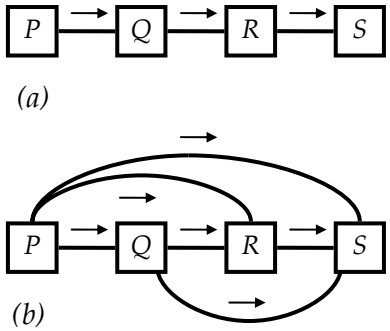


Figure 3-20 *(a) SEQ construct,*
(b) over-specified SEQ construct.

This example is written as

$$P \rightarrow Q \rightarrow R \rightarrow S \quad \Leftrightarrow \quad (P,Q,R,S,\rightarrow) = \big((P,Q,\rightarrow),(Q,R,\rightarrow),(R,S,\rightarrow),\rightarrow\big)$$

A sequential composition in CSP is precisely described by the algebraic expression *P;Q*. This expression defines a process that behaves as *P* and after *P* terminates it behaves as *Q*. The expression *P;Q* is a special case of $P \rightarrow Q$. The symbol '$\rightarrow$' gives more specification freedom than ';'. One can specify $P \rightarrow R$ without knowing whether or not it must be *P;R* or *P;Q;R*. For example, the partial relationships in Figure 3-20 cannot all be represented with ';'. It is obvious that $(P,S,\rightarrow)$ does not represent *P;S*.

The ';' and '$\rightarrow$' operators share common properties; e.g. they have no symmetry laws. The following shows when ';' can be used for '$\rightarrow$'.

The relationship $(P,S,\rightarrow)$ can be written as

$$(P,X,S,\rightarrow)\setminus X$$

where X is a set of processes that can be found on the longest path between P and S.; i.e. P and S are not in X. This algebraic expression has resemblance with the hiding operation in CSP. For example, in case $X=\{Q,R\}$, one can write

$$(P,S,\rightarrow) \quad \Leftarrow \quad (P,Q,R,S,\rightarrow)\setminus\{Q,R\}$$

Here, Q and R are hidden and P and S are visible and of interest. This hiding operation is useful for determining whether or not $P\rightarrow S$ can be written as $P;S$. If no other processes can be found between P and S on the longest path between P and S then this means that X is empty. In other words, they are called *neighbours*. A neighbour sequential relationship is written as $(P,S,\rightarrow)_{\varnothing}$ and equals

$$(P,S,\rightarrow)_{\varnothing}=(P,\{\},S,\rightarrow)\setminus\{\}=P;S$$

A chain of neighbour sequential relationships is written as

$$(P,Q,R,S,\rightarrow)_{\varnothing}=\left((P,Q,\rightarrow)_{\varnothing},(Q,R,\rightarrow)_{\varnothing},(R,S,\rightarrow)_{\varnothing},\rightarrow\right) \quad \Leftrightarrow \quad P;Q;R;S$$

This expression implies the longest paths between the processes P and S from P to S. Generally, one can write

$$(P_0,\ldots,P_{n-1},\rightarrow)_{\varnothing}=\mathop{;}_{i=0..n-1}P_i$$

This form allows transformation to a sequential code-construct as discussed in Section 4.6.2. The longest sequential paths of sequential operators that are pointing in the same direction along the path are eligible for implementation. The redundant relationships are useful for consistency checks in designs. This is discussed on Section 3.8.

3.6.3 Parallel relationships

Equally-prioritized parallel relationships

An equally-prioritized parallel relationship (PAR) between processes *P* and *Q* is denoted by the label '||'. This parallel composition is written as *P*||*Q*. This process will behave as *P* and as *Q* in parallel. This process terminates when all participating processes *P* and *Q* have terminated.

The parallel relationship between *P* and *Q* specifies that these processes are competing at equal priorities. See Figure 3-21. This relationship is written as (*P*,*Q*,||).

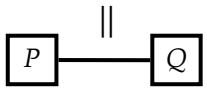


Figure 3-21 *Equally-prioritized parallel relationship.*

A multiple composition (*P*,*Q*,*R*,*S*,||) represents *P*||*Q*||*R*||*S*. See Figure 3-22a. This diagram shows the relationships (*P*,*Q*,||), (*Q*,*R*,||), and (*R*,*S*,||).

What are the relationships (*P*,*R*,$\underset{PR}{\oplus}$), (*P*,*S*,$\underset{PS}{\oplus}$), and (*Q*,*S*,$\underset{QS}{\oplus}$)? See Figure 3-22b. The solution is ambiguous. For example, (*P*,*R*,$\rightarrow$) or (*P*,*R*,$\leftarrow$) are valid specifications between *P* and *R*. This shows that the unspecified relationships do not have to be '||'. The choice of operator could be performed by the design tool based on certain criteria. The choices are

$$\underset{PR}{\oplus},\underset{PS}{\oplus},\underset{QS}{\oplus} \in \{\rightarrow,\leftarrow,||,\overleftarrow{||},\overrightarrow{||},\overrightarrow{\Delta},\overleftarrow{\Delta}\}$$

Note that any choice must not cause compositional conflicts, otherwise the operator is invalid. See Section 3.8.4.

In case the user wants to specify that all processes must be performed in parallel, the diagram must be completed as in Figure 3-22c. The use of hierarchical notations can prevent many lines and cycles. This is discussed in Section 3.7.

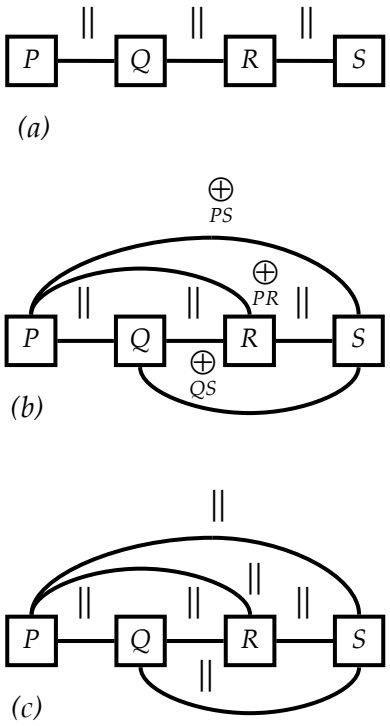


Figure 3-22 *(a) ambiguous PAR construct,*
(b) unambiguous PAR construct,
(c) completely specified PAR construct.

Figure 3-22a is ambiguous without assumptions. Figure 3-22c is uniquely specified and this diagram can be written as

$$\left(P_0,\ldots,P_{n-1},\|\right)=\left(\mathop{\|}_{i=0..n-2} P_i\right)\|\,P_{n-1}$$

The design tool could apply a criterion (assumption) that specifies that Figure 3-22a and 3-22c are equal. In this case, an equally-prioritized parallel composition with n processes in the form of Figure 3-22a and 3-22c is written as

$$\left(P_0,\ldots,P_{n-1},\|\right)=\mathop{\|}_{i=0..n-1} P_i$$

The processes $P_0 \ldots P_{n-1}$ are randomly ordered since operator '||' is symmetrical. Such criterion may simplify the understanding of parallel patterns, such as in Figure 3-22a. Other criteria are possible that can

optimize the performance of the process architecture for a particular target platform. This is not further discussed in this thesis. The user can overwrite any criterion by specifying interrelationships between compositional undefined processes (section 3.7.3).

The previous parallel operator can be described by one of the three kinds of parallel operators in CSP. The alphabets of processes are used to specify the exact parallel process.

The three parallel operators are:

- Alphabetized parallel $P \ {}_X\|_Y \ Q$
  P is allowed to communicate in the set X and Q is allowed to communicate in the set Y. They must agree on events in the intersection $X \cap Y$.
- Interleaving $P \,|||\, Q$
  P and Q run completely independent of each other. They do not synchronize with each other. In the graphical modelling language this implies that there is no channel or barrier interrelationship between them.
- Generalized parallel $P \underset{Z}{\|} Q$
  This is the process where all events in Z must be synchronized and events outside Z can proceed independently. This operator is a hybrid of an alphabetized parallel process and an interleaving process.

Alphabets are subordinate to the design and alphabets can be determined by the design tool once the design is completed. The design tool can determine the exact parallel operator and the hiding of internal communication events.

Unequally-prioritized parallel relationships

An unequally-prioritized parallel relationship (PRIPAR) between processes P and Q is denoted by label '$\vec{\|}$'. This parallel composition is written as $P \,\vec{\|}\, Q$. If process P is waiting to engage in a communication

event then it will behave as *Q* otherwise this parallel composition behaves as *P*. In other words, process *P* is executed with higher priority than process *Q*. This process terminates when all participating processes terminate.

The unequally-prioritized parallel relationship between *P* and *Q* is depicted in Figure 3-23. This relationship is written as $(P,Q,\overleftarrow{\|})$.

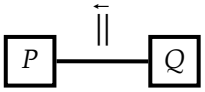


Figure 3-23 *Unequally-prioritized parallel relationship.*

The multiple relationship $(P,Q,R,S,\overleftarrow{\|})$ represents $P \overleftarrow{\|} Q \overleftarrow{\|} R \overleftarrow{\|} S$, see Figure 3-24a. There is no ambiguity involved, since all directed operators point in the same direction. This is similar as for the sequential operator. Therefore, the unspecified relationships can be uniquely derived from the specified relationships. See Figure 3-24b.

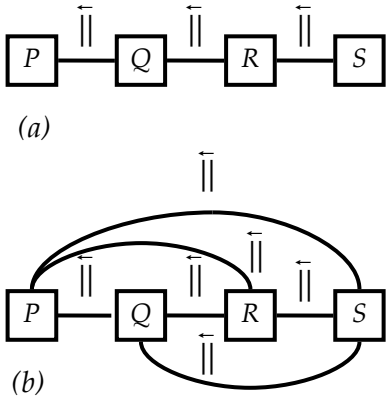


Figure 3-24 *(a) PRIPAR construct,*
(b) over-specified PRIPAR construct.

An unequally-prioritized parallel composition of *n* process is written as

$$\left(P_0,\ldots,P_{n-1},\overleftarrow{\|}\right) = \overleftarrow{\prod_{i=0..n-1}} P_i$$

This form allows immediate transformation to unequally-prioritized parallel code-constructs.

3.6.4 Alternative relationships

Equally-prioritized alternative relationships

An equally-prioritized alternative relationship (ALT) between processes *P* and *Q* is denoted by label '□'. See Figure 3-25. This operator is called the *external choice* in CSP. This alternative composition is written as $P \square Q$. This process will behave as *P* if *P* can engage in a communication event or it behaves as *Q* if *Q* can engage in a communication event. If both processes can engage in a communication event then the alternative construct will choose one arbitrarily. The alternative process terminates when the selected guarded process terminates.

In this thesis a fair choice, based on a fair priority policy, is preferred. This is discussed in Chapters 4 and 5.

The equally-prioritized alternative relationship between *P* and *Q*. is depicted in Figure 3-25. This notation is also written as relationship (*P*,*Q*,□).

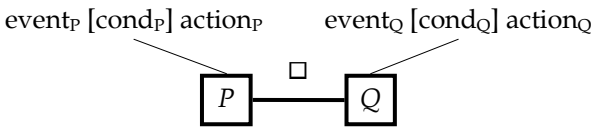


Figure 3-25 *Equally-prioritized Alternative relationship.*

A guard expression to each process in the alternative relationship is required that explicitly expresses a selection criterion that is taken. These processes are called *guarded processes*. A guard expression is depicted as a guard body and contains the format `event [cond] action`. This guard body is depicted next to the guarded process with a thin line connecting

each other. This guard body is used to create an automaton, which was discussed in Section 3.6.1. Figure 3-25 depicts the expression

$$\left(cond_P \,\&\, \left(action_P ; \left(event_P \rightarrow P'\right)\right)\right) \Box \left(cond_Q \,\&\, \left(action_Q ; \left(event_Q \rightarrow Q'\right)\right)\right)$$

where $P = \left(event_P \rightarrow P'\right)$ and $Q = \left(event_Q \rightarrow Q'\right)$. P and Q are the guarded processes that must engage respectively in $event_P$ and $event_Q$. as their first event. The action bodies $action_P$ and $action_Q$ must not engage in any event.

A guarded process has only one guard body attached to it. If the Boolean expression `cond` is true and the guarded process can engage in `event` then the choice operator may select the guarded process. If `cond` is false then `event` will be omitted and the guarded process will not be selected. Once the guarded process is selected then `action` will be executed prior to the guarded process is executed.

A variety of guard expressions is shown in Table 3-1.

`channel`	unconditional *channel-input or channel-output* guarded process
`channel [cond]`	conditional *channel-input* or *channel-output* guarded process
`callchannel.method`	unconditional *channel-call* or *channel-accept* guarded process on specified method
`callchannel.method [cond]`	conditional *channel-call* or *channel-accept* guarded process on specified method
`callchannel`	unconditional *channel-accept* guarded process on any method
`callchannel [cond]`	conditional *channel-accept* guarded process on any method
`skip`	unconditional *skip* guarded process
`skip [cond]`	conditional *skip* guarded process
`else`	unconditional *else* guarded process
`else [cond]`	conditional *else* guarded process
`timeout(t)`	unconditional *timeout* guarded process
`timeout(t) [cond]`	conditional *timeout* guarded process

Table 3-1 *Variety of guard expressions.*

Here, `event` indicates a *channel-input* guard, *channel-accept* guard, *channel-output* guard, *channel-call* guard, *skip* guard, an *else* guard, or a *timeout* guard. After each guard expression an `action` body can be specified that updates the state invariants.

The words `channel` and `callchannel` should be replaced by a channel name. The word `cond` represents a Boolean expression (or condition) and `method` should be replaced with the actual method name. The names `skip`, `else`, and `timeout` are special keywords and `t` represents the specified time. The channel-calls may require additional arguments in order to express the variables that are involved in communication. A distinction between channel-input and channel-output or between channel-accept and channel-call is rendered by respectively the arrow entering or leaving the guarded process. Optionally, symbol '?' or '!' can be appended to the event name in order to render the direction in the guard expression; channel-input and channel-accept use symbol '?', and channel-output and channel-call use symbol '!'.

A *skip* guard does not require a channel-input or channel-output and the guard is ready all the time. An *else* guard cannot be found in CSP but it is like a *skip* guard with the difference that it will be selected if no other guard is ready. The *else* guard can be modelled as a *skip* guard in an unequally-prioritized alternative construction. See Figure 3-26. The unequally-prioritized alternative operator is discussed in the next sub-section.

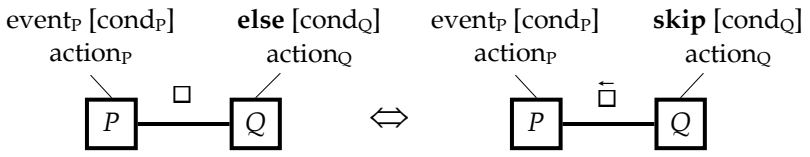


Figure 3-26 *Else guarded construct.*

The guard is said to be *unconditional* when `cond` is always true (or not specified) and the guard is said to be *conditional* when `cond` is some Boolean expression.

A *timeout*-guard becomes ready when the specified time expires. The timeout is relative to the beginning of the execution of the alternative relationship.

Guards can be applied to any-to-any channels but not to barrier configurations. It is important to notice that a channel-input guard and a channel-output guard, using the same channel and specified (at different processes) in the same alternative relationship will *never* commit in communication (Jones, 1987). All guards sharing the same alternative relationship must be disjoint in such a way that no pair of channel-input guards and channel-output guards can become simultaneously ready. This guideline prevents unwanted race conditions.

The composition $(P,Q,R,S,\Box)$ represents a path of relationships. See Figure 3-27a. Since each process specifies a separate guard, the criterion here is that the unspecified relationships may also be a choice operator. See Figure 3-27b, whereby

$$\underset{PR}{\oplus},\underset{PS}{\oplus},\underset{QS}{\oplus} \in \{\rightarrow,\leftarrow,||,\overleftarrow{||},\overrightarrow{||},\Box,\overleftarrow{\Box},\overrightarrow{\Box},\overrightarrow{\Delta},\overleftarrow{\Delta}\}$$

The criterion that specifies that Figure 3-27a implies Figure 3-27c will simplify the understanding of Figure 3-27a. In this case, $(P,Q,R,S,\Box)$ is written as $P \Box Q \Box R \Box S$ or one can write

$$(P_0,...,P_{n-1},\Box) = \underset{i=0..n-1}{\Box} P_i$$

Operator $\Box$ is symmetrical and thus the processes $P_0 \ldots P_{n\text{-}1}$ are randomly ordered.

In case the guards specify a conditional expression *cond*, one can write

$$(P_0,...,P_{n-1},\Box) = \underset{i=0..n-1}{\Box} (cond_i \,\&\, P_i)$$

Process P_i will be omitted when its conditional expression $cond_i$ is false.

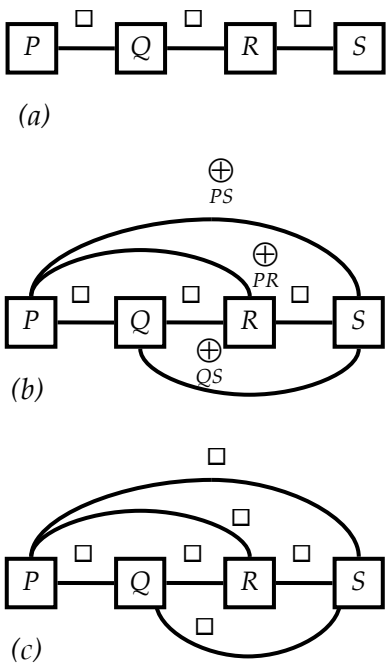


Figure 3-27 *(a) ambiguous ALT construct,*
(b) unambiguous ALT construct,
(c) completely specified ALT construct.

Unequally-prioritized alternative relationships

An unequally-prioritized alternative relationship (PRIALT) between processes P and Q is denoted by the label '$\vec{\Box}$'. This prioritized alternative composition is written as $P \vec{\Box} Q$. This process is almost similar to the alternative relationship, except that when both processes can engage in a communication event, process P will be chosen in preference of Q.

The unequally-prioritized alternative relationship between P and Q is depicted in Figure 3-28. This notation is written as relationship (P,Q, $\vec{\Box}$).

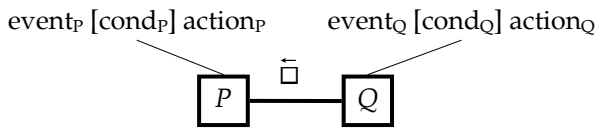


Figure 3-28 *Unequally-prioritized alternative relationship.*

A multiple relationship $(P,Q,R,S,\overleftarrow{\square})$ represents $P \overleftarrow{\square} Q \overleftarrow{\square} R \overleftarrow{\square} S$. See Figure 3-29a. Since '$\overleftarrow{\square}$' is a directed operator and pointing in the same direction along the path, this means that Figure 3-29a implies 3-29b.

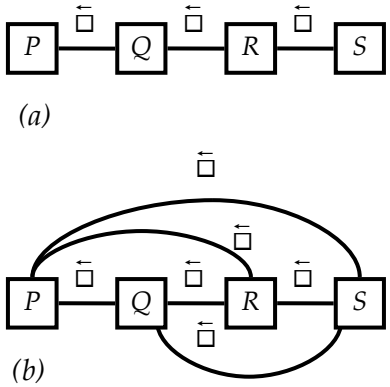


Figure 3-29 *(a) PRIALT construct,*
(b) over-specified PRIALT construct.

In case of a multiple relationship we can write

$$(P,Q,R,S,\overleftarrow{\square}) = \left((P,Q,\overleftarrow{\square}),(Q,R,\overleftarrow{\square}),(R,S,\overleftarrow{\square}),\overleftarrow{\square}\right) \quad \Leftrightarrow \quad P \overleftarrow{\square} Q \overleftarrow{\square} R \overleftarrow{\square} S$$

In the graph this expression is the longest path P to S. The importance of this form is that we can index processes in the compositional construct so that processes are successively ordered and with declining guard priorities.

One can write

$$\left(P_0,\ldots,P_{n-1},\overleftarrow{\square}\right) = \underset{i=0..n-1}{\overleftarrow{\square}} \left(cond_i \,\&\, P_i\right)$$

This form allows immediate transformation to unequally-prioritized alternative code-constructs.

3.6.5 Exception relationships

An exception relationship (EXC) between processes *P* and *Q* is denoted by the label '$\vec{\Delta}$'. This exception composition is written as $P\vec{\Delta}Q$. This process behaves as *Q* when *P* unsuccessfully terminates; otherwise it behaves as *P*. If *P* successfully terminates then *Q* will be omitted. This process is depicted in Figure 3-30.

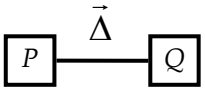


Figure 3-30 *Exception relationship.*

This exception operator is not formally defined in CSP. The exception operator is a new operator that is defined in Appendix C. It originates from the interrupt operator $P\Delta_i Q$ in CSP. This process behaves like *P* until *Q* can engage in event *i* at which point it behaves as *Q*. *Q* is initially awaiting for some event *i* from its environment. If *P* successfully terminates then *Q* will be ignored. The exception operator is a simplified version of the interrupt operator whereby event *i* is represented as an internal event. This internal event is generated by channels, barriers, or instructions that are in exception; e.g. error in hardware, disconnected link, or division by zero. Process *Q* is the exception handling process, called the *exception handler*.

Here, $P\vec{\Delta}Q$ and $P\Delta_i Q$ are both directed interrupt relations between *P* and *Q*, but only $P\vec{\Delta}Q$ is directional commutative. The directional commutative property provides topographical modelling freedom. Consider the differences:

$$P\Delta_i Q \neq Q\Delta_i P$$

$$P\vec{\Delta}Q = Q\overleftarrow{\Delta}P$$

The Δ_i operator requires preemption on event i which needs a sophisticated construct like the prioritized parallel construct. On exception, the Δ_i operator requires that process P releases all the channels and barriers, on which it or its child processes are blocked, in order to prevent deadlocks. Instead, the $\vec{\Delta}$ operator can be implemented with try-and-catch clauses as found in Java and C++. The $\vec{\Delta}$ operator can be implemented with a setjump/longjump construct in C and assembler. The compositional relationships must collect and pass exception objects further on. Furthermore, the inclusion of exception operator does not change the semantics of the other CSP operators.

The $P \vec{\Delta} Q$ construct passes objects on exception to the exception handling processes. The objects embrace the exception type. If an exception is 'thrown' in P then this indicates an exceptional state in P. P returns immediately with the exception object indicating the type of the exception that was raised. A process may also return a collection of exception objects that were raised in its child processes. The exception collection being not empty indicates that the process has terminated unsuccessfully. An empty exception collection indicates successful termination. This distinction between successful and unsuccessful termination does not affect the semantics of the original CSP operators. Computations or primitive communication processes (see Section 3.6.8) convey points in the model where exceptions can rise. Thus, channels, barriers, and instructions are entry points for exceptions. Channels and barriers that are in exception, release their synchronization and throw exceptions at both sides of the communication. Therefore, erroneous channels or barriers cannot lock processes forever.

In design, exception handling can be composed with multiple (redundant) exception relationships. See Figure 3-31.

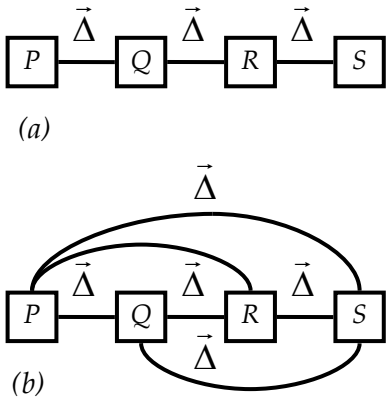


Figure 3-31 *(a) EXC construct,*
(b) over-specified EXC construct.

Multiple exception relationships are possible in a relaxed form where no set of exceptions are yet specified. If an exception is thrown in *P* then this exception may be caught by *Q*, *R*, or *S*. The exception handlers specify which exceptions that they can handle. Those exceptions that are not handled by an exception handler are delegated to the next exception handler until the exception is handled. This is detailed in Section 4.6.4.

3.6.6 Anonymous repetitions

As with many programming languages, this graphical language supports imperative repetition. Repetition in CSP is in a declarative style, called *recursion*. Recursion is a process with a function $P=F(P)$, which involves P like in $P=N;P$. Function F is any CSP term. This kind of recursion is 'named' and involves recursive hierarchies. This complicates imperative extensions. The recursion is also defined as a 'nameless' fixed-point recursion $\mu X.F(X)$ (μ is a Greek letter 'mu'), which does not involve recursive hierarchies. This recursive process is a *repetition* involving X, as in

$$\mu X.(P;X) = P;P;P;P;...$$

This represents a process with a fixed point at which the traces of events maps to itself. For example, $P = a \rightarrow SKIP$ identifies the function μX with the sets of traces

$$\left\{ \langle a \rangle^n \mid n \in \mathbb{N} \right\}$$

Another type of repetition that is mentioned by Roscoe (1998) is *P**, which performs *P* in an infinite sequence. This repetition is not very useful in practice, since no escape is possible. With μX one can escape from the repetition with the help of *if-then-else*; e.g. $\mu X.(P; X \triangleleft cond \triangleright SKIP)$ repeats *P* and terminates when *cond* becomes false. This allows for an imperative approach by which the if-then-else clause is part of an automaton. In an imperative language, this kind of repetition is called a *loop*. Therefore, the graphical modelling language incorporates μX, which is called a *loop process*. In Hilderink (2002), the loop process was labelled 'μ'. This symbol is often used in mathematics for all kind of things, such as micro = 10^{-6}. We choose a different symbol '○' (pronounced as 'loop'), which is more convenient to represent a loop.

A loop process that repeats process *P* until expression *cond* is false, is illustrated in Figure 3-32.

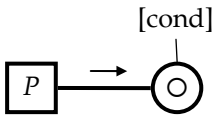


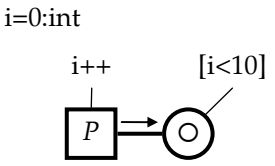
Figure 3-32 *Infinite recursion.*

The loop process is a special primitive process that embraces only one compositional relationship with one associated process. The loop process will always repeatedly execute the associated process until its conditional expression becomes false. The conditional expression is depicted as a guard label, which is evaluated in the order that is specified by the compositional relationship. The loop process terminates when the associated process terminates and the conditional expression is false.

Common kinds of loop constructs can be specified, as illustrated in Figure 3-33. These loop constructs can be realized in many programming languages. In the examples, the action bodies attached to the processes specify an automaton involving the loop process. The last example shows a repetition construct.

DO-WHILE construct:

```
int i = 0;
do { i++; P; } while (i<10);
```

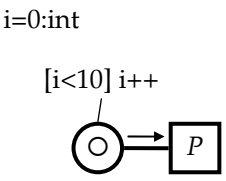


FOR construct:

```
for (int i=0; i<10; i++) { P; }
```

or

```
int i=0;
while (i<10) { i++; P; }
```



Repetition construct:

```
P(0) || P(1) || P(2)
```

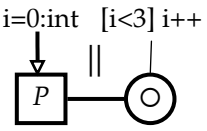


Figure 3-33 *Examples of different kind of loop constructs.*

Variable labels are depicted as floating port labels that are not part of the process interface. Figure 3-34a shows a loop process with variable labels *x* and *y*.

Since these variables are declared within the scope of the parent process, the loop process can use these variables in its guard body. Guard bodies may not always clearly show the data dependencies in the process architecture. It is possible to show the data dependencies between the loop process and the variable labels by using open arrows. Figure 3-34b

illustrates this. Only primitive data types are accepted by the guards of loop processes.

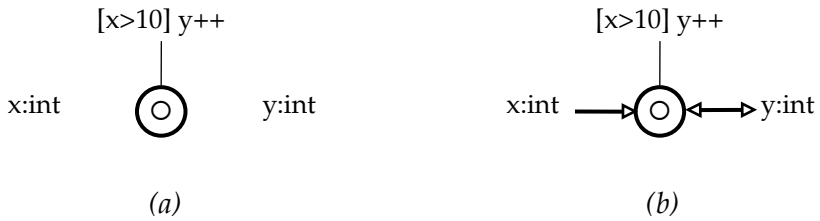


Figure 3-34 *Two equal loop processes:*
(a) without showing data dependencies,
(b) showing data dependencies.

3.6.7 Aliases

An *alias* is another name for the same thing. Multiple floating labels with different names can be part of the same relationship. These are aliases which names are dedicated to their specification context. This can improve the readability of the CSP diagram. An alias can be depicted in CSP diagrams using a line between two floating labels of the same type. See the examples in Figure 3-35.

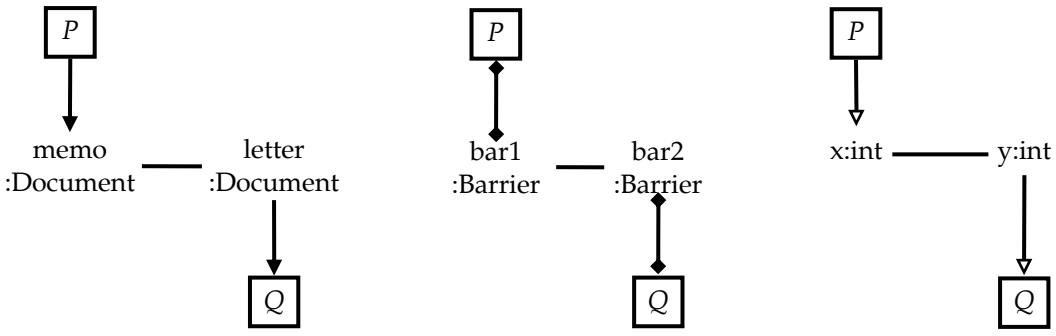


Figure 3-35 *Examples of aliasing:*
(a) channel communication
(b) barrier communication
(c) state communication.

In Figure 3-35a, messages are passed via the channel `memo` and `letter`. In Figure 3-35b a barrier communication is extended via `bar1` and `bar2`. In Figure 3-35c, the variable *x* is the same as *y*. When *x* is updated then *y* is updated as well, visa versa. In this example, a value from *P* is passed to *Q* via *x* and *y* after *P* has terminated and before *Q* is executed.

In a flat hierarchy, the alias labels must be ports in the process they are declared. Arrows are used in a deep hierarchy design. Flat and deep hierarchies are discussed in Section 3.7. The line can also have an identifier label on top of the line, which is another alias.

3.6.8 Primitive communication processes

This section introduces three special processes that express synchronization points in the CSP diagram. These special processes are called *primitive communication processes*.

Data channel, input and output

The primitive communication processes for data channel communication are depicted in Figure 3-36. Figure 3-36a shows channel input. The process reads from channel *c* and outputs the data to variable *x*. Figure 3-36b shows channel output. The process writes the content of *y* to channel *c*.

Figure 3-36 *Primitive communication processes on data channels:*
(a) data channel input, i.e. channel to variable,
(b) data channel output, i.e. variable to channel.

Call channel, call and accept

The primitive communication processes for call channel communication are depicted in Figure 3-37. The method that is involved is a choice or element of the service type. The method is specified in a guard body. Figure 3-37a shows the acceptance of any method on channel *c* that Figure 3-37b illustrates the acceptance of a particular method, namely method *on()*. The method *on()* is an element of *OnOff*. Figure 3-37c depicts a request of the method *on()* on channel *c*.

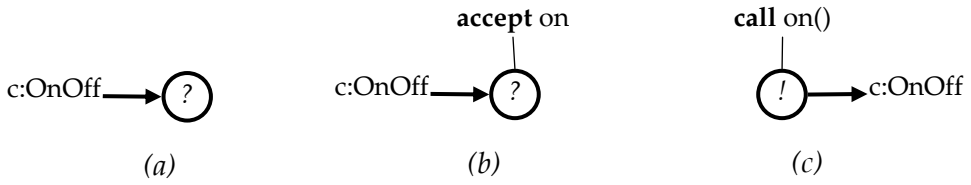


Figure 3-37 *Primitive communication processes on call channels:*
(a) server side accepts any method,
(b) server side accepts only method on(),
(c) client requests method on() on call channel.

The keywords `call` and `accept` in the action bodies, as illustrated in Figure 3-37, specify the behaviour of the primitive communication processes. The keyword `accept` without a specified method implies that any request will be accepted.

Barrier, sync

The primitive communication process for barrier communication is depicted in Figure 3-38. This example synchronizes on barrier *b*.

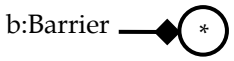


Figure 3-38 *Primitive communication processes on call channels.*

Examples

Examples of communication are given in Figure 3-39. Figure 3-39a shows communication via a data channel. Figure 3-39b illustrates a request for method *func()* via a call channel. One can specify arguments and a return value as shown in the example. Figure 3-39c shows barrier communication on which the variables x, y, and z depend.

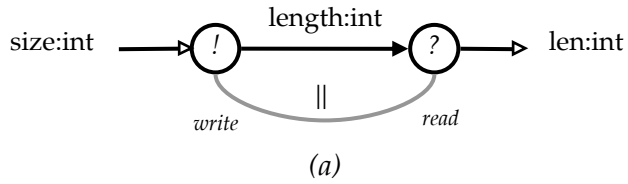
Multipurpose primitives

These primitive communication processes are useful for

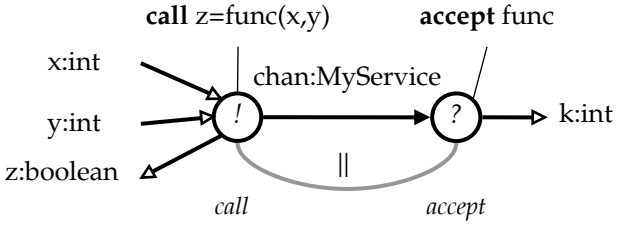
- showing the points of interaction between processes,
- transformation between channels and local variables,
- showing hardware access points (Hilderink et al., 1998),
- checking for deadlocks in design,
- checking for priority inversion problems in design,
- throwing exceptions on internal errors.

Sharing objects

Objects can be passed via a channel or barrier by the methods pass-by-value or pass-by-reference. These mechanisms are discussed in Appendix H. Pass-by-reference may improve performance on large objects on shared memory systems. This requires secure handling to avoid that no more than one process can access the shared object at the same time. Primitive data types do not suffer from this problem since pass-by-value is used, which is instinctively secure. Sharing objects is more restricted than sharing primitive data types. The rules in Section 3.8.1 also apply for objects.

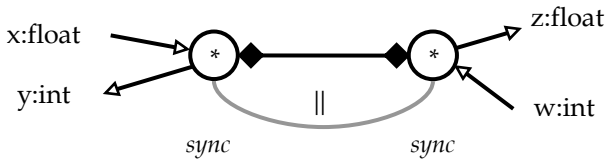


(a)



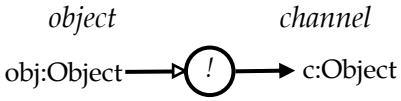
(b)

```
boolean func(int x, int y)
{
  ...
  k = x;
  ...
  return k;
}
```

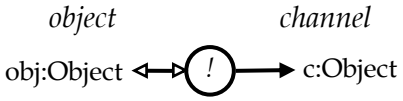


(c)

Figure 3-39 *Examples of communication relationships between channels, barriers, state variables, and input and output processes:*
(a) data channel communication: len = size,
(b) call channel communication: z = func(x,y),
(c) barrier communication: z=x, y=w.



(a)



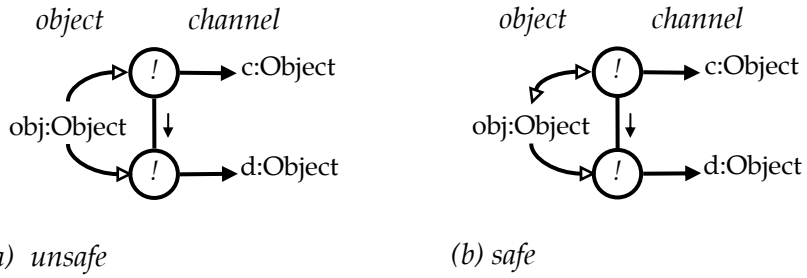
(b)

Figure 3-40 *Output process and state handling:*
(a) pass reference of obj via channel c; obj must not be used after output,
(b) pass reference of obj via channel c and return a clone of obj; obj can be safely used after output.

Figure 3-40a shows an output process that sends an object *obj* via a channel *c*. The reference of the object *or* its content is passed on. One cannot tell in advance which mechanism will be used at this side of the channel. Therefore, *obj* must not be used after the output, since its ownership may be released and claimed by another parallel processes.

Figure 3-40b shows an output process with a two-way open arrow. This implies that *obj* will be valid after output. In case pass-by-value is used, the original object is returned. In case pass-by-reference is used, a clone of the original object is returned. This allows other sequential processes to safely share *obj*.

Reading *obj* in parallel is not allowed, as illustrated in Figure 3-41a, since no guarantee can be given that pass-by-value is used. The ownership of the object can be passed on. A null-pointer may be the result, which will cause a null-pointer exception on use after the reference of the object was send. In Figure 3-41b the problem is solved by a two-way arrow between the first output processes and *obj*. The second output may safely use *obj*. After the second output, *obj* may not be used anymore, unless a two-way open arrow is used.



(a) unsafe *(b) safe*

Figure 3-41 *Example of parallel output using the same object:*
(a) illegal since second output may suffer from a nullpointer exception,
(b) illegal since second output uses a reused or cloned object

Channel communication contributes to a secure memory management concerning objects. Open arrows can be used such that the choice

between pass-by-value and pass-by-reference is made. This is useful for optimizing the communication within the process architecture.

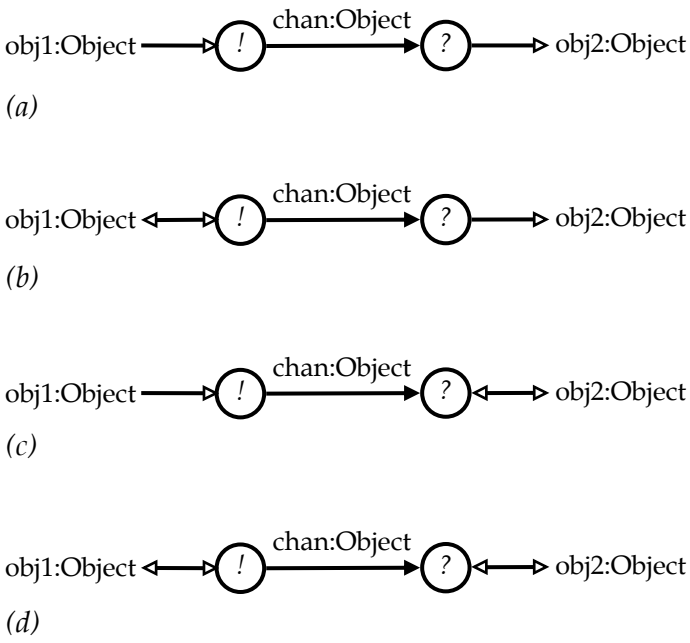


Figure 3-42 *Four different open arrow configurations between objects and input or output processes;*
(a) pass-by-reference is enforced,
(b) pass-by-reference is enforced and a clone is returned at the writer side,
(c) pass-by-value is preferred; in case the object cannot be copied then pass-by-reference is used,
(d) pass-by-value is preferred; in case the object cannot be copied then a clone is returned at the writer side.

Figure 3-42 shows four configurations and each configuration has its properties. Figure 3-42a passes the references from producer to consumer. Figure 3-42b is similar, but it returns a clone of *obj1* so that the next process can use *obj1* without explicitly creating a new object. Figure 3-42c illustrates a configuration that can pass the content of obj1 to *obj2*, since *obj2* can be reused. In case no pass-by-value can be used, pass-by-reference is used instead. Since the choice is uncertain, the producer may not use *obj1* after output. Figure 3-42d depicts a configuration by which pass-by-value is preferred. Object *obj1* may be used after the output. In

case a reference is send then a clone is returned to the producer. This is similar with call channels and barriers.

3.7 Hierarchies

3.7.1 Ambiguity and Unambiguity

A CSP diagram is ambiguous when it describes more than one algebraic expression. The ambiguity is caused by unspecified compositional relationships for which the operator can be a choice of one out of more possibilities.

Consider a model with three processes *P*, *Q* and *R* as shown in Figure 3-43. In this example, the user specifies that process *P* should be executed before *Q* and *Q* should be executed in parallel with *R*. The behaviour between *P* and *R* is not specified by the user and leaves open certain ambiguity. This means that there is more than one valid solution and any of these solutions is accepted.

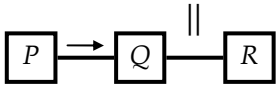


Figure 3-43 *Example of a model with ambiguity.*

Here, the valid solutions are $P;(Q\|R)$ and $(P;Q)\|R$. Every solution should satisfy the requirements. If a solution exists that does not satisfy the requirements then further refinement steps are necessary in order to exclude the invalid solution from the set of solutions.

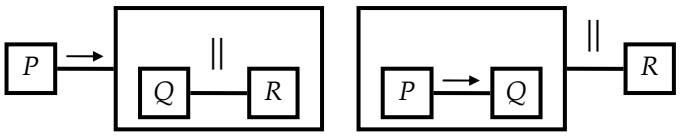
Ambiguity can be avoided by a complete graph or by using hierarchies of processes. A *complete graph* is a diagram for which all interrelationships between all processes are user-specified or uniquely derivable. A complete graph can be transformed into a hierarchical diagram and visa versa. A hierarchical diagram simplifies a complete graph.

The graphical modelling language supports hierarchies in three different ways:

- *Deep hierarchical modelling* abstracts away detail by describing different levels of processes. This approach simplifies a design by hiding detail, which is based in depth browsing.
- *Flat hierarchical modelling* shows the insight of processes in one model in order to understanding the behaviour of a protocol of interaction. This is based on using parenthesis, which is based on flat browsing.
- *Mixture of deep and flat hierarchical modelling* shows the best of both.

In any of these hierarchical modelling approaches one can model a complete graph at a particular level in the hierarchy. *Complete graph modelling* does not directly specify hierarchies, but it gives rise to anonymous compositional hierarchies during implementation.

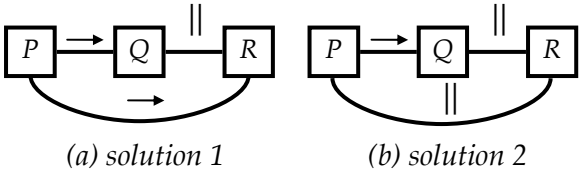
Solutions can be depicted by nested hierarchical processes as illustrated in Figure 3-44a and 3-44b. This deep hierarchy is depicted here in transparent rectangles in order to illustrate the hierarchy and the relationships between the processes in one diagram. In general, transparent rectangles are notation unfriendly. They can easily occupy an unnecessary amount of space in a diagram. Instead a flat hierarchy should be designed using parenthesizing symbols, which is shown later in this section.



(a) solution 1 *(b) solution 2*

Figure 3-44 *Unambiguous solution using deep hierarchies:*
(a) $P;(Q \| R)$
(b) $(P;Q) \| R$

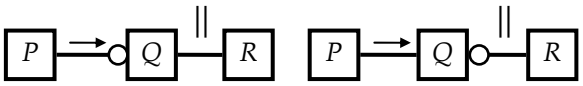
A unique and unambiguous solution can be achieved by specifying a relationship between *P* and *R*, as shown in Figure 3-45. This is a complete graph. Each step towards a complete graph eliminates ambiguous interpretations.



(a) solution 1 *(b) solution 2*

Figure 3-45 *Unambiguous solutions by complete graph modelling:*
(a) $P;(Q \| R)$
(b) $(P;Q) \| R$

This way, unambiguous compositions require many relationships to specify a unique solution. All these lines would make the model complex and likely unreadable. In order to keep the model simple, we introduce the *parenthesis symbol* on compositional relationships. See Figure 3-46. This is represented by an open dot '○' (concatenation of '(' and ')') at the peer-end of the compositional relationship. In these examples, Figure 3-44a is equal to 3-45a and 3-46a. Figure 3-44b is equal to 3-45b and 3-46b. Using parenthesis symbols minimizes the number of relationships. The parenthesis symbol can represent an anonymous process whereby its identifier is unspecified by the user. An unspecified identifier is hidden by default.



(a) solution 1 *(b) solution 2*

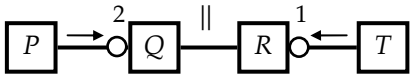
Figure 3-46 *Unambiguous solutions using parenthesizing relationships:*
(a) $P;(Q \| R)$
(b) $(P;Q) \| R$

A compositional relationship with an open dot at one end or both ends becomes a directed relationship. This is a *parenthesizing relationship*. A process to which the open dot is connected belongs to a parenthesized relationship.

3.7.2 Indexed parenthesizing relationships

A dot in the parenthesizing relationships can be indexed with a value greater than zero, i.e. $i \in \mathbb{N}^+ \setminus \{0\}$ (only positive natural numbers without 0). The index is an instrument useful for determining the levels in hierarchy. Its value can be altered by an algorithm that allows for reallocating relationships in a CSP diagram while maintaining its hierarchy and its algebraic expression. This is discussed in Section 3.8.2.

For example, see Figure 3-46. Indexes greater than 1 should be rendered next to the dot to indicate the index. A dot with no index implicitly means that it has index 1. A *zero-order* relationship has index 0, which implies no parenthesis symbol. A parenthesizing relationship with index 1 is said to be a *first-order* relationship, index 2 is a *second-order* relationship, etc.



$$P;(T;(Q \parallel R))$$

Figure 3-47 *Indexed parenthesizing relationship with index i.*

Figure 3-47 illustrates indexed parenthesizing relationships for describing more complex compositions or algebraic expressions using a minimal number $(n-1)$ of interrelationships, with n processes. A model is usually analyzed or read starting at the first-order relationships towards second-order relationships. A systematic method is discussed in Section 3.7.4 that allows the user to brows (or read) the hierarchy by stepping from zero-order relationships to higher-order relationships.

3.7.3 Compositional undefined relationships

In reality and virtually, all processes are compositionally related to each other. The compositional interrelationships that are specified by the user are visual in the CSP diagram. The user-unspecified interrelationships are hidden and they are internally determined by the tool. A CSP diagram that is conflict-free results always in a computational model. A process that is not connected by user-specified interrelationships is called *compositional undefined*. A compositional undefined process has no neighbours in the visual view. Compositional undefined processes can be executed in any order, i.e. in parallel or in some sequence. The behaviour also depends on communication, as specified by the communication diagram. This is in compliance with the computational models of block diagrams. Block diagrams are like CSP diagrams without user-specified compositional relationships.

By not specifying connections, we mean that we do not care what the execution order is and therefore we let the design tool decide. The criterion that is applied here is an internal choice between (equally- or unequally-prioritized) parallel and sequential operators. The choice could be influenced by the communication relationships between the compositional undefined processes. The choice of the hidden operator must be valid, i.e. each solution must be compositional conflict-free (Section 3.8.4). Hidden interrelationships can be visualized (e.g. using transparent lines) to show to the user which operator has been chosen by the tool. Hidden interrelationships can also be parenthesizing in order to simplify the view. See Figure 3-48 where operator $\oplus \in \{ \|, \overleftarrow{\|}, \overrightarrow{\|}, \rightarrow, \leftarrow \}$.

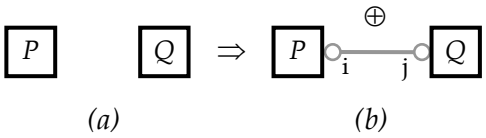


Figure 3-48 *Undefined relationship between two processes:*
(a) compositional undefined processes,
(b) choice of operator and interrelationship visualized.

The graphical modelling language allows us to express the detail of the execution framework. The ability of visualizing the hidden relationships is an ultimate solution for debugging and studying the behaviour of the model at design level.

Compositional undefined processes can be connected by a zero-order interrelationship without an operator in order to group these processes. See Figure 3-49. This is useful when the group is being parenthesized. Again, the operator of an undefined interrelationship can be determined by the design tool.

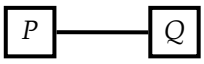


Figure 3-49 *A grouped undefined relationship.*

3.7.4 Deep hierarchies versus flat hierarchies

A process can contain other processes. Therefore, a CSP diagram can contain hierarchies of sub-diagrams. A sub-diagram is a process that is again described by a communication diagram and a composition diagram. The scope of a sub-diagram is determined by its parent process. In CSP diagrams, nested hierarchies are created by deep hierarchical modelling (encapsulation and hiding), by flat hierarchical (encapsulation but not hiding) modelling, or a mixture of the both. See Section 3.7.1.

Although deep hierarchical modelling simplifies a design by hiding detail, it may make the understanding of behaviour complicated when it involves processes at deeper levels in the hierarchy. Flat hierarchical modelling solves this problem, but a flat hierarchy may complicate the design by offering too much detail. However, in a flat hierarchy any process and relationship that is irrelevant for describing a protocol of communication can be hidden from the view. This is called *compression*. Compression is useful for describing the behaviour of processes as observed via their process interfaces.

The user can apply a suitable mix of deep and flat hierarchical modelling. A straightforward transformation between deep and flat hierarchies exists that can assist the user in determining an appropriate view for analyzing a particular specification or behaviour.

Techniques are available that allow the user or the tool to transform a deep hierarchy into a flat hierarchy and visa versa. These techniques can also be applied to mixed (deep/flat) hierarchies. Any transformation results in the same computational expression with differences in transparency. The transformation from a deep hierarchy to a flat hierarchy starts with selecting one sub-process in the sub-diagram that has only outgoing parenthesizing and/or zero-order relationships. If such a sub-process is found then the process is connected with a parenthesizing relationship with index 1 to that sub-process. For example in Figure 3-50a, such a sub-process is *R* in parent process *Q*. The parent process can be transformed from a rectangle to a parenthesizing symbol of the newly created relationship. This is illustrated in Figure 3-50b. The parenthesizing symbol inherits the parent process identifier. Eventually, the reallocation rules can be used to reallocate the relationship to other sub-processes in *Q*. See the reallocation rules Section 3.8.2. Illegal indexes, i.e. *index* < 1, occur when a wrong starting process was selected that has an incoming parenthesizing relationship. If reallocation rules are applied correctly then the indexes do not change the computational expression. This procedure can be repeated until the deep hierarchy has been transformed to a flat hierarchy.

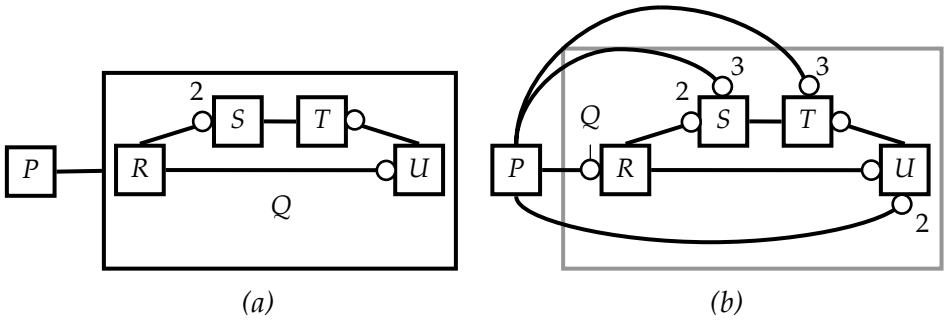


Figure 3-50 *Example of flattening hierarchy:*
(a) P related to Q (Q is transparent),
(b) P related to the processes in Q.

The reverse procedure is also interesting. The reverse procedure can be used to determine design conflicts in more complex designs. A group of processes connected with zero-order relationships forms a hierarchy by default. For example, the processes *S* and *T* in Figure 3-51a are merged into a new process which is identified as *ST*. See Figure 3-51b. The relationships are reconnected to the anonymous process and the index between *R* and *ST* must be decremented by 1. This procedure can be continued until a single process and only zero-order relationships remain. See Figure 3-51c.

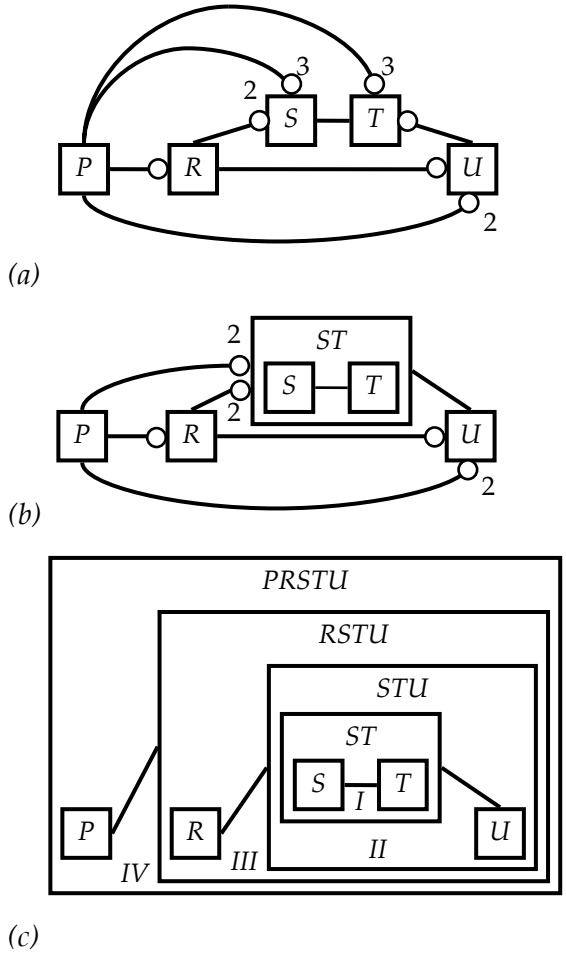


Figure 3-51 *Example: creating deep hierarchies:*
(a) P is related to all processes in a flat hierarchy,
(b) merging first-order relationships first,
(c) complete deep hierarchy.

The relationships $(R,ST,\oplus)$ and $(R,U,\oplus)$ in Figure 3-51b can be merged only if both relationships have the same operator, i.e. one relationship is redundant. If these relationships have different operators then these operators are in conflict and the model has an error. This procedure allows checking the model for compositional conflicts. We assume that these operators are the same. The relationship between *P* and *ST* can be removed due to redundancy. This procedure ends until a single process is the result and all parenthesized symbols have been eliminated. See Figure 3-51c.

Identifier prefixing

Relationships and identities in a CSP diagram must not disappear on translating a deep hierarchy into a flat hierarchy. Also the translation of a flat hierarchy into a deep hierarchy must not create new information.

The names of ports are unique in the process they are defined. However, these names can be the same between processes. This is not a problem in a deep hierarchy, but it may conflict on a flat hierarchy. The solution is to distinguish the names in a flat hierarchy.

An identifier label can be prefixed with the process name to which it belongs. Therefore, an identifier label can have the following format:

process.id:Type

Here, *process* is optionally used to distinguish *id*s with the same name from different processes. In a flat hierarchy, *process* can contain other prefixes separated by '.'. In case the entire name becomes too large due to many prefixes, a word wrap can be used after '.' and the name is depicted by multiple lines.

In the following example are the operators on the compositional relationships omitted.

Figure 3-52 shows a transparent CSP diagram using a deep hierarchy of processes. The independent variables *i* are used in the processes *p:P*, *q:Q*,

r:R, and *s:S*. Some action bodies increment or decrement *i* for some reason. The meaning of *i* is not important in this example.

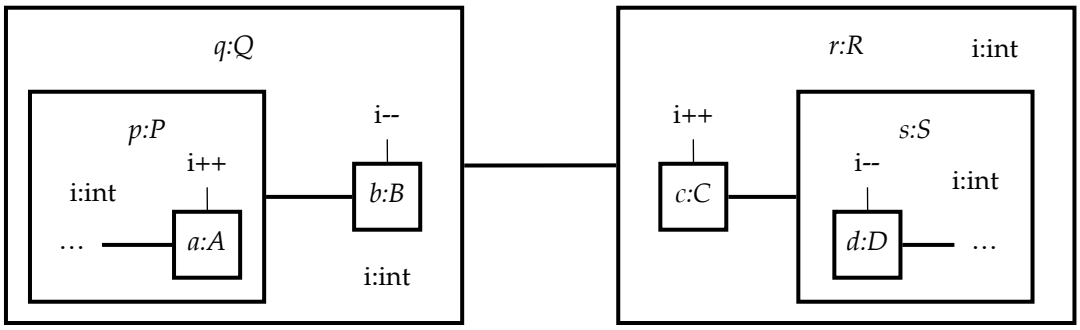


Figure 3-52 *Deep hierarchy with equal variable names.*

Applying these translation steps to Figure 3-52 results in Figure 3-53. This example demonstrates the result of prefixing floating variable labels in order to maintain their locality.

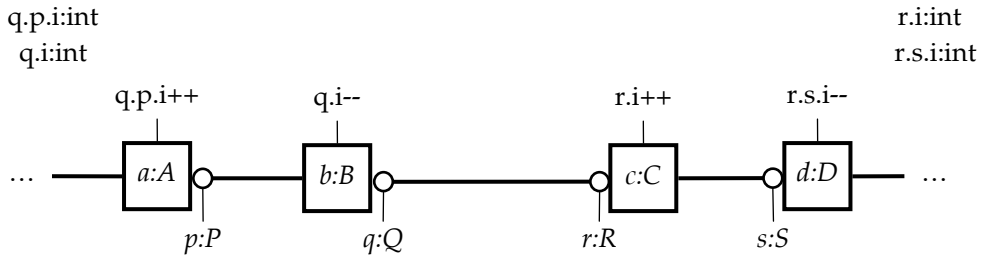


Figure 3-53 *Flat hierarchy with prefixed variable names.*

Variable *i* in process *q* is named *q.i* and *i* in *r* is named *r.i*. Since *i* in process *p* is also nested in *q*, a double prefixing must be used, namely *q.p.i*. This is similar for *i* in *s*, namely *r.s.i*. The variables in the action bodies cannot use prefixes. They do not need prefixes because they are attached to a process which is parenthesized. Therefore, action body of *a:A* cannot use a variable in a different process; e.g. it refers to *q.p.i* but it cannot refer to *r.s.i*.

Translating a deep hierarchy into a flat hierarch also affects the communication diagram. Channels and barriers that are passed via ports are aliased with prefixed identifiers. Figure 3-54 shows a merged

communication and compositional diagram based on a deep hierarchy as in Figure 3-52 (above).

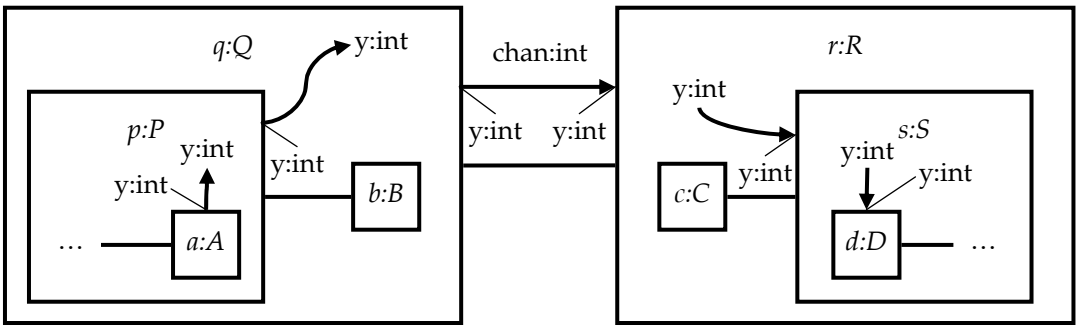
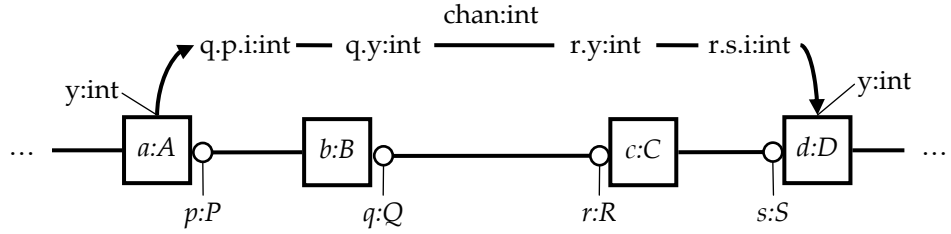
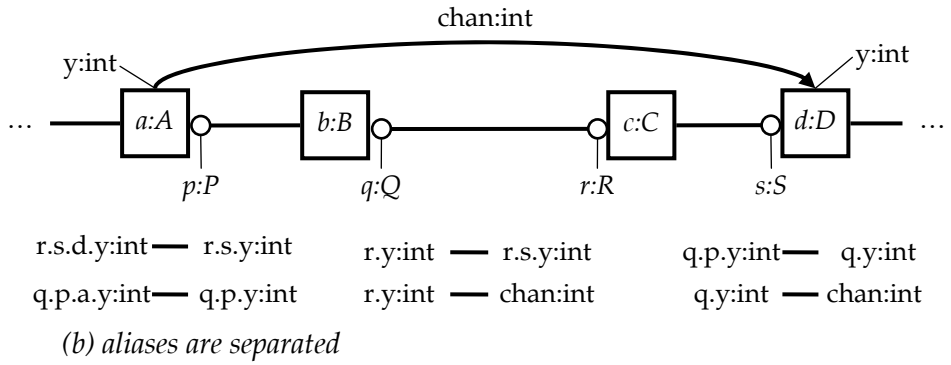


Figure 3-54 *Deep hierarchy with equal port names.*

After translation, the result can be depicted in two different ways. Figure 3-55a shows the result based on aliases that extend the channel. Figure 3-55b shows the result based on aliases that are separated from the channel and these aliases are depicted next to the design. This makes the diagram better readable in case the ports are not of immediate concern. Hiding the ports in Figure 3-55a results in Figure 3-55b.



(a) aliases extend the channel



(b) aliases are separated

Figure 3-55 *Flat hierarchy with prefixed variable names.*

3.8 Analysis techniques and rules

The graphical modelling language includes rules to avoid illegal designs. In this section, analysis techniques are described that apply these rules to determine the validity of the process architecture. These analysis techniques can be applied by the user or it can be automated by a design tool.

3.8.1 State communication rules

Simultaneous updating of the same state variable is forbidden. State communication must not cause race conditions between reading and writing. State variables are allowed to be read in parallel. Figure 3-56 shows a mixed CSP diagram in which these safety rules are depicted. Here, the state variable *x* is shared by *P*, *Q*, *R*, and *S*. The safety rules for *x* are expressed by the choice of operators in this general pattern of compositional relationships. This pattern scales for more or less processes.

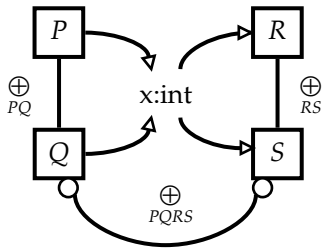


Figure 3-56 *Rules of using variables safely.*

Let *OP* be a set of all operators

$$OP = \{\rightarrow, \leftarrow, ||, \overleftarrow{||}, \overrightarrow{||}, \Box, \overleftarrow{\Box}, \overrightarrow{\Box}, \overrightarrow{\Delta}, \overleftarrow{\Delta}\}$$

The operators on the compositional interrelationships are restricted to

$$\underset{PQ}{\oplus} \in OP - \{||, \overleftarrow{||}, \overrightarrow{||}\},\ \underset{RS}{\oplus} \in OP,\ \underset{PQRS}{\oplus} \in OP - \{||, \overleftarrow{||}, \overrightarrow{||}\}$$

Thus, the interrelationships $\underset{PQ}{\oplus}$ and $\underset{PQRS}{\oplus}$ are not allowed to be performed in parallel.

The rules that apply to two-way open arrows are given in Figure 3-57.

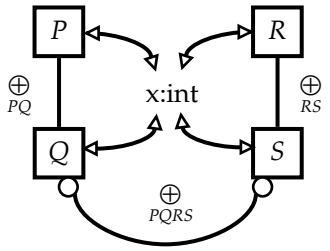


Figure 3-57 *Rules of sharing an object with two-way open arrows.*

The operators on the compositional interrelationships are restricted to

$$\underset{PQ}{\oplus} \in OP - \{||, \overleftarrow{||}, \overrightarrow{||}\}, \ \underset{RS}{\oplus} \in OP - \{||, \overleftarrow{||}, \overrightarrow{||}\}, \ \underset{PQRS}{\oplus} \in OP - \{||, \overleftarrow{||}, \overrightarrow{||}\}$$

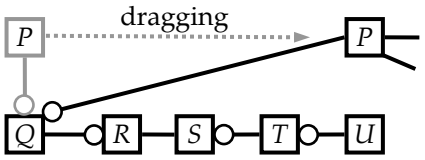
The user or design tool must apply these rules to check whether or not they are violated. Violation results in an error. It is obvious that parallel writing, or parallel writing and reading may cause a race hazard that corrupts the data.

3.8.2 Reallocation rules

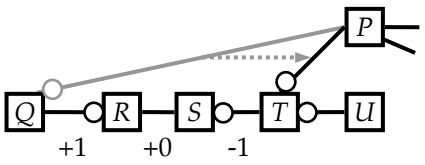
In process architectures, as in CSP diagrams, processes are usually located near to the processes with the highest relationship density. This should give the user the freedom to move processes around while the model grows. The connections between processes are usually kept short and crossings should be avoided as much as possible. However, reallocating processes can result in longer connections and possibly create crossings with other connections. In case a process is related to a group of processes, a technique is presented that allows the process to be related to the nearest process in the group, while preserving the algebraic expression. This technique can significantly shorten the connection or eliminate crossings, which make the model better readable.

Example

Figure 3-58a shows a process *P* that is originally related to process *Q*, but it has been moved to another location in the diagram closer to other processes it is related to. These other processes are not shown in this figure. The technique presented here shows that the relationship between *P* and *Q* can be reallocated to a relationship between *P* and *T* as illustrated in Figure 3-58b.



(a) moving process P in CSP diagram



(b) reallocation of connection

Figure 3-58 *Example of dragging a process in a CSP diagram and reallocating its connection:*
(a) moving process P from left to right,
(b) reallocation its connection by following rules.

Reallocation rules

Each reallocation step along a compositional relationship represents an index increment, an index decrement, an index increment and decrement, or index equality. Figure 3-59a-f show six basic rules for reallocating relationships.

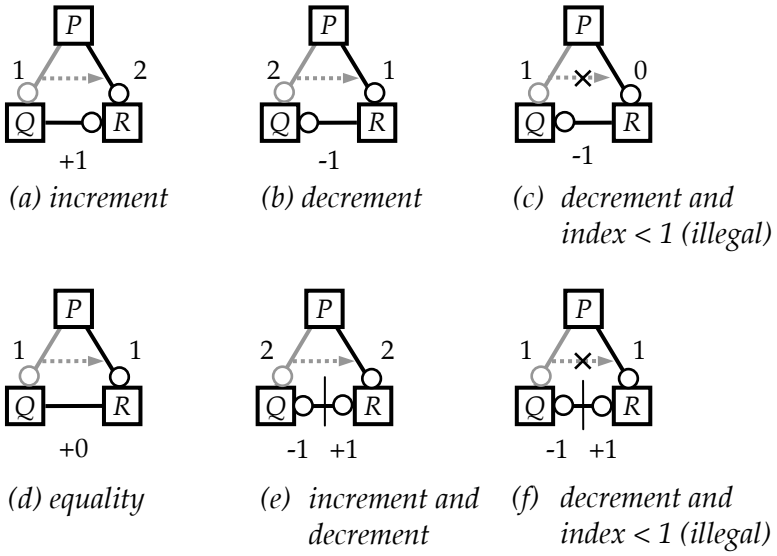


Figure 3-59 *Reallocation rules.*

The operators above the interrelationships have been omitted because they do not really matter for this technique. The operators are assumed to be specification conflict-free and they are assumed to satisfy the requirements.

In Figure 3-59 the rules c and f illustrate the boundaries of reallocation. Once a relationship is given a parenthesis symbol then its index is 1 or higher. Illegal indexing (*index* < 1) indicates an illegal reallocation in that direction and indicates a dead end. In Figure 3-59c and 3-59f, one can see that process *R* is not a member of the group and therefore reallocation should not be applied.

For example, the rules are applied to Figure 3-58a. Each step is illustrated in Figure 3-60a-d. The rules cannot be applied when the index becomes illegal, as illustrated in Figure 3-60e.

These reallocation rules provide a systematic method which can be automated by the design tool when the user drags a process in the diagram for which the tool automatically determines the shortest connections.

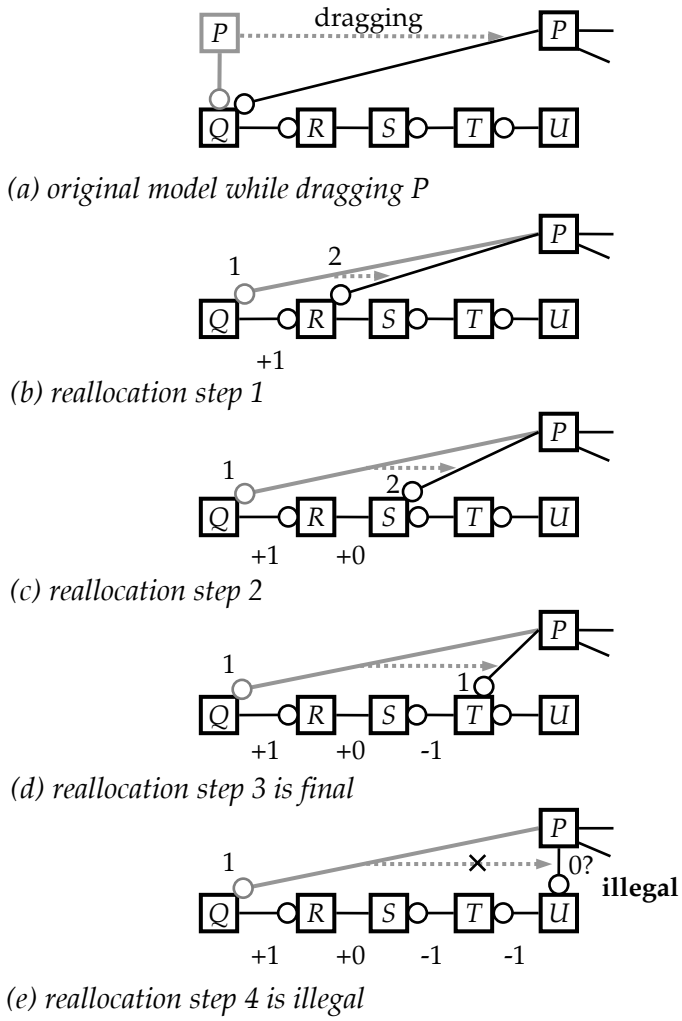


Figure 3-60 *Example of reallocating a connection by incremental step.*

Algebraic expression

Figure 3-60a-d are represented by one algebraic expression for which any operator can be applied on the interrelationships. Let $\overset{\oplus}{}_{PQ}$ be an operator between P and Q with $\overset{\oplus}{}_{PQ} \in \{\rightarrow, \leftarrow, \|, \overleftarrow{\|}, \overrightarrow{\|}, \square, \overleftarrow{\square}, \overrightarrow{\square}, \overrightarrow{\Delta}, \overleftarrow{\Delta}\}$. One can derive the following algebraic expression using the *compositional analysis rule* as defined in appendix G.

$$X = R \underset{RS}{\oplus} S$$
$$Y = Q \underset{QX}{\oplus} X \underset{XT}{\oplus} T$$
$$Z = P \underset{PY}{\oplus} Y \underset{YU}{\oplus} U$$

The process Z is an algebraic expressions in compressed form. Ambiguity may exist between P and U, and between Q and T. After expanding, the complete algebraic expressions is

$$P \underset{PY}{\oplus}\left(Q \underset{QX}{\oplus}\left(R \underset{RS}{\oplus} S\right) \underset{XT}{\oplus} T\right) \underset{YU}{\oplus} U$$

Any ambiguity is conserved in this algebraic expression. A CSP diagram can make ambiguity better observable. The previous algebraic expression is depicted in Figure 3-61.

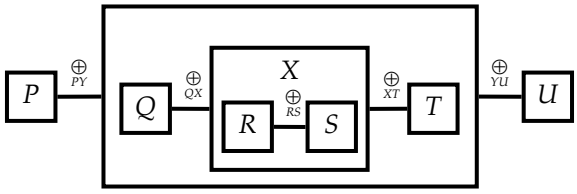


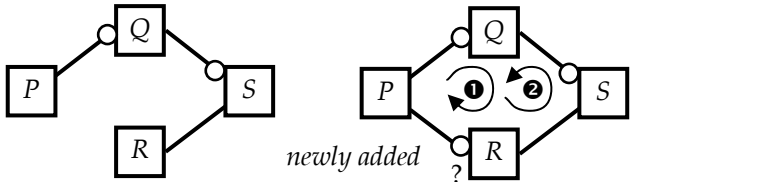
Figure 3-61 *Transparent representation of* $P \underset{PY}{\oplus}\left(Q \underset{QX}{\oplus}\left(R \underset{RS}{\oplus} S\right) \underset{XT}{\oplus} T\right) \underset{YU}{\oplus} U$.

Ambiguity can be observed independently for each level in the hierarchy. In this example, there are no lines between P and U, or between Q and T. These undefined interrelationships can be uniquely derived from the operators on the user-defined relationships, or they can be determined by the tool.

3.8.3 Balanced and unbalanced parenthesized cycles

Cycles of parenthesizing relationships in a design should be *balanced*. This means that in a cycle the weight (sum of indexes) of parenthesizing relationships pointing in one direction should compensate the weight of parenthesizing relationships pointing in the other direction. If these parenthesizing relationships do not compensate opposite parenthesizing relationships in the cycle then one cannot completely determine the algebraic expression of this so-called *unbalanced cycle*. In an unbalanced cycle, the algebraic expression reasoned in one direction is not the same as the algebraic expression reasoned in the other direction.

The following technique can be applied to test whether cycles of parenthesizing relationships are balanced or not. This technique allows the design tool to give an error when a wrong index was specified. This technique can also be used to determine the right index automatically on a newly added interrelationship. Figure 3-62 illustrates an example of a design (Figure 3-62a) where the user adds an interrelationship which creates a cycle (Figure 3-62b). The operators on the interrelationships that do not matter to explain the example are omitted from the illustrations.



(a) previous design *(b) newly added interrelationship*

Figure 3-62 *Balancing cycles or determining indexes.*

The procedure of checking or determining the index is as follows. The weight of parenthesizing relationships pointing clockwise (see ❶ in the example) is 2 and the number of parenthesizes pointing anti-clockwise (see ❷ in the example) is 1. After subtraction the result is 2-1=1 whereas in a balanced cycle the difference should be 0. In this example, it is not difficult to see that the index of the new parenthesizing relationship between *P* and *R* should be 2.

As illustrated above, a systematic approach exists that determines the index of, for example, a newly added parenthesizing relationship that forms a cycle. This technique should be carried out for each cycle. In case two connected cycles determine two different indexes for a shared relationship then the model has a structural error. This indicates that the model needs to be revised.

Of course, the user can override any automatically generated index with a different and valid index on order to specify a different hierarchy.

3.8.4 Compositional conflicts

A compositional conflict is a failure in the design which may cause a specification mismatch, deadlock, or a performance bottleneck in the process architecture. A compositional conflict is defined as follows:

Definition (compositional conflict): A *compositional conflict* is a failure formed by two compositional relationships that are in contradiction.

Three different kinds of compositional conflicts are categorized:

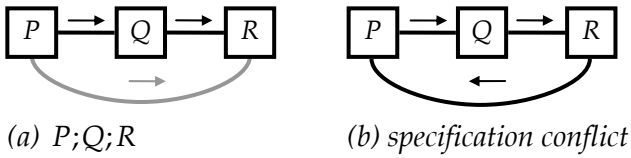
- *Specification conflict*. A process architecture that suffers from a compositional relationship mismatch between processes cannot be code-generated or model-checked because no solution can be found.
- *Deadlock conflict*. A process architecture that is specification conflict-free and suffers from a sequential relationship mismatch between communications is a deadlock.
- *Priority conflict*. A process architecture that is specification conflict-free and suffers from a priority mismatch between communications introduces bottlenecks that slow down the reactiveness or responsiveness of the process architecture.

The graphical modelling language is expressive enough to detect these compositional conflicts in designs, as is shown below. A systematic approach exists for each conflict that is based on a similar technique applied to different contexts.

A process architecture that suffers from a specification conflict requires a redesign. A process architecture that suffers from a deadlock conflict or a priority conflict may require a redesign or it may require a buffered channel to solve the conflict. In some worst-case timing, a priority conflict can cause starvation, which can evolve to livelock or deadlock. Hence, the user is interested whether or not the process architecture is compositional conflict-free. The detection of compositional conflicts in process architectures can be automated by the design tool following the rules, as described in the following sub-sections. The design tool could warn the user if an incorrect operator or a wrong index is applied.

Specification analysis

Specification analysis is the examination of specification conflicts in the design. For example, Figure 3-63a shows a diagram with sequential relationships; (P,Q,R,$\rightarrow$). The relationship between P and R can be derived from the user-specified path, e.g. (P,R,$\rightarrow$). During design, the user can decide to specify an interrelationship between P and R. See Figure 3-63b. The sequential operator causes a compositional conflict. In this case (P,R,$\rightarrow$) and (P,R,$\leftarrow$) are in contradiction. Two or more relationships between two processes with different operators, including derived interrelationship, are forbidden. Thus, Figure 3-63b suffers from a specification conflict.



(a) P;Q;R *(b) specification conflict*

Figure 3-63 *Example of specification conflict:*
(a) derived relationship,
(b) overriding relationship that is in contradiction.

Figure 3-63 shows triangular cycles. A triangular cycle is a cycle of three processes that are completely connected by compositional interrelationships. The compositional analysis rule in Appendix G is applied for each triangular cycle in the CSP diagram. Any *not conflict-free* result is a specification conflict.

Deadlock analysis

A process architecture that is specification conflict-free may still suffer from a deadlock conflict. The deadlock causes the program to stop at run-time. Deadlock is a synchronization conflict between rendezvous communication and sequential relationships between these communications.

Definition (deadlock): A *deadlock* is a failure of two processes to cooperate with each other because of not being able to agree on a common event, although they are willing to participate in other events.

A good solution in finding deadlocks is using formal deadlock checkers. For example, a process architecture could be translated into readable CSP and analyzed by a tool like FDR (2004). The tool will prove if the design is deadlock-free. This is only possible if the process architecture is specification conflict-free, but not necessarily deadlock-free.

During design it would be convenient to detect and to warn about the presence of deadlocks before finishing the model. Here, a technique is described for finding and for reasoning about deadlocks in the design phase of the project. This is based on the conflict-free checking techniques involving primitive communication processes. Deadlock can be traced in the CSP diagram before run-time or code generation. The primitive communication processes play an important role in deadlock analysis.

For example, Figure 3-64a shows a model that is specification conflict-free but suffers from deadlock. In this case, the communication diagram and composition diagram are depicted in one model. Figure 3-64b illustrates that this model is specification conflict-free by applying the specification analysis technique. The merging of temporal processes, being part of the analysis technique, is illustrated with the help of dotted rectangles. These dotted rectangles are not part of the CSP diagram. Since the model is specification conflict-free, the model can be code generated and executed.

At run-time, the primitive communication processes synchronize on channel communication and on barrier synchronization. They maintain in a locked state forever, they deadlock. In the procedure of finding deadlock conflicts we define a preliminary step that allows us to detect these kinds of conflicts.

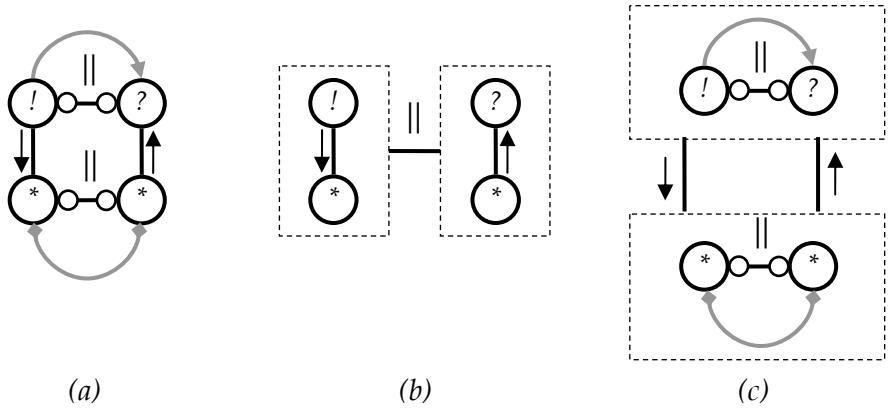


Figure 3-64 *Example of deadlock conflict:*
(a) original design,
(b) specification conflict-free,
(c) compositional conflict using rendezvous processes.

Given the fact that channel-ends and barrier-ends always rendezvous with each other on communication, this instance of communication represents a *rendezvous process* at run-time. The compositional relationships between the primitive communication processes must be (equally- or unequally-prioritized) parallel. Visualizing rendezvous processes is *only* used for deadlock analysis. A rendezvous process merges both ends of a channel or barrier to one anonymous process. See the dotted rectangles in Figure 3-64c. Note that parenthesizing symbols are ignored and the design is not altered by this analysis technique.

For this analysis technique it is useful that the model is flattened.

1. Merge all pairs of primitive communication processes together into rendezvous processes. This is called a *scenario*. In case primitive communication processes are part of an alternative relationship, this must be treated as a choice of communication. Each choices results in another scenario. Multiple scenarios must be analyzed separately.

2. For each scenario the compositional relationships between the rendezvous processes must be checked for sequential conflicts. A *sequence conflict* is a compositional conflict whereby sequential relationships are in contradiction. This is based on the same

technique as for detecting specification conflicts concerning only sequential relationships. This is similar as illustrated in Figure 3-64.

Any deadlock in the process architecture can be shown in the design by highlighting the paths in, for example, the colour red.

This analysis technique can incorporate logical decisions, such as conditional guards and if-then-else constructs. This may results in a large amount of scenarios. The benefit of this technique is that each scenario can be checked individually and it does not cause a state explosion in the model-checker. This analysis technique is a similar to the deadlock-checker developed by Martin and Jassim (1997), which is based on a graph of states. This technique is used by the model-checker FDR (2004). CSP diagrams could be checked by FDR and feedback from FDR can be depicted in the diagrams, which shows the user the conflict in the design. Anyway this graphical modelling language offers the notation to depict deadlock in CSP diagrams.

Priority inversion analysis

With a similar technique as described in the previous section one can find priority conflicts. Priority conflicts are caused by unequally-prioritized parallel operators that are in contradiction.

An example is shown in Figure 3-65. This example suffers from a priority conflict between rendezvous processes, which means that the model suffers from a *priority inversion problem*. The priority inversion problem can have a significantly burden on the performance of the program. It this case, a higher priority process can be blocked by the lower priority process and as a result of that it may be likely that the deadlines of the higher priority process cannot be met.

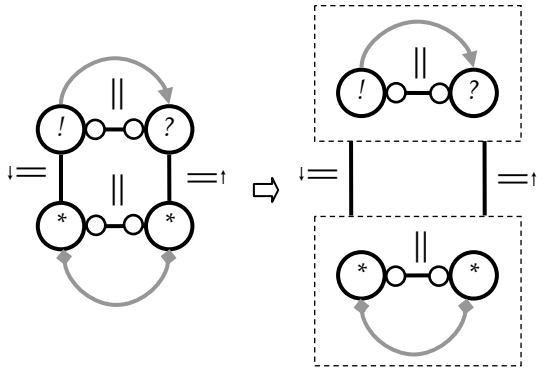


Figure 3-65 *Priority conflict = priority inversion problem.*

Usually, eliminating priority conflicts by correcting unequally-prioritized compositional relationships will result in a better design. In case where priority inversion is inevitable in the design, which is possible, one could solve the problem by use a buffered data channel between processes executing at different priorities in order to avoid blocking. Buffering does not apply to call channels and barriers, for which a redesign is required to solve the priority conflict. This information can be used by the design tool to change rendezvous data channels into buffered data channels, which solve priority conflicts. With this information, together with the frequency of the processes, one can determine over-sampling or sub-sampling type of buffered data channels.

3.8.5 Companionship between communication and composition

Compositional relationships are orthogonal to communication relationships. However, the compositional relationships can determine the need for buffered communication. The valid configurations for data channels are depicted in Figure 3-66a-f.

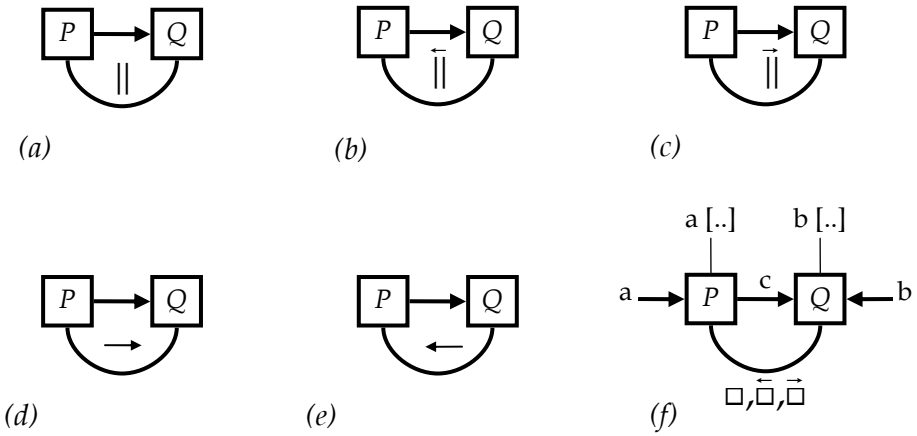


Figure 3-66 *Valid data channel configurations:*
(a) rendezvous channel or buffered fifo channel
(b) buffered sub-sampling channel,
(c) buffered super-sampling channel,
(d) buffered fifo channel,
(e) buffered fifo channel with initial value,
(f) buffered fifo channel with initial value.

For each configuration, the communication relationship and compositional relationship are depicted in one figure. Figure 3-66a-c, buffered channels can improve the performance of a process architecture in circumstances where unbuffered channels cannot sufficiently decouple multiple frequencies. Buffered communication has the side effect that it reduces the number of context-switching or solves priority conflicts. One should be careful with buffering in Figure 3-66a, because buffered communication can jeopardize the reactivity and responsiveness of a concurrent system. Buffered channels can compensate latencies on external channels. Figure 3-66d-f are useful when existing processes with existing process interfaces must be connected in compositions other than parallel. In case a channel is connected between guarded processes in an alternative construct, the channel *c* must not be part of the guards (see Figure 3-66f).

Communication via call channels and barriers are restricted to equally- and unequally-prioritized parallel compositions. See Figure 3-67 and

3-68. Method calls without return values could be buffered by a buffered call channel, but this feature is not support by the graphical modelling language. Thus, call channels and barriers are unbuffered. In case of a priority conflict, a redesign is required to solve the performance bottleneck

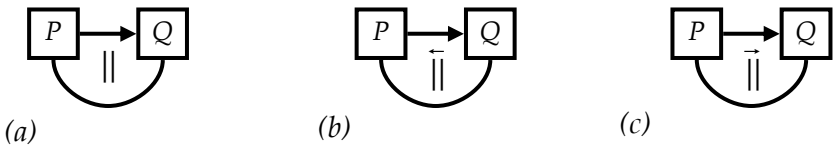


Figure 3-67 *Valid call channel configurations.*

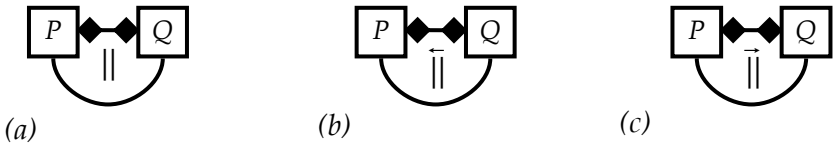


Figure 3-68 *Valid barrier configurations.*

3.9 Design freedom

The proposed graphical modelling language comprises significant freedom during design, which makes the development of CSP diagrams easier in the following ways:

1. The user can leave compositional relationships between processes undefined when any order of execution is accepted.

2. The user is no longer restricted to a predefined framework. The user can influence the framework of code generation by defining compositional relationships between processes. The framework will adapt to the desires of the user.

3. The design can be further refined by adding compositional relationships whereby redundant information is allowed. One can

over-specify the design to be sure that the requirements are met. Over-specifying can be useful when the diagram is viewed from different perspectives and this does not have to make the program more complex. Of course, redundant relationships can also be automatically removed or hidden by the design tool.

4. The run-time environment (kernel) takes care of the non-deterministic behaviours of the parallel and alternative relationships. The program becomes truly event-driven and the user does not have to concern with lower-level states in regard of multithreading or sequencing.

5. A design tool can frequently check the model for design failures, like compositional conflicts. During design, any design failures are treated as warnings. Such a tool can highlight the path(s) in which a particular failure occurs. This could be done similarly as a word processor that underlines the incorrect words or suggest other grammar. In short, the tool can guide the user to improve the model without restricting the design freedom. In the final model any remaining failures are considered as errors.

3.10 Refinement and verification

Process architectures should satisfy the desired requirements. If these requirements are not met then the design is incomplete and it is subject for further refinements. A CSP diagram encompasses information that allows verifying the design prior to executing the code. This makes any *trial-and-error* approach in early stages in the development unnecessary.

In case the design does not meet the requirements, additional relationships must be specified—the model needs to be refined. The user will undertake refinement steps until the requirements are achieved. The refinement and verification approach is a continuously interactive process.

Cycles specify redundant compositional relationships which help with detecting specification conflicts between the specified relationships and

the requirements. The ultimate refinement step results in a complete composition diagram whereby all processes are visually connected to each other. This can be simplified by tree-structures using deep or flat hierarchies. The density of connections in a complete composition diagram makes the model complex and difficult or impossible to read. Although, a complete composition diagram has a unique solution, usually, a unique solution is not the goal of the user. Any valid solution that satisfies the requirements is adequate. This can be useful when the hardware provides feedback and determines an optimal solution that best performs on the embedded computer system.

3.11 Conclusions

In this chapter, a graphical modelling language is defined, which is useful for designing process architectures in the form of CSP diagrams.

A CSP diagram consists of two distinct views, respectively the communication diagram and the composition diagram. Each diagram describes a different concurrency concern in the system. The collaboration between both diagrams provides valuable information about their compositions that is useful to determine design conflicts, such as specification conflicts, deadlocks, and priority inversion problems. This information can determine the exact type of communication (e.g. rendezvous, buffered, sub-sampling, super-sampling) between processes that is necessary to solve design conflicts or to optimize the performance of the process architecture in a systematic way. Composition diagrams can be traced for various design decisions which may be in conflict with the specification or mind set of the user. Thus, CSP diagrams incorporate guidance for the user to avoid design and coding errors.

The language is process-oriented and it extends to object-orientation. The presented graphical modelling language acts as a glue-logic between structured methods and object-orientation and thus providing continuation between the two paradigms. The graphical modelling language does not prescribe the design process of developing concurrent systems, but it offers guidance for stepwise refinement. The language can

be used at every level of abstraction with the same graphical notations and semantics. Furthermore, the language abstracts away from hardware or software implementations. CSP enhancements have been incorporated, such as exception handling, priorities, timing, and imperative facilities. These enhancements are essential for designing real-time process architectures.

The graphical modelling language does not prescribe styles for designing CSP diagrams. The user can design complex diagrams that are unreadable to others. Thus, the user is responsible for the readability of the diagrams.

Essentially, the resulting designs must be implementable. The design process is guided by rules, such as:

- *Compositional analysis rule*—useful for analyzing compositional CSP constructs. It is used for determining operators on hidden interrelationships derived from user-specified paths of relationships, for writing ambiguous or unambiguous algebraic expressions, and for detecting specification conflicts.
- *Reallocation rules*—rules for reallocating relationships with another, possibly nearest, process while preserving the algebraic expression.
- *Balancing cycles*—technique that ensures a balanced cycle of correct parenthesizing indexes.

These rules offer analysis approaches that guarantee consistency and correctness, such as

- *Specification analysis*—finding specification conflicts whereby relationships are in contradiction in the design.
- *Deadlock analysis*—finding deadlock by searching for sequential conflicts between primitive communication processes.
- *Priority inversion analysis*—finding priority inversion problems by searching for priority conflicts between processes.

A CSP diagram can be mathematically analyzed (model-checked), simulated, and finally executed on a dedicated embedded real-time

system. CSP diagrams are a sort of state diagrams that do not suffer from state explosions.

Design tools that support this graphical modelling language are inevitable in order to really benefit from CSP diagrams.

CHAPTER 4

A CSP library for compositional programming of concurrent Software

4.1 Introduction

Processes and their interrelationships, as discussed in Chapter 3, are detailed by an object model. The object model is called *Communicating Threads* (CT). CT is based on object-oriented techniques described by class diagrams and CT is implementable by object-oriented programming languages. This results in a CSP-based library for each object-oriented programming language. The library implements an application programming interface (API), which is used to program communicating processes and compositional constructs.

The CT API for the programming language Java is defined and presented in this chapter, which is called *Communicating Threads for Java* (CTJ). Other libraries have been created for the programming languages C (in object style) and C++. These libraries are called *Communicating Threads for C* (CTC) and *Communicating Threads for C++* (CTC++). CTC and CTC++ are native coded and they are therefore much faster and more compact than CTJ. CTC++ is used for high-performance real-time control applications, which are discussed in Chapter 6. CTC is useful for processors that are not supported by a C++ compiler. CTC is part of CTC++. CTJ is used for prototyping and illustrative purposes. Each

library is influenced by the limitations of the syntax and taxonomy of the programming language.

CTJ illustrates that the user will be freed from explicitly dealing with low level multithreading issues, e.g. creating, destroying, and synchronizing threads. More precisely, the user will be using threads without programming threads directly. CTJ offers clean compositional design patterns without polluting objects with synchronization constructs (e.g. monitor constructs). CTJ outlines the fact that the notion of processes is inevitable in order to let object-orientation succeed in concurrent software.

In Section 4.2, the approach and background of the implementation are motivated. CTJ is described in Sections 4.3 to 4.7. In Section 4.3 the process interface is described. The channel and barrier interfaces are described in Section 4.4 and 4.5 respectively. These implement the *communicational interrelationships* as discussed in Chapter 3. The compositional constructs are described in Section 4.6. These constructs implement the *compositional interrelationships* as discussed in Chapter 3. Timing and sampling are crucial in real-time systems and these issues are discussed in Section 4.7.

4.2 Approach and background

4.2.1 CT object model

The CT object model presents a concurrency model for building reliable, robust, and real-time concurrent software in object-oriented programming languages. The CT object model is described by classes, interfaces, and relationships that separate concerns by object-oriented techniques. The CT object model is the meta-model for CTJ and other CSP libraries. Several aspects of the occam and Ada languages (Burns and Wellings, 1990) have been incorporated into the object model. The development of CT was based on a rapid prototyping strategy, because the development was strongly driven by technical issues and alternative

solutions had to be investigated. A good match between the API and its low level implementation is essential. CT has to be efficient and therefore it was designed in such a way that the performance is eminent for a large class of embedded systems with limited systems resources. The implementation has to be able to deal with common implementation detail, like interrupt handling, timers, memory management, and I/O control. The design of CT is described with the help of class diagrams (UML 1.4) and this process is interleaved with coding in Java.

The semantics of the CT constructs are a subset of the semantics of the CSP operators. The CSP theory comprises non-determinism, which is as such not implementable on computers. For example, fairness and unfairness of thread scheduling is based on priorities, which is not judged or captured by the CSP operators. CSP abstracts away from that.

Priorities are important for developing real-time software. On a single processor, priority policies are important to accomplish efficient execution of the program. Priority policies can be seen as a gloss on the semantics of the CSP operators. They bring about restrictions to the theoretical ordering of events in event-traces. This should have no effect on deadlock or livelock checking with untimed CSP. Note that priorities have effect on a timely basis like performance. However, a poor priority scheme may result in poor performance. The worst-case scenario is starvation, which may lead to livelock or deadlock. This kind of starvation indicates that the program cannot meet its real-time requirements anyway. Since the CSP operators are compositional, priorities and priority policies must also be compositional. That is, priorities are relative and not absolute.

The CT object model must be consistent with the graphical modelling language as defined in Chapter 3.

4.2.2 Java thread model

Java encompasses a thread-model in the language, supported by its run-time environment. Currently, Java suffers from significant run-time overhead and high memory footprint. Furthermore, Java is not suitable

for real-time applications. Welch (1996) illustrated this problem by the "The Starving Philosophers" example. This example shows that the Java monitor is statistically correct, but the monitor can cause starvation on certain timing constraints. It is possible that with certain timing, certain threads always stay on the waiting queue of the monitor. These threads will starve to death, even though there is enough time for the program to meet their real-time requirements.

4.2.3 Communicating Threads for Java

CTJ presents an alternative for the Java thread model for building reliable, robust, and real-time concurrent software in Java. CTJ is the implementation of the CT object model in Java. CTJ was not intended for high-performance control systems. Despite the previously mentioned disadvantages of Java, Java is considered to be a good programming language for exercises and educational purposes on different target computers. The programming languages C and C++ are used for high-performance embedded real-time systems. Therefore, CTC, CTC++, and CTJ were developed to illustrate the portability of the object-model to C, C++, and Java. CTC and CTC++ illustrate optimized performance (test show that CTC and CTC++ are about 400 times faster than CTJ), low memory-footprint, and portability to a large variety of processors.

Consistency between CT and the graphical modelling language in Chapter 3 should allow for a straightforward implementation of CSP diagrams to Java with CTJ (or to C/C++ with CTC/CTC++).

4.2.4 Aspects

Applying object-oriented techniques, object-oriented programming languages, and applying heterogeneous computer hardware at a reasonable performance, are desired requirements that were previously discussed. Three other requirements are important aspects that were considered during the design and implementation of the CT object model and the CSP libraries.

These aspects are:

- Simplicity
- Portability
- Generality

These three aspects are already an integral part of the graphical modelling language in Chapter 3. This is because the CSP theory comprehends simplicity, portability, and generality by offering abstraction and fundamental concepts.

Simplicity

Usually, libraries and frameworks grow and become more complicated with addtional features, that were forgotten, that compensate limitations of the already implemented features, or that one thinks that *may* be needed in the future. However, it takes real skills to keep features out that are not really necessary. This is what the occam's razor is intended for. With the occam's razor in mind, every feature is judged for necessity.

CT was developed with the Occam's razor in mind. The Occam's razor (Beckett, 1994; Hiroshi, 1997; Skeptic, 2004) is an approach to make things as simple as possible, but not simpler. Relevant information should not be lost due to simplification. It is important to meantion that the Occam's razor is inherently part of CSP. Consequently, CSP offers a minimal set of necessary concepts to describe concurrent systems. This explains the name *occam* as the name for the programming language that implements a subset of CSP (Inmos, 1988).

Portability

CT aims to be portable to a large variety of embedded computer systems and to a variety of object-oriented programming languages. Java is a portable programming language. Other object-oriented programming languages, such as C and C++, are not always portable. These native languages can address platform specific instructions.

In order to gain portability and maintain program structure, it was decided to write code in portable C/C++ and as little as possible in assembler language. About 99% of CTC/CTC++ is written in portable C/C++ and less than 1% is processor-specific. Portable C/C++ code is written in C/C++ with no platform specific pointers or dependencies. The processor-specific methods must be implemented for each CPU and perhaps specific for each different operating system. An overview of the processor-specific methods is given in Appendix B. Assembler language can be used when processor-specific instructions are required (e.g. swapping stack pointers or saving and restoring the processor context).

Generality

CT aims to be generally applicable for concurrent computer systems. CT should be applied to all kinds of concurrent and real-time applications and was not made for control applications alone.

The API of CT should obey the semantics of the CSP concepts and it is designed in such a way that the API solely serves the application. The implementation of the API is devoted to the CPU, but the API is devoted to the user. This holds for any platform or for different programming languages. This implies that the user should not notice major differences in the semantics of the CSP concepts for different programming languages.

The CSP constructs, channels, and barriers simplify concurrent software by abstracting away from the underlying thread control. Therefore, CT can implement its own threads using an embedded scheduler (as in the current implementation) or CT can borrow threads from an operating system (this can involve a POSIX library). The API of CT should abstract away from single- and multi-processor systems.

4.3 Processes

A process is a component that can play the role of a *CSP process* or a *process instance*. When the process is performing its (real-time) task, it plays the role of a CSP process. A process that is not running (e.g. has not started or has terminated) can be treated as a process instance. A process instance is the existence of a process in memory, like an object. Either way, a process is *not* an object.

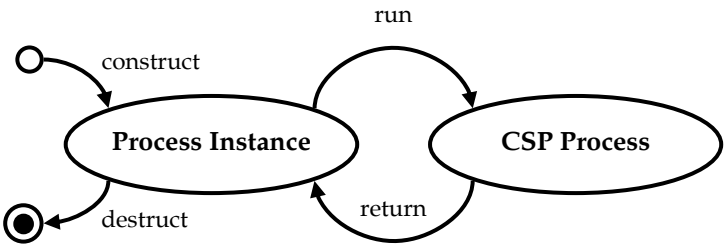


Figure 4-1 *Role game of a process (state diagram).*

Figure 4-1 illustrates the role game between the process instance and the CSP process. A process that is constructed and instantiated becomes a process instance. Once the process instance is invoked to run, it plays the role of a CSP process. When a CSP process terminates, it returns back into a process instance. A process dies when the process instance is destructed.

The distinction between CSP processes, process instances, and objects serves separate concerns in the program. This distinction was discussed in Section 3.2. In CT, process classes are distinguished from object classes. A process class describes and implements a *process communication interface* and a *process communication interface* for processes of the same kind. An object class describes and implements an *object interface* for objects of the same kind. Java supports only object classes. The Java syntax can be used to describe process classes by imposing an arrangement of rules and semantics.

4.3.1 Process instance interface

The process class describes a constructor, a destructor, support methods, and a single `run()` method. These elements specify the *process instance interface*. The implementation relationship between the class and interface is shown in Figure 4-2a-b. Figure 4-2a illustrates the construction of the `MyProcess` class. This can be simplified in the UML by using the stereotype label `<<process>>`, see Figure 4-2b.

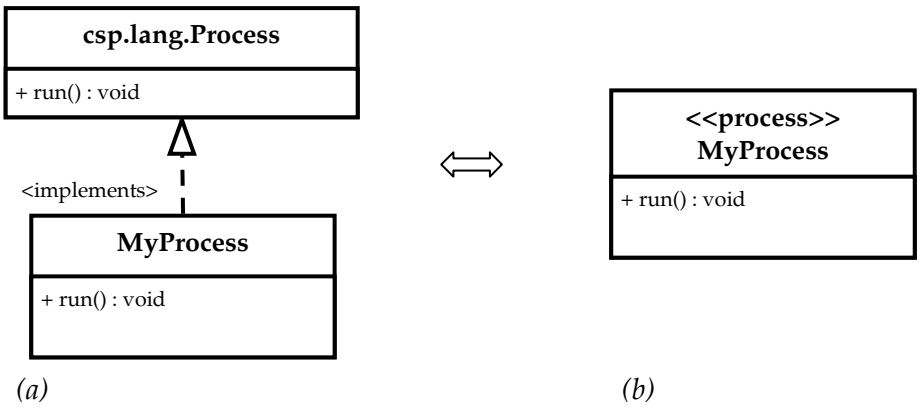


Figure 4-2 *UML class diagram of MyProcess process:*
(a) by interface inheritance,
(b) by stereotyping.

The `run()` method, as defined by the `csp.lang.Process` interface, is required for every process. Therefore, a process must implement the `csp.lang.Process` interface, which specifies a public `run()` method. This is similar as for Java's `java.lang.Runnable` interface. is shown in Listing 4-1.

```
public interface csp.lang.Process {
  public void run()
    throws ExceptionSet;
}
```

Listing 4-1 *The process interface.*

The `run()` method implements a sequential task that the process performs when this method is invoked by its parent process. In real-time systems this `run()` method performs a real-time task. The process can throw a set of exceptions of type `ExceptionSet` when one or more

exceptions occur in the process. See Section 4.6.4 for more information on exceptions.

The parent process is allowed to call public methods on its child process instances to set up its pre-condition, to retrieve its post-condition, or to execute its `run()` method. The `run()` method or other methods can never be invoked simultaneously by multiple processes since a child process is owned by one parent process at the time. This simple design rule strictly separates each thread of control and enables a secure multithreading environment. Once the `run()` method is called on the process instance, the role of a process instance switches immediately to the role of a CSP process. Calling a public method on a CSP process is illegal and only channels or barriers should be used to change the state of a CSP process. When the process terminates (i.e. the `run()` method returns) then the role of a CSP process switches back to process instance. See Figure 4-1.

Figure 4-3 depicts a class diagram that shows the associations of a process with other processes, channels and barriers.

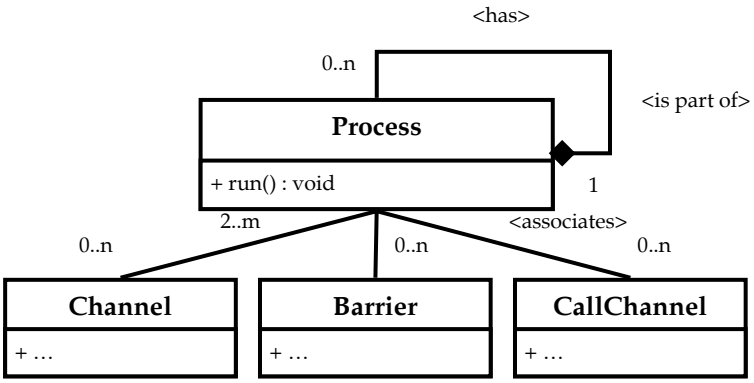


Figure 4-3 *Process associations.*

Listing 4-2 shows an example of a template of a process class.

```
class MyProcess implements csp.lang.Process {

  ... private or protected declarations for local use

  public MyProcess(process communication interface) { ... }

  ... methods defining the process instance interface
```

```
  public void run()
    throws ExceptionSet { ... do something }

  public int get_parameter() { ... }
  public void set_parameter(..) { ... }
  public void add(..) { ... }
  public void remove(..) { ... }

  ... private or protected methods for local use
}
```

Listing 4-2 *Example of a process class.*

The public constructor is called once on the instantiation of the process class. The constructor configures the process. The constructor sets up all of the initial resources, like references to channels, references to barriers, initial parameters, and the requisite pre-condition for the `run()` method. After construction, the reference to the process instance is available and offers the `run()` method waiting to be invoked.

The pre-condition of a process is the constraint that must be true when the `run()` method is invoked. The post-condition is the constraint that must be true after the completion of the `run()` method. The post-condition is usually the pre-condition for the next run. The initial state of a process is usually set by the constructor at instantiation. *State handling methods* (e.g. `add(..)`, `remove(..)`, `set_parameter(..)`, `get_parameter()`) can be used to initiate or retrieve the state of a process after instantiation, but before or after invoking the `run()` method. These methods are thread-safe since they are exclusively used by the parent process. These methods offer dynamic construction of the process at run-time and they must strictly serve the `run()` method. An example of using state handling methods is described in Appendix D.5.

4.3.2 Process communication interface

The channel-ends and barrier-ends that are passed via the constructor specify the *process communication interface*. These channel-ends and barrier-ends are the ports through which the process communicates with other processes. One can observe the behaviour of the process through

these ports. The process communication interface does not offer public methods, but it may specify the services (methods) that can be requested via channels.

4.4 Channels

Channels establish interactions between processes. Channels are intermediate objects that allow anonymous and point-to-point communication between two processes. Processes know their channels but they do not know the processes they interact with. Channels take care of synchronization, scheduling, and message delivery of data via the underlying hardware. There are two types of channels supported by CT.

These channels are:

1. data channels
2. call channels

Data channels are lower-level channels for sending primitive data or objects from a producer process to a consumer process. Data channels do not return objects or data. Call channels are higher-level channels for requesting a method call from a client process to a server process that will perform the call once it has accepted the request. Call channels may return objects or data.

4.4.1 Synchronization

Channels are *thread safe*, i.e. they are synchronized objects so that no race hazards can occur within channels. A channel protects its internals by mutual exclusion between its readers and writers, or callers and callees. Thus, CTJ channels can be safely shared between multiple processes.

4.4.2 Scheduling

A writer process can write to a channel at any time and the process will be blocked to wait for some other process to read from the channel. Similarly, a reader process may read from a channel at any time and it will be blocked to wait for some other process to write on the channel. Blocking is entirely passive, the blocked process consumes no processor time and another (non-blocking) process gets the processor time to execute instead. When both processes are willing to communicate then instantaneous communication will happen and no withdraw is possible—this is called *rendezvous*. On rendezvous they engage in the *communication event*. After the communication event both processes are unblocked and both can continue in parallel. On a single processor system, the channel determines which thread of control is scheduled first. The channel complies with a prioritized scheduling policy to ensure fairness. Due to the rendezvous principle one can abstract away from thread states. This is explained in Chapter 5.

Buffered data channels can store data that is to be delivered at a latter time. Buffered data channels extend the rendezvous principle in circumstances that requires unblocking communication between processes, which are executing at different frequencies (Chapter 5). In this circumstance, buffered data channels extend the scheduling policy that is required to guarantee that these processes (at different priorities) can meet their deadlines. The total scheduling policy of a process architecture is composed by the type of channels.

4.4.3 Message delivery

Channels can pass messages (e.g. data, objects, requests) with *pass-by-value* or with *pass-by-reference*. Each mechanism has its advantages and disadvantages.

- The *pass-by-value mechanism* is safe, fast for small messages, reuses memory efficiently, independent of shared and distributed memory system, and has a good reputation with occam on transputers.

- The *pass-by-reference mechanism* is fast for large messages on shared memory systems. Precautions are required to use it safely. Cloning messages is required on message delivery between distributed memory systems. Cloning requires expensive memory management. A simple and elegant solution is passing ownership of the message. That is, a sender sends a reference of an object and immediately releases its ownership of the reference. Consequently, the receiver will be the only process claiming the ownership of the object.

Data channels support both message delivery mechanisms. Call channel support pass-by-value for primitive data types and pass-by-reference for objects. These message delivery mechanisms are further discussed in Appendix H.

4.4.4 Data channels

A data channel transfers primitive data types or objects in *one* direction. Data channels are initially unbuffered and do not store messages in the channels. A *generic data channel* has been developed from which other data channels can be derived.

Generic channel type

In CTJ, a *data channel interface* consists of a *channel input interface* and a *channel output interface*. The channel input interface specifies one or more `read(..)` methods and the channel output interface specifies one or more `write(..)` methods. Processes communicate by reading or writing on channels using these methods.

The generic data channel is implemented by the `Channel` class. Its `read(..)` and `write(..)` methods are protected and only available for its subclasses. The `Channel_of_Object` class makes these methods public. The `Channel_of_Object` channel can communicate all kinds of objects, which provides a great deal of flexibility and generality. The `Channel_of_Object` class implements a channel input interface and a channel output

interface; the `ChannelInput_of_Object` interface and the `ChannelOutput_of_Object` interface respectively. For generalization, the `ChannelInput_of_Object` is of type `ChannelInput` and `ChannelOutput_of_Object` is of type `ChannelOutput`. See Figure 4-4.

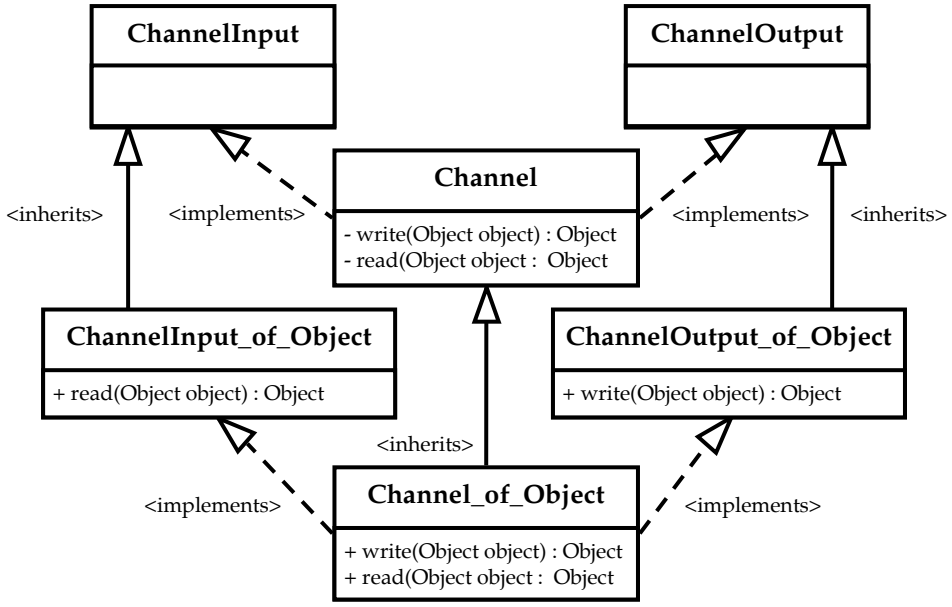


Figure 4-4 *UML class diagram of the Channel_of_Object channel.*

The `ChannelOutput_of_Object` interface specifies the method

```
public void write(Object object)
```

and the `ChannelInput_of_Object` specifies the method

```
public Object read(Object object)
```

These `read()` and `write()` methods support both pass-by-value and pass-by-reference. The choice of message delivery mechanism depends on the type of channel in a distributed or shared memory system and it depends on the ability of the receiver to reuse memory.

The `write(object)` and `read(object)` statements are used to allow pass-by-value. On rendezvous the channel copies the content of the specified object in `write(object)` at the producer side to the specified object in `read(object)` at the consumer side of the channel.

In circumstances where the size of messages can change, cloning or reference passing is required. In order to enforce pass-by-reference, statement `object = read(..)` is required. This allows receiving objects, arrays, or strings of variable length. It is important that the sender releases its ownership of the message that is passed. The `object = read(object)` statement can choose between pass-by-value and pass-by-reference. The choice depends on the platform and the implementation of the data channel. For example, a distributed system may require cloning objects over a remote channel. Since the objects are specified by the `read()` and `write()` methods, their memory can be efficiently be reused at both sides of the data channel.

The channel may provide additional `read(..)` and `write(..)` methods that can support cloning or reusing message objects. The following `read(..)` methods always returns a clone of the message or an `IOException` when cloning is not supported.

```
obj = chan.read(null);
obj = chan.read();
```

The programmer or the garbage collector must destroy the message object `obj` when it is no longer used. These kind of `read(..)` methods depend on dynamic memory management and for this reason these methods are usually avoided for real-time programs.

If the messages are of fixed length and reusable then the return value can be ignored, as in:

```
chan.read(obj);
```

A producer/consumer example is given in Appendix D.1

Restrictions and compatibility

Processes can only be connected when their process communication interfaces specify pairs of channel-inputs and channel-outputs of the same type. Also, the source and destination message objects must be of the same type otherwise an exception will be thrown to both processes.

This implementation of the generic data channel can only pass objects with public attributes. Protected and private attributes are inaccessible by the channel `copy()` method. The `copy()` method allows for deep copying (i.e. copying object within objects) and reference passing.

Overview of features

The generic data channel was designed and implemented in such a way that:

- different data channels can be derived from this implementation,
- hardware and software concerns are separated,
- implementation complexity problems can be explored,
- one can learn about the feasibility of a generic implementation.

The generic data channel provides the following features:

- a primitive interface,
- support of object transfer,
- support of pass-by-reference and pass-by-value,
- safe for multiple readers and multiple writers (Any2Any),
- usable as a guard in alternative constructs,
- initially zero-buffered, but can be extended with a buffer,
- support of shared memory and distributed communication,
- support of timed communication events,
- support of a priority scheduling policy for optimal performance on single processor systems.

The generic data channel implements the Any2Any channel. An Any2Any type of channel is a safe channel between any (one or more) writers to any (one or more) readers. One2Any, Any2One, and One2One channel types, as in JCSP (Welch and Austin, 1999), can be derived from the generic data channel. These channel types restrict channel sharing

and therefore they can offer optimized performance and documentary help. On the other hand, using an One2One data channel in an Any2Any connection makes the channel crash. The generic data channel will not crash and processes behave in a natural way, such as blocking or deadlocking. For now One2Any, Any2One, and One2One channel types have been postponed and put on the list of recommendations. Furthermore, the use of templates (as found in C++) would make creating channels simpler. Templates in Java may become a useful feature in the upcoming Java 1.5 (Sun Microsystems, 2004).

Specific channel types

The `Channel` class implements generic methods that one can use to create channels for specific message types. The reads and writes on the channel are delegated to the generic `read(..)` and `write(..)` methods of the `Channel` class. In Chapter 3, these `read(..)` and `write(..)` methods are the primitive communication processes on data channels.

In CTJ, `Channel_of_Integer` channels send objects of the `Integer` class (in the `csp.lang` package) or data of primitive data type `int`. The `Integer` object type is basically a wrapper for `int` with additional methods. The `Integer` wrapper is a modified version of the `Integer` class in the `java.lang` package; the `int` value attribute has been made public rather than private. Hence, channels can only copy public attributes.

The `ChannelOutput_of_Integer` interface specifies two `write(..)` methods:

```
public void write(Integer integer)
public void write(int integer)
```

The `ChannelInput_of_Integer` interface specifies two `read(..)` methods:

```
public Integer read(Integer integer)
public int read()
```

Any `write(..)` method can be used in conjunction with any `read(..)` method. The user can mix `Integer` and `int` types at the sender or receiver

side. If Java would support multiple return types then `public Integer read()` could be added. This is on the wish list for Java 1.5.

The `Channel_of_Integer` channel can contain a special link driver as specified by its constructor:

```
Channel_of_Integer(LinkDriver linkdriver)
```

When a link driver is specified, the reads or writes on this channel are delegated to the specified link driver; otherwise they are delegated to the `Channel` class. This concept allows for the plugging in of link drivers that can delegate communication to hardware or via a special buffer.

csp.lang....	Channel_of_...	ChannelInput_of_ ...	ChannelOutput_of_ ...
Object	Channel_of_Object	Object read(java.lang.Object) Object read()	Void write(java.lang.Object)
Boolean	Channel_of_Boolean	boolean read(csp.lang.Boolean) boolean read()	void write(csp.lang.Boolean) void write(Boolean)
Byte	Channel_of_Byte	byte read(csp.lang.Byte) byte read()	void write(csp.lang.Byte) void write(byte)
Char	Channel_of_Character	char read(csp.lang.Character) char read()	void write(csp.lang.Character) void write(char)
Double	Channel_of_Double	double read(csp.lang.Double) double read()	void write(csp.lang.Double) void write(double)
Float	Channel_of_Float	float read(csp.lang.Float) float read()	void write(csp.lang.Float) void write(float)
Integer	Channel_of_Integer	int read(csp.lang.Integer) int read()	void write(csp.lang.Integer) void write(int)
Long	Channel_of_Long	long read(csp.lang.Long) long read()	void write(csp.lang.Long) void write(long)
Short	Channel_of_Short	short read(csp.lang.Short) short read()	void write(csp.lang.Short) void write(short)
Reference	Channel_of_Reference	Reference read(csp.lang.Reference) Reference read()	void write(Reference) void write()
extra	Channel_of_Trigger	void read()	

Table 4-1 *CTJ wrappers and data channel interfaces.*

The channel interface `Channel_of_Integer` is an example for all other primitive data-type channels. Table 4-1 gives an overview of the default data channel types that are provided by the CTJ library. A special channel `Channel_of_Trigger` has been included which does not send

information but it is used to trigger processes. This strong typing allows safety and interface matching.

Reference channel type

The `Channel_of_Reference` channel accepts only `Reference` objects. In fact, the `Channel_of_Reference` channel is a wrapper of a `Channel_of_Object` channel. A `Reference` object has a public `Object object` attribute for sending object references, as shown in Listing 4-3. The `tag` value can be used to identify the `object`. What is special about the `Channel_of_Reference` channel is that the attribute object at the writer side will be set to `null` by the `write(..)` method after communication and will be set to `null` by the `read(..)` method before communication.

```
public final class Reference implements java.io.Serializable
{
  public static final int UNDEFINED = -1;

  // Public place holder for an object reference and tag.

  public int    tag    = UNDEFINED;
  public Object object = null;

  ... support methods
}
```

Listing 4-3 *The Reference class.*

Hardware link control

The channel interface abstracts away from the hardware, such as memory or some device. Data channels encapsulate hardware control by means of *link drivers* (Hilderink et al., 2000; Hilderink et al., 1998; Hilderink et al., 1998). The concept of this framework is illustrated in Figure 4-5.

Communication via channels provides platform independency that is two-fold:

1. Processes communicate with their environment (or hardware) via channels and never directly.
2. The semantics and behaviour of channel communication is identical for single processor systems as for distributed processor systems.

Figure 4-5 illustrates data channel communication between two processes at different levels of detail. The top figure shows the conceptual data-flow between two processes, which are connected with a communication relationship. The second figure illustrates the communication relationship as an intermediate channel object. Thus far, this is hardware independent. The third figure shows a link driver as part of the channel object to which the write and read methods are delegated. The write and read methods of the link driver carries out the hardware dependent code. The last figure shows that this mechanism scales for a distributed system.

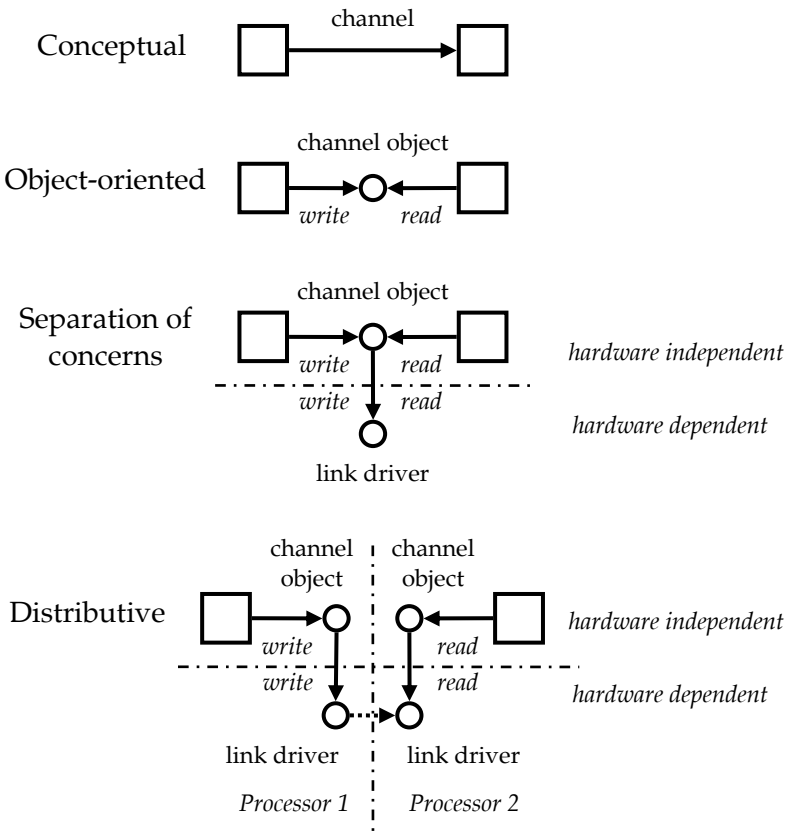


Figure 4-5 *CT channel framework.*

The link drivers establish the connection and must be compatible to each other. A special *link driver framework* has been developed that is used to implement data channels for dedicated hardware, like analogue-digital converters, counters, serial port, CAN, TCP/IP sockets, digital pins, etc. These kinds of channels are called *external channels*. Data channels using shared memory on a single processor are called *internal data channels*. Processes do not know if they communicate with internal or external data channels. Call channels do not support the link driver framework and are therefore always hardware independent in CT. Data channels should be used for communication via hardware.

There are two ways to create a hardware specific data channel in CT, namely by means of:

1. *Delegation*—A link driver object can be plugged into a data channel. The reads and writes will be delegated to the link driver object.

2. *Inheritance*—A data channel inherits the `LinkDriver` class and implements channel interfaces. The `LinkDriver` class provides the necessary methods for synchronizing and controlling kernel functions. The channel implements a link driver but processes access the channel through the channel interface and not via the link driver interface.

A guideline to separate concerns is that link drivers are the only objects that strictly control the underlying devices. The CT link driver framework is abstractly defined in such a way that it can be extended as needed, without affecting the process architecture. Processes usually do not use or create link drivers and become fully hardware independent. Hardware dependent processes are considered to be *network builders* as they setup and configure a network of processes and map the software on the topology of the hardware. The use of network builders allows a systematic approach, which makes the program highly hardware independent and consequently portable. Porting a program to a different target involves changing the network builders and leaves the other processes intact.

4.4.5 Call channels

A *call channel* allows a client process to pass a message to a server process with the instruction that the server should perform a particular service (method). When the server is willing to accept the request (or call) then the service will be performed. It is the server process that performs the service and not the client process. A client blocks on the service call on the call channel until the server is ready to accept the method call and has performed the methods. The `accept(..)` method is invoked on the call channel by the server process when it is ready to accept a call. Similarly, the server process will be blocked on the `accept(..)` method until a client is invoking a method on the channel. Likewise, with data channels, both processes must rendezvous to engage in the communication event. Conceptually, this type of message passing is different from method invocation on objects where the invokee must follow the invoker. The channel strictly separates the behaviour of the client and server, which simplifies compositional programming.

The call can specify arguments to be used by the service and any results of the service can be returned to the client. Hence, data transfer via call channels can be bidirectional and the arguments are passed by pass-by-reference for objects and pass-by-value for primitive data types. The call must release the ownership of objects that are passed as arguments. This will prevent object sharing between the client and server. A service call is light weight on single processor systems. In case the request is accepted, the thread of control of the client process is borrowed and performs the method on the server process at the priority of the server process, as if the server performed the method.

As with data channels, the client process sees the call channel and not the server process. Call channels were part of the occam3 language specification (Barrett, 1993). They provide *rendezvous* in the sense of an Ada `entry-accept`, but are considerably more flexible and lightweight.

The services a server process can offer are specified by its *call interface*. A client can communicate with the server process via a call channel with the same call interface. Services can be partitioned by multiple call

interfaces. The call channel concept is generic, but there is no one generic call channel. A call channel can be straightforwardly generated from a set of call interfaces (service types).

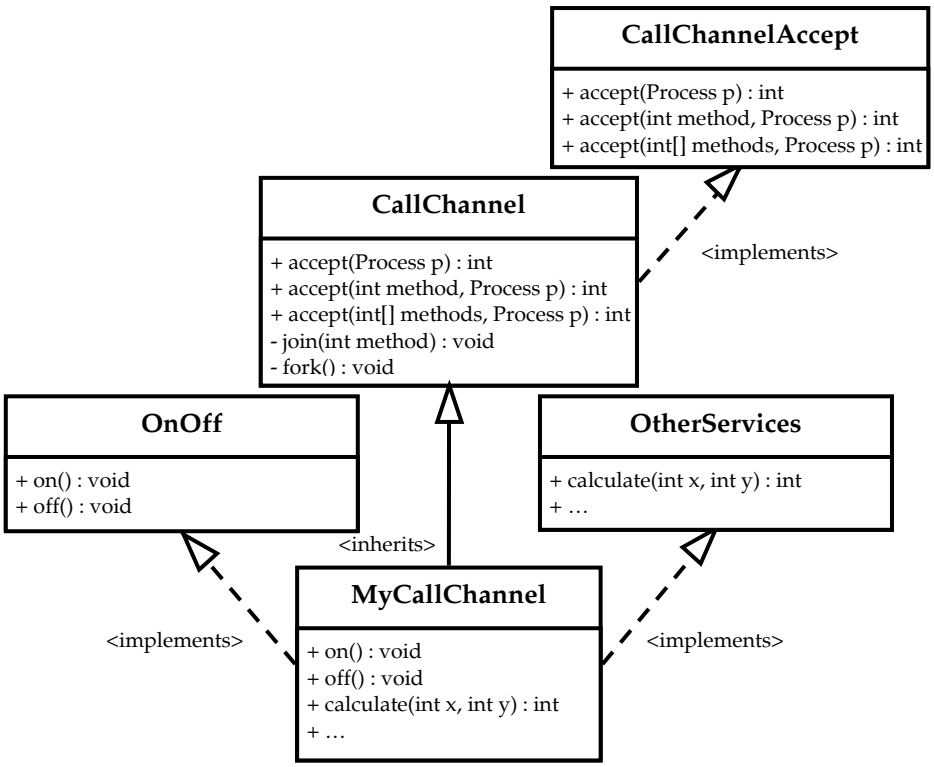


Figure 4-6 *UML class diagram of MyCallChannel class.*

The class diagram in Figure 4-6 depicts the relationships between the different classes that constructs a call channel. In this example, the call channel `MyCallChannel` supports the call interfaces `OnOff` and `OtherServices`. A client process can call the methods

- `On()`
- `Off()`

and an association with OtherServices allows the client to call

- `Calculate(..)`
- other methods prototyped in `OtherServices`

A call channel supports multiple call interfaces. Each different call interface allows the separating of different groups of clients to

communicate via the call channel. A client/server example is illustrated in Appendix D.2.

4.5 Barriers

A barrier provides a multi-way synchronization point, which may involve a number (two or more) of processes. Any process synchronizing on a barrier will be blocked until all processes associated with the barrier reach that synchronization point. On rendezvous with the synchronization point, a special communication process could exchange data between all participating processes. The barrier releases the participating processes when the communication process has completed. The barrier can represent a communication event, a bag of events (Smith et al., 2003), or merged events (Lawrence, 1998).

The theory of Bulk Synchronous Parallelism (BSP) (McColl, 1996) exclusively makes use of the barrier primitive to determine an optimized communication/processing trade-off for shared variable models. Roscoe (1998) notes that the BSP model is appropriate for large computations of numerical problems; it does not give any insight into the way parallel systems interact at a low level. CSP can be used to model the communication process that is performed by the barrier. The barrier, as proposed in this thesis, provides a process-layering concept. This process-layering concept allows one to create parallel programs using the BSP model in one upper layer and the CSP model in a lower layer. The CSP model implements the barrier.

A barrier is an instance of the `Barrier` class and it can be set up with an initial number of associated processes. This is specified as

```
Barrier barrier = new Barrier(int number);
```

The synchronization point, at which processes associated with the barrier must rendezvous, is the point where these processes invoke the `sync(..)` method on the barrier.

```
void sync() throws ExceptionSet;
```

The process will synchronize with all associated processes. Its implementation is restricted for shared memory systems. The next `sync(..)` method can be used for distributed memory systems.

```
void sync(csp.lang.Process process) throws ExceptionSet;
```

With this method the processes will synchronize with all associated processes and the barrier performs the specified `process` at each side of the barrier in parallel on rendezvous. `process` can be part of a network of processes with data channels that can describe the information exchange on a distributed system. In the current version of CTJ, a distributed barrier (i.e. a barrier connected to distributed processes) may not release all processes at the same time. After termination of `process` the `sync(..)` method releases the associated process. This allows one to create a distributed barrier with a particular `process` at each end of the barrier that communicates on external data channels (or with special link drivers). The appropriate behaviour of the barrier can be implemented with CSP concepts. The `sync(..)` methods will throw an exception set on error in the barrier.

In Appendix D.3, an example is given which shows two processes that synchronize two times on a barrier and this illustrates the differences between `sync()` and `sync(process)`.

In a dynamic network of processes, the number of associated processes with a barrier could grow or shrink in time. In this case the `number` is unknown and one can specify a `barrier` without a fixed number, as in

```
Barrier barrier = new Barrier();
```

Additional methods allow enrolling and resigning processes with the barrier at run-time.

```
void enroll() throws ExceptionSet;
```

The process will be associated with this barrier. Also, any process that is associated with the barrier can resign itself from the barrier with

```
void resign() throws ExceptionSet;
```

If a process is the last process for which the barrier waits to synchronize and it resigns from the barrier then the barrier completes and releases all the remaining associated processes. If a process resigns from the barrier and only one process remains to synchronize on a barrier then the process must wait for a second process to enrol and synchronize with. This is because an event can only happen between at least two processes. Both `enroll()` and `resign()` methods can throw an exception set on error in the barrier. If the barrier was instantiated with constructor `Barrier(10)` then the number of processes that must synchronize is 10 and the methods `enroll()` and `resign()` do not alter that number.

4.6 Compositional constructs

In the previous section, the communication relationships (i.e. data channels, call channels, and barriers) in Java have been discussed. Processes are also related by compositional relationships. The parallel relationship was already shown in the examples of Appendices D.1, D.2, and D.3. The compositional relationships are implemented as compositional constructs in CTJ. These constructs are discussed in this section. A compositional construct composes a set of processes as one process and executes these processes in a particular order. The execution order is determined by communication and by compositional constructs.

The set of compositional constructs that is supported by CT are:

- *Equally-prioritized parallel*—the parallel construct
- *Unequally-prioritized parallel*—the priparallel construct
- *Sequential*—the sequential construct
- *Equally-prioritized choice*—the alternative construct
- *Unequally-prioritized choice*—the prialternative construct
- *Exception*—the exception construct

These constructs are processes themselves and therefore they implement the `Process` interface (stereotyped `<<process>>`). Therefore, these constructs allow nesting of other compositional constructs. There are a few processes to which the process interface is implicit. For example, the `switch-case` clause and the `try-throw-catch` clause provided by the programming language can be used as anonymous processes for an alternative construct and an exception construct respectively. These anonymous processes are discussed in Section 4.6.3 and 4.6.4.

4.6.1 The parallel construct

The implementation of the CSP parallel operator in CT is prioritized and it is divided into

- Equally-prioritized parallel construct
- Unequally-prioritized parallel construct

Equally-prioritized parallel construct

The equally-prioritized parallel construct (parallel or PAR) executes a list of processes in parallel with equal priorities. A parallel construct is based on a process instance that is instantiated by the `Parallel` class, as in

```
Parallel par = new Parallel(Process[] processes);
```

The argument `processes` is an array of processes that begins executing when the `run()` method of the parallel construct is invoked,

```
par.run();
```

The parallel construct will assign a separate thread of control to each associated process. Each thread of control will perform the `run()` method of the associated processes at the same priority as the main thread of control that entered the parallel construct.

The parallel construct is running when at least one of its associated processes are running. The parallel construct terminates when all associated processes have terminated.

The following example shows a parallel composition of three parallel processes.

```
Parallel par = new Parallel(new Process[] {
  new Process1(process interface),
  new Process2(process interface),
  new Process3(process interface)
});

par.run();
```

Listing 4-4 *Parallel construct.*

The listed processes `Process1`, `Process2`, and `Process3` will be executed in parallel when `par.run()` is invoked. These processes execute with the same priority as the `Parallel` process. The `par` process finishes successfully when *all* three of its associated processes have successfully finished.

The parallel construct supports a few additional public state handling methods that allow the adding or removing of parallel processes in the software architecture at run-time. These methods are part of the process instance interface and they may only be invoked when the parallel construct is not running. This is similar as for the other compositional constructs in the next sections.

New processes may be added at run-time, using

```
par.add(new Process4(..));
```

or by adding multiple processes at a time:

```
par.add(new Process[] { new Process4(..), new Process5(..) });
```

A process may be removed from the process list, using

```
par.remove(process);
```

Unequally-prioritized parallel construct

The unequally-prioritized parallel construct (priparallel or PRIPAR) executes a list of processes in parallel with declining (unequally) priorities. The execution of the first process in the process list is given the highest priority and the execution of the last process in the process list is given the lowest priority. These different priorities can improve the reactivity and responsiveness of the program.

A process itself has no priority. In other words, one cannot assign a priority number to a process and one cannot ask a process what priority it has. The priority is given to the thread of control that is encapsulated within the process. The priparallel construct avoids one using priority index numbers. Priority indexes are an implementation issue and not a design issue. Priority indexes have only a meaning in relation to other priority indexes. Therefore priparallel constructs implement priority relationships that specify higher, equal, or lower priority between processes.

The priparallel instance is constructed with

```
PriParallel pripar = new PriParallel(Process[] processes);
```

The following example (Listing 4-5) shows a priparallel construct of three processes.

```
PriParallel pripar = new PriParallel(new Process[] {
  new Process1(process interface),    // priority highest
  new Process2(process interface),    // priority next highest
  new Process3(process interface)     // priority lowest
});

pripar.run();
```

Listing 4-5 *Priparallel construct.*

The processes `Process1`, `Process2`, and `Process3` will be executed in parallel with successively lower priorities. Process `Process1` has the

highest priority. The `pripar` process finishes successfully when all three processes have successfully finished.

Unfortunately, it is not always possible to move away from certain implementation issues. Due to improving the performance and saving memory, each priparallel construct is limited to 8 priorities; where 7 are for user defined processes and one is reserved. The reserved priority is private to the priparallel construct and can be used for an idle task, skip task, or garbage collector task. The restriction to 8 priorities allows quick priority sorting with the efficiency of order $O(2)$, i.e., a process can be placed into the correct priority queue in a maximum of two steps. Increasing the maximum number of priorities, i.e., more than 7, is possible by nesting. The following example, in Listing 4-6, illustrates a priparallel construct with 49 ($=7^2$) priorities.

```
PriParallel pripar = new PriParallel(new Process[] {
  new PriParallel(new Process[] {       // priority 0
    Process1_1(..)                      // priority 0.0
    Process1_2(..)                      // priority 0.1
    Process1_3(..)                      // priority 0.2
    Process1_4(..)                      // priority 0.3
    Process1_5(..)                      // priority 0.4
    Process1_6(..)                      // priority 0.5
    Process1_7(..)                      // priority 0.6
  }),
  new PriParallel(new Process[] {       // priority 1
    Process2_1(..),                     // priority 1.0
    ...                                 // priority 1.1-5
    Process2_7(..)                      // priority 1.6
  }),
  new PriParallel(new Process[] {..}),  // priority 2.0-2.6
  new PriParallel(new Process[] {..}),  // priority 3.0-3.6
  new PriParallel(new Process[] {..}),  // priority 4.0-4.6
  new PriParallel(new Process[] {..}),  // priority 5.0-5.6
  new PriParallel(new Process[] {..})   // priority 6.0-6.6
});

pripar.run();
```

Listing 4-6 *Example of a nested priparallel construct.*

Note that the indexes in the comments show the internal indexing that is generated by the nested priparallel construct.

As with the parallel construct, methods like `add(..)` and `remove(..)` exist and an additional method is added to insert a process at run-time, using

```
pripar.insert(process, index);
```

Process `process` will be inserted at `index` of the process list. The order of priorities will automatically be applied to the new process list.

4.6.2 The sequential construct

A sequential construct (SEQ) performs processes in a particular fixed sequence. The sequential construct is the instance of the `Sequential` class. The sequential process instance is created with

```
Sequential seq = new Sequential(Process[] processes);
```

The argument `processes` is an array of processes that begins executing when the Sequential construct's `run()` method is invoked,

```
seq.run();
```

When the `run()` method of a sequential composition construct is invoked then the associated processes are executed one at a time by the same thread of control. The sequential construct process terminates when all associated processes have terminated.

The following example shows a sequential composition of three processes.

```
Sequential seq = new Sequential(new Process[] {
  new Process1(process interface),
  new Process2(process interface),
  new Process3(process interface)
});

seq.run();
```

Listing 4-7 *Sequential construct.*

In this case, `Process1` executes to completion first, followed by `Process2` and then by `Process3`. The `seq` process finishes successfully when `Process3` successfully finishes, i.e. when all three processes have successfully finished running in order. Sequential processes should not communicate with each other using (unbuffered) rendezvous channels, as this would cause deadlock.

Additional to the `add(..)` and `remove(..)` methods, a new process can be inserted at a specific `index` in the list of processes at run-time, using

```
seq.insert(process, index);
```

4.6.3 The alternative construct

Sometimes a choice must be made of one process out of a set of processes that are simultaneously committed in communication. Sequential programming languages, like Java, offer `if-then-else` clauses for making choices in the flow of control of the program. An `if-then-else` construct works for Boolean expressions but *not* for events, since an event cannot return `true` or `false`. An event occurs or does not. An `if-then-else` construct is only suitable for checking for conditions and not for catching events.

CSP provides a choice operator that allows choosing one process out of many processes which are ready to engage in the first event. This process is also called *alternative construct*. The alternative construct combines a number of processes guarded by channel inputs, channel outputs and channel timeouts. The alternation performs the process associated with a guard which is ready (Roscoe, 1998). This process, to which a guard is associated, is called a *guarded process*. A guard is ready when the guarded process can engage in a first communication event, as the first action of the process. This is called *conditional communication*. If no guard is ready, the alternation will suspend until a guard becomes ready. A suspended alternative construct consumes no time. As soon as one guard becomes ready (i.e. an *alting process* at the other end of one of the channels is willing to communicate) it will resume the alternative construct followed by the execution of the guarded process. When the selected guarded

process finishes, the execution of the alternative construct finishes as well.

The implementation of the CSP choice operator in CT is prioritized and is divided into

- Equally-prioritized alternative construct
- Unequally-prioritized alternative construct

Equally-prioritized alternative construct

The equally-prioritized alternative construct (alternative or ALT) is instantiated by

```
Alternative alt = new Alternative(Guard[] guards);
```

The argument `guards` is an array of guard objects. A guard is an instance of the `Guard` class. There are two ways to create alternative constructs in CTJ: as a composition-based construct or as a select-based construct.

Composition-based construct

The compositional approach is almost similar to the sequential and parallel constructs as described in the previous sections. The following example shows an `Alternative` composition for three guarded processes.

```
Alternative alt = new Alternative(new Guard[] {
  new Guard(channel1, new Process1(channel1, ..)),
  new Guard(channel2, new Process2(channel2, ..)),
  new Guard(channel3, new Process3(channel3, ..))
});

alt.run();
```

Listing 4-8 *Composition-based alternative construct.*

The alternative process starts by invoking its `run()` method. Here, $\texttt{channel}_i$ is an input channel or output channel of $\texttt{Process}_i$. The `Guard` with $\texttt{Process}_i$ is ready when a process at the other end of the channel is waiting. A guard that becomes ready is then candidate for selection. The

`alt` process waits until at least one guard becomes ready and completes successfully when one of the ready guards is selected and its respective guarded process has successfully executed. If more than one guard is ready then one guard will be *randomly* selected; theoretically this is a non-deterministic choice and practically any selection mechanism is applicable. CT's alternative construct makes its selections *fairly*, i.e., when more than one guard is ready, the guard to execute will be selected according to a first-come-first-served policy and the process that it guards will then be executed.

New guards may be added at run-time, using

```
alt.add(guard);
```

or by adding multiple guards at a time:

```
alt.add(new Guard[] { guard1, guard2, ..});
```

A specific guard may be removed from the guard list, using

```
alt.remove(guard);
```

Processes that are specified in a guard can also be written as anonymous processes, as shown in Listing 4-9.

```
Integer n = new Integer();      // n is an Object

Process alt = new Alternative(new Guard[] {
  new Guard(inChannel[0], new Process() {
    public void run()
    throws ExceptionSet {
      inChannel[0].read(n);
      ... do something with n
    }
  }),
  new Guard(inChannel[1], new Process() {
    public void run()
    throws ExceptionSet {
      inChannel[1].read(n);
      ... do something with n
    }
  })
});
```

```
for (int i=0; i<20; i++) {
  alt.run();       // make the selection and run the response
}                  // 20 times
```

Listing 4-9 *Example of a composition-based alternative construct.*

Select-based construct

The select-based alternative construct starts by invoking the `select()` method, as with

```
i = alt.select();
```

Index `i` specifies the guard that was selected; $i \in [0,n\text{-}1]$ and n is the number of guards. This method does not execute any specified process of the selected guard. Therefore, guards in a select-based construct do not specify processes. A `switch-case` clause can execute guarded processes. Examples are given in Listing 4-10 and Listing 4-11.

```
Alternative alt = new Alternative(new Guard[] {
  new Guard(inChannel[0]),
  new Guard(inChannel[1])
});

Integer n = new Integer();

for (int i=0; i<20; i++) {
  int index = alt.select(); // wait for a channel
  inChannel[index].read(n); // read from selected channel

  ... do something with n
}
```

Listing 4-10 *Example of a select-based alternative construct.*

In CT, a channel can also play the role of a guard which simplifies the code. Every object that inherits the `Guard` class can play the role of a guard. This can only be used with the select-based alternative construct.

```
Alternative alt = new Alternative(new Channel[] {
  inChannel[0],
  inChannel[1]
});
```

```
Integer n = new Integer();

for (int i=0; i<20; i++) {
  int index = alt.select(); // wait for a channel

  switch(index) {
    case 0: inChannel[0].read(n);
            ...
            break;
    case 1: inChannel[1].read(n);
            ...
            break;
  }
  ... do something with n
}
```

Listing 4-11 *Example of a select-based alternative construct with simplified guards.*

For call channels, the following guards exist. A guard that guards a call channel for a particular method, denoted by constant number `channel.METHOD`, is:

```
Guard guard = new Guard(callchannel, callchannel.METHOD,
                new Process(callchannel,..));
```

A guard with a range of methods, for example `on()`, `off()`, and `calculate()`, is specified by

```
Guard guard = new Guard(callchannel,
                int [] { callchannel.ON, callchannel.OFF,
                      callchannel.CALCULATE},
                new Process(callchannel,..));
```

The guard that accepts any method is specified by

```
Guard guard = new Guard(callchannel,
                new Process(callchannel,..));
```

The same guard can be used at a client process *or* a server process. If the alting process calls a method on the call channel then this guard will assume an `accept()` by the guarded process. If the alting process is

accepting on the call channel then this guard will assume a call to `method()` by the guarded process.

The call channel `callchannel` can play the role of a guard, as with data channels, without specifying the `Guard` class. Barriers cannot be used as guards in the alternative construct.

Unconditional and conditional guards

The guard object signals the alternative construct when it becomes ready. The policy of a guard to become ready can be conditional or unconditional.

As shown in the previous section, a guard object may be declared as follows,

```
Guard guard = new Guard(channel, new Process(channel,..));
```

The guard becomes `true` when argument `channel` is ready. The guard described above always participates in the alternative construct and is called an *unconditional guard*. A guard is a *conditional guard* when it is enabled and when some condition is true; otherwise the guard is disabled and omitted by the alternative construct. A disabled guard will never be selected.

For example, in Listing 4-12 the variable `condition.value` represents the result of a Boolean expression.

```
Boolean condition = new Boolean(false);
condition.value = (temperature > 30);
Guard guard = new Guard(condition, channel, new Process(channel,..));
```

Listing 4-12 *Example of a conditional guard.*

If `condition.value` is `true` then the guard will be ready when the specified `channel` is ready, otherwise the guard is omitted and the guarded process will not be selected. The parent process of the alternative construct and guarded processes may update variable `condition.value` at any time.

A conditional guard is declared as

```
new Guard(new Boolean(true), channel, new Process(channel,..))
```

and is equivalent to

```
new Guard(channel, new Process(channel,..))
```

Applying conditional guards is useful for implementing a state machine on communication events in a safe and elegant manner. Similarly, conditional guards also exist for call channels.

The skip guard allows the alternative constructs to withdraw and continue when no guard is true. An overview of skip guards is given in Appendix D.4.1 The timeout guard allows the alternative construct to withdraw and to continue after the expiration of a specified time when no guard is true. An overview of timeout guards is given in Appendix D.4.2.

Unequally-prioritized alternative construct

The unequally-prioritized alternative construct (prialternative or PRIALT) is similar to the alternative construct and is created by the `PriAlternative` class. The `PriAlternative` class extends the `Alternative` class and overrides the *equally-prioritized choice* mechanism with a *unequally-prioritized choice* mechanism. The prialternative construct is instantiated with

```
PriAlternative prialt = new PriAlternative(Guard[] guards);
```

The `prialt` process waits until at least one guard becomes ready and finishes successfully when one of the three guarded processes is selected and has successfully executed. The `Guard` with $\texttt{Process}_i$ may be selected when $\texttt{channel}_i$ is ready. Here, $\texttt{channel}_i$ is an input channel or output channel of $\texttt{Process}_i$. If more than one guard is ready than the guard with the lowest index will be selected and the guarded process of the selected guard will be executed.

As with the alternative construct, methods like `add(..)` and `remove(..)` exist and an additional method is added to insert a process at run-time, using

```
prialt.insert(guard, index);
```

Guard `guard` will be inserted at `index` in the guard list. The order of priorities will automatically be applied to the new guard list.

4.6.4 The exception handling construct

Reliable software should deal with all situations in the environment, which have an effect on the behaviour of the software. Unusual situations can cause *exceptional occurrences* or *exceptional states* in software which, when unhandled, can cause an undesirable behaviour of the program. The exception manifests an error. Exceptions should be handled by a proper design concept that deals with its complexities, such as compositionality and state explosions. On the occurrence of an exception, it requires switching from the main process to the exception handling process. The exception handling process is called an *exception handler*. The main process is not concerned with exception handling. A proposed CSP-based exception handling construct is discussed in this section.

The exception mechanism

Exceptions are events and states of disruption of the current flow of control that occur at a particular time and space. Exceptions are observed by two behaviours:

- **An exception as an unsuccessful termination.** An illegal state in a process can be seen as an exception from which point the process should stop continuing, like division by zero or a temperature value being out of range. In case of an exception, the process must abort its actions. In other words, the process must terminate unsuccessfully. The exception operator will

capture the unsuccessful termination of a process and preempts to the exception handling process.

- **An exception as an interrupt.** Channels or barriers that are in exception prevent communication events. Processes that are willing to engage in communication events, on so-called corrupted channels or corrupted barriers, should escape from blocking forever. In this case, exceptions are a gloss on channels or barriers. In case a higher-priority process is waiting for a channel or barrier that becomes corrupt, the higher-priority process will preempt the running lower-priority process in order to terminate unsuccessfully. The exception will cause an interrupt of the running lower-priority process.

The exception construct has been developed in CT in such a way that it goes with the other compositional constructs. The exception construct fulfils several desirable properties:

1. The exception construct is derived from a theoretical model of an exception operator described in the CSP language (see Appendix C). The exception operator is simple enough and no changes to the semantics of the other CSP operators are required. The exception operator can be used for formal model checking and for reasoning about the behaviour of exception handling.

2. This version of the exception operator abstracts away from exception types.

3. The CT implementation of the exception handling should fulfil the semantics of the theoretical exception operator and encompasses exception types on which the exception handler can make decisions.

4. The exception handling concept is lightweight and thread-safe.

5. This exception mechanism implements the *termination model* of exception handling (Burns and Wellings, 1990). In the termination model, control never returns to the point in the program execution where the exception was thrown. The mechanism does not implement the *resumption model* of exception handling (Burns

and Wellings, 1990). The resumption model allows an exception handler to correct the exception and then return to the point where the exception was thrown. Error recovery is often not possible, or difficult to realize, with acceptable overhead costs.

The exception mechanism collects the exceptions of each parallel construct. All compositional constructs contribute to this mechanism. The exception set will be passed (thrown) back to the caller of the process (construct), i.e. up the hierarchy of processes. At the level of the exception constructs, the exception set is caught and the set is available to the exception handler. The handled exception must be removed from the set by the exception handler. The set of remaining unhandled exceptions must be thrown by the exception handler so that it can be intercepted by another handler.

The exception operator requires an exception set which contains individual exceptions that occurred in concurrent processes at run-time. In CT, the exception set is an instance of the `ExceptionSet` class. The `ExceptionSet` object collects the exceptions that occurred in the process and its child processes. Thus a process throws an `ExceptionSet` object on exception, see the process interface in Section 4.3. Each sequential construct has its own `ExceptionSet` object and thus an `ExceptionSet` object exists for every branch at the parallel and priparallel constructs.

Errors can also occur in a channel, e.g. a hardware failure occurs in the system. Those errors which cannot be repaired by the channel must be thrown as exceptions. The exception handler should deal with one or more exceptions and therefore the exception set is important. Each exception must be collected in the exception set. The `throw` keyword in Java allows for throwing single exceptions and this is not according to the proposed theoretical model. Therefore, the `throwExceptionSet` method should be used instead of `throw`. The `throwExceptionSet` method appends the specified exception to the exception set. The exception set is a system attribute per sequential construct. Successively, the exception set is thrown by the method. For example, the channel (or link driver) can throw an `IOException` with

```
System.throwExceptionSet(new IOException("cable failure"));
```

The specified `IOException` will be appended to the exception set and the exception set will be thrown to the invoking processes using the Java `throw` statement. The exception construct or the Java `try-throw-catch` clause must be used to catch the exception set. Three static `throwExceptionSet` methods are provided by the `System` class.

The method

```
System.throwExceptionSet(Exception exception)
```

adds the specified `exception` to the exception set and throws the exception set instead of the `exception`.

Similarly, the method

```
System.throwExceptionSet(Exception[] exceptions)
```

adds multiple exceptions to the exception set when more than one exception occurred at the same instance of time. If the `throwExceptionSet` method specifies an exception object that is already in the set then the exception object will be omitted and the set will be thrown.

If the exception handling process does not deal with all exceptions then the handler must pass the remaining exception set to the next exception construct using

```
System.throwExceptionSet()
```

If the exception set is empty then this method will not throw the exception set but will return instead.

The exception handling process can get the exception set of the current thread of control with

```
ExceptionSet es = System.getExceptionSet()
```

The `ExceptionSet` class offers a small set of public methods for the exception handling process. The exception handling process can use the iterator (of type `ExceptionIterator`), provided by `ExceptionSet`, to wander

through the exception elements and deal with the individual exceptions. When the exception is handled, it should be marked as handled using the `handled()` method.

An exception construct is defined by the `ExceptionCatch` class and can be instantiated with

```
ExceptionCatch exc = new ExceptionCatch (new P(), new E());
```

This process performs process *P* and it switched to exception handler *E* on exception in *P*, according to the semantics of $P\vec{\Delta}E$ (Section 3.6.5 and Appendix C). As with any other process this construct can be executed with

```
exc.run();
```

This process encapsulates the `try-throw-catch` construct and allows nesting with other compositional constructs. An example of exception handler `E` is illustrated in Listing 4-13.

```
import csp.lang.*;
import csp.lang.Process;

public class E implements Process
{
  public E() { ... }

  public void run()
    throws ExceptionSet {
      ExceptionSet es = System.getExceptionSet()
      for (ExceptionIterator i = es.iterator(); i.hasNext();) {
        Exception e = i.next();
        ... handle exception
        i.handled();
      }
      System.throwExceptionSet();
  }
}
```

Listing 4-13 *Exception handler process class.*

Java supports a language-based exception mechanism that implements the termination model of exception handling. The three keywords for

exception handling in Java are `try`, `throw`, and `catch` (Arnold et al., 2000). This is similar for C++. A method can throw a single exception-object up the hierarchy until a `try-catch` clause catches the exception. This `try-throw-catch` mechanism is usually fast and it can implement the proposed exception mechanism in an efficient way. This `try-throw-catch` mechanism has similarities with the proposed exception handling mechanism. See the example in Listing 4-14. The exception will be removed from the exception set `es`. If the exception set is not empty then the exception set will be thrown further upwards; otherwise the exception handler terminates. The `try-throw-catch` construct is a fixed construct and not truly compositional.

```
try {
  ... perform process  -- process P
} catch (ExceptionSet es) {
    for (ExceptionIterator i = es.iterator(); i.hasNext();) {
      Exception e = i.next();
      ... handle exception
      i.handled();
    }
    System.throwExceptionSet();
  }
```

Listing 4-14 *Example of a try-throw-catch clause with exception handling.*

Exceptions and sequential construct

A sequential process that cannot engage in a communication event on a corrupt channel or corrupt barrier, is in exception. The channel or barrier will throw an exception and the sequential construct will unsuccessfully terminate. The exception is added to the exception set.

Exceptions and parallel construct

A parallel construct terminates when all parallel processes terminate whether or not they terminate successfully or unsuccessfully. The parallel construct terminates unsuccessfully when the exception set is not empty and the exception set is thrown. In other words, if one of the parallel processes in the parallel construct terminates unsuccessfully then

the parallel construct also terminates unsuccessfully. The parallel construct collects all exceptions that occurred in the parallel processes and the construct will throw the exception set on termination.

Exceptions and alternative construct

The alternative construct will terminate unsuccessfully when one or more channels in the guards are in exception. In this case the alternative construct cannot make a fair selection. The alternative construct will collect all exceptions in the exception set. Of course, the alternative construct will throw all exceptions that occurred in the guarded process it executes. This implies that when a guarded process unsuccessfully terminates then the alternative construct also unsuccessfully terminates.

Exceptions and priparallel construct

The mechanism is similar to the parallel construct. The exception handling at a higher priority can perform preemption of processes executing at lower priority. This preemption happens only when a process at a higher priority is invoking on a channel or barrier that is corrupt, or when the process is waiting for a channel or barrier that becomes corrupt (refused by the environment).

Exceptions and prialternative construct

The mechanism is identical to the alternative construct.

Exceptions and exception construct

The exception handler can throw a new exception, can pass unhandled exceptions, and can remove exceptions from the set that are handled by the handler. If the exception handler terminates unsuccessfully then the exception construct terminates unsuccessfully.

Other applications

The exception construct can be used for things other than handling errors. For example, if processes must quit their tasks then the channels they use can be poisoned and these processes will terminate on reading and writing on poisoned channels. See the `System.refuse(..)` method in Section 4.7.2 .

4.6.5 Nested compositional constructs

The `Sequential`, `Parallel`, `PriParallel`, `Alternative`, `PriAlternative`, and `Exception` are compositional processes that may be nested (composed) within other compositional processes. Here, compositional programming is illustrated by an arbitrary example, see Listing 4-15. A single `run()` invocation starts the composition. At declaration of the construct, the initial state is set before the `run()` method is invoked. This makes the execution of the composition efficient.

```
Process process = new Exception(
  new Sequential(new Process[] {
    new Parallel(new Process[] {
      new Process1(..),
      new Process2(..)
    }),
    new Alternative(new Guard[] {
      new Guard(channel1, new Process3(channel1, ..)),
      new Guard(channel2, new Process4(channel2, ..))
      new PriAlternative(new Guard[]
        new Guard(channel4,
          new Sequential(new Process[] {
            new Process5(channel4, ..),
            new Process6(..)
          })),
        new Guard(channel5,
          new Sequential(new Process[] {
            new Process7(channel5, ..),
            new Process8(..)
          })),
    }),
    new Parallel(new Process[] {
      new Process9(..),
```

```
        new Process10(..)
      })
    }),
    new Process {
      public void run() {
        ... exception handling process
      }
    }
);

process.run();
```

Listing 4-15 *Example of a nested construct with alternative construct.*

This example shows that the alternative construct can play the role of a process or the role of a guard.

4.7 Timing and Sampling

Control theory assumes a constant sampling period between successive inputs (sampling) and outputs (actuation) and the outputs should be calculated before the end of the sampling interval. Variations of the sampling interval can degrade the performance of the controlled system and even lead to instability of the system. This is called *jitter*, which is defined as the variation of a point in time around a reference point in time. Jitter should also be prevented between the multiple inputs that are supposed to be sampled at a predefined reference point in time. This is similar with multiple outputs.

In control software, sampling and actuation should be independent from priority-based preemptive scheduling methods. Preemption causes variations in process execution. Timed threads are provided by operating systems for creating timely activated tasks or timed processes. Even when timed processes operate at the highest priority there is no guarantee of jitter-free behaviour when interrupts can preempt timed processes at any time. Therefore, sampling and actuation should not be performed by timed processes. Timed threads and timed processes are inadequate solutions for hard real-time control software.

The CSP paradigm offers a solution that is conceptually clean and without surprises. Sampling and actuation are timed events that are related to channel communication between the controller and the plant. Therefore, channels are concerned with sampling and actuation. This section introduces timed communication events.

4.7.1 Timed communication events

Timed communication events have been proposed in (Hilderink and Broenink, 2003). This proposal allows for a conceptual approach that incorporates timing on channels and barriers. The concept specifies that the environment may engage in communication events on a permanent or timely basis.

Time is part of the environment. Therefore, CT provides some methods or system services that can be used to command the environment, say the *environmental process*, to accept or refuse communication events on channels or barriers, on a permanent or timely basis. This results in timed communication events *or* exceptions when the timing requirements are not met.

The hard real-time timer and the associated interrupt service routine are part of the environment of the program. The environmental process can assign channels or barriers to the timed interrupt service routine and programs the timer. On the interrupt, the environmental process will engage in the communication event and at that moment communication takes place. The interrupt service will execute the link driver of the data channel, which performs sampling or actuation, at precise moments in time. Barrier and call channels do not provide link drivers and they will be released at the instance of time.

This proposed concept offers,

- atomic, accurate, and high-performance sampling and actuation,
- notion of time to *untimed* CSP,
- separation of concerns increases the reusability of processes,

- exception handling on crossing hard deadlines.

If necessary, on the basis of timed communication events one can also create timed processes.

4.7.2 System services

The following methods are services that can be carried out by the environmental process. These methods are called by the control application and they are served by the environmental process when the environmental process is willing to accept the method call. The `csp.lang.System` class provides a global static call channel whose service can be invoked by any process at any time. These services are used by network builders which setup the timing on the created network of communication processes.

Accept communication event at specified time

```
System.at(channel, time);                  // single-shot
System.at(channel, time, interval_time);   // periodical
```

The producer and consumer processes must be willing to communicate on the specified channel or barrier, before the environmental process is willing to accept the communication event at the specified time (in microseconds) or period. If this is not the case and the processes engage in the event after the specified time expires then the real-time requirement has not been met. In this case, any blocked process will be released and a `RealTimeException` exception is thrown at the producer and at the consumer. The first `at(..)` method specifies a single deadline and the second method specifies periodical deadlines.

```
System.at(barrier, time);                  // single-shot
System.at(barrier, time, interval_time);   // periodical
```

The environmental process will participate in the barrier synchronization and will commit to the synchronization at the specified time. If one participant does not sync before the environmental process then

`RealTimeException` exceptions will be thrown to all processes and all processes will be released. Hence, the real-time requirement has not been met.

Accept communication event `after` specified time

```
System.after(channel, time);                // single-shot
System.after(channel, time, interval_time); // periodical
```

The communication between the producer and consumer processes will be delayed until the specified time. Any communication after the specified time will be accepted and they both immediately continue. No exceptions are thrown. If an interval time is specified then the next waiting time will be incremented with the interval time.

```
System.after(barrier, time);                // single-shot
System.after(barrier, time, interval_time); // periodical
```

The environmental process will participate in the barrier synchronization and will commit to the synchronization at the specified time. No exceptions are thrown. If an interval time is specified then the next waiting time will be incremented with the interval time.

```
System.after(guard, time);                  // single-shot
System.after(guard, time, interval_time);   // periodical
```

If the alternative construct is waiting and the alting process at the other end is willing to communicate before the specified time then the guard will become ready at the specified time. This guard is called a *timed guard*. No exceptions are thrown. A timed *skip guard* is used for specifying a *timeout-guard*. The skip guard will be ready at the specified time and no exceptions are thrown at timeout. As with channels and barriers, the guard can be periodically timed. The specified interval time increments time each period. Since guards are local to a process, this implies that `after(guard,time,..)` is locally used and no other process can alter time on a local guard.

The timed guard can be used with `at(..)` and `after(..)`. For example, the method `after(guard,time1,..)`, with `channel` being part of `guard`, can be used with `at(channel,time2,..)` or with `after(channel,time2,..)`. The method `after(guard,time1,..)` could endanger the deadline, as specified by `at(channel,time2,..)` when `time1 > time2`. In this case, the hard deadline will not be met and an exception may occur. Although this is a valid behaviour, this combination is not very useful and should be avoided.

The timing on a channel or barrier stops when `accept(..)` or `refuse(..)` are used on the channel or barrier.

`Refuse` communication and (optionally) throw exception

```
System.refuse(channel, exception_message);
System.refuse(barrier, exception_message);
```

This method will let the environmental process refuse the acceptance of the communication event on the specified channel or barrier. If an exception message is specified then it will let the channel or barrier throw the exception message to the participating processes. If no exception message is specified then the channel or barrier will block the invoking processes until the environment is willing to accept the events.

The `refuse(..)` method can be used to command the environment that an artificial refusal should be carried out. This method can be used for two main reasons:

1. The application can be tested by deliberately refusing communication. This way the robustness of the application can be tested.

2. In case the application deadlocks or livelocks then there is no way the program or a particular part of the program can terminate. In case the program deadlocks or livelocks, `refuse(..)` can be used to let channels or barriers throw exceptions. The exceptions will release synchronization and exception handling may gracefully

terminate the program. This is called *poisoning* a channel or a barrier (Section 3.5.1 and Appendix C.4).

A `UnacceptableException` results if the environmental process cannot refuse events on the specified channel or barrier.

`Accept` communication

```
System.accept(channel);
System.accept(barrier);
```

The environmental process will accept any communication event on the specified channel or barrier. This will cancel any timing, as specified with `at(..)` or `after(..)`, or any refusal that was specified with `refuse(..)`. If a channel or barrier that was refused cannot be accepted then an `UnacceptableException` exception is thrown. The method will be ignored and returns when it was called before `at(..)`, `after(..)`, or `refuse(..)`.

Get the actual `time`

```
long System.time();
```

Returns the absolute time read by the environmental timer.

4.7.3 Thread services

The `csp.lang.Thread` class in CTJ can delay the thread of control for a specified duration of time. The services offered by the `csp.lang.Thread` class are thread-oriented and therefore local to the process. This service does not involve channels or barriers.

```
Thread.sleep(relative_time);
```

Let the thread of control in a process sleep for the specified relative time.

```
Thread.sleepUntil(absolute_time);
```

Let the thread of control in a process sleep until the specified absolute time.

4.7.4 Example real-time timing

In this Section, an example illustrates real-time sampling and actuation that are based on timed channels.

Consider the two controller processes HController and LController as depicted in Figure 4-7.

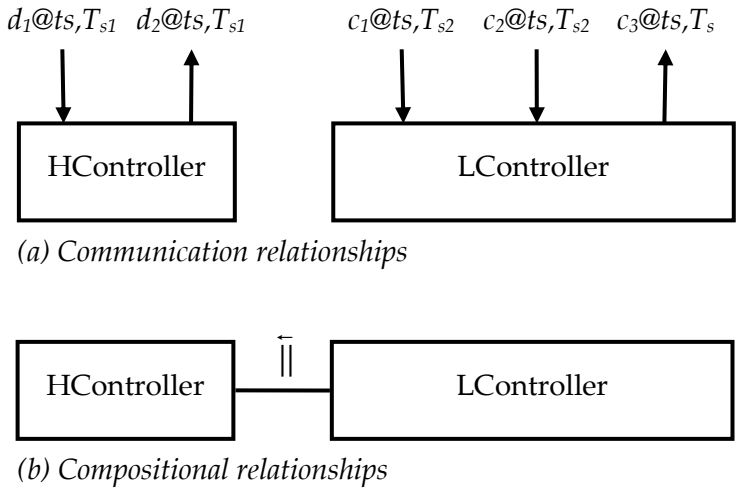


Figure 4-7 *Control application consisting of a higher-priority controller process and a lower-priority controller process.*

The external channels d_1 and d_2 are timed on sample interval T_{s1} and external channels c_1, c_2 and c_3 on sample interval T_{s2}. The start time *ts* is some delay after which the program is completely instantiated. In the following code we create the external channels and assign them to the environmental process with specified start time and sampling interval. The sampling rate for HController is 1 kHz and the sampling rate for LController is 0.1 kHz. The start time is specified such that sampling

starts when everything is constructed, otherwise deadlines may be passed on start-up.

```
//--- create external channels
Channel_of_Integer d1 = new ADC(0);
Channel_of_Integer d2 = new DAC(0);
Channel_of_Integer c1 = new ADC(1);
Channel_of_Integer c2 = new IncCounter(0);
Channel_of_Integer c3 = new DAC(1);

//--- set up sampling timing and register channels to environment
long Ts1 =  1000; // in usec
long Ts2 = 10000; // in usec

long ts = System.time() + 100000;  // start time

// firstly the inputs
System.at(d1, ts, Ts1);
System.at(c1, ts, Ts2);
System.at(c2, ts, Ts2);

// secondly the outputs
System.at(d2, ts, Ts1);
System.at(c3, ts, Ts2);

//--- create processes and compositional relationships
...
```

Listing 4-16 *Creating timed-events using external channels.*

Although processes can read and write on these channels in parallel, the actual conversions will be performed in some atomic sequence by the timed interrupt service routine.

Every registration with the same start time and sampling interval belong to the same atomic group and its order of execution is determined by the sequence of registration. See the sequence of `at(..)` statements in Listing 4-16. The sequence of inputs and outputs will be sorted by its time stamp and when the time stamps are equal then the sequence is determined by the sequence of registration. Due to this constraint, the programmer can minimize conversion latencies by choosing an optimal order of registration. The implementation is omitted in this thesis. The link driver

framework takes care of sequencing on a timer interrupt. Link drivers act as interrupt handlers.

This mechanism is illustrated using CSP diagrams, as depicted in Figure 4-8 and in Figure 4-9. Figure 4-8 shows the communication relationships of both controllers with their input-output counterparts in hardware (the input/output bubbles in the grey rectangle).

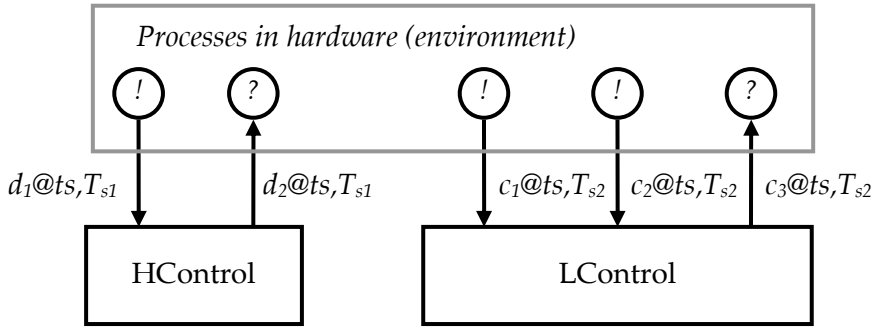
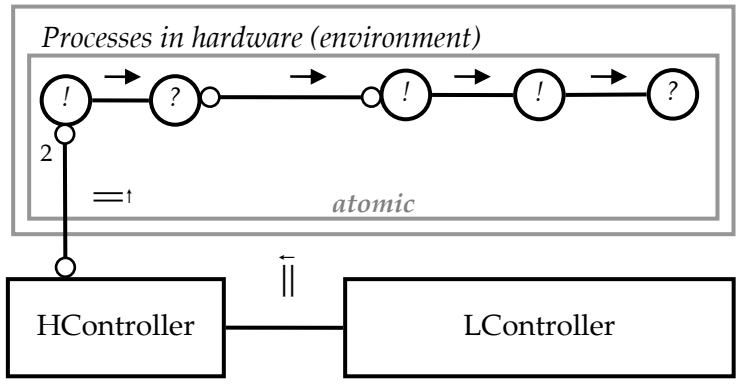


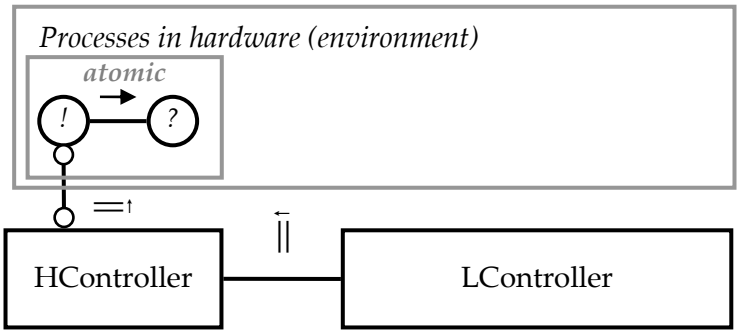
Figure 4-8 *Controllers communicating with devices.*

In Figure 4-9a-c, the compositional relationships between these hardware inputs/outputs are rendered for different scenarios. This is the solution for using interrupt handling on the internal timer. Process LController has a lower sampling frequency ($1/T_{s2}$) than the sampling frequency ($1/T_{s1}$) of process HController, with $T_{s1} < T_{s2}$. Thus, we specify that LController gets a lower priority than HController.

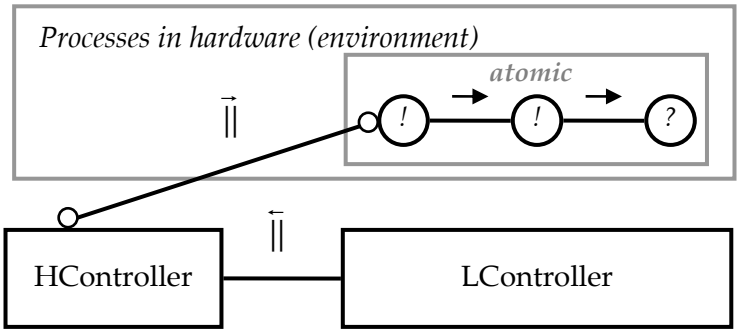
The conversions are atomically performed by the devices. That is, they cannot be interrupted by the application. This is depicted by the atomic rectangles in grey. Processor interrupt mechanisms are sequential and mostly priority-based or preemption-based. This is depicted by the sequential relationship. The prioritized parallel relationship between the hardware processes and the software processes are enforced by this environment.



(a) Compositional relationships on $t = n.T_{s1} = m.T_{s2}$.



(b) Compositional relationships on $t= n.T_{s1}$ and $t \neq m.T_{s2}$.



(c) Compositional relationships on $t \neq n.T_{s2}$ and $t = m.T_{s2}$.

Figure 4-9 *Atomic sequence of inputs and outputs by the environmental process.*

This mechanism adapts to three scenarios, where:

$t = n.T_{s1} = m.T_{s2} \quad \Leftrightarrow n.T_{s1} = m.T_{s2}$

$t = n.T_{s1}$ and $t \neq m.T_{s2} \quad \Rightarrow n.T_{s1} \neq m.T_{s2}$

$t = m.T_{s2}$ and $t \neq n.T_{s1} \quad \Rightarrow n.T_{s1} \neq m.T_{s2}$

with variable t being the actual time and $m,n \in [0,1,2,3,..]$ and $T_{s1} \neq T_{s2}$.

Scenario 1: Figure 4-9a shows the scenario of two equal time stamps, namely $n.T_{s1}=m.T_{s2}$. Sampling and actuation are performed in a predefined sequence; the sampling and activation for HController is performed before the sampling and activation for LController. The sampling and activation of the HController has a low-pass character that will ensure that c_1 and c_2 are not influenced by d_2. Furthermore, the delay between the first sampling and the last actuation is usually constant and small enough so that this does not affect the stability of the controlled system. A timing scheme is depicted in Figure 4-10.

Figure 4-10 *Timing scheme for HController and LController.*

Scenario 2: Figure 4-9b shows the situation when only time stamp t_{s1} is reached. A timing scheme is depicted in Figure 4-11.

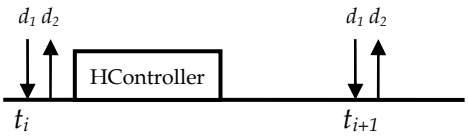


Figure 4-11 *Timing scheme for HController.*

Scenario 3: Figure 4-9c shows the situation when only time stamp t_{s2} is reached. A timing scheme is depicted in Figure 4-12. However, in practice, Scenario 3 will never occur when frequencies are multiplicities.

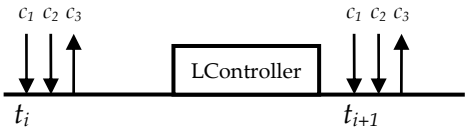


Figure 4-12 *Timing scheme for LController.*

In case all three scenarios are applied then there will be small variations (jitter) between the input and output conversions. In Figure 4-10, input d_1 is converted at t_i and c_1 at $t_i+\Delta T$. In scenario 3, input c_1 is converted at t_i which is earlier than in scenario 1. Similar variations happen between the output conversions. These variations are very small compared to the timing interval. Since the parameters of a controller are a function of time, these variations may cause inaccurate values of the parameters. In case this has a significant and wrong effect on the behaviour of the controller (e.g. instability), one can choose scenario 1. In scenario 1, all conversions are performed at the highest frequency and variations are eliminated. Buffered processes are required to decouple the inputs and outputs from lower-frequent controllers. This scenario is depicted in Figure 4-13.

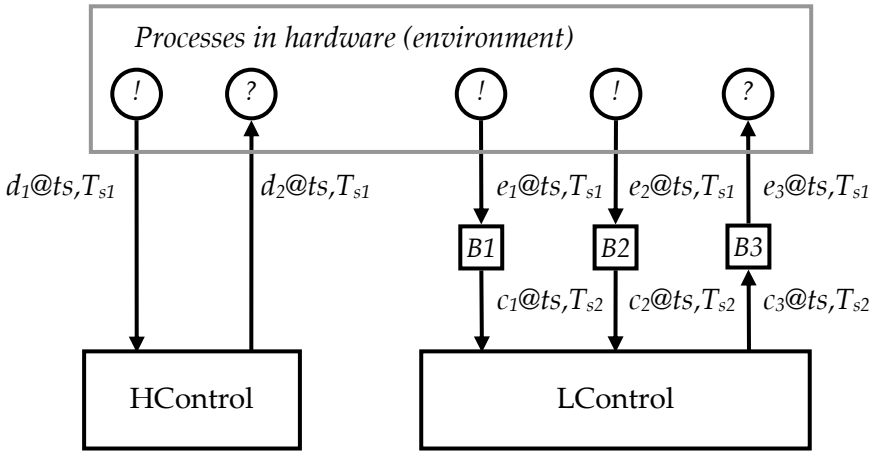


Figure 4-13 *Controllers communicating at different frequencies but with the same sampling and actuation frequency.*

In this example, *ts* is the start time for the timer to start. The buffered processes B_1 and B_2 are sub-sampling and buffered process B_3 is super-sampling. Here, e_1, e_2, and e_3 are additional timed channels required for sample interval T_{s1}. T_{s2} is a multiplicity of T_{s1}.

4.8 Conclusions

The CT library offers a set of process-oriented design patterns or constructs for implementing concurrent software with object-oriented programming languages. These constructs allow true compositional programming of reactive software.

The semantics of the proposed constructs are the building-blocks on which the user can build reliable and reasonable concurrent software. The concept of reasoning in terms of processes, channels, and barriers provide a logical separation of hardware dependent and hardware independent concerns. Multithreading is freed from the mind set of the user.

The proposed constructs provide a systematic way of handling exceptions and timing in concurrent programs. The proposed solutions to timed events (i.e. timed channels and timed barriers) are more accurate than timed processes and this solution is useful for sampling and actuation in control systems.

The aspects simplicity, portability, and generality are demonstrated in Chapter 6. See conclusions in Section 6.7.

CHAPTER 5

Notion of priorities

5.1 Introduction

Priority is meant as a solution for optimizing program execution in order to increase its reactivity and responsiveness. Priority specifies the importance or urgency between tasks concerning a shared resource to which some kind of precedence rule is applied. The precedence rule determines which task can precede the other task, since no two tasks can or are allowed to perform on the shared resource at the same time. At a low level of abstraction—devoted to the CPU and its threads of control, priority is seen as a scheduling parameter used by a scheduler. At a high level of abstraction—appropriate for the human mind—priority is an urgency or priority relationship between two event handling processes. This notion of priorities for the CT object model (Chapter 4) is defined in this chapter. The precedence rules are defined for the communication relationships in presence of compositional constructs. In order to get some trust in the efficiency of the CT libraries (Chapter 4 and 6) and CSP diagrams (Chapter 3), the scheduling policy is specified in this chapter. The implementation of the scheduler is not treated.

The notion of priority relationships is discussed in Section 5.2. The scheduling policies of the equally-prioritized and unequally-prioritized parallel constructs are described in Section 5.3. Particular patterns of compositions and communications between processes can result in inefficiency problems such as the *priority inversion problem* (Lauer and

Satterwaite, 1979; Sha et al., 1990), is discussed in Section 5.4. The channel communication is burdened with the task to solve these problems in a way that determines the quality of service. This affects the scheduling of the communication primitives as described in Section 5.5. CT implements enhanced alternative constructs, which improves the performance of concurrent software with respect to fairness and real-time requirements. Alting with notion of priorities is discussed in Section 5.6. Its efficiency is briefly discussed in Section 5.7. Although output guards are forbidden in occam for safety and implementation reasons, CSP allows output guards and so does CT. Output guards are discussed in Section 5.8. Conclusions to this chapter are drawn in Section 5.9.

The occam programming language (Inmos, 1988) is used to illustrate the listings, rather than using CTJ. Occam uses abbreviations for the compositional constructs, which keep the listings compact. The sequential construct is abbreviated as SEQ, the equally-prioritized parallel construct as PAR, the unequally-prioritized parallel construct as PRIPAR, the equally-prioritized alternative construct as ALT, and the unequally-prioritized alternative construct as PRIALT.

5.2 Priority relationship

In real-time systems, sporadic and periodic processes must be scheduled on a single CPU in order to meet their specific deadlines. Periodic processes in embedded systems may involve control loops, data acquisition, signal generation, etc. Sporadic processes may involve emergency buttons, safety switches, user interaction, etc. In any case, the process architecture as well as the run-time scheduling mechanism must be harmonized so that the total scheduling policy is able to guarantee that every deadline is met. Priorities are used to specify the scheduling of processes that are involved with some shared resource; e.g. a single CPU. More precisely, the threads of control within processes are scheduled. Processes do not know that they are scheduled.

Priority is defined as follows:

Definition (priority): *Priority* is a relationship between two or more processes that defines a precedence rule that determines which process has the permission to claim a shared resource at run-time.

Priority concerns the simultaneous use of a shared resource and the precedence rule only applies when a shared but exclusively used resource is involved. A shared resource can be a channel, a device, an object, memory, critical region, or a single processor. The precedence rule may require certain parameters, such as an index or time. These parameters concern the scheduling mechanism but they do not directly concern the user. Instead, the user is concerned with the fairness and unfairness of the system.

Priorities indicate the relationship of importance between processes—the importance of one process is *greater-than*, *lower-than*, or *equal-to* another process in the process architecture. The difference or equality of importance between processes is called the *priority relationship*. Priorities are relevant when processes engage in events. Therefore, the interrelationships between processes specify the priorities and apply precedence rules in process architectures. Consequently, the communication and compositional relationships are priority relationships that compose the total priority policy of process architectures. The prefix PRI as in PRIPAR and PRALT or the arrow on top of the interrelationships $\overleftarrow{||}$ and $\overleftarrow{\Box}$, specify priority relationships to which the scheduling policy will adapt in the process architecture. A priority relationship between processes can be fixed or preferred. The latter may change in time when the context changes.

Processes and events are unaware of their priorities and priorities are encapsulated in the *execution* of processes. Priorities are related to event handling. *Event handling* is the task that is executed by a process upon an event. Priorities may propagate via events from event handling to event handling. Communication and termination events perform the precedence rules.

5.3 Equally- and unequally-prioritized parallel constructs

Real-time processes cannot meet their deadlines when they have to wait for lesser-urgent processes to complete for which the total processing time exceeds the maximal CPU time. The lesser-urgent processes must not consume more processor time than actually necessary. Ultimately, there should be enough CPU time for remaining non-real-time processes to meet their requirements. The equally- and unequally-prioritized parallel relationships are used to specify the priorities of execution in process architectures.

Commonly, multithreaded programming interfaces consider priority as an index that can be assigned to each thread of control. A common policy in many operating systems is the lower the index, the higher the priority. Often, index 0 is the highest priority. The comparison between indexes expresses the priority relationship between the threads of control. Such explicit indexing of priorities has a global and absolute character. Consequently, the user must determine the absolute index values by global knowledge. The CSP constructs in this thesis abstract away from priority indexing and compose relative priority relationships between pairs of processes. The PAR and PRIPAR constructs assign separate threads of control with respectively equal and different priorities to its child processes. These priorities are relative between pairs of processes and local to the parent process. The PRIPAR is like a PAR construct with the additional property that the PRIPAR executes its child processes with declining priorities. The process on top of the process list gets the highest priority of all processes in the list. The highest priority is equal to the priority of which the parent process is executing. An example of declining priority relationships is given by the PRIPAR composition in Listing 5-1.

```
PRIPAR
  Process1    -- priority 0
  Process2    -- priority 1
  PRIPAR      -- priority 2
    Process3  -- priority 2.0
```

```
      Process4   -- priority 2.1
    Process5     -- priority 3
```

Listing 5-1 *A nested PRIPAR construct.*

The PAR performs non-preemptive scheduling between competitive parallel processes. The PRIPAR performs preemptive scheduling. *Non-preemption* allows scheduling to another process when the current process blocks or terminates. *Preemption* occurs on external communication events and lets a higher-priority process takes precedence over a lower-priority process. The lower-priority process can continue when the higher-priority processes have terminated or are blocked on communication. A composition of PAR and PRIPAR provides a composition between non-preemptive and preemptive scheduling that is optimal for the process architecture. With optimal is meant that context-switching is only performed when essentially required. Therefore, time-slicing is not part of the scheduling policy, but time-slicing can be built by the following construct in Listing 5-2.

```
PRIPAR
  Timeslicer(time)     -- priority 0
  PAR                  -- priority 1
    Process1           -- priority 1
    Process2           -- pritoriy 1
```

Listing 5-2 *Time-slicing construct.*

The `Timeslicer` process contains a simple infinite WHILE loop with a sleep statement. The process repeatedly sleeps for the specified time and sleeps again after the time has expired. Each time the `Timeslicer` wakes up it will preempt the PAR construct and when the `Timeslicer` sleeps again it will reschedule the next process in the `PAR` construct. The `PAR` will alternately schedule its processes in a round-robin fashion—this is fair.

The PRIPAR construct provides fixed-priority scheduling and is used to implement a *rate-monotonic* (RM) scheduling scheme (Sha et al., 1990). This scheme assigns priority to the execution of processes based on their periods. The *rate* is the inverse of the period; the shorter the period, the higher the priority. Rate-monotonic scheduling is common for control systems and many other classes of real-time systems. The priority

assignment is simple and its implementation is lightweight compared to dynamic-priority scheduling schemes, such as an *early deadline first* (EDF) scheduling scheme (Sha et al., 1990). Generally, control applications can be optimal scheduled using rate-monotonic scheduling and do not require a dynamic-priority scheduling scheme.

The priority relationships of Listing 5-1 are graphically depicted in a CSP diagram in Figure 5-1.

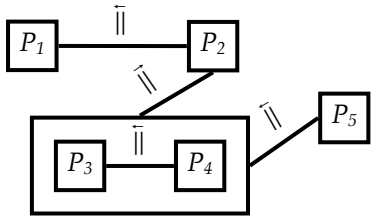


Figure 5-1 *Example of a composition diagram of a nested PRIPAR construct.*

The precedence rules, as specified by these unequally-prioritized parallel relationships in this composition diagram, are applied on each communication event and termination event within this system. Preemption of lower-priority processes happens when a higher-priority process can proceed upon an event from an external channel or an external barrier.

In composition diagrams (as in Figure 5-1) and in occam (as in Listing 5-1) the priority relationships between processes are static. In CT, priority relationships can change at run-time by moving processes in the list of processes in the PRIPAR construct. See the methods `add(..)` and `remove(..)` in Section 4.6.1. For every change in the unequally-prioritized parallel relationship a separate composition diagram is required to *express* each change.

5.4 The priority inversion problem

Section 4.5.1 illustrates a technique to determine whether or not a process architecture is priority conflict-free. A priority conflict can increase the

performance of lower-priority processes at the cost of the performance of high-priority processes. This is known as the *priority inversion problem* (Sha et al., 1990). Sometimes a priority conflict seems to be inevitable. The solution offered here is a design refinement, which results in priority conflict-free designs. It deals with eliminating the source of the problem rather than fixing the problem by makeshift solutions.

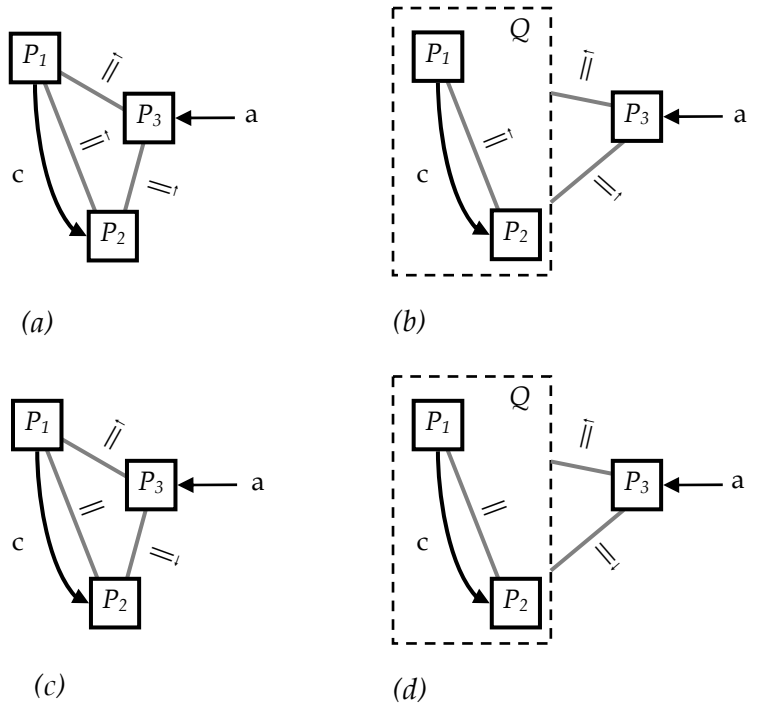


Figure 5-2 *(a) priority inversion problem in design,*
(b) priority analysis shows priority conflict design,
(c) priority inheritance in design,
(d) priority analysis shows priority conflict-free design.

Figure 5-2a illustrates an example of a priority inversion problem with processes and a channel *c*. Should a high-priority process (P_1) be blocked on a channel, waiting for communication with a lower priority process (P_2), it may have to wait a longer time than seems reasonable. A third process (P_3) of middling priority might be hogging the CPU. The channel (shared resource between P_1 and P_2) causes the priority inversion problem. The network of processes seems to be conflict-free, but if one considers the communication between P_1 and P_2 as one communication

process Q then a priority conflict raises between the processes Q and P_3. See the conflicting unequally-prioritized operators between Q and P_3 in Figure 5-2b. The analysis technique to find priority conflicts is described in Section 3.8.4. In other words, at the moment of rendezvous a priority inversion problem raises.

In order to do justice to the overall system performance, it would be reasonable to elevate the priority of P_2 to the level of the blocked P_1 when P_1 gets blocked on the channel with P_2. In other words, the lower-priority process inherits the priority of the blocked higher-priority process until the high-priority process can carry on. This solution is called *priority inheritance* (Lauer and Satterwaite, 1979; Sha et al., 1990). Figure 5-2c illustrates the effect of priority inheritance on the relationships. On priority inheritance, the priority relationships become different as was originally specified. This is also clarified in Figure 5-2d, which shows that the priority conflict is solved. Immediately after communication via the channel the priority will be restored to the lower-priority as was specified by the priority relationships, see Figure 5-2a. Here, priority is no longer static and can temporarily change in order to serve a higher-priority process. The *ceiling protocol* provides a solution for transitive blocking (chain of blockings) (Cornhill et al., 1978). The ceiling protocol could prevent deadlock, but deadlock is a pathological problem of the process architecture and not a problem of scheduling. In addition, the inheritance and ceiling protocols are problematic for channel-based software architectures, since channels can only retrieve the priorities between processes on the moment when both threads of control enter the channel. The channel does not know a priori which processes (or threads) access the channel.

Priority inversion comes from a bad design in the first place and the priority inheritance and the ceiling protocol are bad solutions to a bad design. From the point of view of a higher-priority process P_1, the last thing it wants is the priority to be raised of another process P_2. Priorities are set for a reason. This does not imply that priorities are static at all times. Priorities should be able to change by external influences in order to improve the performance of the program. The priorities that are initially set are called *preference priorities*.

A design pattern, which follows the following rule, can avoid the priority inversion problem.

> *Do not communicate with a lower-priority process unless you do not have any real-time guarantees to deliver; otherwise, feel free to communicate with a lower-priority process (and maybe get blocked) if you currently have no real-time service commitments.*

This solution to solve priority inversion is different to priority inheritance, which follows the design pattern:

> *Give the high-priority servicing process an equal-priority buddy process that has the only task to communicate with a lower-priority process.*

Thus, when a high-priority servicing process needs to communicate with a lower-priority process, get its buddy process to do it. The buddy process is listening out for the servicing process so the servicing process will not be blocked communicating with its buddy. The buddy may get blocked communicating with the lower-priority process but no matter the higher-priority process is still alive and servicing. The servicing process needs to remember *not* to communicate with its buddy until its buddy communicates back after dealing with the lower-priority process. If this is necessary, some more buddies are needed. The buddy process needs to be of equal-priority with the servicing process so that the buddy will succeed as soon as the low-priority process is ready to communicate with it so that it gets the attention of the servicing process when that is been done.

This design pattern needs no priority rising, but the design must be refined with additional processes and handshaking between the higher-priority process and the buddy process. A simplified refinement that overcomes the priority inversion problem is a buffer process that has the role of a buddy process as depicted in figure Figure 5-3a. The higher-priority process writes to the buffer and can immediate continue without being blocked. After servicing the buffer process waits until the lower-priority process consumes the message.

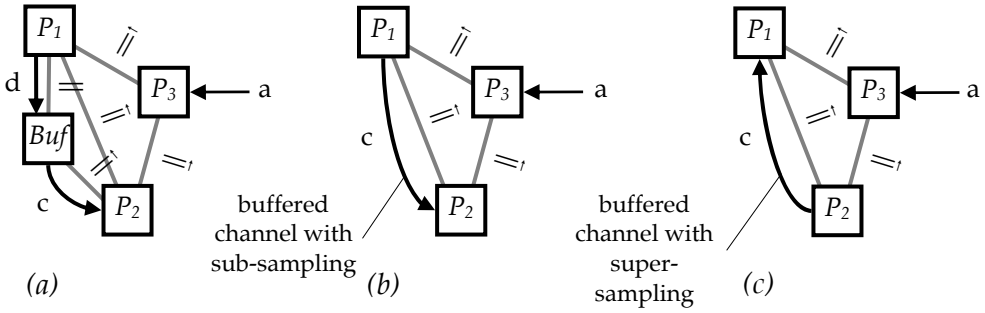


Figure 5-3 *(a) Solution with buffer process,*
(b) sub-sampling buffered channel,
(c) super-sampling buffered channel.

A buffered data channel replaces a buffered process and additional data channels. Such an implicit buffer simplifies the CSP diagram. See Figure 5-3b and Figure 5-3c. The buffer is a property of a data channel as a means to solve priority inversion problems. Figure 5-3b uses a sub-sampling buffered data channel that overwrites values and Figure 5-3 uses super-sampling which generates values that are equal to the last value that was written to the channel. In Appendix F, a proof is given that a sub-sampling or super-sampling buffered data channel can solve a priority inversion problem in case data channels cause priority conflicts.

A buffered data channel may save context switches but at the same time it can decrease the reactiveness of the program. In CSP-based process architectures one can reason about where to place a specific kind of buffered data channel instead of a rendezvous data channel. This is also discussed in Section 3.8.5. One should start the design of a process architecture with rendezvous channels. From this point on one can refine the model with buffered channels with the right kind of buffer at places where a buffer does improve the throughput and does not decrease the reactivity and responsiveness of the system.

The reverse approach, by starting a design with an asynchronous communication model, complicates the preservation of reactivity and economically using memory. In small computer systems memory can be scarce. Note that asynchronous behaviour does not solely come from buffered communication. Asynchronous behaviour is described by ALT

and PAR compositions (Roscoe, 1998; Welch and Bakkers, 1992). A buffered data channel is described as a CSP process with an ALT, a PAR, and an additional CSP channel (Roscoe, 1998). Furthermore, a deadlock-free program with rendezvous data channels will be deadlock-free with buffered data channels. The reverse may not be true (Roscoe, 1998).

Call channels and barriers are not buffered in CT since they require strict rendezvous. Solving the priority inversion problem with call channels and barriers is not as easy as with data channels. The process architecture should be designed such that it is free from priority conflicts. Method calls that do not return data can be buffered, but this is of no use when methods must be served in a strict sequence. Processes that participate in the barrier should have equal priorities.

5.5 Scheduling of communication primitives

To give some insight and trust in the scheduling policy of the CSP-based synchronous communication model, the scheduling policy on data channels, call channels, and barriers are described in this section.

5.5.1 Scheduling of data channels

Hoare (1974; 1985) suggests that the order of scheduling of processes on a channel should be fair according to a *first-come-first-served* policy. Figure 5-4 illustrates an example of fair scheduling on a channel in steps from ❶ to ❹ in an equally-prioritized parallel relationship. Ignore the unequally-prioritized operator on the compositional interrelationship in the figure for now, which will be discussed later.

In Figure 5-4a, the writer process is first to write on the channel *c* and it gets blocked until the reader comes along to read the data from the channel. After communication the channel schedules the writer process first. This is similar for Figure 5-4b where the reader is first to reclaim the

channel and continues before the writer. For each channel communication two context-switches are performed.

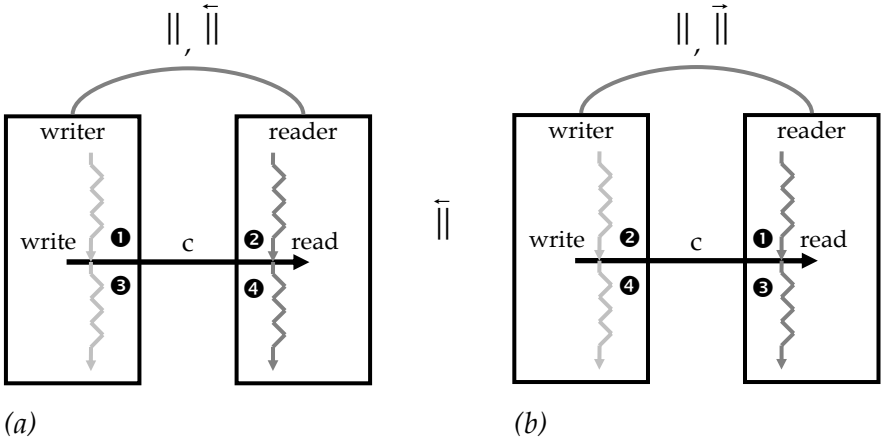


Figure 5-4 *Rendezvous and first-come-first-served scheduling:*
(a) writer first.
(b) reader first.

In systems with frequent channel communications and small computations, the total amount of context-switch time can consume a significant amount of processor time. A control system, where control loops continuously input and output on internal and external channels, is such a class of system. The performance can be significantly improved by a scheduling policy that eliminates one context-switch per channel communication, while preserving its reactivity. This scheduling policy is based on a *last-come-first-served* policy as illustrated in Figure 5-5a and Figure 5-5b, which has been adopted in CT. The improvement relates to the entire program and not to individual processes.

In Figure 5-5a the writer process is first to write on the channel and it gets blocked until the reader comes along to read the data from the channel. After communication the channel lets the current thread of control continue; thus the reader process will be scheduled first and the writer process will be scheduled at a later time. See the steps ❶ to ❹. This is similar for Figure 5-5b where the reader process is first and the writer continues before the reader. For each channel communication one context-switch is performed. Cyclic processes will alternatively read and write on channels which do not defect the reactive behaviour of the total

architecture. In periodical control processes this is always the case. For example, if the processes in Figure 5-5 are cyclic executing and repeatedly read or write on the channel then the behaviour alternates between Figure 5-5a and Figure 5-5b.

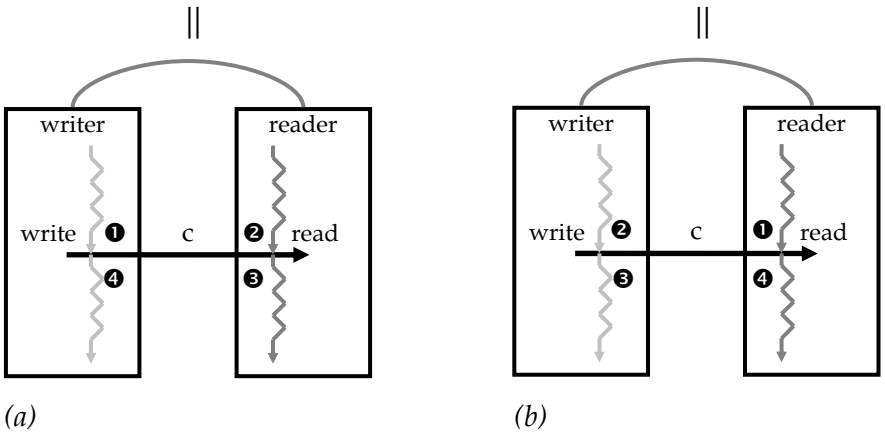


Figure 5-5 *Rendezvous and last-come-first-served scheduling:*
(a) writer first,
(b) reader first.

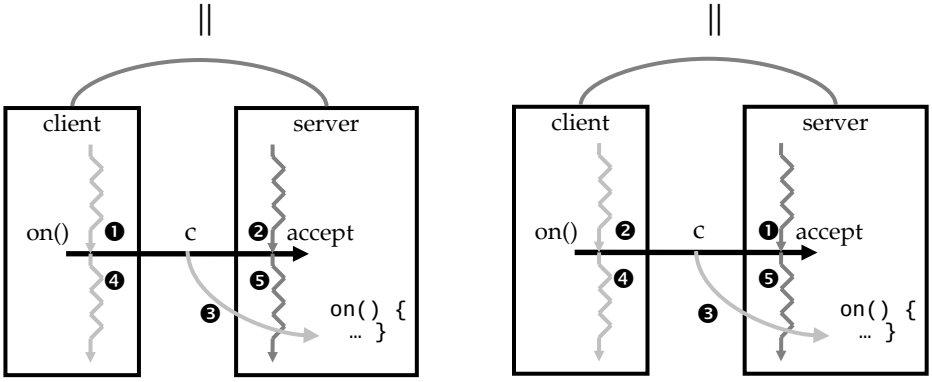
In case the writer and reader processes have different priorities, the scheduling policy falls back to the policy as described in Figure 5-4. In this case, consider the unequally-prioritized relationships on the compositional interrelationships.

Shared data channels apply the first-come-first-served policy between multiple writers and readers of equal priorities and it applies *highest-priority-first* policy for multiple writers and readers of unequal priorities.

5.5.2 Scheduling of call channels

The scheduling policy of a client-server relationship with equal priorities is illustrated in steps from ❶ to ❺ in Figure 5-6. No matter which process accesses the call channel first, after communication the client (or caller) continues before the server, we call this the *caller-first* policy. The caller-first policy is an optimal solution in a run-to-completion execution framework, like in programs written in Java or C++. Hence, step ❸ is

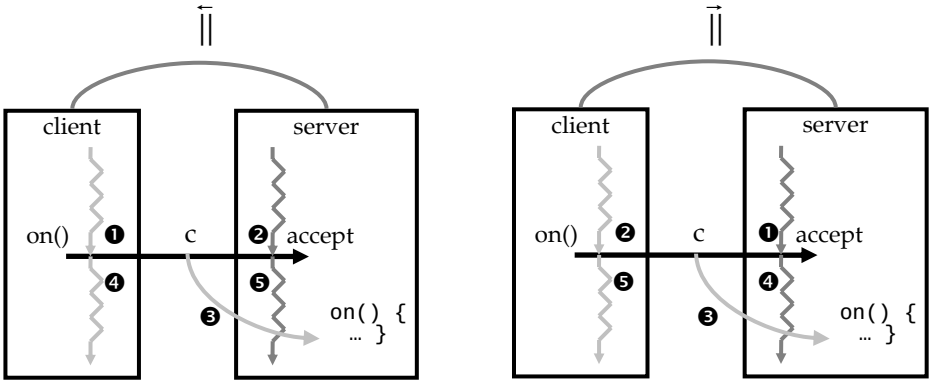
always performed by the caller's thread of control. Although, the policies of data channels and call channels differ, but the rendezvous concept remains equal.



(a) *(b)*

Figure 5-6 *Rendezvous and caller-first scheduling:*
(a) client first,
(b) server first.

The scheduling policy of a client-server relationship with equal priorities is illustrated in steps from ❶ to ❺ in Figure 5-7.



(a) *(b)*

Figure 5-7 *Rendezvous and highest-priority--first scheduling:*
(a) client first,
(b) server first.

As with shared data channels, a first-come-first-served queuing policy between multiple clients and servers with equal priorities is applied for

shared call channel. Once a client and server are committed to communication on a shared call channel, the caller-first policy applies. Multiple clients and servers are scheduled with unequal priorities are scheduled with the highest-priority-first policy.

5.5.3 Scheduling of barriers

The scheduling of processes that participate in barrier synchronization is a combination of a last-come-first-served policy and a first-come-first-served policy. The last-come-first-served policy applies for the last process that participates in the barrier synchronization. The thread of the last process invokes the parallel process at a lower layer. See the process with parallel sub-processes *A*, *B*, *C* and *D* in Figure 5-8 (step ❺).

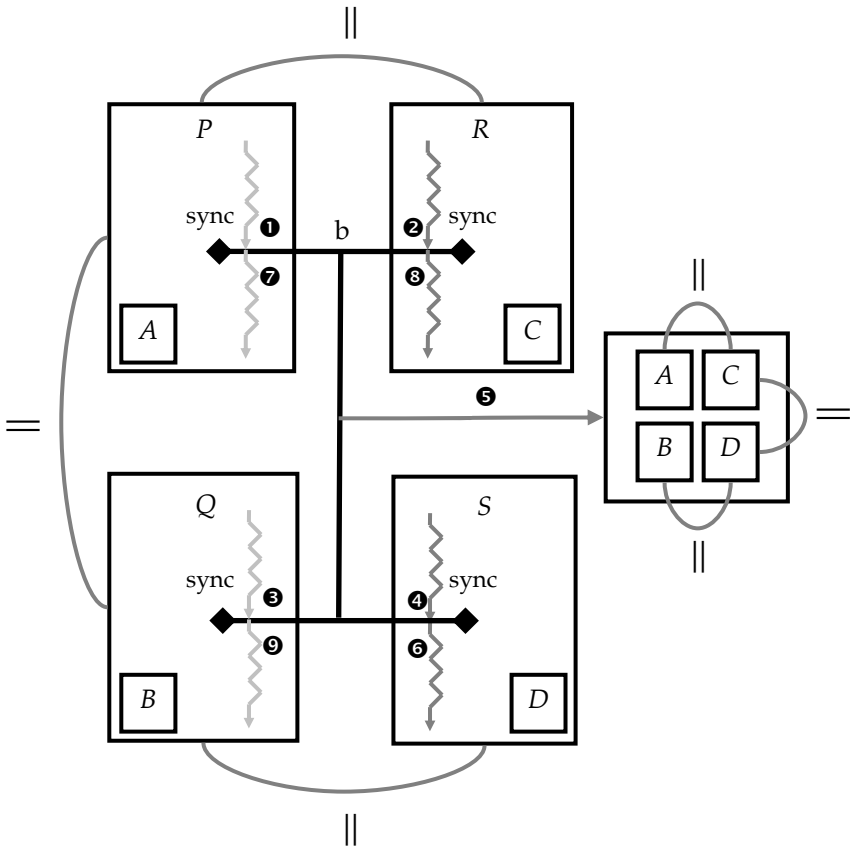


Figure 5-8 *Rendezvous of a barrier synchronization primitive.*

Each sub-process is provided by a participating process respectively *P*, *Q*, *R*, and *S*. After the lower-layer process is performed, the first-come-first-server policy is applied that schedules the other three participating processes in that order. When the processes *P*, *Q*, *R*, and *S* are cyclic then the policy rotates.

The barrier scheduling policy is the same as for the PAR construct that is applied when it releases its parallel processes at termination. Hence, the PAR construct is also a barrier construct. The PRIPAR also performs a barrier.

5.6 Alting with notion of priority

The run-time environment of software is deterministic and therefore the software must specify the appropriate deterministic decisions. Decisions can be fair or unfair. A fair decision is made with respect to previous decisions the mechanism has made. An unfair decision can be any decision that is not a fair decision. In this section we present the ALT as a fair alternative construct and the PRIALT as an unfair alternative construct. This criterion is based on local priorities between guards.

In circumstance where surrounding priorities are involved, the alternative constructs must be fair and serve the alting process with highest priority first. This precaution guarantees that the priorities of guards do not cause priority inversion problems that likely decrease the overall performance of process architecture.

Two types of alting are discussed, namely resolute alting and preference alting. Resolute alting is known in occam and preference alting is an improved approach that is proposed in this thesis. This section will show that preference alting is superior to resolute alting. Preference alting has been adopted in CT.

5.6.1 Resolute alting versus preference alting

In occam the ALT and PRIALT constructs are identical in that they share the same PRIALT implementation. The ALT and PRIALT implementations are unfair choice constructs, i.e. declining priorities are assigned to the guards. The ALT should have been a fair choice construct as suggested by Roscoe (1987). Listing 5-3 illustrates this fair choice construct based on a PRIALT with the use of conditional input guards. This is what the occam's ALT should have been, but unfortunately this fair alting implementation is difficult to realize on the transputer. In this thesis, the ALT refers to Listing 5-3 so that the ALT represents a fair choice construct and the PRIALT an unfair choice construct.

```
SEQ
  PRIALT
    (i ≤ 0) & g0
      j = 0
    (i ≤ 1) & g1
      j = 1
    ...
    (i ≤ n-2) & gn-2
      j = n-2
    gn-1
      j = n-1
    (i > 0) & g0
      j = 0
    (i > 1) & g1
      j = 1
    ...
    (i > n-2) & gn-1
      j = n-2
```

```
CASE j
  0
    P0
  1
    P1
  ...
  n-2
    Pn-2
  n-1
    Pn-1
i = (i+1) mod n
```

Listing 5-3 *Fair alternative construct with conditional input guards.*

Here, g_i is a guard and the CASE performs branches to the process P_i. Index *i* is the priority parameter used by the preference rule that is expressed by the conditional guards. The conditions provide a fair priority ordering among the guards. The fairness criteria used is that the guards are cyclic prioritized with the guard chosen last time getting lowest priority next time. This will be interpreted as if the guards under

the ALT have equal priority. This solution guarantees that no guard can be activated twice while there is another one waiting. A slightly simpler solution of a fair ALT is given by Lau and Shea (1988) using occam 2, which is basically the same as the one above.

The prioritized choice of the fair ALT as described by Roscoe (1987) and the PRIALT as described by Lawrence (1998) are isolated to the alternative process. This implies that the decision is not completely externally influences. The implementations of the ALT and priority-ordering of the PRIALT are basically cyclically (non-busy) polling mechanisms that test the readiness of each guard in a cyclic fashion (Barrett et al., 1988). These decision mechanisms are focused on the local priorities of its guards and not on the priorities of its alting processes. We call this type of alting *resolute alting*. In this section we will refer to the fair ALT as described by Roscoe and not to the occam ALT which is equal to the PRIALT.

An unfortunate mapping between the local priorities of the guarded processes in a PRIALT construct and the priorities of the alting processes in a PRIPAR construct can cause a priority mismatch (Burns, 1987; 1990). This mismatch results in a performance penalty. In circumstances where the priority of a process is changing (e.g. due to the use of deadline-driven scheduling or priority inheritance), the use of a static mapping would no longer be adequate. In this case, even the fairness of the ALT can become unfair when the wrong choice has been made and a lower-priority client process is served before a higher-priority client process. For real-time applications these problems can have a significant burden on the deadlines. Therefore resolute alting is not optimal for real-time software. A solution is to adapt the decision mechanism in such a way that it uses *preference priorities* of its guards. Preference priorities are locally set in the alternative process and they adapt to the surrounding priorities of the alting processes. Important is that the priorities of the alting processes should dominate over the priorities of the guards. Burns (1987; 1990) calls this type of alting *preference alting*.

The choice of resolute alting is not completely externally influenced and this is in contradiction to the fact that they represent the external choice operator in CSP; meaning that the choice can be externally influenced.

The problem of resolute alting will be illustrates in the next examples in Listing 5-4a-d. Preference alting is explained as a solution that is suitable for real-time systems.

```
PAR                    PAR                    PRIPAR                 PRIPAR
  c2!a      -- P1        c2!a      -- P1        c2!a      -- P1        c2!a      -- P1
  c1!b      -- P2        c1!b      -- P2        c1!b      -- P2        c1!b      -- P2
  FOR 0 TO 1             FOR 0 TO 1             FOR 0 TO 1             FOR 0 TO 1
    ALT     -- P3          PRIALT  -- P3          ALT     -- P3          PRIALT  -- P3
      c1?x                   c1?x                   c1?x                   c1?x
        P(x)                   P(x)                   P(x)                   P(x)
      c2?y                   c2?y                   c2?y                   c2?y
        Q(y)                   Q(y)                   Q(y)                   Q(y)
```

(a) *(b)* *(c)* *(d)*

Listing 5-4 *Scheduling behaviour of alting;*
(a) PAR – ALT composition,
(b) PAR – PRIALT composition,
(c) PRIPAR – ALT composition,
(d) PRIPAR – PRIALT composition.

ALTing in the presence of a PAR.

Listing 5-4a illustrates an ALTernative process communicating with two alting processes in parallel. The `PAR` executes its processes in a cyclic fashion and starts with the first process in the list of processes. Process `P1` outputs on channel `c2`, process `P2` outputs on channel `c1`, and process `P3` alternates two times to serve both alting processes. The `PAR` starts with process `P1`. Due to the deterministic behaviour of the `PAR` we assume that `P1` and `P2` are waiting for communication when the `ALT` is executed.

The trace of communication in this example will be `<c1,c2>` and this sequence is determined by the cyclic selection mechanism of the ALT. The execution order of the guarded processes is $(c1?x \to P(x));(c2?y \to Q(y))$. The selection fairly alternates between the two guarded processes `P(x)` and `Q(y)`. This behaviour is exactly according to our expectation. This description applies for a resolute ALT and for a preference ALT since they behave equally under the PAR.

PRIALTing in the presence of a PAR

Listing 5-4b illustrates a PRIALTernative process communicating with two alting processes in parallel. The PAR starts with process P1. The trace of communication will be <c1,c2> and this sequence is forced by the selection mechanism of the PRIALT. The execution order of guarded processes is $(c1?x \rightarrow P(x));(c2?y \rightarrow Q(y))$. This behaviour is exactly according to our expectations. Note that the PRIALT under the PAR gets close to the behaviour of the ALT considering the random arrival times on which alting processes access the channels. Again, this description applies for a resolute ALT and for a preference ALT since they behave equally under the PAR.

ALTing in the presence of a PRIPAR

Listing 5-4c illustrates an ALTernative process communicating with two alting processes with different priorities. The PRIPAR executes its processes in a preemptive fashion and starts with the first process in the list of processes. The PRIPAR starts with process P1. With a resolute ALT, the trace of communication will be <c1,c2>, whereby P2 is served before P1. The resolute ALT determines the sequence of this trace. By looking at the urgencies of the alting processes P1 and P2, we would expect that process P1 should be served (read) before process P2, because P1 has more important things to do. The resolute ALT starts with checking the first guard and therefore it will serve P2. The resolute ALT will check the second guard first on the second run, this time P1 will be served, but then this can be too late for P1 to meet its deadline. The desired trace of communication should be <c2,c1>, whereby P1 is served before P2. This sequence must be forced by the PRIPAR. The preference ALT is exactly doing this. The preference ALT will determine the priority order of its guards based on the priority of the alting processes under a PRIPAR. Thus, in Listing 5-4c the priority ordering of the guards will be determined by the priorities of the alting processes. With preference alting, the trace will be <c2,c1>.

PRIALTing in the presence of a PRIPAR

Listing 5-4d illustrates a PRIALTernative process communicating with two alting processes with different priorities. With a resolute PRIALT the trace of communication will be <c1,c2>. This is not an optimal trace, because P1 has a higher priority than P2 and therefore P1 should be served before P2. The preference PRIALT will adapt its decision to the urgency of its surrounding client processes and therefore the trace of communication will be <c2,c1>. Thus, the sequence of the trace is primarily determined by the PRIPAR. This is the optimal trace that is desired. Remarkably, a resolute PRIALT is known as an unfair ALT, but the preference PRIALT adapts its behaviour to the priorities of the surrounding client processes and becomes fair.

5.6.2 Preference alting implementation

The mechanism of preference alting is briefly discussed in this section. Both the preference ALT and preference PRIALT are based on the same mechanism. The difference between these ALTs is that the ALT assigns equal priorities to its guards and the PRIALT assigns declining priorities to its guards; as they were resolute ALTs. These priorities are so-called preference priorities which are preferred by the alternative process. However, these priorities can be overruled by surrounding priorities that are specified by unequally-prioritized parallel relationships between the alting processes. Equally-prioritized parallel relationships do not overrule the priorities of the guards and thus the priorities of the guards are applied to the choice mechanism. The basic idea is the same as for a shared channel with multiple writers that are communicating with a single reader. A communication diagram is given in Figure 5-9a. We omit multiple readers for the moment.

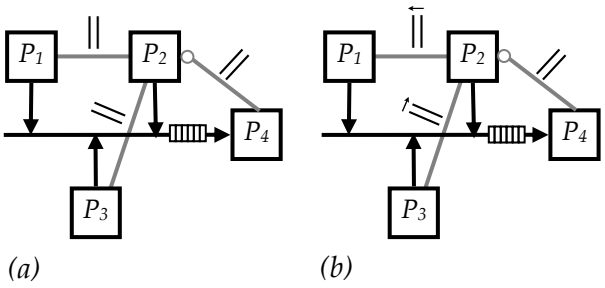


Figure 5-9a-b *Alting on an any-to-one channel.*

All processes P_1, P_2, P_3 and P_4 run in parallel. The figure shows an imaginary queue annotated to the channel. This queue stores the threads from the blocking writers on the channel. The thread on front (the right side) of the queue claims the channel and it is the first one to be released after communication with the reader. The order is determined by a prioritized sorting algorithm. After every release the queue is resorted.

The sorting algorithm implements two sorting policies. If the writers have the same priority then they will be queued in a *first-come-first-served* order, because this is fair with respect to their arrival time. Otherwise they are queued according to their relative priorities, namely *highest-priority-first*, because this is fair with respect to the specified priorities.

In Figure 5-9a, the processes P_1, P_2, P_3, and P_4 execute with equal priorities and assume that the arrival time on the channel is P_2, P_3 and P_1. Consequently, these processes are queued in that order; P_2 followed by P_3 and P_3 followed by P_1. Process P_4 will serve P_2.

In Figure 5-9b, the processes P_1, P_2, and P_3 execute with unequal priorities. Writer P_1 has the highest priority, writer P_3 has the lowest priority, and the priority of the writer P_2 is somewhere in the middle. The reader process P_4 runs in parallel to the writer processes. The sorting algorithm will store the process in order of priority whereby the highest priority process will be stored in front of the queue. Thus, if process P_3 is the first process waiting on the queue for the reader and when P_1 comes along, then process P_1 will take the place of P_3 and P_3 will be the next element in the queue. Thus, a process will be released in prioritized order. The reader will serve P_1 first since it has the highest priority of all

other waiting processes. The channel will adapt its alting queue to any changes in priorities of its alting processes. This is fair.

The reader side of the channel has a similar prioritized queue for multiple readers. Multiple readers accessing a shared channel simultaneously are rarely applied since the race condition between the readers can give unpredictable or undesired results. A design tool could warn the user for race conditions in the design.

The preference alternative process uses the same idea as with any-to-any channels. The previously described prioritized queuing policy for any-to-any channels is equivalent to the queuing policy for preference alting. The semantics, properties, and behaviour of any-to-any channels and preference alting are discussed in Appendix E. In CT, the guards are stored in the alting queue instead of the threads of alting processes. The alting queue is a linked-list of guards. Each guard can be chained to other guards and created a queue in which guards can easily be added, removed, or moved. Also, each guard in the queue has reference to its associated guarded process. The behaviour of the ALTs automatically adapts to every new situation. This includes dynamic scheduling. In circumstances whereby the priority of alting processes changes, while some guards are already on the alting queue, the selection may not be adequate. It is necessary to reorder alting queues on every change of unequally-prioritized parallel relationships.

5.7 Efficiency

The efficiency of CT depends on the optimization of all the queues. These are the waiting queues, ready queues, and alting queues. All these queues are prioritized in one way or the other.

5.7.1 Waiting queues

Semaphores and monitor constructs are commonly used to synchronize threads (Brinch-Hansen, 1972; Dijkstra, 1965; Hoare, 1974; Silberschatz

and Galvin, 1994). Semaphore and monitor constructs are integral part of the CSP constructs, channels, and barrier implementations. These synchronization constructs maintain one or more waiting queues. These waiting queues function as the alting queues in channels. These queues are efficiently implemented as a prioritized link-list mechanism performing both the first-come-first-served and highest-priority-first sorting policies.

5.7.2 Ready queues

The scheduler has a ready queue of processes waiting to be scheduled by the dispatcher. The prioritized sorting algorithm of the ready queue is based on recursive index-table technique (Labrosse, 1992). This technique takes two steps to point the right ready queue for storing a pointer to a process thread that is ready to execute. Each PRIPAR creates its own set of ready queues and every PAR assigns a process thread for each process to the ready queues of nearest surrounding PRIPAR construct. The idea of nesting is similar as with a nested alternative construct. Every PRIPAR creates a separate scheduler that is scheduled by its parent PRIPAR construct. The result is an advanced nested scheduler. Nesting of static and dynamic schedulers is possible and this is an interesting topic for further research. A PAR construct that has no parent PRIPAR becomes a PRIPAR with a PAR as its first and highest priority process in the program; otherwise the kernel is not setup and no threads can be scheduled.

5.7.3 Alting queues

The prioritized alternative implementation maintains a separate alting queue as a linked-list of guards. The queuing mechanism supports input guards, output guards, call guards, accept guards, timeout guards, skip guards, and nested ALTs. The implementation is reasonably efficient for a number of reasons:

- The entire implementation is divided in simplified objects. Most of the algorithm is performed by the alting queuing object, which performs sorting of a linked-list of guards in prioritized order. The guard objects allow nesting of other guards and implements a few simplified recursive methods. The ALT construct is a guard itself and inherits the implementation of the guard. The channels also carry out bits of the implementation only when the channel is part of a guard and only when a process reads/accepts or writes/calls on a channel.
- Each PRIALT holds an alting queue. The ALT uses the alting queue of a surrounding PRIALT. Therefore, the overhead of sorting of the alting queue scales with the sum of nested PRIALTs. Exceptionally, if there is no surrounding PRIALT then the root ALT will always create a root alting queue.
- When comparing the CT implementation with the code of the transputer-based implementation then we can conclude that:
    - The worst case of the sorting algorithm for each PRIALT is the same worse case as for the algorithm of the transputer-based implementation.
    - Adding guards to the queue, removing guards from the queue, and moving guards in the queue as a result of sorting, are the additional overheads compared to the algorithm of the resolute ALTs. These queue manipulations are just a few pointer assignments.

Last but not least, the fairness that can be achieved by preference alting is expected to have a greater effect on the performance than the latency of sorting. This statement should be studied in future research.

5.8 Output guards

As a consequence of the queuing mechanism, the alternative constructs are flexible enough to support output guards. In this research output guards are investigated since CSP supports them. Output guards are discussed in this section and can be used to simplify a design.

5.8.1 Alting disagreement

Jones (1987) describes that if each process tries to communicate by a conditional communication then they must both make the same decision about whether they want to communicate. Any attempt to make the decision independently at each process is likely to lead to a disagreement, especially for 'truly' concurrent or physically separated processors. This is called the *alting disagreement problem*. For example, Listing 5-5 illustrates a scenario with two communicating alting processes (performing conditional communication at each end of a channel) that will never commit in communication.

```
PAR
  ALT
    chan!x            -- write x value to chan
      P()             -- perform process P
    ...
  ALT
    chan?y            -- read y from chan
      Q(y)            -- perform process Q
    ...
```

Listing 5-5 *Alternative disagreement.*

A solution is imposing restrictions on the way in which conditional communications can legally be used in programs. The restriction adopted in occam is to ensure that no pair of conditional communications ever meet. Whenever a pair of communications matches, at least one is guaranteed to be unconditional. Jones shows that the restriction of eliminating output guards and allowing input guards is sufficient; programming without input guards is less natural than programming without output guards. This means that in each pair of communicating processes there is an output, which is necessarily unconditional. He illustrates that each output guard can be replaced by a communication pattern with an input guard. Listing 5-6 shows an example of three processes in parallel, i.e. `Process1`, `Process2`, and `Process3`. `Process1` is the alternative process and `Process2` and `Process3` are the alting processes.

```
PAR
  ALT                  -- Process1
    chan1!x            -- output guard
      P()
    chan2?y
      Q(y)
  SEQ                  -- Process2
    chan1?z
    S(z)
  SEQ                  -- Process3
    chan2!w
    T()
```

Listing 5-6 *Example with an output guard.*

In occam, the output guard (`chan1!x`) is forbidden and therefore this example must be transformed into Listing 5-7. An additional request channel is required to trigger the guard. The associated alting process `Process2` must agree with the protocol of first outputting a request on `chanx` and then reading the object from `chan1`.

```
PAR
  ALT                  -- Process1
    chanx?request      -- request first
      SEQ
        chan1!x
        P()
    chan2?y
      Q()
  SEQ                  -- Process2
    chanx!true         -- perform request
    chan1?z
    S(z)
  SEQ                  -- Process3
    chan2!w
    T()
```

Listing 5-7 *Example with only input guards.*

This workaround calls for an additional channel and an expansion of the communication protocol. In legal circumstances, an output guard may simplify the design and the result is likely to be faster than applying a workaround with an input guard. Instead of imposing restrictions, as suggested by Jones, another solution is to allow output guards when it

suites best. The above mentioned alting disagreement problem should be detected by tools before run-time and results in an error message.

The workaround is a source for an extra priority inversion problem when the alternative process has a lower priority than its alting processes. Listing 5-8 shows the example of Listing 5-7 with the alting process executing at a higher priority than the alternative process. Process `Process4` executes at an intermediate priority and it could preempt `Process3`. We assume that `Process3` gets enough time to perform since the whole system should obey the real-time requirements.

```
PRIPAR
  PAR
    SEQ                -- Process2
      chanx!true       -- perform request
      chan1?z
      S(z)
    SEQ                -- Process3
      chan2!w
      T()
  SEQ                  -- Process4
    ...
  ALT                  -- Process1
    chanx?request      -- request first
      SEQ
        chan1!x
        P()
    chan2?y
      Q()
```

Listing 5-8 *Priority inversion problem with alting and input guard.*

This example suffers from a double priority inversion problem on the channel input and on the channel output. Buffering `chanx` may solve the first priority inversion problem, but due to preemption between `chanx?request` and `chan1!x`, the buffer in `chan1` will be empty and it will block `Process2`. A super-sampling buffer could be useful, but then the request signal has no useful function.

Listing 5-9 illustrates a single priority inversion problem with the use of an output guard. Channel `chan1` could contain a super-sampling buffer to solve the priority inversion problem.

```
PRIPAR
  PAR
    SEQ                -- Process2
      chan1?z
      S(z)
    SEQ                -- Process3
      chan2!w
      T()
  SEQ                  -- Process4
    ...other process with intermediate priority
  ALT                  -- Process1
    chan1!x
      P()
    chan2?y
      Q()
```

Listing 5-9 *Priority inversion problem with alting and output guard.*

A buffered channel `chan1` makes the guard (i.e. `chan1!x`) initially true and `P()` may be selected even when the higher-priority process did not read from the channel. The guard could always be true and this could be an unwanted behaviour. Adequate buffer synchronization can prevent that the alternative construct never synchronizes on an output guard. For example, a super-sampling buffered data channel should make its output guard true after it was read at least once by the alting process; otherwise the guard should be false since the previous value was not consumed.

5.8.2 Alting agreement

Output guards come with limitations in design and implementation and they should be applied carefully. The graphical modelling language as described in Chapter 3 can protect the user from the alting disagreement problem as described in Section 5.8.1. The design tool could select the right type of buffering for channels to solve priority inversion problems.

With the current alting queuing mechanism, it is foreseen that the implementation can be extended in such a way that it can solve the alting disagreement problem for internal data channels and call channels. For external data channel communication this is more complicated and requires additional handshaking over the hardware link. This extension

has not been implemented since CT is kept compact and efficient. Certainly, this is a subject for further research.

5.8.3 Model checking and priorities

CSP abstracts away from priorities and the model-checker FDR (2004) cannot perform a performance analysis based on priorities. The PRI has no meaning in CSP. FDR can still be used to determine if the program is deadlock-free simply by removing the PRI; i.e. $\overset{-}{[\!|} \rightarrow [\!|$. Priorities makes choices determinism and it reduces event traces to particular sequences of events which represent the best reactivity or responsiveness. A performance analysis tool can check these event traces whether an optimal sequence of events is achieved, or whether the program suffers from starvation or other performance problems.

5.9 Conclusions

CT implements a task scheduling mechanism, which makes the library suitable for embedded real-time software. The efficiency of scheduling is determined by the design of the process architecture. The scheduling policy is composed by the compositional relationships.

Data channels can be buffered in circumstances where a rendezvous data channel is a source for a priority conflict. Sub-sampling or super-sampling buffered data channels can solve the problem in an elegant way. Buffered call channels and buffered barriers are not supported. Priority conflicts must be solved by correcting the design by buddy processes that unblock any higher-priority processes.

Communication between multiple readers and/or writers via an any-to-any channel has equivalence with alternative constructs. The fair scheduling policy that applies to any-to-any channels should also apply to the alternative process. Not surprisingly, the queuing implementations of the ALT have strong similarities with the queuing implementation of the any-to-any channel. The PRIALT is a specialization of the ALT

providing a particular unfairness between competitive alting processes, i.e. having equally-prioritized parallel relationships.

Preference alting contributes to a better performance than resolute alting. Preference alting allows priorities to propagate over events. This way, alternative constructs can make efficient decisions, which are influenced by the surrounding priorities between alting processes. Important is that any inadequate mapping between an unequally-prioritized parallel construct (PRIPAR) and an unequally-prioritized alternative construct (PRIALT) is corrected by the preference alting. Preference alting and preference priorities provide the ability of dynamic scheduling. The ability of preference priorities allows a CSP-based program to schedule a static process architecture in a dynamic way in order to achieve optimal performance. The notion of preference priorities will determine time-critical paths of event handling processes. By observing those paths of event traces and event handling processes one can reason about the length and deadlines of those paths. Timing analysis has not been described and may require further research.

The implementation of the preference ALT constructs in CT supports output guards. Output guards can be used to simplify a design. One should keep the alting disagreement problem in mind to ensured safety.

CHAPTER 6

CSP concepts applied to control systems

6.1 Introduction

The applicability of the proposed methodology for control applications and embedded computer systems is illustrated in this chapter. Several applications are discussed to which CSP diagrams, CTC or CTC++ was applied. It is shown that this methodology offers a concurrency paradigm that has the ability to manage complexities in control software.

Two low-cost DSP-based embedded computer systems are discussed in Section 6.2, to which CTC and link drivers were applied. In Section 6.3, a test bed is briefly discussed for which the application, written in CTC++, is portable between platforms. CT for the PC architecture is discussed in Section 6.4. It is applied to two mechatronic systems: ARTY and JIWY. ARTY is discussed in Section 6.5 and JIWY is discussed in Section 6.6. Conclusions are drawn in Section 6.7.

6.2 20-Controller

There are ways to create transputer-like embedded computer systems based on heterogeneous CPUs (other than transputers). These systems benefit from the CSP paradigm. For this research, two different low-cost

embedded computer boards were used: one equipped with OS-links (transputers class links) and one equipped with a CAN bus (common field bus in industry). This initiative was called *20-Controller*. The suffix "20" is the abbreviation for "Twente" (refers to the University of Twente) and "Controller" denotes that the board was made to perform a variety of control applications. These embedded computer systems were programmed with CTC or CTC++ and these systems inherit the scalability and distribution of transputer technology.

OS-links

A low-cost processor board, based on the Texas Instruments TMS320F240 low-cost 16-bit fixed-point DSP, has been developed for educational purposes and demonstrations (Lahpor, 1998). The board was specially developed to be able to distribute a concurrent controller application over multiple 20-Controllers using external links. The concept was based on a transputer-based architecture of heterogeneous processors. See Figure 6-1.

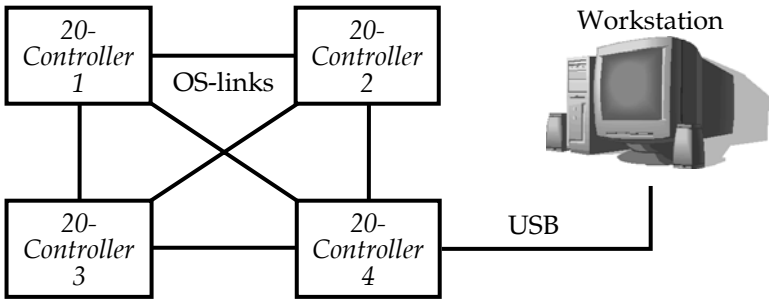


Figure 6-1 *Network of 20-Controllers via transputer links.*

Each 20-Controller has 3 OS-links (i.e. transputer links according to the IEEE 1355 standard) implemented on an FPGA. Each OS-link establishes a reliable connection with another 20-Controller. OS-links represent rendezvous channels in hardware. Channel communication via OS-links is deterministic and can guarantee hard real-time requirements. Unfortunately, only one 20-Controller was built and thus the OS-links were not used. The 20-Controller is shown in Figure 6-2. An additional USB link was intended for configuring and monitoring the board via a

workstation. For this feature, a footprint larger than the one implemtented on the prototype board is required.

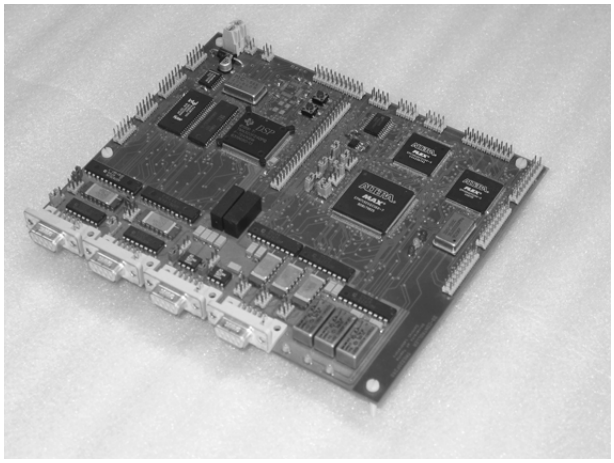


Figure 6-2 *The 20-Controller prototype based on a TMS320F240 DSP.*

The processor is optimized for digital motor/motion control applications and has some I/O-functionality integrated on chip for this purpose. 20-Controller is equipped with external devices to increase its applicability for the larger variety of motor control applications, which are necessary for the student practicum.

CTC was ported to the TMS320F240 and for almost every device on the 20-Controller a link driver was made (van Drunen, 2000). Link drivers were plugged into data channels so that processes can communicate with the hardware via data channels. The methodology provided guidance, which is imposed by the CSP concepts and CT for developing embedded real-time software in a sound and systematic way. The distinction between processes, channels, compositional constructs, and link drivers helped a great deal in separating concerns and simplifying the code structures and the documentation. Once the link drivers were created, one could entirely focus on the implementation of the application rather than on the platform-specific technical difficulties or thread synchronization. The methodology managed complexities by separating concerns and simplification. This simplification decreased the development time of the software without being an expert in programming embedded systems.

CAN bus

The laboratory of Control Engineering was sponsored with ADSP-21992 processor boards by Analog Devices. See Figure 6-3. These embedded processor boards are equipped with a CAN bus (Bosch, 2003; ESD, 2003) which is commonly used as a field bus in industry.

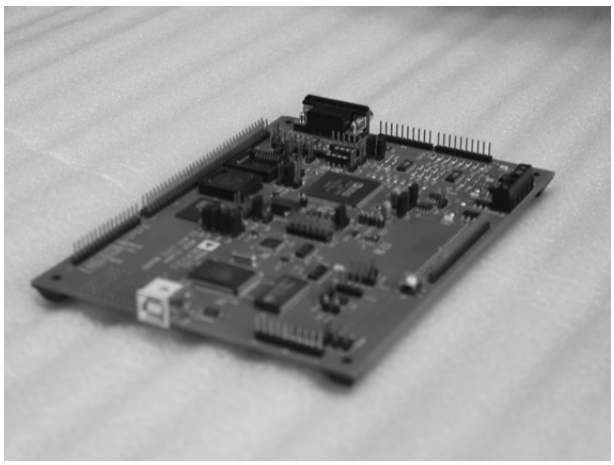


Figure 6-3 *The 20-Controller prototype based on a ADSP-21992 DSP*

The CAN network is used to distribute control applications on different embedded processor boards. An example is depicted in Figure 6-4. CTC and later CTC++ have been ported to the ADSP-21992 on 160 MHz (Orlic et al., 2003). CAN link drivers have been developed which implement channels via the CAN bus.

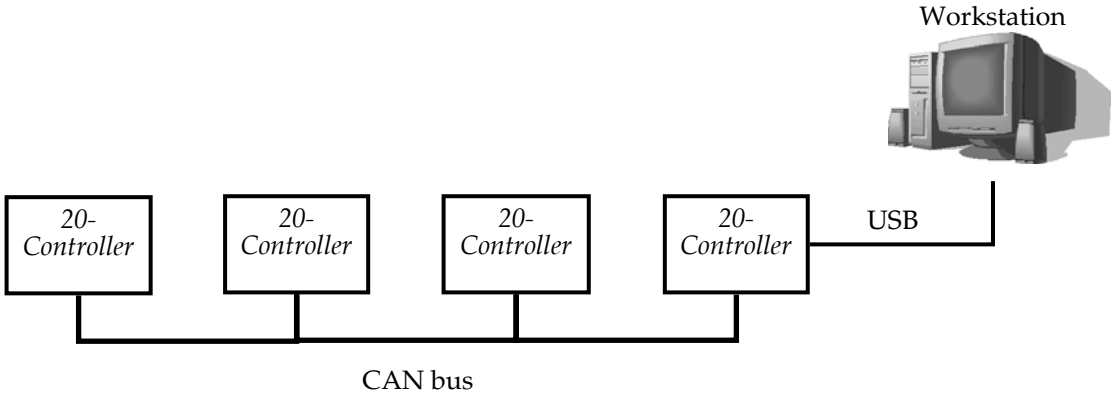


Figure 6-4 *Network of 20-Controllers via a CAN bus.*

6.3 MIMO-OFDM test bed

At the laboratory of Systems and Signals, CTC++ was ported to the TMS320C6711 DSP architecture. CTC++ was used to implement a CSP-based processing architecture for a flexible Multiple Input Multiple Output Orthogonal Frequency Division Multiplexing (MIMO-OFDM) test bed (Cronie et al., 2003). High-priority processes had to perform hard real-time tasks (taking advantage of data streaming for which the TMS320C6711 is optimized) and low-priority processes performed soft real-time communication with the user. Multiple external channels (link drivers) were developed that allowed the processes to communicate with the setup. The CSP paradigm guided this project without the requirement of extensive knowledge about multithreading and more importantly this project was accomplished the first time right.

The TMS320C6711 ran on 166 MHz which appeared too slow for the job and therefore a PC with a 2.4 GHz Intel Pentium PC was used. The application that ran on the TMS320C6711 also ran on the PC. Only the link drivers for the TMS320C6711 had to be replaced by link drivers for the PC I/O cards.

This application shows several benefits of using CTC++, such as

- one is freed from programming threads,
- the application is highly portable between platforms,
- guidance and development speedup,
- the application was done the first time right.

6.4 Laboratory PC and embedded PC

PC's with special I/O cards are often used as laboratory equipment connected to laboratory set-ups. Such a PC is called a *laboratory PC*. The Control Engineering Laboratory uses laboratory PC's for controller design and controlling the laboratory set-ups. A PC that is embedded in a machine is called an *embedded PC*.

The development of CTC++ for the PC platform resulted in the support of the following four targets:

- bare processor without file system support
- DOS
- Linux
- Real-Time Linux (RTAI)

CTC++ programs made for Laboratory PC's run almost unchanged on embedded PC's, and visa versa. It may only require a different set of link drivers which are declared in the top network builder. The processor-specific methods of these targets are very similar. Therefore, the API of CTC++ allows a concurrent program to migrate to any of the previously mentioned targets without changing the process architecture.

GNU compiler tools (GNU, 1996) have been used for all four targets. CTC++ can run in a single thread from the CPU or threads borrowed from an operating system, such as Linux or RTAI. CTC++ does not require an operating system if no file system or any other operating system resource is required. Due to the embedded scheduler, the context switch times were equal on DOS and on Linux. The channel communication time is 1 μs on a 266 Mhz Pentium PC.

6.5 ARTY, an autonomous robot

Student projects in 1998-1999 resulted in an autonomous robot, called *ARTY*. ARTY was developed as a mobile robot that can autonomously drive to a destination (Balkema et al., 1999; Beneder, 1998). See a picture of ARTY in Figure 6-5.

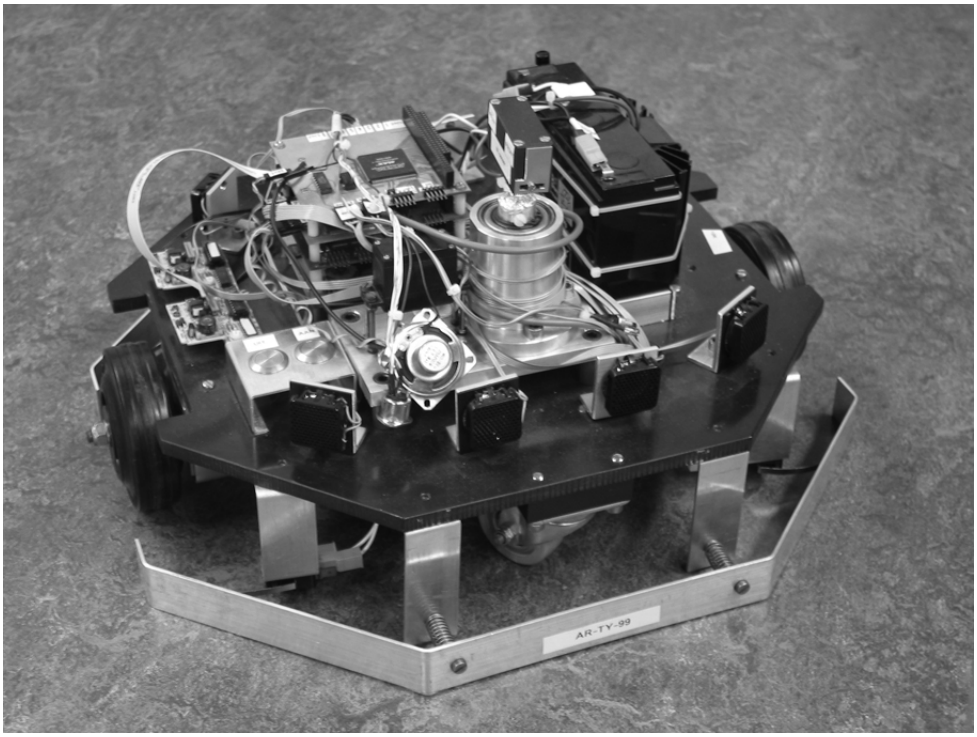


Figure 6-5 *Photo of ARTY.*

ARTY takes its own decisions on the basis of its destination and the obstacles it encounters. ARTY knows the distances between obstacles by using ultrasonic sensors and it knows its location via surrounding beacons. See Figure 6-6.

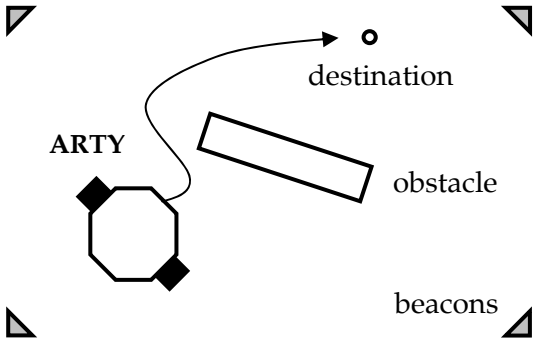


Figure 6-6 *Behaviour of ARTY.*

ARTY must be able to drive 3 to 4 hours on one battery cell. Low power consumption was an important design goal. The maximum speed of ARTY is about 13 cm/second.

ARTY contains the following parts:

- PC/104 board (486 CPU with 4 Mb onboard memory)
- Two navigation wheels for moving forward, backward, or turning
- Two supporting wheels to keep the frame horizontal
- Eight Polaroid ultrasonic sensors for measuring distance with surrounding objects
- Two bumpers for detecting obstacles that were missed by the ultrasonic sensors
- One beacon sensor for detecting its position between infrared beacons
- Control hardware programmed on an Altera Programmable Logic Device (PLD)
- 6 Volts battery

The first software was agent-oriented without a proper understanding of concurrency. The software resulted in a sequential execution framework of agents in C++ without multithreading. At the time, multithreading was considered complicated. The agent-based method by van Breemen (2001) provided a framework for structuring the problem domain towards an agent-based design of solutions. The divide and conquer approach resulted in a structured agent-based software architecture whereby each agent is responsible for a sub-task. Concurrency, which was natural in hardware and natural in the agent-based concepts, vanishes in a sequential software framework. Timing, priorities, interrupt handling, and processor utilization are problems that were implemented in an ad-hoc manner and became sources of complexities. Especially when it comes to concurrent event handling and preemption, these hurdles caused discontinuities between the design and the implementation. Van Breemen (2001) recommended that the CTC++

library would have given a good foundation to create a truly concurrent agent-based framework. As a follow-up, the proposed methodology and CTC++ were applied to ARTY and resulted in a concurrent control application, which has the task of controlling the two motors (Engelen, 2004; Modderkolk, 2003) independently from each other. The design and implementation of the process architecture using CSP diagrams and CTC++ are described in the next sections. The agent-based method has been omitted here since it did not yet comprehend concurrency in a systematic way.

6.5.1 Motor controller description

The task description of the motor controller is shaped by the appearance of the physical hardware and the way it should service the artificial intelligence of ARTY. The hardware structure of a single motor control system is depicted in Figure 6-7. The hardware structure is the same for the other motor control system.

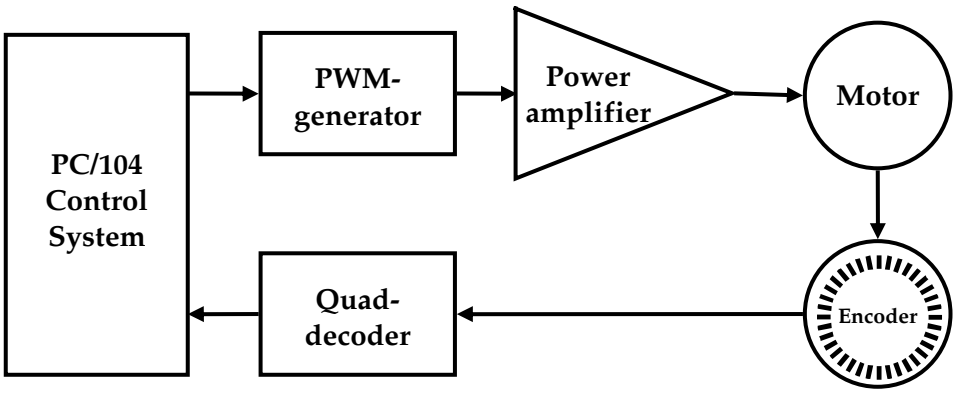


Figure 6-7 *Single motor loop controller.*

The control application comprises two loop control processes and one sequence control process. Each motor is controlled by an independent control loop, which has the duty of keeping the motor at a desired speed. The desired speed is provided by the sequence control process. The sequence control process concerns the artificial intelligence of ARTY and it can be replaced by any sophisticated process. The concurrent nature of

the loop control processes and the sequence loop process should allow them to operate independently and eventually at different frequencies.

6.5.2 Process architecture

The process architecture of the motor control part of ARTY is derived from the task description (specification) in Section 6.5.1. The process architecture is designed using CSP diagrams. Figure 6-8 shows the communication diagram of the control application. This diagram is the top CSP diagram and is called the context diagram. This was discussed in Section 2.2.1.

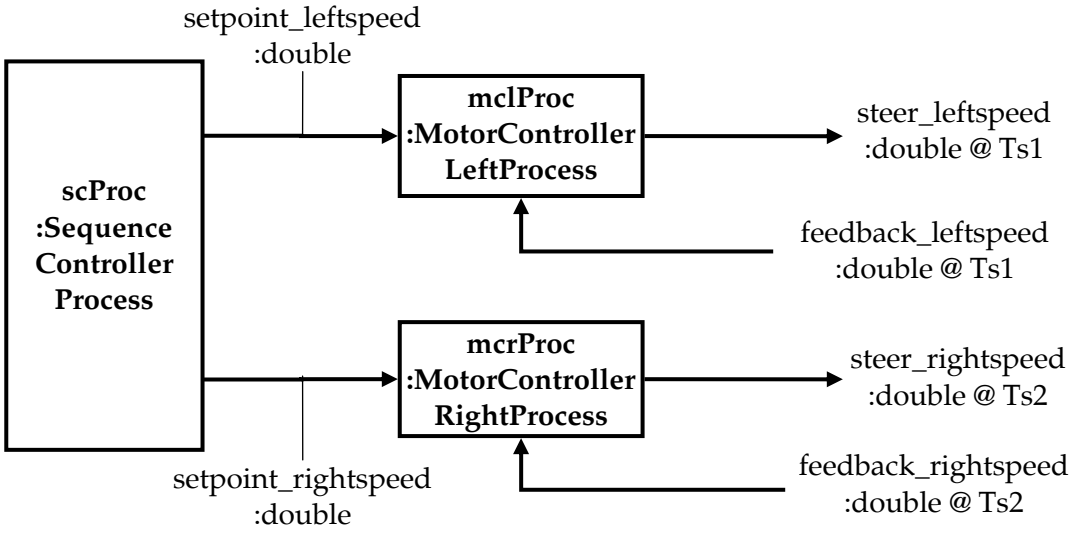


Figure 6-8 *Communication diagram of motor control part.*

The process class `MotorControllerLeftProcess` describes the left loop controller and the process class `MotorControllerRightProcess` describes the right loop controller. Process class `SequenceControllerProcess` specifies the sequence control process which commands the loop controllers to perform a specified speed. The channels that concern sampling and actuation are specified with sampling intervals T_{s1} and T_{s2}. In this application, the intervals for both loop controllers are equal, T_{s1}=T_{s2}. See suffix "@ Ts1" and "@ Ts2" in Figure 6-8.

The process class `SequenceControllerProcess` describes a sporadic process without a periodic interval. It implements an algorithm that allows ARTY to drive a simple path. This process can be substituted by a more intelligent process (Balkema et al., 1999). This process could very well operate on sampling intervals $T_{s3} > T_{s1}, T_{s2}$. Such a change in the software does not affect the processes in the process architecture.

Figure 6-9 depicts the compositional relationships between the processes. In this application, the motor controllers perform at a higher frequency than the sequence controller. Processes that are executing at different frequencies can be prioritized using a rate-monotonic priority scheme. This rate-monotonic priority scheme is implemented by the unequally-prioritized parallel operator on the interrelationship between the sequence controller and the motor controllers.

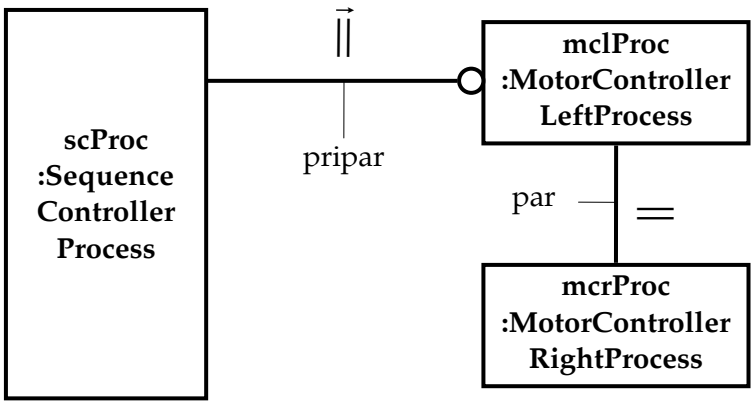


Figure 6-9 *Composition diagram of motor control part.*

Instead of decoupling the sequence controller and the motor controllers with super-sampling buffered data channels, we used an alting construct in the motor controller process from the sequence controller process.

The blueprint of the `MotorControllerLeftProcess` is given by the communication diagram and the composition diagrams in Figure 6-10 and Figure 6-11 respectively. The blueprint of `MotorControllerRightProcess` is similar and therefore omitted.

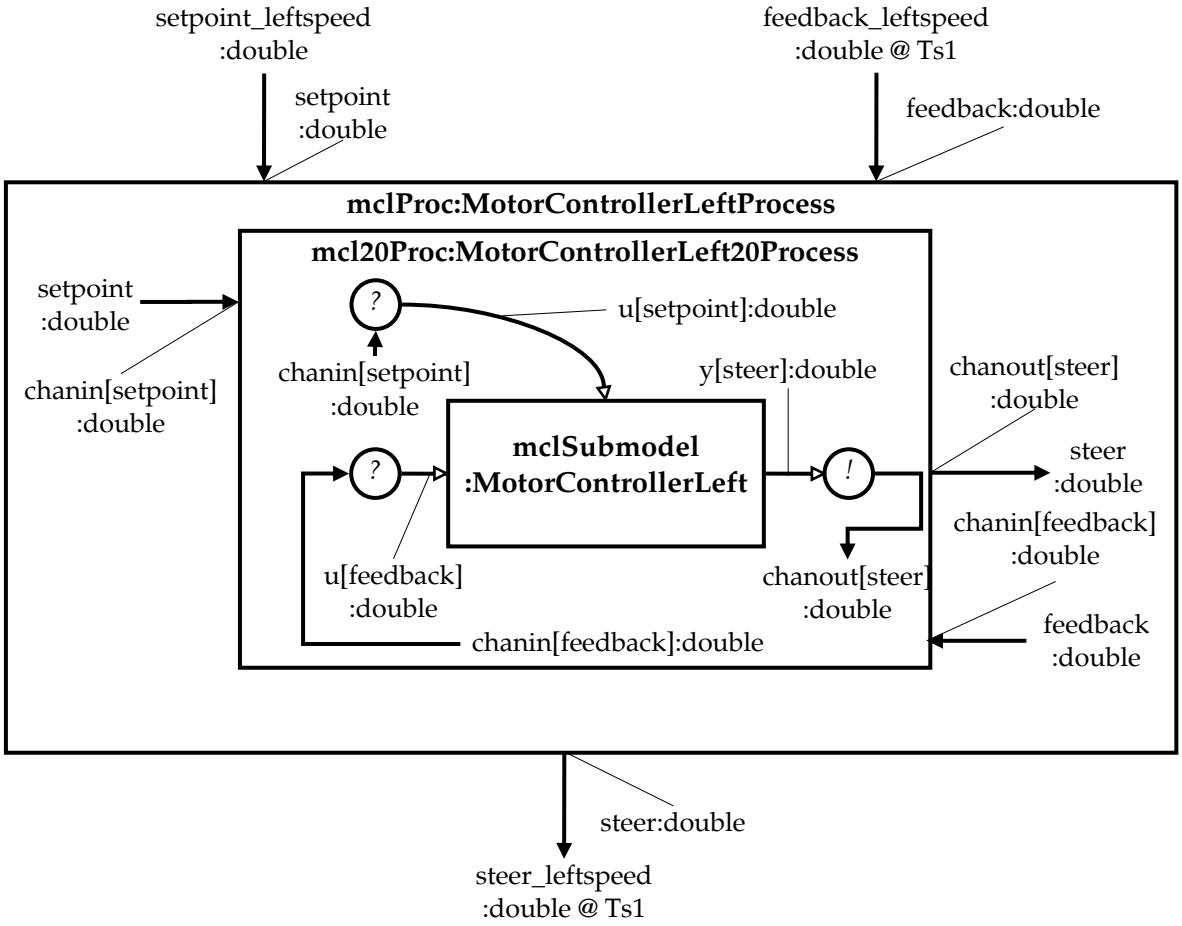


Figure 6-10 *Communication diagram of MotorControllerLeftProcess.*

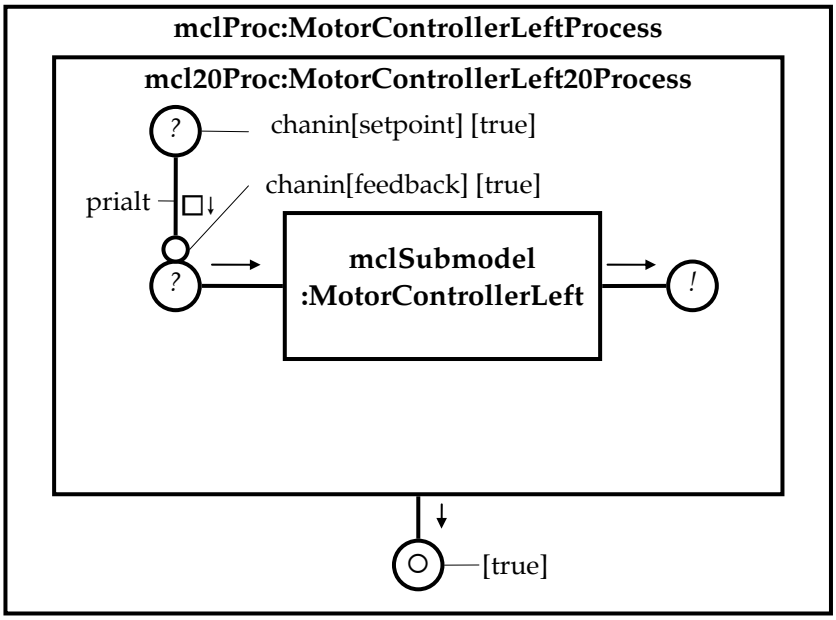


Figure 6-11 *Composition diagram of MotorControllerLeftProcess.*

The communication diagram and the composition diagram are viewed by a transparent hierarchical structure, which of course can be viewed in a layered hierarchical structure of black boxes. The code is hierarchically structured according to the process architecture.

The process `mclSubmodel` represents a leaf process that is developed in 20-sim. Code is generated by 20-sim with C++ templates. This is a process without channels and it solely performs the control laws using an array of input-variables `u[]`, an array of output-variables `y[]` and its internal state. The arrays of `u[]` and `y[]` are a property of 20-sim code generation. The names `setpoint`, `feedback`, and `steer` between square brackets refer to index numbers in the arrays. Process `mcl20Proc` is a so-called *20-process* which always terminates and is dedicated to invoke the methods of the `submodel` object. Process `mcl20Proc` specifies an array of input-channels `chanin[]` and an array of output-channels `chanout[]` on which the process communicates with its concurrent environment. Process `mclProc` performs process `mcl20Proc` in an infinite loop.

In the composition diagram one can see that the process is reactive on channels `chanin[setpoint]` and `chanin[feedback]`. The points of channel communication (see the ?- and !-processes) safely synchronize data transfer between parallel processes. While the process is waiting on the `feedback_leftspeed` (or `chanin[feedback]`) channel, the sequence controller can alter the setpoint via the `setpoint_leftspeed` (or `chanin[setpoint]`) channel. The prioritized alternative construct takes care of the proper event handling. The synchronization of the variable elements in `u[]` and `y[]` is based on the sequence operator and does not require a channel.

In process class `MotorControllerLeft20Process` the channels are indexed and in process class `MotorControllerLeftProcess` these channels are given sensible names. A process-interface of sensible port names is easier to use. Process class `MotorControllerLeftProcess` transforms the indexed channels to a process-interface with sensible names. The mapping between the sensible names and indexed channels was not automated since it would require changes to the 20-sim code-generator. With the template `%submodel%.info`, 20-sim creates a `MotorControllerLeft.info` text file which is named after the sub-model and contains the indexes and

corresponding variable names that are derived from the 20-sim sub-model. This file can be used to automate the mapping of the variables and channels on the corresponding indexed array elements.

6.5.3 Controller design

The process class `MotorControllerLeft` in Figure 6-10 and Figure 6-11 is modelled, simulated, and code-generated with 20-sim. The block diagram of `MotorControllerLeft` is given in Figure 6-12. The controller design of the right motor is identical. This figure is a direct copy-and-paste from the 20-sim model editor.

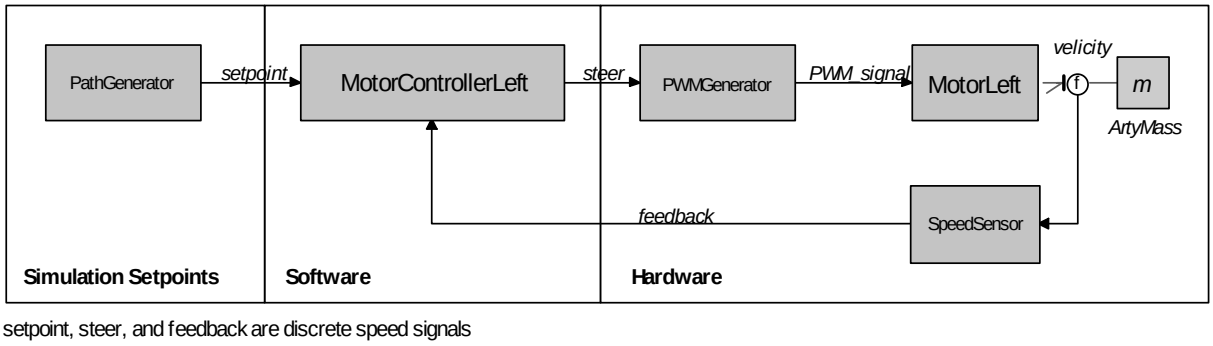


Figure 6-12 *Block diagram of MotorControllerLeft.*

The dynamics of the sub-models `PathGenerator`, `MotorControllerLeft`, `PWMGenerator`, and `SpeedSensor` were designed using block diagrams. The dynamics of sub-model `MotorLeft` and the mass was described using a bond graph. The CSP diagram for the right motor is similar but with slightly different parameters. All these sub-models are relevant to the total dynamics of the system and they are relevant to the design of the control laws. In fact, this behaviour should include the dynamics of the process architecture—the engagement of events. Most of this behaviour is part of the simulation framework of 20-sim, i.e. its behaviour corresponds to a sequence of inputs, calculation, and outputs. However, the choice between the joystick setpoint and the control loop feedback had to be artificially solved in 20-sim. This solution is based on a busy polling workaround, which complicated the sub-model `PathGenerator`.

An alternative construct (as shown in Figure 6-11) is a more elegant solution that makes the application truly reactive without the need for busy polling constructs. The `MotorControllerLeft` sub-model is code-generated and merged with the code of the software architecture as explained in Section 6.5.2.

The controller processes are implemented by a PID controller. The 8-bit resolution of the signals steer and feedback was determined by the creators of ARTY and fixed in hardware. The signals in this model are real numbers and the PID controller itself operates with real numbers. Quantization of these signals is performed by the `PWMGenerator` and `SpeedSensor` sub-models.

The `PWMGenerator` is responsible for translating a discrete `steer` value from the controller to a true PWM signal. This sub-model matches the real PWM device on ARTY. The steer signals are 8-bit values with range [-127, +127] which corresponds to the speed [-13, +13] cm/s.

The `SpeedSensor` sub-model converts and scales the velocity of the wheel (the actual speed of ARTY) to a discrete value of ticks per sample. This value is also an 8-bit value with range [-127, +127].

The sub-model `PathGenerator` generates a simple stream of steps from 0 to 120 or from 0 to -120.

6.5.4 Implementation

The implementation of the control software is the translation of the CSP diagrams to C++ using CTC++ and code from 20-sim. The listings that resulted from this translation are described in Appendix D.6.

Straightforward translation

All elements in a CSP diagram are declared by the parent process they describe. The type of the elements and their identifiers specify the type of declarations or the required code constructs. This allows for a straightforward translation from CSP diagrams to its C++ code using

CTC++. The translation was done manually, but this can be automated by a software design tool that supports CSP diagrams.

The controller sub-models in 20-sim are automatically translated to C++, using C++ templates, by 20-sim. This results in generated processes for each sub-model that perform the control laws. These processes require state manipulation to initiate their input and to retrieve their output.

Network building processes

A process that is responsible for constructing a network of processes is called a *network builder*. A program can contain a set of network builders. The network builder that describes the top CSP diagram or context diagram is called a *top network builder*. The context diagram is usually the part of the process architecture that connects the program with the real-world via hardware links. Therefore, the top network builder is usually hardware dependent and declares external channels. The external channels are further connected to sub-processes. Except for the top network builder, all other network builders are hardware independent. This makes a CSP-based program highly portable.

Figure 6-8 and 6-9 specify the context diagram of the motor part of ARTY. In C++, this top network builder is implemented in the `main()` method. The `main()` is the first method called upon to execute the program. Therefore, the `main()` method is equivalent to the constructor and the `run()` of a process. The implementation of the `main()` method and the `run()` methods of the sub-processes are described in Appendix D.6.

The top network builder is constructed and executed according to a uniform pattern, consisting of the following ingredients:

1. All top elements are declared, i.e. external channels, internal channels, and processes in the context diagram.

2. The top compositional construct is created, which is usually a PAR or PRIPAR. The declared processes are assigned to the constructs and connected via the channels.

3. The timing on channels is set up.
4. The `run()` method of the construct is invoked.
5. After the `run()` method terminates, all the declared objects and processes are deleted.
6. Now the `main()` method can safely terminate.

Since CTC and CTC++ are written in portable C/C++, the minimal requirements depend on the quality of the C/C++ compiler and linker. Compiler optimization can make the code fast, but also large. On the PC architecture, the CTC and CTC++ libraries indicate the following minimal requirements:

- 50K bytes of program memory,
- 4K bytes of data memory,
- simple register model of 16-bit registers,
- a 16-bit real-time timer.

These minimal requirements are based on code without debug information and optimized for space. These minimal requirements exclude the amount of memory that is eventually used by the application and link drivers.

6.5.5 Experiments

In open loop, ARTY could not perfectly drive straight forward or backwards. ARTY tended to go left or sometimes right. The condition of the battery was an important factor for this drift. The wheels reached their maximum speed in approximately 0.5 seconds on a steer signal from 0 to 120 (0-12.2 cm/s). In close-loop, the maximum speed on steps from 0 to 120 was reached in 0.25 seconds. ARTY drove nicely in a straight line and the condition of the battery no longer caused drift. The response of ARTY has become twice as fast.

A circle-test program showed good results. With the input parameters `speed`, `radius` and `direction`, ARTY obeys this command and drives circles with the correct speed, correct radius and in the right direction. See Figure 6-13. After many circles a small error was noticeable but acceptably small due to slip of the wheel with the floor. This slip was not taken into account when designing the controller.

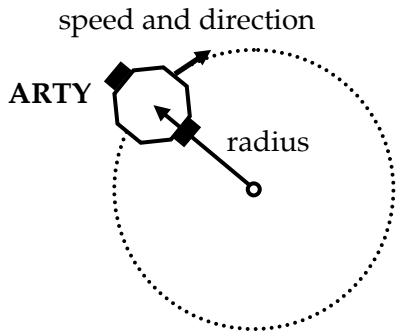


Figure 6-13 *Circle-test program.*

We took the opportunity to experiment with different frequencies for each loop controller, $T_{s1} \neq T_{s2}$. For example, the sampling interval of one loop controller was set to 10 Hz and the other to 100 Hz. Instead of controlling both motors on 100 Hz, processor time can be saved so that the real-time requirements of other processes can be met more easily. The experiment showed that these different sampling intervals did not change the behaviour of ARTY. The implementation of periodical events is an easy task with CTC++, which is orthogonal to the program structure and scales well on multiple frequency MIMO systems.

6.6 JIWY, a robotic servo system

A two-degree-of-freedom robotic servo system was reused for the purpose of applying a stepwise refinement trajectory for designing and implementing the control software. The servo system is called *JIWY*, see Figure 6-14. The construction contains two revolute joints that allow a mounted device, like a camera, to rotate on a horizontal axis and a vertical axis. JIWY could be used as a surveillance device, whereby

people can remotely view the room in which JIWY resides. Users can interact with JIWY via the internet. The images from the webcam can be remotely viewed and JIWY can be steered with a remote joystick or a remote keyboard. A prototype with a webcam is described in Smith (2002).

Figure 6-14 *Photo of JIWY servo system.*

The joints are equipped with DC motors and incremental encoders. JIWY is controlled by a PC equipped with a National Instruments I/O card (PCI-6024E, 2000), an analogue joystick, and supplied with a power-supply/amplifier/circuit box (I/O box) for driving the motors and converting sensor and actuator signals. The PC is connected to the internet via a LAN network. Details on JIWY can be found in Jovanovic et al (2002).

An overview of the hardware structure of one control loop is given in Figure 6-15.

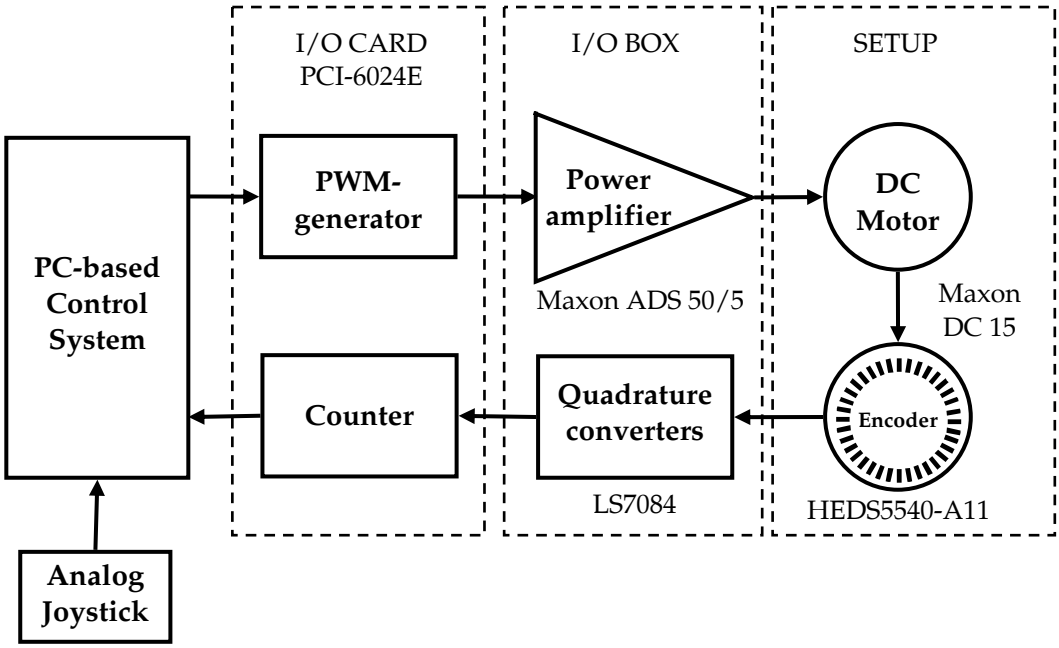


Figure 6-15 *Hardware structure of one motor control system.*

6.6.1 Motion control description

JIWY uses incremental encoders and therefore the centre position for each axis has to be determined by alignment. It is required that the outer boundaries of each axis are determined first in order to calculate the centre position of the axis. Each joint rotates to its end-stop, at which point the process terminates. It remembers the maximum value. The next process rotates the joint with constant velocity to the other end until it reaches the end-stop. Again this process remembers the maximum value. Subsequently, the main servo position motion controller takes over. It uses the two maximum values from the alignment processes to determine the centre position of the joint. The main controller can be stopped by the user by pressing a specific button on the joystick. After the main controller is stopped by the user, it is required that the joints return back to their centre position as a safe state. This process is called homing. After homing, the motors will be disabled.

6.6.2 Process architecture

The process architecture of JIWY has been designed using CSP diagrams. The components were derived from the description (specification) in Section 6.6.1. Some freedom of viewing CSP diagrams, with slightly different degrees of detail, is illustrated.

Modes of operation

The functional description, as described in Section 6.6.1, gives rise to four control modes per joint. Each mode identifies a separate process. These processes are depicted in Figure 6-16 and Figure 6-18.

Figure 6-16 shows the communication diagram representing the data-flow between these four processes. Each process has an input-channel (`feedback_horizontal`) that provides the angle of the joint and an output-channel (`control_horizontal`) that steers the motor. The main controller `motionControlH` receives the joystick set-points and joystick buttons from the channels `joystick_horizontal` and `joystick_buttons`.

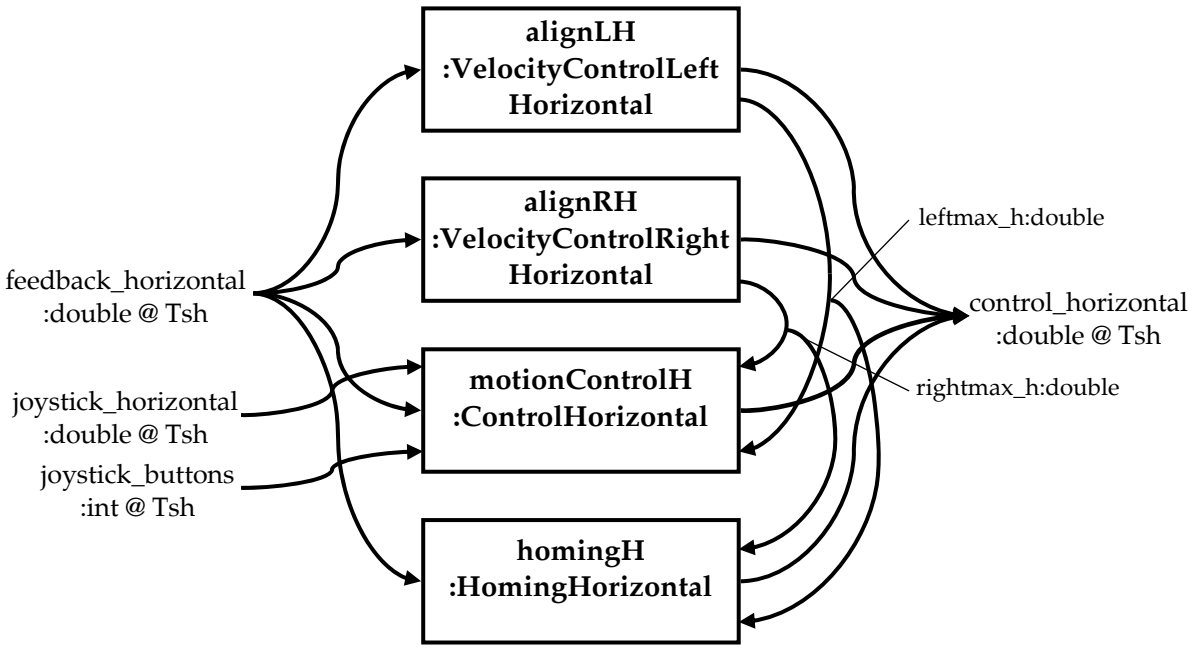


Figure 6-16 *Communication diagram of the horizontal loop controller.*

The names `feedback_horizontal`, `control_horizontal`, `joystick_horizontal`, and `joystick_buttons` identify process-interface elements or ports of the parent process. Here, the parent process is the main process or the top network builder. These names are used inside the parent process and are connected to communication relationships outside the process. Suffix "@ Tsh" specifies that we deal with a sampled data system, where the samples arrive at equidistant moments in time. Clearly, each sample must be processed before the next arrival. This suffix means that the external channels are triggered on sampling period T_{sh} for the horizontal control loop. T_{sv} is the sampling period for the vertical control loop. The CSP diagram for the vertical control loop is not shown here as it is similar to the horizontal control loop. This information is used to set up timing in the process architecture, see Section 6.6.4.

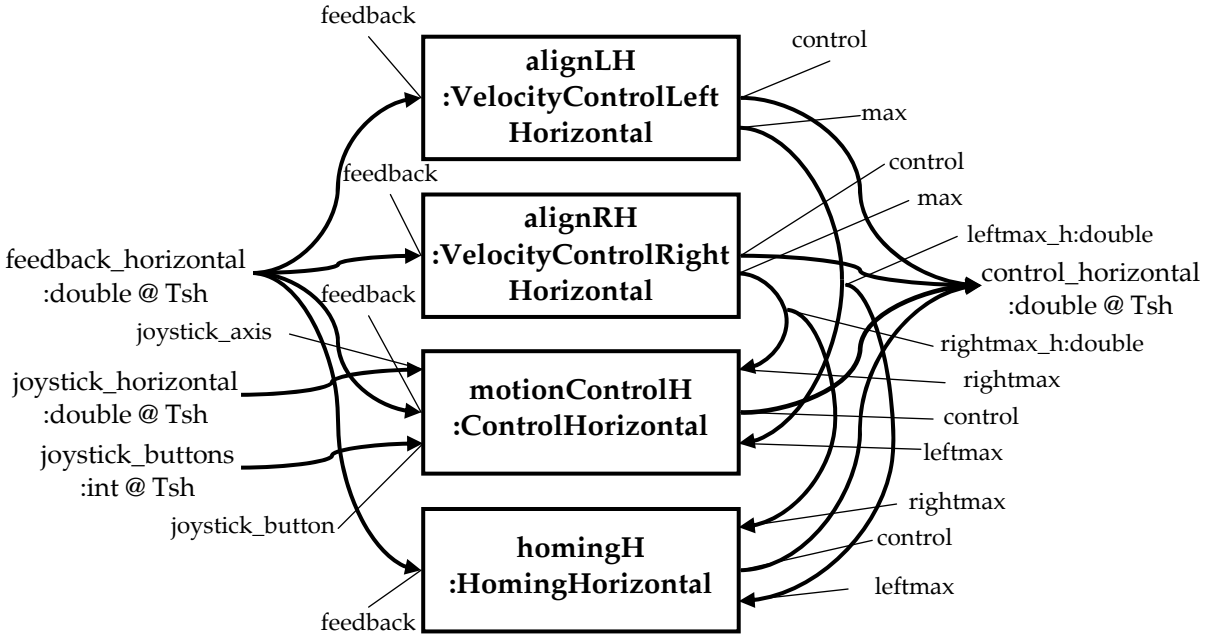


Figure 6-17 *Communication diagram with port detail.*

Figure 6-17 shows the necessary data-flow or dependencies between the inputs, outputs, and the processes. In order to determine the correctness of connectivity, ports labels can be shown next to the processes. This is illustrated in Figure 6-17. The diagram becomes more difficult to read for the user and for outsiders. Detail reduction by hiding port labels

improves the readability of the diagram. This illustration shows that the complexity or readability of CSP diagrams comes from the user style (or lay-out) and the amount of detail that is shown by a CSP diagram. The user should be able to toggle the visibility of port labels, i.e. switching between Figure 6-17 and Figure 6-16. Therefore, design tool support is inevitable.

The communication types of ports at each end of a communication relationship should be of the same type, otherwise they are incompatible and cannot be connected. The type indication `:double` or `:int` have been hidden from the ports in this example since they can be derived from the label-type of the arrow. Thus, derivatives can also be hidden in order to make CSP diagram readable.

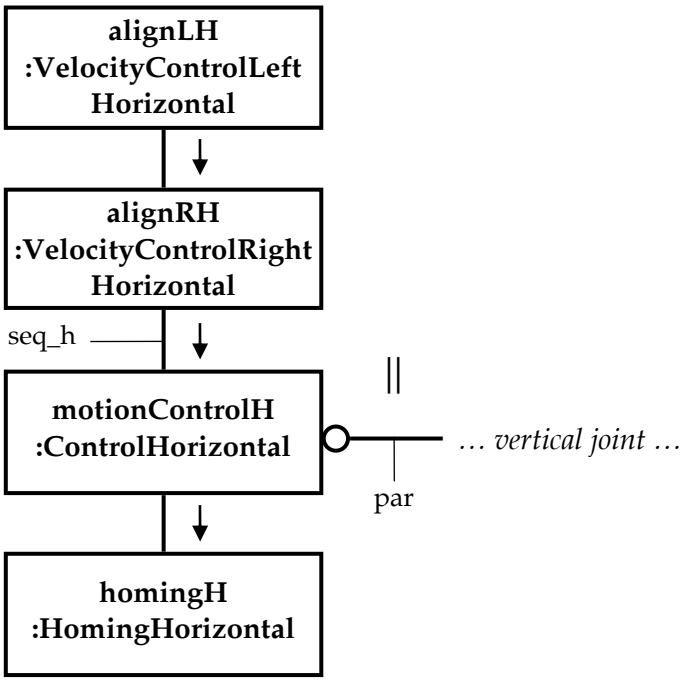


Figure 6-18 *Composition diagram of the horizontal loop controller.*

Figure 6-18 shows the compositional diagram which depicts the control-flow between these processes (modes). Identifiers for sequential relationships between processes and primitive processes are not really necessary. Here, we add one identifier `seq_h` which identifies the sequential construct. It can be found in the resulting code. Each joint

operates in parallel, hence there is a parallel interrelationship between the horizontal and vertical processes. The little circle at the end of the parallel relationship denotes a parenthesis, which determines a group of control processes for the horizontal joint. The parenthesis symbol forms one anonymous parent process containing child processes. This is similar for the vertical joint, which is not shown in Figure 6-18. The parameters for the horizontal joint are slightly different than the vertical joint. We will mainly restrict ourselves to the horizontal joint.

This composition diagram is like an abstract state-transition diagram, where the processes represent states of operation (modes) and their termination represent the transition between these states (modes).

Motion controller process

The `motionControlH` process in Figure 6-16 is the main controller that receives the set-points from the joystick. This process terminates when the stop button on the joystick is pressed. This process contains a 20-process `servoHorizontal` which computes the control law for each sample. See Figure 6-19 and Figure 6-20. The looping is specified by the loop process.

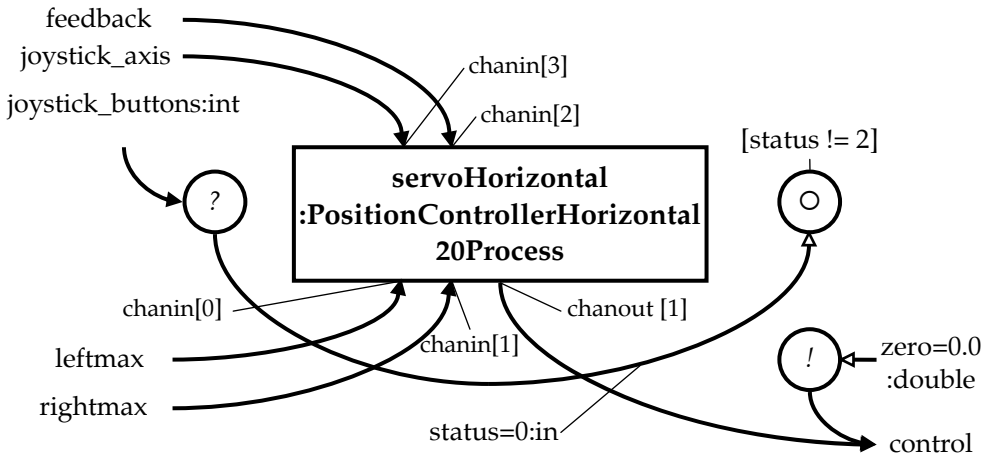


Figure 6-19 *Communication diagram of motionControlH.*

In Figure 6-19, communication relationships between the external ports and the 20-process `servoHorizontal` are specified. The 20-process ports

are rendered by indexed channel array names. This indexing rises from the sub-model; `chanin[i]` → `u[i]` and `y[j]` → `chanout[j]`.

Two local communication relationships, `status` and `zero`, have been added. `status` holds the status of the joystick buttons when it is pressed. The μ-process will continue repeating the alternative relationship $\overleftarrow{\square}$ until `status` is 2. `zero` is a constant value 0 in the communication diagram. Therefore these variables are declared and initialized with the default value in the code.

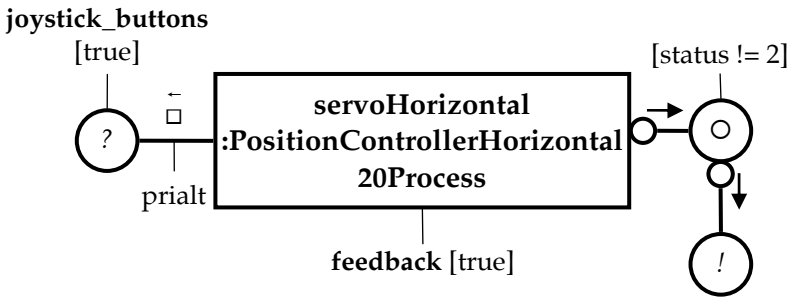


Figure 6-20 *Composition diagram of motionControlH.*

The alternative construct will wait until the channels `feedback` or `joystick_buttons` become ready. See Figure 6-20. The construct will select the controller when feedback is ready to communicate and when the joystick button is not pressed. If the joystick was pressed and feedback is not yet ready to communicate, then the joystick buttons will be read. If both the joystick button was pressed and the feedback is ready then reading the buttons will preceed the feedback. In case the stop button was pressed (`status != 2`), the loop controller terminates and a number zero will be sent to the actuator in order to release the steering of the joint. Successively, the entire process terminates and the joint resides in a safe state. Identifiers for sequential relationships between processes and primitive processes are not necessary.

Alignment controller process

Each alignment mode is based on a velocity controller. Process `alignLH` has been depicted in Figure 6-21 and Figure 6-22. This process contains a 20-process `vleftHorizontal` and a loop construct which repeats itself until

`stop` is true. The variable `stop` becomes true when the end-stop has been reached. This is specified by the 20-sim sub-model. Process `alignRH` is identical to `alignLH` except that the motor rotates right.

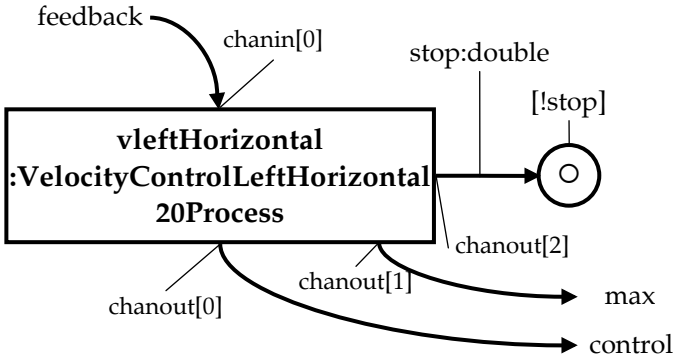


Figure 6-21 *Communication diagram of alignLH.*

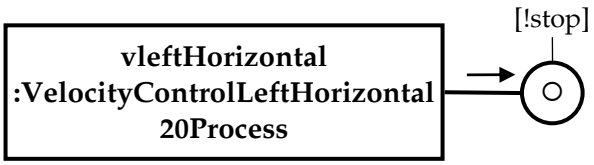


Figure 6-22 *Composition diagram of alignLH.*

Homing controller process

The homing controller uses the same position controller process as in the motion controller. This process requires a set-point that is set to the centre position, i.e. zero. See Figure 6-23 and Figure 6-24. For a change we hide the port names in Figure 6-23. The homing process stops when the joint resides in a sufficiently small neighbourhood of the centre position. This is measured within 10 sample periods.

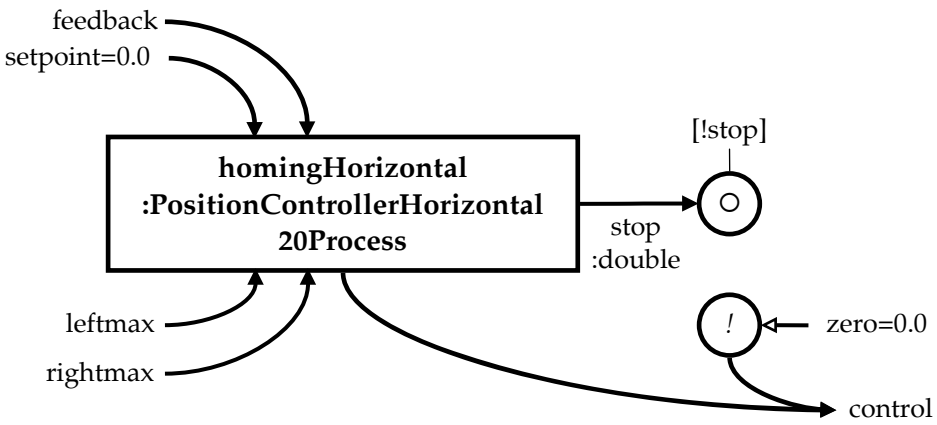


Figure 6-23 *Communication diagram of homingH.*

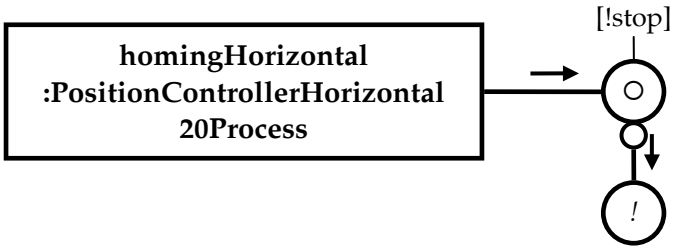


Figure 6-24 *Composition diagram of homingH.*

6.6.3 Controller design

Separate from CSP diagrams, simulate-able models in 20-sim have to be designed in order to determine the control laws. The 20-sim model comprises all relevant dynamics of the system and the controller itself. Simulations of the model can be found in (Lammertink, 2003).

Position controller

The position motion controller is the main operational mode of this servo system. The model renders the context of the closed-loop system. The context contains two servo-controlled axis and the joystick as a position

set-point generator. Figure 6-25 presents the top-level block diagram of this control mode for the horizontal joint. The diagram for the vertical joint is similar. The joystick input for the x-axis is simulated by some function as described in sub-model `JoystickHorizontal`. The `ioHorizontal` sub-model models and simulates the hardware input/output interfacing between the physical system and the control software. These sub-models allow for a more realistic dynamic behaviour, which one can expect from the real setup. The physical system is modelled by the sub-model `MotorHorizontal` using bond graphs.

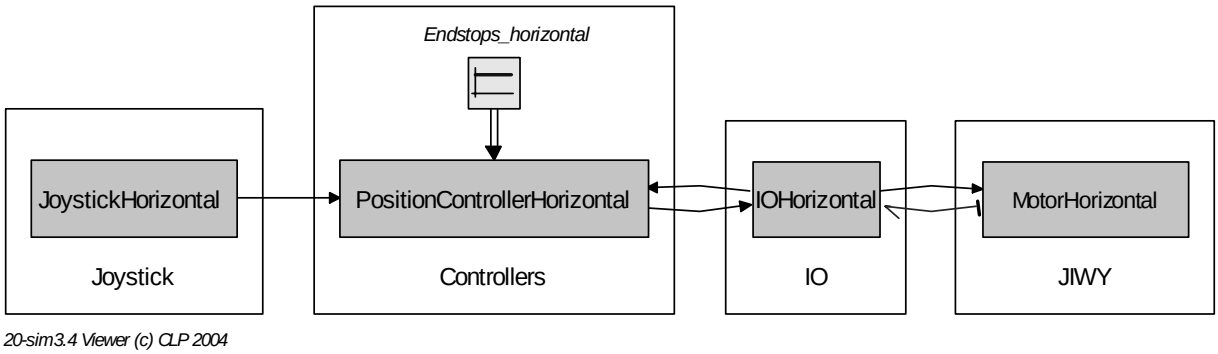


Figure 6-25 *Top-level model of position motion controller in 20-sim.*

The sub-model `PositionControllerHorizontal` is the software controller that will be code generated by 20-sim as 20-processes using C++ templates. The controller `PositionControllerHorizontal` is described using block diagrams in 20-sim. This is similar for `PositionControllerVertical`, which may differ in parameters.

Alignment controller

The alignment determines the maximum angles of the joints. The alignment is velocity controlled, whereby the end-stops must be detected while the joint moves slowly and with constant speed. The top-level block diagram of the alignment mode for the horizontal joint is given in Figure 6-26.

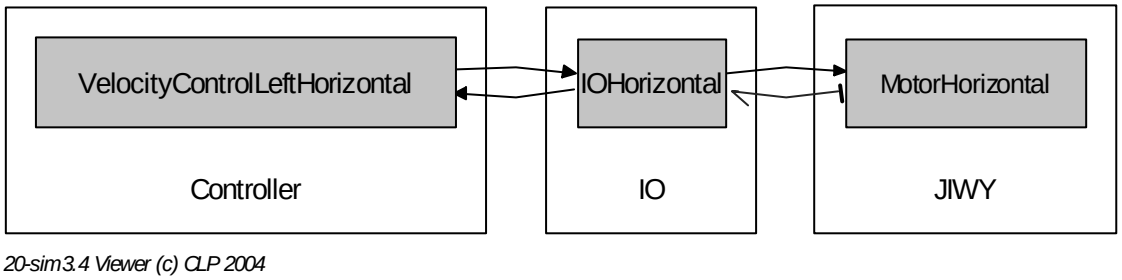


Figure 6-26 *Alignment mode for left rotation.*

The sub-model `VelocityContolLeftHorizontal` is the controller that will be code generated by 20-sim as 20-processes.

Homing controller

The homing controller `HomingControllerHorizontal` is almost identical to `PositionControllerHorizontal`. See also Figure 6-25. The only difference is that the joystick input is overwritten by value 0 and enforces the joint to move to the centre position. This is similar for homing controller `HomingControllerVertical`.

6.6.4 Implementation

The implementation of the top network builder and the sub-processes are described in Appendix D.7. The translation of CSP diagrams to C++ is similar as with ARTY in Appendix D.6.

Both top network builders from ARTY and JIWY show a common pattern of construction (template). Both templates are divided in the following parts, as shown in Listing 6-1.

```
void main() {
  //--- external channels declarations
  ...

  //--- internal channels declarations
  ...

  //--- processes declarations
  ...
```

```
  //--- compositional construct declaration
  ...

  //--- set timed events
  ...

  //--- run the process
  ...

  //--- delete all instances
  ...
}
```

Listing 6-1 *Template of a network builder.*

This assignment showed that the quality of the program was related to the process architecture and the quality of the link drivers. Test-runs simulated that the process architecture was correct. Link drivers for the National Instruments I/O card (PCI-6024E, 2000) were developed for Linux, RTAI, and DOS (Stephan, 2002). Once these link drivers were correct on all platforms, the program was portable between these platforms without changing the top network builder.

It was observed that students involved in these projects had modest knowledge and experience in programming. They appreciated the CSP concepts from which they understand the ins and outs of embedded system programming at an appropriate level of abstraction. The methodology eliminated programming pitfalls which increased the development speed. Above all, they appreciate that they can build embedded and real-time software by following the guidelines of the CSP concepts.

6.6.5 Tests

Once the CSP diagrams and the 20-sim model were completed, the generated controller software worked the first time right.

The sensitivity of detecting the end-stops had to be fine-tuned in the controller design in order to reduce the force on hitting the end-stops.

JIWY has been successfully used for demonstrations. It was observed that the dynamics of JIWY was close to the dynamics of the 20-sim models. To protect the motors, the current to the motors was limited to 1.5A.

The control software has been tested in DOS and in RTAI. RTAI and Linux offered the ability to use the webcam and internet services. Both platforms worked equally well.

6.7 Conclusions

The proposed methodology demonstrates a sound foundation for systematic designing and implementing control software on embedded control systems. It has been demonstrated that this methodology applies to laboratory setups and to DSP-based embedded computer systems. The graphical modelling language has been used to design the process architecture of the control software. The resulting CSP diagrams were essential in describing and understanding the desired behaviour of the software and hardware. Unfortunately no design tool was present yet to design CSP diagrams or to translate CSP diagrams to a code framework. The translation of CSP diagrams to code was manually done in a straightforward and systematic way which eliminated surprises and simplified the implementation. The code is documented with CSP diagrams.

The aspects of simplicity, portability, and generality that were mentioned in Section 4.2.4 have been demonstrated.

The simplicity, as a result of applying the occam's razor, resulted in a compact and coherent set of graphical notations, methods, and constructs. Redundancy has been minimized in this set without reducing flexibility. The proposed methodology comprises the necessary fundamental concepts for developing concurrent software. Simplicity comes from abstraction, semantical consistencies between design and code, and rapid prototyping through plug and play. This significantly decreased the development time of the software, because the user priory knows exactly where to put code in which processes or objects.

The portability of the applications comes from the channel concept and the small set of processor-specific methods. The channel concept makes the applications highly hardware independent. Each different platform requires a different set of link drivers. The processor-specific methods have been successfully ported to the following processors that were used by the laboratory of Control Engineering and the laboratory of Signals and Systems:

- Texas Instrument TMS320F240 DSP (TMS320F2000 series)
  (Texas Instruments, 1996)
- Texas Instrument TMS320C6711 DSP (TMS320C6000 series)
  (Texas Instruments, 1999)
- Analog Devices ADSP-21992 DSP (ADSP 2199x series)
  (Analog Devices, 2003)
- Intel i386 series microprocessors
  (Intel, 1996)

Generality is demonstrated by translating the CT object model to different programming languages, different operating systems, and to different CPUs. The methodology abstracts away from from low level technical issues; the low level technical issues are well-ordered by using CSP diagrams. The approach of designing and implementing the process architectures of ARTY and JIWY are very similar. Design patterns easily map on the underlying hardware. Also, 20-sim integrated nicely in this methodology. There were no engineering surprises that complicated the designs or implementions. An important observation was that the outcomes of these projects were predictable early in the design phase. CSP diagrams and CT offered the right paradigm to deal properly with sophisticated concurrent applications on embedded computer systems.

CHAPTER 7

Discussion

7.1 Conclusions

A methodology for building embedded real-time software for heterogeneous embedded control systems is developed. The proposed methodology comprehends a foundation based on CSP concepts, which deals with the technical "how to's" for building concurrent software. The foundation deals with common sources of complexities in programming concurrent software, such as multithreading, interrupt handling, exception handling, inter-processor communication, priority scheduling, reactivity, responsiveness, safe-guarding and fault-tolerance, etc. These technical issues are elevated to a high level of abstraction. The abstraction simplifies the design of embedded real-time applications and their implementation in software and it simplifies the mindset of the user. The proposed methodology allows the user to focus on the control application at hand rather than spending time on difficult and low level technical issues early in the development phase.

The proposed methodology comprises two ingredients:

1. A *graphical modelling language* is developed for creating concurrent designs, called *process architectures*. The language captures the CSP concepts by a graphical notation which one can use to specify, design, and to program process architectures for embedded real-time systems. The resulting designs are called *CSP diagrams*.

2. A *CSP object model* (CT) is developed that implements the CSP concepts using object-oriented techniques and can be implemented in object-oriented programming languages. The results are *CSP libraries* for Java, C and C++. Hence, this proposed methodology shows that the CSP concepts offer a process-oriented paradigm which marries well with object-orientation.

The graphical modelling language and the CT object model provide a foundation to specifying, designing, and implementing process architectures of control applications. This foundation is an important provision for building tool support. The CSP concepts bring about well-defined, distinct, and coherent concerns. This leads to abstraction, complexity reduction, and complexity absorption. The continuity and consistency of concurrency between the different phases in the development trajectory are guaranteed by model checking. The results of this methodology embrace what-you-see-is-what-you-get. Consequently, this methodology allows for rapid prototyping and round-trip engineering.

Process architectures capture hierarchies of processes, communication between processes, the role of processes, their specific execution order, and timing. The graphical modelling language divides the relationships between processes into communication relationships and composition relationships. This relationship-oriented design approach allows for scalable designs and different views. A CSP diagram consists of two distinct views, respectively the communication diagram and the composition diagram. Each diagram describes a different concurrency concern in the software design trajectory. The collaboration between both types of relationships provide valuable information about their compositions that is useful to determine design conflicts, such as deadlocks, livelocks, and priority inversion problems in a process architecture. This information can determine the exact type of communication (e.g. rendezvous, buffered, sub-sampling, and super-sampling) between processes necessary to solve design conflicts or to optimize the performance of the process architecture in a systematic way. Composition diagrams can also be traced for various design decisions, which may be in conflict with the specification or mind set of the user.

Thus, CSP diagrams incorporate guidance for the user to avoid design and coding errors.

The graphical modelling language does not prescribe styles for designing CSP diagrams. The user can design complex diagrams that are unreadable to others. Thus, the user is responsible for the readability of the diagrams.

The design process is guided by design rules, such as

- *Compositional analysis rules*—useful for analyzing compositional CSP constructs. Compositional analysis rules are used for
  - determining the operators on hidden interrelationships,
  - for writing ambiguous or unambiguous algebraic expressions,
  - and for detecting specification conflicts.
- *Reallocation rules*—rules for reallocating relationships with another possibly nearest process, while preserving the algebraic expression. This allows for a free topology of processes.
- *Balancing cycles*—technique that ensures a balanced cycle of correct parenthesizing indexes. A design must be consistent when viewed from different angles, i.e. reading a CSP diagram starting from different processes and possibly in different directions.

These rules offer analysis approaches that guarantee consistency and correctness, such as

- *Specification analysis*—finding specification conflicts whereby relationships are in contradiction in the design.
- *Deadlock analysis*—finding deadlock by searching for sequential conflicts between primitive communication processes.
- *Priority inversion analysis*—finding priority inversion problems by searching for priority conflicts between processes.

This research showed that the CSP concepts reach further than the good-old transputer technology. The CSP concepts provide theoretical and

pragmatic solutions to complex problems in embedded software engineering. This methodology comprises several enhancements which are essential in embedded real-time software. The enhancements are:

- *Shared channels*—data channels or call channels can be shared by multiple producers/ consumers and clients /servers.
- *Internal and external data channels*—internal data channels transfer data (i.e. primitive data types or objects) via shared memory and external data channels transfer data via hardware devices. A process cannot distinguish between an internal or external data channel.
- *Buffered data channels*—primitive data types or objects can be passed and temporarily stored in the channel via some FIFO, sub-sampling, or super-sampling queue.
- *Call channels*—high level channels for requesting methods on a server process.
- *Barrier*—performing a complex communication process as a layered process consisting of communicating processes via channels.
- *Notion of priorities*—support for preference priorities, propagation of priorities, and fair scheduling.
- *Improved parallel construct*—supports nested and compositional priorities and scheduling set up by the PAR and PRIPAR compositions.
- *Improved alternative constructs*—the decisions made by the ALT and PRIALT can be influenced by the priorities of alting processes (priority propagation imposed by the surrounding PAR/PRIPAR composition).
- *Exception handling*—escaping from exceptions and gathering exceptions to be handled.
- *Timing*—postponing events in untimed CSP models.

It has been demonstrated that this methodology works in practice. The use of CSP diagrams were essential in describing and understanding the desired behaviour of the software for the control systems ARTY and

JIWY in Chapter 6. A design tool was in development and unfortunately the tool was not mature enough for designing CSP diagrams or to translate CSP diagrams to a code framework. The translation of CSP diagrams to code was manually done in a straightforward and systematic way which eliminated surprises and simplified the implementation. A prototype design tool is under construction by Jovanivic (2001)

The aspects of simplicity, portability, and generality have been demonstrated. The simplicity comes from abstraction, semantically consistencies between CSP concepts and rapid prototyping through plug and play. The simplification decreased the development time of the software without being an expert in programming embedded systems. The portability of the applications comes from the channel model, which makes the applications highly hardware independent. Each different platform requires a different set of link drivers. Generality is demonstrated by translating the CT object model to different programming languages, to different operating systems, and to different CPUs.

7.2 Suggestions for future research

A software design tool is inevitable in order to truly benefit from CSP diagrams. The prototyped design tool that is currently in development can do a fraction of the possibilities that CSP diagrams can offer. The software design tool can generate concurrent frameworks for various software engineering tools. The integration or coupling of such a software design tool with other tools requires further investigation.

The graphical modelling language does not prescribe how process architectures must be designed. It is very well possible that the user models unreadable CSP diagrams. It shoulds be investigated in what way the software design tool can impose a systematic design process, which results in readable designs.

The model-checker FDR cannot check the real-time performance of process architectures, except for deadloack and livelock. For example, FDR does not support priorities and timing. Performance analysis should determine whether or not the process architecture suffers from starvation or other performance problems. Performance analysis takes priorities and the timing parameters into account. Performance analysis was not part of this research. A performance analysis tool should be investigated.

CSP diagrams marry well with block diagrams. Therefore, CSP diagrams should be used for designing hybrid systems. This requires research in the field of control theory for which CSP can describe the interaction between discrete and continuous-time systems. This is more promising than using traditional state diagrams, which tend to be only applicable for small control problems.

The exception construct in CSP diagrams and in CT showed to be useful for the applied control applications. However, the semantics of the exception construct should be reconsidered. It should be investigated if the semantics of the exception operator can get closer to the semantics of the interrupt operator. The implementation of such exception construct may not be as complicated as was suggested in this thesis.

The CT library has been tested by test programs. These tests showed appoved behaviour. However, this does not prove that everything is correct and perhaps something slipped one's mind. In order to gain trust in the implementation and semantics of the CSP libraries, it is required that the CSP libraries are model-checked using CSP. A professional tool is available, called the Failure-Divergence-Refinement tool (FDR, 2004). Model checking should prove that the semantics of the CT elements are conform to the theory.

The use of CSP diagrams could improve the concurrency model in the UML. Modelling concurrency in the UML is complex due to discontinuities between the different views. CSP diagrams could add a new view to the UML that integrates with the other views in the UML. This approach can guarantees safe and reliable multithreading without dealing with threads directly. This addition of CSP diagrams to the UML may result in real-time UML for embedded systems.

APPENDIX A

The CSP Language

A.1 Introduction

CSP stands for *Communicating Sequential Processes*, which was introduced by C.A.R. Hoare in 1978 (Hoare, 1978). The first textbook was published in 1985 (Hoare, 1985). Since then CSP, as a language, has significantly evolved. Roscoe (1998) updated CSP and published it in 1998. This version is referred to as CSP II. This version has been used in this thesis. CSP remains untimed because it abstracts away from notion of time. A timed CSP variant exists, which was developed by Schneider (2000). This extension to untimed CSP is not used in this thesis.

This appendix outlines the syntax and semantics of untimed CSP as it is now used. An illuminating and brief tutorial was written by Martin and Jassim (1997). This text has been used in this appendix to give an overview of CSP. A complete outline of the syntax and semantics of CSP can be found in (Roscoe, 1998; Schneider, 2000).

A.2 Evolving Theory

CSP encompasses the fundamental aspects of concurrency. From this point the theory is evolving with extreme caution for correctness. Extensions to untimed CSP exists which enhance the use of CSP for particular areas in software design. For example, a timed CSP variant

was developed, which allows for reasoning about timing behaviour (e.g. performance, timeouts, delays) in processes. Another variant is prioritized CSP which was developed by Lawrence (1998) which includes notion of priorities as was found in the occam programming language; i.e. the PRIALT and PRIPAR processes.

The textbooks by Nissanke (1997) and Schneider (2000) illustrate how CSP can be applied to real-time systems. In Maggee & Kramer (1999) CSP provides a systematic and practical approach to designing, analyzing and implementing concurrent programs in Java.

A.3 The CSP Language

Basic Syntax and Informal Description

The basic syntax of CSP is described by the following grammar.

$$
\begin{aligned}
Process :== \; & STOP \mid \\
& SKIP \mid \\
& event \rightarrow Process \mid \\
& Process ; Process \mid \\
& Process \,|[\, alph \,|\, alph \,]|\, Process \mid \\
& Process \,|||\, Process \mid \\
& Process \sqcap Process \mid \\
& Process \,\Box\, Process \mid \\
& Process \setminus event \mid \\
& f(Process) \mid \\
& \text{name}
\end{aligned}
$$

Here *event* ranges over a universal set of events, $\sum$, *alph* ranges over subsets of $\sum$, f ranges over a set of function names, and *name* ranges over a set of process names.

A process describes the behaviour of a component in terms of the events in which it may engage. The simplest process of all is *STOP*. This is the process which represents a deadlocked component. It never engages in any event. Another primitive process is *SKIP* which does nothing but terminate successfully; it only performs the special event $\surd$, which represents successful termination.

An event may be combined with a process using the prefix operator, written $\rightarrow$. The process $bang \rightarrow UNIVERSE$ describes a process which first engages in event *bang* then behaves according to process *UNIVERSE*. This new process can be given a name *CREATION* as in

$$CREATION = bang \rightarrow UNIVERSE$$

Processes may be defined in terms of themselves using the principle of recursion. Consider a process to describe the ticking of an everlasting clock.

$$CLOCK = tick \rightarrow CLOCK$$

CLOCK is a process which performs event *tick* and then starts again. This is a somewhat abstract definition. No information is given as to the duration or frequency of ticks. We are simply told that the clock will keep on ticking. A duration of the tick can be specified with timed CSP (Davies and Schneider, 1995). The recursion notation is commonly extended to a set of simultaneous equations where a number of processes are defined in terms of each other. This is known as mutual recursion.

There are a number of CSP operations which combine two processes to produce a new one. The first of these that we shall consider is sequential composition.

$$UNIVERSE = EXPAND;CONTRACT$$

Is the process which first behaves like *EXPAND*, but when *EXPAND* is ready to terminate it continues by behaving like *CONTRACT*. However it may be possible that *EXPAND* will never terminate.

It is rather more complicated to compose two processes in parallel than in sequence. It is necessary to specify a set of events for each process, known as its *alphabet*. The process denoted

$$PANTOMIMEHORSE =$$

$$FRONT \,|[\{forward, backward, nod\}|\{forward, backward, wag\}]|\, BACK$$

represents the parallel composition of two processes: *FRONT* with alphabet $\{forward, backward, nod\}$ and *BACK* with alphabet $\{forward, backward, wag\}$. Here each behaves according to its own definition, but with the constraint that events which are in the alphabet of both *FRONT* and *BACK*, i.e. *forward* and *backward*, require their simultaneous participation. However they may progress independently on those events belonging solely to their own alphabet, If a situation were to arise where *FRONT* could only perform event *forward* and *BACK* could perform event *backward* then deadlock would have occurred. Note that a *pantomime* is a traditional British theatrical entertainment which often features a "horse" consisting of two actors in a single costume, one of whom plays the front legs and head while the other plays the hind legs and tail.

Parallel composition may be extended to three or more processes: given a sequence of processes $V = \langle P_1, .., P_n \rangle$ with corresponding alphabets $\langle A_1, .., A_n \rangle$ we write their parallel composition as

$$PAR(V) = ||_{i=1}^{n} (P_i, A_i)$$

Note that it is implicitly assumed that the termination event $\surd$ requires the joint participation of each process P_i, whether or not it is included in their process alphabets.

An alternative form of parallel composition is *interleaving*, where there is no communication between the component processes. In the parallel combination

$$BRAIN \,|||\, MOUTH$$

The two processes, *BRAIN* and *MOUTH*, progress independently of each other and no cooperation is required on any event, except of $\surd$, the termination event. Any other actions which are possible for both processes will only by one process at the time. Interleaving is a commutative and associative operation and so we may extend the notation to various indexed forms, such as

$$|||_{i=1}^{n} P_i, \quad |||_{x:X} P_x$$

A useful feature of CSP is the ability to describe nondeterministic behaviour, which is where a process may operate in an unpredictable manner. The process

$$BUFFER = TWOPLACE \sqcap THREEPLACE$$

may behave either like process *TWOPLACE* or like process *THREEPLACE*, but there is no way of telling which in advance. The purpose of the $\sqcap$ operator is to specify concurrent systems in an abstract manner. At the design stage, there is no reason to provide any more detail than is necessary and, where possible, implementation decisions should be deferred until later.

This operation is known as *internal choice*. CSP also contains an *external choice* operator $\Box$ which enables the future behaviour of a process to be controlled by other processes running along side it in parallel, which, collectively, we call its *environment*.

The process

$$MICROWAVE = DEFROST \Box COOK$$

may behave like *DEFROST* or like *COOK*. Its behaviour may be controlled by its environment providing that this control is exercised on the very first event. If an initial event *button1* is offered by *DEFROST* that is not an initial event of *COOK*, then the environment may coerce *MICROWAVE* into behaving like *DEFROST*, by performing *button1* as its initial event. If, however, the environment were to offer an initial event that is allowed by both *DEFROST* and *COOK* then the choice between them would be nondeterministic.

Both the choice operators may be extended to indexed forms. We write

$$\Box_{x:A}\ x \to P_x$$

To represent the behaviour of a process which offers any event of a set A to its environment. Once some initial event x has been performed the future behaviour of the process is described by the process P_x. However, the process

$$\sqcap_{x:A}\ x \to P_x$$

offers exactly one event x from A to its environment, the choice being nondeterministic.

Sometimes it is useful to be able to restrict the definition of a process to a subset of relevant events that it performs. This is done using the hiding operator($\backslash$). The process

$$CREATION \setminus bang$$

behaves like *CREATION*, except that each occurrence of event *bang* is concealed. Note that it is not permitted to hide event $\surd$.

Concealment may introduce non-determinism into deterministic processes. It may also introduce the phenomenon of *divergence*. This is a drastic situation where a process performs an endless series of hidden actions. Consider, for instance, the process

$$CLOCK \setminus tick$$

which is clearly a divergent process. It is conventional to extend the notation of $P\backslash A$, where A is a finite set of events.

Finally let us briefly consider process relabelling. Let f be an alphabet transformation function $f : \Sigma \to \Sigma$, which satisfies the property that only finitely many events may be mapped onto a single event. Then the process $f(P)$ can perform the event $f(e)$ whenever P can perform event e. As an example consider a function *new* which maps *tick* to *tock*. Then we have

$$new(CLOCK) = tock \rightarrow new(CLOCK)$$

Denotational Semantic

The meaning of a CSP process is defined in terms of the circumstances under which it might deadlock or diverge. This is the *Failures-Divergences model*.

A *trace* of a process P is any finite sequence of events that it may initially perform. A *divergence* of a process is a trace after which it might diverge. A *failure* of a process P consists of a pair (s,X) where s is a trace of P and X is the set of events if offered to P by its environment after it has performed trace s, might be completely refused.

Each CSP process is then uniquely defined by a pair of sets (F,D), corresponding to its *failures* and *divergences*. The failures and divergences of the fundamental CSP terms are defined by equations such as

$$\begin{aligned} divergences(STOP) &= \{\ \} \\ failures(STOP) &= \{\langle\rangle\} \times \mathbb{P}\Sigma \\ divergences(x \rightarrow P) &= \left\{\langle x\rangle^\frown s \middle| s \in divergences(P)\right\} \\ failures(x \rightarrow P) &= \left\{(\langle\rangle, X) \middle| X \subseteq \Sigma - \{x\}\right\} \\ &\quad \cup \left\{\left(\langle x\rangle^\frown s, X\right) \middle| (s,X) \in failures(P)\right\} \end{aligned}$$

A complete set of equations can be found in (Roscoe, 1998). These particular equations define the meaning of *STOP* and the event-prefix operator $\rightarrow$. First we are told that *STOP* does not diverge, but refuses to perform any event whatever set of events is offered to it. Then we are told that the divergent traces of $x \rightarrow P$ are the divergent traces of P which event x prefixed to them, and the failures of $x \rightarrow P$ to be failures of P with x prefixed to the trace of each failure, *together with* parings of the empty trace with all subsets of $\sum$ which exclude x.

This model is also used for formal reasoning about the behaviour of concurrent systems defined by CSP equations. There is a natural *partial ordering* on the set of all processes given by

$$(F_1, D_1) \sqsubseteq (F_2, D_2) \Leftrightarrow F_1 \supseteq F_2 \wedge D_1 \supseteq D_2$$

The interpretation of this is that process P_1 is worse than P_2 if it can deadlock or diverge whenever P_2 can. This partial ordering is very important to the stepwise refinement of concurrent systems. Starting from an abstract nondeterministic definition, details of components may be independently flashed out whilst preserving important properties of the overall system such as freedom from deadlock and divergence. The FDR tool of Formal Systems Europe (FDR, 2004) can automatically verify this refinement relation in the failures-divergences model.

A process P is deadlock-free if there is no trace after which it might refuse to perform any event, i.e. $\nexists s \bullet (s, \Sigma) \in \mathit{failures}(P)$. It is divergence-free if it has an empty set of divergences. A particularly important point to stress is that when a network of CSP processes is composed in parallel it becomes a single CSP process. So these definitions apply equally to parallel networks of processes as they do to single sequential processes, for which deadlock and divergence are not usually a problem.

Algebraic Semantics

From the failures-divergences model, a complete set of algebraic laws can be deduced, which govern CSP processes, for instance

$$\begin{aligned}
(P;Q);R &= P;(Q;R) \\
(a \rightarrow P);Q &= a \rightarrow (P;Q) \\
P \,|[A|B]|\, Q &= Q \,|[B|A]|\, P \\
P \,|[A|B \cup C]|\, (Q \,|[B|C]|\, R) &= (P \,|[A|B]|\, Q) \,|[\mathrm{A} \cup B|C]|\, R \\
P \,\Box\, (Q \sqcap R) &= (P \,\Box\, Q) \sqcap (P \,\Box\, R) \\
P \sqcap (Q \,\Box\, R) &= (P \sqcap Q) \,\Box\, (P \sqcap R) \\
P \,\Box\, STOP &= P
\end{aligned}$$

There are many more such rules, but there is insufficiently room for their inclusion here. The rules are used to derive correctness properties of CSP systems using algebraic manipulation. See (Roscoe, 1998) for more details.

Operational Semantics

So far two ways of looking at communicating processes have encountered: firstly as algebraic expressions and secondly in terms of abstract mathematical sets based on their observable behaviour. There is no obvious way of seeing from either of these representations how CSP might be realized on a machine. A more concrete approach is given by operational semantics. The operational semantics of CSP is a mapping from CSP expressions to *transition systems*. For example, Figure A-1 illustrates the transition system for the process $a \to (b \to STOP \,\Box\, c \to STOP)$.

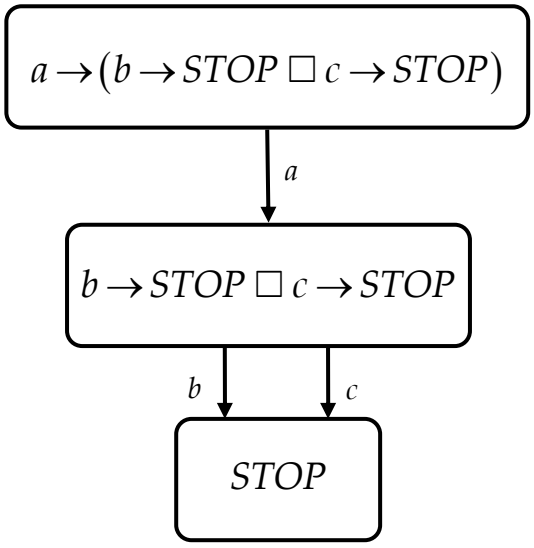


Figure A-1 *CSP Transition System.*

The behaviour of a process predicted by its failures and divergences will be the same as that which can be observed of its operational representation. So we may use the operational semantics of CSP in order to prove properties of process behaviour which are phrased in the

Failures-Divergences model. This feature turns out to be particularly useful when the operational representation of a process is finite although its failures and divergences are infinite, as is usually the case in practice. Therefore this is the representation of processes which us used inside the various CSP verification programs, such as FDR (2004) and Deadlock Checker (Martin and Jassim, 1997).

Language Extensions

The core CSP syntax described above is really abstract, and lacks certain useful features found in conventional sequential and parallel programming languages. The extensions outlined below are useful for writing more detailed specifications and may be defined in terms of the core constructors.

Sometimes a process is defined with parameters, such as

$$BUFF(in,out) = in \rightarrow out \rightarrow BUFF(in,out)$$

This is a process-schema, rather than an actual process. It defines a CSP process for each combination of parameter values. CSP parameters may be integers, real numbers, channels (representing events), sets, matrices, etc.

A *communication* is a special type of event described by a pair $c.v$, where c is the name of the channel on which the event takes place, and v is the value of the message that is passed.

The set of messages communicable on channel c is defined

$$type(c) = \{v | c.v \in \Sigma\}$$

Input and output are defined as follows. A process which first outputs v on channel c, then behaves like P is defined simply as

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

Outputs may involve expressions of parameters such as $P(x) = c!x^2 \rightarrow Q$. The expressions are evaluated according to the appropriate laws.

A process which is initially prepared to input any value x communicable on the channel c, then behave like $P(x)$ is defined.

$$(c?x \rightarrow P(x)) = \Box_{v:type(c)} (c.v \rightarrow P(v))$$

It is usual for a communication channel to be used by at most two processes at any time: one for input and the other for output. This restriction, which is known as *triple-disjointedness*, is not enforced in the modern version of CSP.

Another important aspect to real programming languages is the use of conditionals. Let b be a Boolean expression (either true or false). Then

$$P \lhd b \rhd Q \qquad ("P \text{ if } b \text{ else } Q")$$

is a process which behaves like P if the value of expression b is true, or like Q otherwise.

These extensions are especially useful for specifying fine detail during the later stages of program refinement. At the design stage one should tend to stick to more abstract, nondeterministic definitions of processes. The deadlock issue will usually be addressed at this point. In this way one can build robust programs for which deadlock-freedom cannot be compromised by implementation decisions made at a later stage.

APPENDIX B

Processor-specific methods

CTC and CTC++ are written in 99% portable C/C++ and less than 1% is processor-specific. The processor-specific methods must be programmed for each different type of processor. The processor-specific methods have been reduced to a minimal set of methods, which are categorized as follows:

Context-switching specific methods

```
void Processor__startswitch(void);
```

Start the first process and hold the thread of control of the main program.

```
void Processor__stopswitch(void);
```

Return the thread of control back to the main program.

```
void Processor__contextswitch(void);
```

Perform a context switch to the next process thread that is waiting on the ready queue of the dispatcher.

```
void Processor__enterAtomic(void);
```

Enter an atomic region in which no interrupt or context switch may occur.

```
void Processor__exitAtomic(void);
```

Leave the atomic region and allow interrupts and context switching.

```
void Processor__initiateStack(ProcessThread pt, unsigned int size ,
     void *run);
```

Allocate and initiate a stack for the process thread `pt` with specified `size` and `run` pointing to the run method of a process.

```
int  Processor__defaultStackSize(void);
```

Return a default stack size. Every process that is added to a parallel or priparallel construct with method `add(Process)` will get a stack with the default stack size. This stack size can be overridden with method `add(Process, stacksize)` with `stacksize` is the new stack size in CTC and CTC++.

Memory specific methods

```
Object Processor__malloc(unsigned int size);
```

Allocate memory of `size` and return the pointer to the allocated memory. This method returns `NULL` if no memory can be allocated.

```
Void   Processor__free(void *ptr);
```

Free allocated memory pointed as specified by `ptr`.

```
Void   Processor__copy(Object src, Object dest, unsigned int size);
```

Copy the content of an object `src` to object `dest`. The size of the memory block that will be copied is specified by `size`.

Interrupt specific methods

```
int    Processor__registerInterruptService(unsigned int irq,
       InterruptService is);
```

Register an interrupt service `is` to an interrupt with number `irq`.

```
int    Processor__unregisterInterruptService(InterruptService is);
```

Unregister interrupt service `is`.

```
void   Processor__irqEnable(void);
```

Allow interrupts.

```
void   Processor__irqDisable(void);
```

Disallow interrupts.

Timer specific methods

```
void   Processor__initiateTimer(void);
```

Initiate the hardware timer.

```
int    Processor__readTimer(void);
```

Read the time latch.

```
void   Processor__latchTimer(int value);
```

Set a new `value` in the latch of the timer.

```
void   Processor__setTimer(int value);
```

Override the counter with a new `value`.

These methods are prototyped and documented in the source file `processor.c`. These methods are private to the kernel and not publicly available for the user. The classes `ProcessThread` and `InterruptService` take care of some organization and are written in portable C. The timer

object is a sophisticated object that makes a 64-bits timer from a 16-bits system real-time timer. 16-bit timers are commonly part of computer systems. Usually, the system timer is coupled to the processor which handles the timer interrupts. Therefore, the timer specific methods are related to the `Processor` class.

APPENDIX C

The exception operator $\vec{\Delta}$

C.1 Introduction

A theoretical model of a compositional exception operator is proposed in this appendix. The semantics of the exception operator does not alter the semantics of the CSP operators so that we do not have to redefine the CSP operators .

C.2 Proposed exception handling in CSP

A process *P* can face an error on which it should not continue or the process gets blocked forever waiting on an event that will never happen due to an error in the system. In either case *P* will behave as *STOP*. A process that behaves suddenly as *STOP* is often an undesired behaviour in programs. One would like to escape from *STOP* and handle the error at hand. Such an escape manifests exception handling.

CSP offers an interrupt operator that could be used to escape from *STOP* on an internal event. The CSP interrupt operator is depicted as

$$P\Delta_i E \tag{1}$$

This process behaves as *P* and on the occurrence of internal event *i* this process will behave as *E*. Event *i* is not part of the alphabets of *P* and *E*,

i.e. $i \notin \alpha P \cap \alpha E$ (αP and αE are the alphabets of P and E). The exception in P or in the system can be represented as an internal event i which is intercepted by the interrupt operator and performs a preemption from P to E. Process E is the exception handler of P.

One problem with the interrupt operator is that process E does not know about the different exceptions occurred in P and it does not know what exceptions to handle. Another problem is that the interrupt operator is difficult to implement in software with low performance costs and a small memory footprint. Due to the preemption from P to E, the interrupt operator should somehow release all channels that are being claimed by the child processes in P. Hence, an exception should not cause deadlock because an any-to-any channel was claimed and never released. In a dynamic network of communicating processes the implementation is even more complex. This management will be time consuming and it will increase a significant amount of code.

We define an exception operator $\vec{\Delta}$ that is a simplified version of the interrupt operator Δ_i which is able to collect exceptions in the program.

The proposed exception operation is

$$P\vec{\Delta}E \tag{2}$$

This process behaves as P and on the occurrence of unhandled exceptions it will behave as E. A process with unhandled exception behaves as *STOP* for which the exception operator manifests an escape to the exception handling process that handles the exception. A process with handled exceptions is not in exception and does not require exception handling.

The properties of the exception operator are

$$P\vec{\Delta}E \begin{cases} P & \text{if } P \text{ is not in exception} \\ E & \text{if } P \text{ suffers from one or more unhandled exceptions} \end{cases} \tag{3}$$

The exception operator defines in what way the CSP/CSPP operators should cooperate with exceptions. This is orthogonal to the original semantics of the CSP/CSPP operators. The arrow on top illustrates

explicitly the priority difference between *P* and *E*; i.e. *E* is more urgent than *P* on the occurrence of the exception. This direction can help tracing design conflicts on exception handling in process architectures, as discussed in Chapter 5. This exception operator treats an exception as an event and as a state.

Unusual situations in the environment can affect the behaviour of software. Such an unusual situation can be detected as an exception and passed through (or thrown) by channels and barriers. The exception affects the process when it reaches the point where it wants to communicate with a corrupted (or poisoned) channel or barrier. This may also include exceptions, such as a division-by-zero or null pointers, which involve internal events. The proposed exception operator can capture these kinds of exceptions.

As previously mentioned, exceptions can occur on communication with channels or barriers. For example, consider the following process:

$$(c \rightarrow SKIP)\vec{\Delta}E \tag{4}$$

Event *c* denotes a channel-end or barrier-end, e.g. a channel-input *c?x*, channel-output *c!v*, channel-call *c.call*, channel-accept *c.accept*, or a barrier-sync *c.sync*. On the occurrence of an exception the environment will refuse *c* and the environment will engage in event *ex.c* instead. Event *ex.c* is hidden by the exception operator.

It is useful that the exception handler *E* knows the nature of the exception in order to take the appropriate actions. The proposed exception operator collects the trace of exceptions that have been raised in *P*. The trace of exceptions is a shared set, called *error*, and is passed to $E(error)$.

$$P\vec{\Delta}E = P\vec{\Delta}E(error) \tag{5}$$

The exception handler $E(error)$ may use *error* to discover which exceptions are involved and should be handled. After handling the exception, the exception must be taken from the set *error*; i.e. $error = error - ex.c$

The expression behind the exception operator is described as follows:

$$P\vec{\Delta}E = \left(\begin{pmatrix} EX(P); \begin{pmatrix} read?error \rightarrow stop \rightarrow \\ \begin{pmatrix} E(error) \\ \triangleleft error \neq \langle\ \rangle \triangleright SKIP \end{pmatrix} \end{pmatrix} \\ \| EXSET(\langle\ \rangle) \end{pmatrix} \setminus \begin{Bmatrix} read, \\ write, \\ stop \end{Bmatrix} \right) \quad (6)$$

with $error \subseteq EXCEPTIONS \wedge \langle\ \rangle$ and $error = \langle\ \rangle$ as the initial state of this process. Here, *EXSET* is a process that collects the set of exceptions. Its *read*, *write*, and *stop* events are hidden from outside the exception process. This process is defined as

$$EXSET(es) = \left(\begin{pmatrix} (write?e \rightarrow EXSET(es^\frown e)) \Box (read!es \rightarrow EXSET(es)) \\ \Delta\,(stop \rightarrow SKIP) \end{pmatrix} \right) \quad (7)$$

Event *write* is used to put an exception to the set and *read* is used to get the set. Event *stop* will terminate EXSET.

Process $EX(P)$ distinguishes between event c and event $ex.c$.

$$EX(P) = \left(\begin{pmatrix} (c \rightarrow EX(P \text{ after } \langle c \rangle)) \Delta_{ex.c} \\ (ex.c \rightarrow write!\langle ex.c \rangle \rightarrow SKIP) \end{pmatrix} \right) \setminus \{ex.c\} \quad (8)$$

Here, $c \in fs(P)$ is the *first step* returning the first communication event of P. $EX(P)$ operates on the chain of prefixes and excludes termination events ✓. The recursive function $EX(P)$ is used for tracing exception events occurred in P. Process $EX(P)$ may engage in $ex.c$ with its environment, but $ex.c$ is hidden from other processes.

The environment decides to engage in event c or in event $ex.c$, but not simultaneously. With this assumption we can replace the interrupt operator with an external choice, as described in

$$EX(P) = \left(\begin{pmatrix} (c \rightarrow EX(P \text{ after } \langle c \rangle)) \Box \\ (ex.c \rightarrow write!\langle ex.c \rangle \rightarrow SKIP) \end{pmatrix} \right) \setminus \{ex.c\} \quad (9)$$

This external choice operator can be efficiently implemented as discussed in Chapter 5.

C.3 Compositional semantics

Process P can be any composition of simpler processes. It is important that these compositions cooperate in gathering exceptions and pass through these exceptions to an exception handler higher in the hierarchy. This behaviour is part of the exception operator description, as will be discussed next.

On the termination events ✓ of each process the set *error* must be checked whether it is empty or not. In case *error* is not empty, the process has unsuccessfully terminated and otherwise the process has successfully terminated. This behaviour is described as

$$\begin{aligned} &EX(Q;R) = \\ &EX(Q);\left(read?error \rightarrow EX(R) \triangleleft error = \langle\ \rangle \triangleright SKIP\right) \end{aligned} \tag{10}$$

SKIP is performed on the unsuccessful termination of Q and R is performed on the successful termination of Q.

For the parallel composition we can write:

$$EX(Q \parallel R) = EX(Q) \parallel EX(R) \tag{11}$$

The parallel operator collects the exceptions from each branch in the set *error*.

For the external choice composition we can write:

$$\begin{aligned} &EX(Q \Box R) = \\ &\left(\begin{array}{l} \left(EX(Q);\left(read?error \rightarrow EX(R) \triangleleft ex.fs(Q) \in error \triangleright SKIP\right)\right)\Box \\ \left(EX(R);\left(read?error \rightarrow EX(Q) \triangleleft ex.fs(R) \in error \triangleright SKIP\right)\right) \end{array}\right) \end{aligned} \tag{12}$$

This description implies that the choice operator will collect all exceptions that occurred on the first event in the guarded processes. This

is a desired behaviour since the choice operator cannot make a fair choice due to exceptions on the first events of the guarded processes.

The prioritized parallel and prioritized choice operators have similar expressions as their non-prioritized counter parts in respectively (11) and (12). More generally, we can write the following expressions for $EX(P)$ for each CSP/CSPP operator:

The sequential process $\left(\underset{i=0..n-1}{;} P_i\right)\vec{\Delta}E$ is

$$EX\left(\underset{i=0..n-1}{;} P_i\right) = \underset{i=0..n-1}{;} \left(read?error \rightarrow \left(EX(P_i) \triangleleft error = \langle\ \rangle \triangleright SKIP\right)\right) \quad (13)$$

The parallel process $\left(\underset{i=0..n-1}{\|} P_i\right)\vec{\Delta}E$ is

$$EX\left(\underset{i=0..n-1}{\|} P_i\right) = \underset{i=0..n-1}{\|} EX(P_i) \quad (14)$$

The alternative (choice) process $\left(\underset{i=0..n-1}{\Box} P_i\right)\vec{\Delta}E$ for one recursion is

$$EX\left(P_0 \Box \left(\underset{i=1..n-1}{\Box} P_i\right)\right) =$$

$$\left(\begin{array}{l} \left(\begin{array}{l} EX(P_0);read?error \rightarrow \left(\begin{array}{l} EX\left(\underset{i=1..n-1}{\Box} P_i\right) \\ \triangleleft ex.fs(P_0) \in error \triangleright \\ SKIP \end{array}\right) \end{array}\right) \Box \\ \left(\begin{array}{l} EX\left(\underset{i=1..n-1}{\Box} P_i\right);read?error \rightarrow \left(\begin{array}{l} EX(P_0) \\ \triangleleft ex.fs\left(\underset{i=1..n-1}{\Box} P_i\right) \in error \triangleright \\ SKIP \end{array}\right) \end{array}\right) \end{array}\right) \quad (15)$$

with $ex.fs\left(\underset{i=1..n-1}{\Box} P_i\right) \in error = ex.fs(P_1), ex.fs(P_2), ..., ex.fs(P_{n-1}) \in error$

The prioritized parallel process $\left(\underset{i=0..n-1}{\overleftarrow{\|}} P_i\right)\vec{\Delta}E$ is

$$EX\left(\underset{i=0..n-1}{\overleftarrow{\|}} P_i\right) = \underset{i=0..n-1}{\overleftarrow{\|}} EX(P_i) \tag{16}$$

The prioritized alternative (choice) process $\left(\underset{i=0..n-1}{\overleftarrow{\Box}} P_i\right)\vec{\Delta}E$ for one recursion is

$$EX\left(P_0 \overleftarrow{\Box}\left(\underset{i=1..n-1}{\overleftarrow{\Box}} P_i\right)\right) =$$

$$\left(\begin{array}{l} \left(\begin{array}{l} EX(P_0); read?error \rightarrow \left(\begin{array}{l} EX\left(\underset{i=1..n-1}{\overleftarrow{\Box}} P_i\right) \\ \triangleleft\, ex.fs(P_0) \in error \,\triangleright \\ SKIP \end{array}\right) \end{array}\right) \overleftarrow{\Box} \\ \left(\begin{array}{l} EX\left(\underset{i=1..n-1}{\overleftarrow{\Box}} P_i\right); read?error \rightarrow \left(\begin{array}{l} EX(P_0) \\ \triangleleft\, ex.fs\left(\underset{i=1..n-1}{\overleftarrow{\Box}} P_i\right) \in error \,\triangleright \\ SKIP \end{array}\right) \end{array}\right) \end{array}\right) \tag{17}$$

with $ex.fs\left(\underset{i=1..n-1}{\overleftarrow{\Box}} P_i\right) \in error = ex.fs(P_1), ex.fs(P_2), ..., ex.fs(P_{n-1}) \in error$

Nested exception handling is incorporated, as in

$$\left(\left(P\vec{\Delta}E_0\right)\vec{\Delta}E_1\right)\vec{\Delta}E_2 = P\vec{\Delta}E_0\vec{\Delta}E_1\vec{\Delta}E_2 = P\underset{i=0..2}{\vec{\Delta}} E_i \tag{18}$$

C.4 Livelock and deadlock

An exception can cause a *STOP* indicating a livelock or deadlock for which the exception operator does not manifest an escape. Consider the following example,

$$\left((e \to d \to P) \underset{\{d\}}{\|} (f \to d \to Q)\right) \vec{\Delta} E \tag{19}$$

Assume event f fails and therefore the environment offers $ex.f$ instead; $error = \langle ex.f \rangle$. The right hand side of $\|$ will *SKIP* and will not engage in d. The left hand side will block forever on d and consequently the entire process behaves as *STOP*, as in

$$\left((e \to STOP) \underset{\{d\}}{\|} (ex.f \to SKIP)\right) \vec{\Delta} E \xrightarrow{\text{after } e \text{ and } ex.f} (STOP \| SKIP) \vec{\Delta} E \tag{20}$$
$$= STOP \vec{\Delta} E = STOP$$

This is a correct behaviour. However, the desired behaviour is that E takes over control so that the program can continue without deadlocking or livelocking. A possible solution is

$$\left((e \to d \to P) \| \left(\left((f \to d \to Q) \vec{\Delta} (refuse(d))\right)\right)\right) \vec{\Delta} E \tag{21}$$

When f fails then $error = \langle ex.f \rangle$ and the inner exception handler performs $refuse(d)$, which is a function that will cause d to fail also. Successively, the left hand process will fail on d and exception event $ex.d$ will happen instead. Then E will be performed with $error = \langle ex.f, ex.d \rangle$.

$$\left((e \to ex.d \to SKIP) \| (ex.f \to SKIP)\right) \vec{\Delta} E \xrightarrow{after\ e, ex.f\ and\ ex.d} (SKIP) \| (SKIP) \vec{\Delta} E$$
$$= E$$

In order to avoid livelock or deadlock one should refine the design of the process architecture with exception operators and $refuse(channel \mid barrier)$ functions. This technique is also known as *poisoning channels or barriers*.

APPENDIX D

Examples

D.1 Producer/Consumer example

An example of a producer process and a consumer process connected via a generic data channel is described in this section. The `Producer` class defines the producer process that sends a value to the consumer process. Figure D-1 depicts a class diagram of the producer/consumer communication relationship. A channel is stereotyped with `<<channel>>`. The channel input and output interfaces are stereotyped `<<channelinput>>` and `<<channeloutput>>`. Stereotypes provide constraints in UML diagrams for model checking.

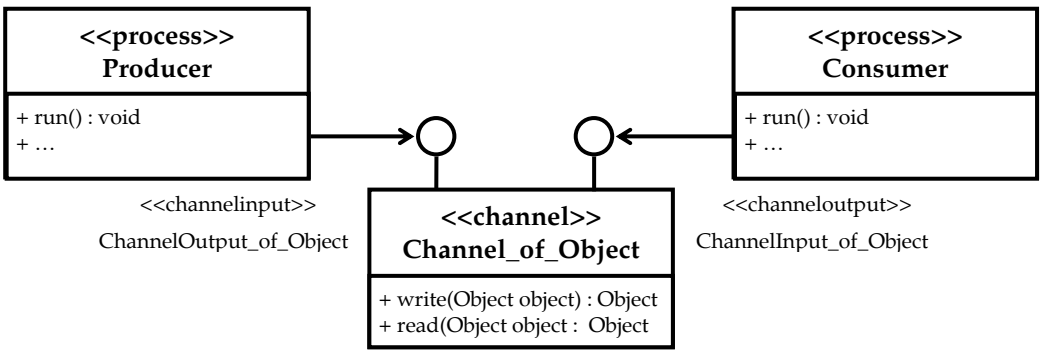


Figure D-1 *UML class diagram of a producer/consumer communication relationship.*

The `Producer` class defines the producer process that sends a value to the channel. See Listing D-1. The `Consumer` class defines the consumer process that receives the value and prints it on screen. See Listing D-2. The producer and consumer run in parallel, see Listing D-3 and Section 4.6.1. Both processes terminate when they have communicated via the channel.

```
import csp.lang.*;
import csp.lang.Process;
import csp.lang.Integer;

class Producer implements Process
{
  ChannelOutput_of_Object chanout;
  int value = -1;

  Producer(ChannelOutput_of_Object channel) {
    chanout = channel;
  }

  public void run()
    throws ExceptionSet {
      Integer obj = new Integer();
      ...
      obj.value = value;
      chanout.write(obj);
      ...
  }

  public void setValue(int v) {
    value = v;
  }
}
```

Listing D-1 *Producer class.*

```
class Consumer implements Process
{
  ChannelInput_of_Object chanin;
  int value = -1;

  Consumer(ChannelInput_of_Object channel) {
    chanin = channel;
  }

  public void run()
```

```
    throws ExceptionSet {
      Integer obj = new Integer();
      ...
      obj = chanin.read(obj);
      value = obj.value;
      ...
      System.out.println("value = "+value);
  }

  public int getValue() {
    return value;
  }
}
```

Listing D-2 *Consumer class.*

In the consumer process the `read(..)` method has a double function. The `read(..)` may use the specified `obj` argument to copy data in or if that fails for some reason a clone is returned. The consumer has become independent from the message delivery mechanisms pass-by-value or pass-by-reference.

```
public static void main(String[] args) {
  Channel_of_Object channel = new Channel_of_Object();

  Parallel par = new Parallel(new Process[] {
    new Producer(channel),
    new Consumer(channel)
  });

  try {
    par.run();
  } catch (ExceptionSet es) {
    ... exception handling
  }
}
```

Listing D-3 *Producer and consumer in parallel.*

D.2 Client/Server example

In this section, the coding of a client process and a server process that communicate via a call channel is shown. This example shows that a

server process can have multiple interfaces. Each interface provides a different view of services for different types of clients. Here, `Server` implements two interfaces: `OnOff` and `OtherService`. The server process is given in Listing D-4 and the two interfaces are given in Listing D-5 and Listing D-6. Their implementations are not shown. The call channel is defined by the `MyCallChannel` class and its relationships with other classes and interfaces are shown in Figure 4-6.

```
import csp.lang.*;
import csp.lang.Process;

public class Server implements Process, OnOff, OtherServices
{
  MyCallChannel channel;

  public Server(MyCallChannel channel) {
    this.channel = channel;
  }

  public void run()
    throws ExceptionSet {
    ... perform server task
    channel.accept();
    ...
  }

  public void on() { ... }

  public void off() { ... }

  public XYZ calculate(..)
    throws ExceptionSet { ...; return ...; }

  public void add(..) { ... }

  public void remove(..) { ... }

  public void setGain(..) { ... }

  public double getGain() { ...; return ...; }
}
```

Listing D-4 *The server process.*

Although these methods are publicly accessible they should only be invoked when the process plays the role of a process instance (in between runs) or otherwise only through call channels that offer these services.

The interfaces `OnOff` and `OtherServices` are defined as:

```
public interface OnOff {
  public void on();
  public void off();
}
```

Listing D-5 *The OnOff service interface.*

```
public interface OtherServices {
  public XYZ calculate(..) throws ExceptionSet;
  public void add(..);
  public void remove(..);
  public void setGain(..);
  public double getGain();
}
```

Listing D-6 *The OtherServices service interface.*

The `run()` method is not part of call interfaces but belongs to the process instance interface as specified by the `Process` interface in Listing 4-1. Thus, a call channel does not implement the `Process` interface.

The class diagram of the client/server communication relationship is given in Figure D-2.

The call channel is stereotyped with `<<callchannel>>`. The channel call and accept interfaces are stereotyped `<<channelcall>>` and `<<channelaccept>>`.

The call channel class is given in Listing D-7. `MyCallChannel` inherits `CallChannel` and all interfaces. Each method has a constant tag that is required for the `accept(..)` methods to let the server process know about the method that was accepted and performed. The `process` argument is the reference to the server process. This argument is usually `this`; the server process itself. On acceptance the tag (or number) of the method

that was performed is returned. The tag can be useful for implementing a state machine.

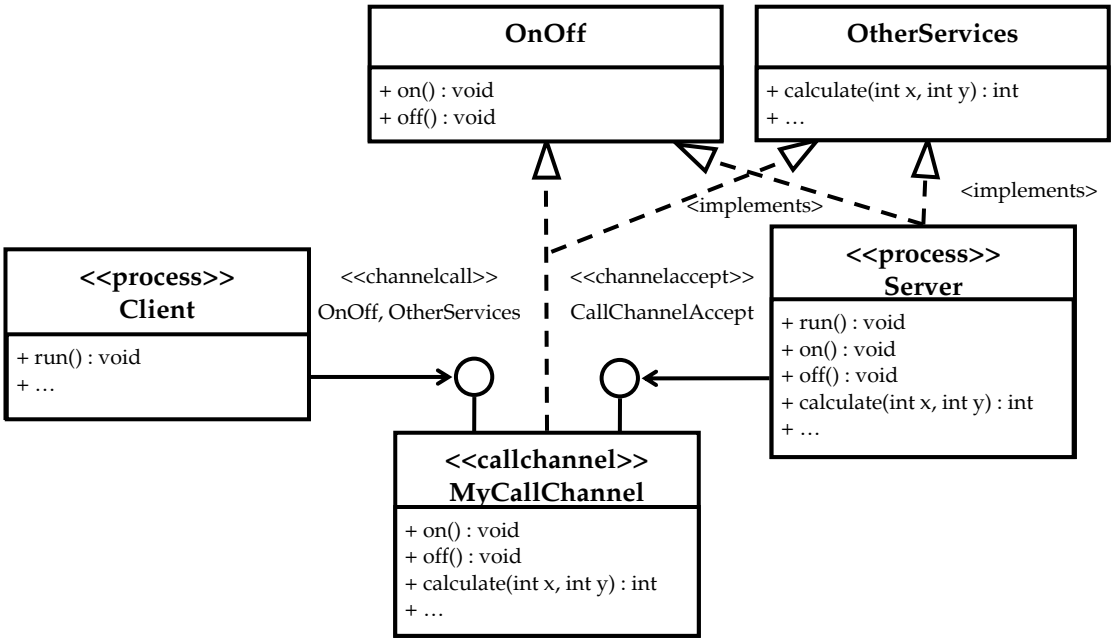


Figure D-2 *UML class diagram of the client/server communication relationship with the server process implementing the call interfaces OnOff and OtherServices.*

The `join(int method)` synchronizes the caller (client) and callee (server). On rendezvous, one thread of control will invoke the method on the server process and both processes continue after `fork()`. This behaves as if the server process performs the method in rendezvous.

```
import csp.lang.CallChannel;

public class MyCallChannel extends CallChannel
  implements OnOff, OtherServices {
  public static final int ON        = 0;
  public static final int OFF       = 1;
  public static final int CALCULATE = 2;
  public static final int ADD       = 3;
  public static final int REMOVE    = 4;
  public static final int SETGAIN   = 5;
  public static final int GETGAIN   = 6;
```

```
  public void on() {
    join(ON);
    ((OnOff)process).on();
    fork();
  }

  public void off() {
    join(OFF);
    ((OnOff)process).off();
    fork();
  }

  public XYZ calculate(..)
  throws ExceptionSet {
    join(CALCULATE);
    XYZ object = ((OtherServices)process).calculate(..);
    fork();
    return object;
  }

  public void add(..) {
    join(ADD);
    ((OtherServices)process).add();
    fork();
  }

  public void remove(..) {
    join(REMOVE);
    ((OtherServices)process).remove();
    fork();
  }

  public void setGain(..) {
    join(SETGAIN);
    ((OtherServices)process).setGain();
    fork();
  }

  public double getGain() {
    join(GETGAIN);
    double value = ((OtherServices)process).getGain();
    fork();
    return value;
  }
}
```

Listing D-7 *Call channel MyCallChannel class.*

The `CallChannel` class implements a few default `accept(..)` methods:

```
int accept(csp.lang.Process process)
  throws ExceptionSet
```

- accept any method

```
int accept(int method, csp.lang.Process process)
  throws ExceptionSet
```

- accept only specified method

```
int accept(int[] methods, csp.lang.Process process)
  throws ExceptionSet
```

- accept any method within the specified method range

These methods are implemented in the `CallChannel` base-class and are specified in the `CallChannelAccept` interface, see Listing D-8. The `accept(..)` methods should exclusively be used by the server process and no other methods should be called by the server on the call channel.

```
public interface CallChannelAccept {
  public int accept(csp.lang.Process process)
    throws ExceptionSet;
  public int accept(int method, csp.lang.Process process)
    throws ExceptionSet;
  public int accept(int[] methods, csp.lang.Process process)
    throws ExceptionSet;
}
```

Listing D-8 *Call channel accept interface.*

In Listing D-9, a client class is shown. The client and server are executed in parallel as shown in Listing D-10.

```
import csp.lang.*;
import csp.lang.Process;

public class Client implements Process
{
  OnOff channel;

  public Client(MyCallChannel channel) {
```

```
    this.channel = channel;
  }

  public void run()
    throws ExceptionSet {
    ...
    channel.on();
    ...
    channel.calculate();
    ...
    channel.off();
    ...
  }
}
```

Listing D-9 *Client process.*

```
public static void main(String[] args) {
  MyCallChannel channel = new MyCallChannel();

  Client client = new ClientA(channel);
  Server server = new Server(channel);

  Parallel par = new Parallel(new Process[] {
    client,
    server
  });

  try {
    par.run();
  } catch (ExceptionSet es) {
  ... exception handling
  }
}
```

Listing D-10 *Client-server example.*

The server process is depicted in Listing D-4. The parallel construct is elaborated in Section 4.6.1.

The use of data channels and call channels can be mixed as illustrated in the following example. Consider a mechatronic system with an electrical motor, which should be turned on and off by an embedded control system.. The `on()` method turns on the electrical motor and the `off()` method turns it off. Although the server process offers these services; it

does not control the hardware directly to turn the motor on or off. Instead, the server process uses a data channel that performs the actual hardware control.

D.3 Barrier Example

The following example shows two processes that synchronize on a barrier two times and this illustrates the differences between `sync()` and `sync(process)`.

```
import csp.lang.*;
import csp.lang.Process;

public class SyncWriter implements Process
{
  Process process;
  Barrier barrier;

  public SyncWriter(ChannelOutput_of_Object channel,Barrier barrier) {
    this.barrier = barrier;
    proces = new Producer(channel);
  }

  public void run()
    throws ExceptionSet {
    ...
    barrier.sync();              // sync example 1
    ...
    process.setValue(100);       // sync example 2
    barrier.sync(process);
    ...
  }
}

public class SyncReader implements Process
{
  Process process;
  Barrier barrier;
  int value

  public SyncReader(ChannelInput_of_Object channel, Barrier barrier)
    this.barrier = barrier;
    proces = new Consumer(channel);
```

```
  }

  public void run()
    throws ExceptionSet {
    ...
    barrier.sync();               // sync example 1
    ...
    barrier.sync(process);        // sync example 2
    value = process.getValue();
    ...
  }
}

public static void main(String[] args) {
  Channel_of_Object channel = new Channel_of_Object();
  Barrier           barrier = new Barrier(2);

  Process syncwriter = new SyncWriter(channel, barrier);
  Process syncreader = new SyncReader(channel, barrier);

  Parallel par = new Parallel(new Process[] {
    syncwriter,
    syncreader,
  });

  try {
    par.run();
  catch (ExceptionSet es) {
  ... exception handling
  }
}
```

Listing D-11 *Barrier example: embedded example 1 performs a barrier synchronization without communication and embedded example 2 communicates on the barrier synchronization.*

The processes `SyncWriter` and `SyncReader` will synchronize on the first `sync()` and secondly they will synchronize on `sync(process)` and the barrier will execute each `process` at each end of the barrier. Successively, both processes will synchronize on `channel`. In this case, the `channel` performs communication (event) and the parallel construct in the barrier performs the barrier synchronization (event). The processes `Producer` and `Consumer` are given in Listing D-1 and Listing D-3.

D.4 Additional Guards

D.4.1 Skip guards

In circumstances where the alternative construct should continue when no channel is ready then a skip guard provides this behaviour. A skip guard is a guard that does not wait for an event to be ready. Skip guards can be created by one of the following constructors:

Unconditional skip guards

`Guard()`	is always true and performs a skip if selected
`Guard(process)`	is always true and performs the process if selected

CTJ provides a special process Skip that can play the role of a process or the role of a guard, as in,

`Guard(new Skip())`	is always true and performs a skip if selected; this is the same as `Guard()` in the role of a guard
`Skip()`	is always true and performs a skip if selected; this is the same as `Guard(new Skip())` in the role of a guard

Conditional skip guards

`Guard(condition)`	performs skip if selected
`Guard(condition, process)`	performs process if selected

The conditional skip guards are,

`Guard(condition, new Skip())`	performs a skip if selected
`Skip(condition)`	performs a skip if selected

D.4.2 Timeout guards

A process can also be guarded by a timeout event. A timeout guard can be specified by one of the following constructors:

Unconditional timeout guards

`Guard(time)`	becomes ready after the specified time and performs a skip if selected
`Guard(time, process)`	becomes ready after the specified time and executes the specified process if selected
`Timeout(time)`	same as `Guard(time)`
`Timeout(time, process)`	same as `Guard(time, process)`

Conditional timeout guards

`Guard(condition, time)`	performs a skip after the specified time and if selected
`Guard(condition, time, process)`	performs the specified process after the specified time and if selected
`Timeout(condition, time)`	same as `Guard(condition, time)`
`Timeout(condition, time, process)`	same as `Guard(condition, time, process)`

When multiple timeout guards in the guard list of the alternative construct are specified then the guard with the smallest timeout will be active. When there are multiple timeout guards with the smallest and equal timeout then one will be arbitrarily selected. One timeout guard per alternative construct is recommended.

D.5 State handling methods

After the construction of a process, the state handling methods are used to dynamically set or get the process state before or after its execution.

In CSP diagrams, the ends of the solid arrows attached to a process are implemented by the constructor. The ends of the open arrows are implemented by the state handling methods. The use of the constructor is elaborated in Chapter 4. In this section, the use of the state handling methods for implementing the open arrows is illustrated.

Figure D-3 is a copy of Figure 3-11 with visible port labels attached at the ends of the open arrows. As discussed in Section 3.5.3, the causality determines that process *f* must be performed before *g*.

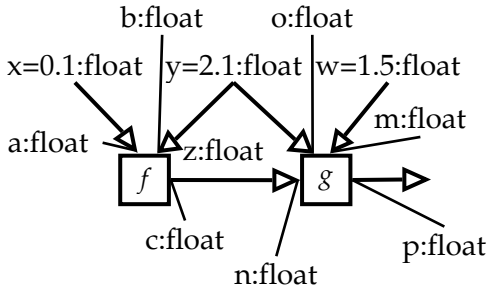


Figure D-3 *Example of state initialization.*

The code construct that results from Figure D-3 is given in Listing D-12. The variables and the process instances are declared on top of the program. At this stage, input variables are initiated with an initial value.

```
float v, w=1.5, x=0.1, y=2.1,z;
F f = new F();
```

```
G g = new G();
...
f.set_a(x);
f.set_b(y);
f.run();
z = f.get_c();
...
g.set_m(w);
g.set_n(z);
g.set_o(y);
g.run();
v = g.get_p();
```

Listing D-12 *Example of initiating processes using state handling methods.*

The constructor is a special initiate method, which initiates the process instance after allocation in memory. In CTJ, the constructor is used to assign channel-ends and barrier-ends to the process during its construction. This can be very-well be done by state handling methods before the process is executed.

D.6 ARTY Implementation

The implementation of the top network builder for ARTY and its sub-processes are given in this section.

D.6.1 Top network builder

The CSP diagram of Figure 6-8 and Figure 6-9 describes the top network builder of this control application. The CSP diagram translates to the main source code in Listing D-13. This top network builder declares channels, link drivers (external channel), variables, processes, and constructs, which are required to build the top-level network of communicating processes. In this case, the top network builder is the only hardware dependent process in the software since it is the only one that sets up hardware dependent objects. All other processes are hardware independent since they solely use channels.

In order to keep the listings readable we will omit the header files and the inclusion of headers in the source code.

```
int main(void) {
  //--- external channels declarations (Figure 6-8)
  ChannelIn<double>  *feedback_leftspeed  = new Encoder( Encoder::LEFT);
  ChannelIn<double>  *feedback_rightspeed = new Encoder(Encoder::RIGHT);
  ChannelOut<double> *steer_leftspeed     = new Motor(Motor::LEFT);
  ChannelOut<double> *steer_rightspeed    = new Motor(Motor::RIGHT);

  //--- internal channels declarations (Figure 6-8)
  Channel<double> *setpoint_leftspeed  = new Channel<double>();
  Channel<double> *setpoint_rightspeed = new Channel<double>();

  //--- processes declarations (Figure 6-9,Figure 6-8)
  SequenceControllerProcess   *scProc = new SequenceControllerProcess(
    setpoint_leftspeed, setpoint_rightspeed);
  MotorControllerLeftProcess  *mclProc = new MotorControllerLeftProcess(
    setpoint_leftspeed, feedback_leftspeed, steer_leftspeed);
  MotorControllerRightProcess *mcrProc= new MotorControllerRightProcess(
    setpoint_rightspeed, feedback_rightspeed, steer_rightspeed);

  //--- compositional constructs declarations (Figure 6-9)
  Parallel *par = new Parallel();
  par->add(mclProc);
  par->add(mcrProc);

  PriParallel *pripar = new PriParallel();
  pripar->add(par);
  pripar->add(scProc);

  //--- set timed events (Figure 6-8)
  //    (channel, start_sec, start_usec, interval_sec, interval_usec)
  System::at(feedback_leftspeed,  0, 500000, 0, 10000);
  System::at(feedback_rightspeed, 0, 500000, 0, 10000);
  System::at(steer_leftspeed,  0, 500000, 0, 10000);
  System::at(steer_rightspeed, 0, 500000, 0, 10000);

  //--- run the process
  pripar->run();

  //--- delete all instances
  delete feedback_leftspeed;
  delete feedback_rightspeed;
  ...
```

```
  return 0;
}
```

Listing D-13 *Top Network builder or main source code.*

This network builder is further detailed by its child processes. The child processes are described in the next sections.

D.6.2 MotorControllerLeftProcess

Process `mclProc` in Figure 6-8 and Figure 6-9 is described by process class `MotorControllerLeftProcess`. The source code is given in Listing D-14.

The constructor specifies the process-interface of ports to which this process can be connected via channels to other processes. The `ChannelIn` and `ChannelOut` types specify whether a process can respectively read or write on a channel. Reading and writing on respectively `ChanOut` and `ChanIn` channels is prohibited by a compiler check. These ports are generic whereby the type between brackets, i.e. `<double>`, specifies that the channel only accepts data of type `double`. The references of channels are kept local so that the `run()` method can read or write on these channels. In the constructor the arrays of indexed channels are declared and they are private to the process. The constructor assigns the named channels to the indexed channels. Also the 20-process `mcl20Proc` is declared and it is connected to other processes via the array of input-channels and they array of output-channels.

```
/** Construct MotorControllerLeftProcess. **/
MotorControllerLeftProcess::MotorControllerLeftProcess(
ChannelIn<double> *setpoint, ChannelIn<double> *feedback,
ChannelOut<double> *steer) {
  chanin  = ChannelIn<double> * [2];
  chanout = ChannelOut<double> * [1];

  chanin[0]  = setpoint;
  chanin[1]  = feedback;
  chanout[0] = steer;

  mcl20Proc = new MotorControllerLeft20Process(chanin, chanout);
}
```

```
/** Process body. **/
void MotorControllerLeftProcess::run(void) {
  while(true) {
    mcl20Proc->run();
  }
}

/** Destruct MotorControllerLeftProcess. **/
MotorControllerLeftProcess::~MotorControllerLeftProcess(void) {
  delete mcl20Proc;
  delete chanin;
  delete chanout;
}
```

Listing D-14 *Process class MotorControllerLeftProcess.*

The `run()` method simply performs the 20-process in an infinite loop as specified by the composition diagram in Figure 6-11. The destructor `~MotorControllerLeftProcess` deletes all objects and processes that were created by its constructor.

D.6.3 MotorControllerLeft20Process

The 20-process `mcl20Proc` is dedicated to invoking methods on the sub-model object. Its process class `MotorControllerLeft20Proces` is shown in Listing D-15. The sub-model object `mclSubmodel` is created and initialized. The alternative construct as specified in Figure 6-11 is declared by the constructor and performed by the `switch(..) {..}` clause in the `run()` method.

```
/** Construct MotorControllerLeft20Process. **/
MotorControllerLeft20Process::MotorControllerLeft20Process(
ChannelIn<double> **chanin, ChannelOut<double> **chanout) {
  this->chanin  = chanin;
  this->chanout = chanout;
  this->u= (double *) malloc (2 * sizeof (double));
  this->y= (double *) malloc (1 * sizeof (double));

  //--- Create and initialize Submodel
  mclSubmodel = new MotorControllerLeft;
  mclSubmodel->Initialize(this->u, this->y, 0);

  //--- Create Alternative construct
```

```
  prialt = new PriAlternative(chanin[0], chanin[1], NULL);
}

/** Process body. **/
void MotorControllerLeft20Process::run(void) {
  switch(prialt->select()) {
    case 0: // on setpoint
      chanin[0]->read(&(u[0]));
      break;
    case 1: // on feedback
      chanin[1]->read(&(u[1]));
      mclSubmodel->Calculate (u, y);
      chanout[0]->write(&(y[0]));
      break;
  }
}

/** Destruct MotorControllerLeft20Process. **/
MotorControllerLeft20Process::~MotorControllerLeft20Process(void) {
  free(u);
  free(y);
  delete mclSubmodel;
  delete prialt;
}
```

Listing D-15 *Process class MotorControllerLeft20Process.*

D.7 JIWY Implementation

The implementation of the top network builder for JIWY and its sub-processes are given in this section.

D.7.1 Top network builder

The main source code is derived from the context diagram, see Figure 6-16 and Figure 6-18. The source code is given in Listing D-16. Some parts are discussed separately. The channel declarations are discussed first.

```
int main(void) {
  //--- external channels declarations
```

```
  // Analog joystick
  AnalogJoystick *joystick = new AnalogJoystick();
  ChannelIn<double> *joystick_horizontal = new AnalogJoystickX(joystick);
  ChannelIn<double> *joystick_vertical   = new AnalogJoystickY(joystick);
  ChannelOut<int>   *joystick_buttons    = \
    new AnalogJoystickButtons(joystick);

  //--- DAQSTC, National Instruments 6024E IO Board
  DAQSTC *daqstc = new DAQSTC();
  daqstc->Initialise();

  // Analog Output
  daqstc->SetAOTM(AOTM::Primary, AOTM::CPUDriven);
  ChannelOut<double> *control_horizontal = daqstc->GetDAC(DAC::DAC0);
  ChannelOut<double> *control_vertical   = daqstc->GetDAC(DAC::DAC1);

  // Two counters for sensors
  ChannelIn<double> *feedback_horizontal = \
    daqstc->GetCounter(GPC::Counter0);
  ChannelIn<double> *feedback_vertical   = \
    daqstc->GetCounter(GPC::Counter1);

  //--- internal channels declarations ('chan' prefix to variable names)
        Figure 6-18
  Channel<double> leftmax_h  = new ChannelVar<double>;
  Channel<double> rightmax_h = new ChannelVar<double>;
  Channel<double> leftmax_v  = new ChannelVar<double>;
  Channel<double> rightmax_v = new ChannelVar<double>;

  //--- processes declarations
  ... see Listing D-17.

  //--- compositional constructs declarations
  ... see Listing D-18.

  //--- set timed events
  ... see Listing D-19.

  //--- run the process
  par->run();

  //--- delete all instances
  ...
}
```

Listing D-16 *Top Network Builder: declaration of internal and external channels.*

Firstly, the external channels (i.e. link drivers) are declared. The communication relationships `leftmax_h`, `rightmax_h`, `leftmax_v`, and `rightmax_v` are prefixed by `chan`. These become `ChannelVar` channels which are unblocking channels representing synchronized shared variables. These special channels are useful for communication between processes in a sequential composition. `ChannelVar` is a sub-class of `Channel` and therefore a `ChannelVar` can replace a channel or a `ChannelVar` can be replaced by another channel in code; they can be intertwined because they have the same channel-interface. The process instance interface does not specify state handling methods, otherwise state handling methods (open arrows) could have been used instead of `ChannelVar` channels. The `ChannelVar` channels are used to illustrate that buffered data-channels are not restricting to strictly parallel relationships in case processes are reused.

The processes are declared in the main source file as follows:

```
ControlHorizontal *motionControlH =   new ControlHorizontal(
     joystick_horizontal, joystick_buttons, feedback_horizontal,
     control_horizontal, leftmax_h, rightmax_h);
ControlVertical   *motionControlV =   new ControlVertical(
     joystick_vertical, joystick_buttons, feedback_vertical,
     control_vertical, leftmax_v, rightmax_v);
VelocityControlLeftHorizontalProcess  *alignLH = new
     VelocityControlLeftHorizontalProcess(feedback_horizontal,
     control_horizontal, leftmax_h);
VelocityControlRightHorizontalProcess *alignRH = new
     VelocityControlRightHorizontalProcess(feedback_horizontal,
     control_horizontal, rightmax_h);
VelocityControlLeftVerticalProcess    *alignLV = new
     VelocityControlLeftVerticalProcess(feedback_vertical,
     control_vertical, leftmax_v);
VelocityControlRightVerticalProcess   *alignRV = new
     VelocityControlRightVerticalProcess(feedback_vertical,
     control_vertical, rightmax_v);
HomingHorizontal *homingH = new HomingHorizontal(feedback_horizontal,
     control_horizontal, leftmax_h, rightmax_h);
HomingVertical   *homingV =  new HomingVertical(feedback_vertical,
     control_vertical, leftmax_v, rightmax_v);
```

Listing D-17 *Declaration of processes*.

The arguments correspond to the port names of the process-interface for each process. This is consistent with the communication diagram of the process architecture.

The compositional construct as specified in Figure 6-18 is coded with CTC++ in Listing D-18.

```
Sequential *seq_h = new Sequential(alignLH, alignRH, motionControlH,
     homingH, NULL);
Sequential *seq_v = new Sequential(alignLV, alignRV, motionControlV,
     homingV, NULL);
Parallel   *par   = new Parallel(seq_h, seq_v, NULL);
```

Listing D-18 *Declaration of compositional constructs.*

The environmental process is set to a particular timing so that it will accept events at certain moment in time with periodical intervals. The sampling time for the horizontal control loop T_{sh} and for the vertical control loop T_{sv} are specified in the communication diagram on the timed external channels. See `@Tsh` for the external channels `feedback_horizontal`, `joystick_horizontal`, and `control_horizontal` in Figure 6-16. For example, we tested with T_{sh} = 100000 µs (=10 Hz) and T_{sv} = 10000 µs (=100 Hz).The code fragment in Listing D-19 must be include before the `par->run()` in the main source file. See Listing D-18.

```
System::at(feedback_horizontal, starttime, Tsh);
System::at(joystick_horizontal, starttime, Tsh);
System::at(control_horizontal, starttime, Tsh);
System::at(feedback_vertical, starttime, Tsv);
System::at(joystick_vertical, starttime, Tsv);
System::at(control_vertical, starttime, Tsv);
```

Listing D-19 *Timing initialization part.*

The `starttime` is set to a value that is long enough for the first events to occur after all initializations has completed. We set the `starttime` to 10 ms.

The timed channels will throw a `TimeoutException` when processes do not arrive on the channel before the environmental process is willing to engage in the event. In this case, processes will terminate unsuccessfully and exceptions must be handled. JIWY is hard real-time and the PC is

fast enough so that timeout-exceptions did not occur. We omit exception handling, which is further discussed in (Engelen, 2004). Exception handling is an ongoing topic for further research.

The entire process will be executed by invoking `par->run()`. Thus before the `run()` method of the top-level construct is invoked, all processes, channels, constructs, timing, and objects have been prepared. Once this `run()` method is invoked, the real-time run bodies of the processes will be performed accordingly to the compositional relationships in the CSP diagrams. After the top `run()` method terminates it can be invoked again or the declared entities can be deleted.

D.7.2 Motion controller process

Process `motionControlH` is depicted in Figure 6-19 and Figure 6-20. The process-interface of `motionControlH` is specified by the constructor in Listing D-20. This constructor assigns its ports to the ports of the child processes. The internal channels for `setpoint`, `stop`, and `zero` are declared. Also the 20-process `servoHorizontal` is declared and connected to the `chanin` and `chanout` channel arrays.

```
PositionControllerHorizontal::PositionControllerHorizontal(
  ChannelIn<double> *joystick_axis, ChannelIn<double> *joystick_buttons,
  ChannelIn<double> *feedback, ChannelOut<double> *control,
  ChannelIn<double> *leftmax, ChannelIn<double> *rightmax) {

  //--- create channel-input array and a channel-output array
  this->chanin  = new ChannelIn<double> * [4];
  this->chanout = new ChannelOut<double> * [2];

  chanin[0]  = leftmax;
  chanin[1]  = rightmax;
  chanin[2]  = feedback;
  chanin[3]  = joystick_axis;
  chanout[0] = NULL;
  chanout[1] = control;

  //--- create the 20-process
  servoHorizontal = new PositionControllerHorizontal20Process(
       chanin, chanout);
```

```
  //--- set up the alternative construct process
  prialt = new PriAlternative();
  alt->add(joystick_buttons); // first preference
  alt->add(feedback);         // second preference
}

void PositionControllerHorizontal::run(void) {
  int status = 0;
  double zero = 0.0;
  do {
    switch(prialt->select()) {
      case 0: //--- controller process
           servoHorizontal->run();
        break;
      case 1: //--- stop button process
        joystick_buttons->read(&status);
        break;
    }
  } while (status != 2);
  //--- release output by setting to zero
  control->write(&zero);
}

PositionControllerHorizontal::~PositionControllerHorizontal() {
  ...destruct all objects and processes
}
```

Listing D-20 *Process class PositionControllerHorizontal.*

Here `status` and `zero` are variables. This is determined by the ?- and !- processes in Figure 6-19. Because process `servoHorizontal` is a 20-process the channels must be mapped on `chanin[]` and `chanout[]` arrays. This process is generated by 20-sim and the C++ template. The `switch(alt->select()) {}` clause in the `run()` body performs the alternative construct. The alternative construct chooses between `feedback` and `joystick_buttons` with preference to `joystick_buttons` so that one can always stop the controller.

D.7.3 Alignment controller process

The alignment process `alignH` in Figure 6-21 and Figure 6-22 is coded in Listing D-21. Process `vleftHorizontal` is the velocity controller of the alignment process.

```
VelocityControlLeftHorizontal::VelocityControlLeftHorizontal(
  ChannelIn<double> *feedback, ChannelOut<double> *control,
  ChannelIn<double> *max) {

  //--- create variable channel
  stop_ = new ChannelVar<double>; //!!

  //--- global variables
  double stop;

  //--- create channel-input array and a channel-output array
  this->chanin  = new ChannelIn<double> * [1];
  this->chanout = new ChannelOut<double> * [3];  //!! [3]->[2]
  chanin[0]  = feedback;
  chanout[0] = control;
  chanout[1] = max;
  chanout[2] = stop_;             //!!

  //--- create the 20-process
  vleftHorizontal = new VelocityControlLeftHorizontal20Process(
       chanin, chanout);
}

void VelocityControlLeftHorizontal::run(void) {
  do {
    vleftHorizontal->run();
    stop_->read(&stop);           //!!
  } while (!stop);
}

VelocityControlLeftHorizontal::~VelocityControlLeftHorizontal() {
  ...destruct all objects and processes
}
```

Listing D-21 *Run body of alignment process.*

Here, `stop` is a variable with a channel-interface that enables variable sharing between sequential processes. The channel name is suffixed with a '_' to distinguish between the channel `stop_` and the variable `stop`. This is not an exact translation of the CSP diagram. The ChannelVar implementation should be replaced by state handling methods as described in Appendix D.5. The C++ templates for 20-sim have not yet been adapted to support state handling methods. We left the ChannelVar implementation in here to illustrate that buffered data-channels can be

used to connect processes that are performed in sequence. This increases the reusability of existing processes in process architectures.

The use of state handling methods would include the following code to the previous code in Listing D-21. The `//!!` Statements must be removed or changed.

```
public void setStop(int stop) {
  this.stop = stop;
}
```

D.7.4 Homing controller process

The homing process steers the joint to its centre position. Subsequently, the motors are turned off. The CSP diagram in Figure 6-23 and Figure 6-24 is coded in Listing D-22.

```
HomingHorizontal::HomingHorizontal(ChannelIn<double> *feedback,
  ChannelOut<double> *control, ChannelIn<double> *leftmax,
  ChannelIn<double> *rightmax) {

  //--- create variable channels
  setpoint_ = new ChannelVar<double>(0.0);
  stop_ = new ChannelVar<double>;

  //--- create channel-input array and a channel-output array
  this->chanin  = new ChannelIn<double> * [4];
  this->chanout = new ChannelOut<double> * [2];
  chanin[0]  = leftmax;
  chanin[1]  = rightmax;
  chanin[2]  = feedback;
  chanin[3]  = setpoint_;
  chanout[0] = stop_;
  chanout[1] = control;

  //--- create the 20-process
  homingHorizontal = new PositionControllerHorizontal20Process(
       chanin, chanout);
}

void HomingHorizontal::run(void) {
  double zero = 0.0;
  do {
```

```
    homingHorizontal->run();
    stop_->read(&stop);
  } while (!stop);
  //--- release output by setting to zero
  control->write(&zero);
}

HomingHorizontal::~HomingHorizontal() {
  ...destruct all objects and processes
}
```

Listing D-22 *Constructor and run body of homing process.*

Here, `setpoint` and `stop` are variables passed as channels (respectively named `setpoint_` and `stop_`) to the `homingHorizontal` process.

APPENDIX E

Alting

E.1 Introduction

The word *alting* is frequently used as a verb representing the operation of the alternative process. The processes that are connected to the alternative process via a separate channel are called *alting processes*. The alternative process chooses one guarded process out of many guarded processes that can communicate via a channel with its alting process. On two or more alting processes that are willing to communicate with guarded processes, the alternative process will select one guarded process that is able to commit in communication. In software, the choice or the decision policy is prioritized.

The decision policy, semantics, and properties of alting for real-time systems are discussed in this appendix. In Section E.2 the notion of fair alting is analyzed and illustrates the decision policy. Section E.4 discusses the semantics of preference alting in contrast to resolute alting. Preference alting allows for optimal performance between compositions of ALT/PRIALT and PAR/PRIPAR.

E.2 Fair alting

In order to understand the fairness of an ALT construct we consider the ideal behaviour of a shared any-to-any channel. We will motivate in this

appendix that the behaviour of a fair ALT should be equal to the behaviour of an any-to-any channel. Reversely, the technique that is used to implement an any-to-any channel can be used to implement a fair ALT. This shows that the any-to-any channel have lots in common with fairly alting and visa versa. This gives confidence that the choice of fairness policy and implementation for the ALT (and PRIALT) is a good choice. The notion of preference priorities allows us to distinguish between an ALT and a PRIALT in a reasonable way that is intuitive for software engineering.

E.3 Any-to-any channel

An any-to-any channel is a CSP channel that can be safely used by multiple reader processes and multiple writer processes. Only one pair of reader and writer can communicate one at the time.

Multiple writers or readers should claim their peer-end of the any-to-any channel in a fair fashion according to a mixed policy of a first-come-first-served policy between equally-prioritized processes and a highest-priority-first policy between unequally-prioritized processes. We consider it fair that a process of higher priority should be served before a process with a lower priority.

A semaphore construct with a prioritized queue according to the above mention policies at the peer-end of a channel can fairly synchronize between multiple processes and takes their prioritized parallel relationships into account. The implementation of a prioritized queuing mechanism is not difficult. The mixed policy and implementation of a semaphore is the same to the behaviour and implementing of the fair ALT.

The any-to-any channel can be described by two alternative processes. For example, consider an any-to-any channel c with at one end three writer processes P, Q, and S, and at the other end three reader processes X, Y, and Z. It may be obvious that the above described mixed policy a

fair selection between pairs of a writer and reader processes. This system can be described as

$$SYSTEM = W \underset{\{c.T\}}{\|} R \text{ with } W = P \| Q \| S \text{ and } R = X \| Y \| Z .$$

where all processes are willing to communicate over channel c. The processes are defined as follows

$$\begin{array}{lll} P = c!a_1 \rightarrow P' & & X = c?x \rightarrow X'(x) \\ Q = c!a_2 \rightarrow Q' & \text{and} & Y = c?y \rightarrow Y'(y) \\ S = c!a_3 \rightarrow S' & & Z = c?z \rightarrow Z'(z) \end{array}$$

Process R can be described by a choice construct instead of a parallel construct of multiple reader processes.

$$c?x \rightarrow \left(X'(x) \| \left(\begin{array}{l} \left(c?y \rightarrow \left(Y'(y) \| \left(c?z \rightarrow Z'(z) \right) \right) \right) \square \\ \left(c?z \rightarrow \left(\left(c?y \rightarrow Y'(y) \right) \| Z'(z) \right) \right) \end{array} \right) \right) \square$$

$$c?y \rightarrow \left(Y'(y) \| \left(\begin{array}{l} \left(c?x \rightarrow \left(X'(x) \| \left(c?z \rightarrow Z'(z) \right) \right) \right) \square \\ \left(c?z \rightarrow \left(\left(c?x \rightarrow X'(x) \right) \| Z'(z) \right) \right) \end{array} \right) \right) \square$$

$$c?z \rightarrow \left(Z'(z) \| \left(\begin{array}{l} \left(c?x \rightarrow \left(X'(x) \| \left(c?y \rightarrow Y'(y) \right) \right) \right) \square \\ \left(c?y \rightarrow \left(\left(c?x \rightarrow X'(x) \right) \| Y'(y) \right) \right) \end{array} \right) \right)$$

Process W can be described by a choice construct instead of a parallel construct of multiple writer processes.

$$c!a_1 \rightarrow \left(P' \| \left(\left(c!a_2 \rightarrow \left(Q' \| \left(c?a_3 \rightarrow S' \right) \right) \right) \square \left(c!a_3 \rightarrow \left(\left(c!a_2 \rightarrow Q' \right) \| S' \right) \right) \right) \right) \square$$

$$c!a_2 \rightarrow \left(Q' \| \left(\left(c!a_1 \rightarrow \left(P' \| \left(c!a_3 \rightarrow S' \right) \right) \right) \square \left(c!a_3 \rightarrow \left(\left(c!a_1 \rightarrow P'(x) \right) \| S' \right) \right) \right) \right) \square$$

$$c?a_3 \rightarrow \left(S' \| \left(\left(c!a_1 \rightarrow \left(P' \| \left(c?a_2 \rightarrow Q' \right) \right) \right) \square \left(c!a_2 \rightarrow \left(\left(c!a_1 \rightarrow P' \right) \| Q' \right) \right) \right) \right)$$

The choice between the processes is arbitrary and allows for input guards and output guards. In practice, an input guard and output guard will never meet each other and the choice is prioritized to ensure fairness or

unfairness. The notion of preference priorities is the most appropriate solution.

Fairly alting on a any-to-any channel

The alting policy between alting processes with equally-prioritized parallel relationships should based on their arrival time, as in

$$\left\{(a \to V) \middle| (a \to V) \square (b \to W) \wedge t_a < t_b \right\}$$

Here, t_a and t_b are timestamp of the alting processes willing to engage in respectively event a and b.

In case of an any-to-any channel there should be a fair selection between the multiple readers and a fair selection between the multiple writers. For example, Q is willing to communicate before P and P is willing to communicate before S then this equals:

$$c!a_2 \to \left(Q' \| \left(c!a_1 \to \left(P' \| \left(c!a_3 \to S'\right)\right)\right)\right)$$

At the reader side of the any-to-any channel, consider Z is willing to communicate before X and X is willing to communicate before Y. This equals:

$$c?z \to \left(Z'(z) \| \left(c?x \to \left(X'(x) \| \left(c?y \to Y'(y)\right)\right)\right)\right)$$

Consequently, *SYSTEM* will behave as

$$SYSTEM = \left(\begin{array}{l} \left(c!a_2 \to \left(Q' \| \left(c!a_1 \to \left(P' \| \left(c!a_3 \to S'\right)\right)\right)\right)\right) \| \\ \left(c?z \to \left(Z'(z) \| \left(c?x \to \left(X'(x) \| \left(c?y \to Y'(y)\right)\right)\right)\right)\right) \end{array} \right)$$

After a_1, a_2, and a_3, have been offered then *SYSTEM* continues as:

$$SYSTEM = \left(Q' \| P' \| S'\right) \underset{\{c.x\}}{\|} \left(Z'(a_2) \| X'(a_1) \| Y'(a_3)\right)$$

If we take the prioritized parallel relationships between processes into account then we apply the following refinement:

Two threads that claim a channel *a* (i.e. one at the reader side and one at the writer side of the channel) participate in one communication event *a*. Property *a.thread* is one of the two threads with the highest priority (i.e. lowest *a.thread.priority*) which is defined as

$$a.thread = \left\{ a.thread_1 \middle| \begin{array}{l} a.thread_1.priority \le a.thread_2.priority; \\ priority \in \mathbb{N}; thread_1, thread_2 \in THREAD; a \in \sum \end{array} \right\}$$

The priority values are assigned to $thread_i.priority$ by the equally-prioritized and unequally-prioritized parallel relationships between processes.

The function *pri*(*a*) returns the priority index of the executing thread with the highest priority engaging in communication event *a*. Function *pri*(*a*) is defined as

$$pri(a) = \left\{ maxpriority - a.thread.priority \middle| \begin{array}{l} \forall b \in \sum \bullet maxpriority \ge b.thread.priority; \\ maxpriority, priority \in \mathbb{N}; \\ thread \in THREAD; a \in \sum \end{array} \right\}$$

with *THREAD* as the non-empty set of all threads and Σ as the set of all communication events. The constant *maxpriority* is the highest priority value (i.e. the lowest priority) in the set of possible events. Here, *maxpriority* can be equal to the total of prioritized parallel relationships + 1.

The comparison of priorities between two events *a* and *b* in a (equally-prioritized) parallel relationship of processes, executing at equal priorities, is defined as

$$\left\{ pri(a) = pri(b) \middle| \begin{pmatrix} a.thread.priority = b.thread.priority \wedge \\ a.thread \ne b.thread \wedge a \ne b \end{pmatrix} \vee (a = b); a, b \in \sum \right\}$$

The comparison of priorities between two events a and b in a (unequally) prioritized parallel relationship of processes, executing at different priorities, is defined as

$$\left\{ pri(a) > pri(b) \middle| \begin{pmatrix} a.thread.priority < b.thread.priority \wedge \\ a.thread \neq b.thread \wedge a \neq b \end{pmatrix}; a,b \in \Sigma \right\}$$

A fair ALT is a preference ALT with operator $\tilde{\Box}$, which semantics is defined as

$$(a \to P)\tilde{\Box}(b \to Q) \Rightarrow \left\{ (a \to P) \middle| \begin{matrix} (a \to P)\tilde{\Box}(b \to Q) \wedge \\ \begin{pmatrix} (t_a < t_b \wedge pri(a) = pri(b)) \vee \\ pri(a) > pri(b) \end{pmatrix} \end{matrix} \right\}$$

The occurrence times t_a and t_b of the events a and b in a deterministic environment can never be the same, thus $t_a \neq t_b$. Preference ALT $\tilde{\Box}$ is a refinement of a deterministic $\Box$.

Operator $\tilde{\tilde{\Box}}$ is the prioritized version of $\tilde{\Box}$, which is a possible valid refinement for $\tilde{\Box}$. The semantics of $\tilde{\tilde{\Box}}$ is defined as

$$(a \to P)\tilde{\tilde{\Box}}(b \to Q) \Rightarrow \left\{ (a \to P) \middle| \begin{matrix} (a \to P)\tilde{\tilde{\Box}}(b \to Q) \wedge \\ pri(a) \geq pri(b) \end{matrix} \right\}$$

and

$$(a \to P)\tilde{\tilde{\Box}}(b \to Q) \Rightarrow \left\{ (b \to Q) \middle| \begin{matrix} (a \to P)\tilde{\tilde{\Box}}(b \to Q) \wedge \\ pri(a) < pri(b) \end{matrix} \right\}$$

These preference choice operators $\tilde{\Box}$ and $\tilde{\tilde{\Box}}$ are further discussed in Section E.4.

E.4 Semantics of alting

In the previous section the semantics of the preference choice operators have been discussed. In this section these semantics are put in contrast to the classical choice operators. Particularly, the definitions and properties of resolute and preference alting are described. The semantics of preference alting given in this section is a wish list of behaviour that is most applicable for software engineering. Preference alting is an extension of the semantics of resolute alting. This behaviour provides an optimal scheduling policy that is adaptive to the presence of surrounding prioritized parallel relationships between processes. Preference alting and any-to-any channels share similar properties, desired behaviours, and consequently a similar implementation.

Definitions of alting

The (symmetric) choice operator □ is defined in the following manner:

$$(x : A \to P(x)) \square (y : B \to Q(y)) \stackrel{def}{=} z : A \cup B \to R(z)$$

where

$$\begin{aligned} R(z) &= P(z) && \text{if } z \in A - B \\ &= Q(z) && \text{if } z \in B - A \\ &= P(z) \sqcap Q(z) && \text{if } z \in A \cap B \end{aligned}$$

If the first event offered by P and Q are disjoint, the new process behaves according to ordinary choice '|', otherwise according to the internal choice operator ⊓ that is characterized by the fact that the environment cannot influence the choice between the processes $P(z)$ and $Q(z)$. The process $P(z) \sqcap Q(z)$ behaves either as $P(z)$ or as $Q(z)$. The choice between them is made internally and, therefore, it is not possible to predict which one of the process $P(z)$ or $Q(z)$ may emerge from $P(z) \sqcap Q(z)$. The non-deterministic behaviour of process $P(z) \sqcap Q(z)$ is impossible to realize in a deterministic environment, say on a sequential processor.

We introduce the resolute choice operators $\dot{\square}$ and $\dot{\tilde{\square}}$ which are deterministic versions of the theoretical choice operator $\square$. Both resolute choice operators are respectively equally-prioritized and unequally-prioritized.

The operator $\dot{\square}$ is defined in the following manner:

$$(x : A \to P(x)) \dot{\square} (y : B \to Q(y)) \stackrel{def}{=} z : A \cup B \to R(z)$$

where

$$\begin{array}{lcll} R(z) & = & P(z) \text{ and } i{=}1 & \text{if } z \in A - B \\ & = & Q(z) \text{ and } i{=}0 & \text{if } z \in B - A \\ & = & \left.\begin{array}{l} \text{if } (i{=}0)\ P(z) \\ \text{if } (i{=}1)\ Q(z) \\ i = (i{+}1) \text{ modulo } 2 \end{array}\right\} & \text{if } z \in A \cap B \end{array}$$

with $i \in \mathbb{Z}$ and i is initially 0.

The (asymmetric) choice operator $\dot{\tilde{\square}}$ is a particular implementation of the $\tilde{\square}$ operator; similarly the $\tilde{\tilde{\square}}$ is a different implementation of $\tilde{\square}$. These resolute choice operators are restricted in that they do not allow propagation of priority over events. The operator $\dot{\tilde{\square}}$ is defined in the following manner:

$$(x : A \to P(x)) \dot{\tilde{\square}} (y : B \to Q(y)) \stackrel{def}{=} z : A \cup B \to R(z)$$

where

$$\begin{array}{lcll} R(z) & = & P(z) & \text{if } z \in A - B \\ & = & Q(z) & \text{if } z \in B - A \\ & = & P(z) & \text{if } z \in A \cap B \end{array}$$

The preference symmetric and asymmetric choices are denoted by respectively the operators $\tilde{\square}$ and $\tilde{\tilde{\square}}$, which must know the priorities of

the alting processes that are committed in communication with the alternative process. If we refer to the priority of a process we mean implicitly the priority of the thread of control that performs event handling within a process. This property holds for a process which has no knowledge about its priority. These definitions allow propagation of priorities over events.

The operator $\tilde{\Box}$ is defined in the following manner:

$$(x : A \to P(x)) \tilde{\Box} (y : B \to Q(y)) \stackrel{def}{=} z : A \cup B \to R(z)$$

where

$$\begin{aligned}
R(z) &= P(z) && \text{if } z \in A - B \\
&= Q(z) && \text{if } z \in B - A \\
&= P(z) && \text{if } z \in A \cap B \text{ and } pri(x) > pri(y) \\
&= Q(z) && \text{if } z \in A \cap B \text{ and } pri(x) < pri(y) \\
&= P(z) && \text{if } z \in A \cap B \text{ and } pri(x) = pri(y) \text{ and } t_a < t_b \\
&= Q(z) && \text{if } z \in A \cap B \text{ and } pri(x) = pri(y) \text{ and } t_a > t_b
\end{aligned}$$

The operator $\tilde{\tilde{\Box}}$ is defined as

$$(x : A \to P(x)) \tilde{\tilde{\Box}} (y : B \to Q(y)) \stackrel{def}{=} z : A \cup B \to R(z)$$

where

$$\begin{aligned}
R(z) &= P(z) && \text{if } z \in A - B \\
&= Q(z) && \text{if } z \in B - A \\
&= P(z) && \text{if } z \in A \cap B \text{ and } pri(x) > pri(y) \\
&= Q(z) && \text{if } z \in A \cap B \text{ and } pri(x) < pri(y) \\
&= P(z) && \text{if } z \in A \cap B \text{ and } pri(x) = pri(y)
\end{aligned}$$

E.5 Properties of alting

In this section we will give some properties of alting for the operators $\Box$, $\overleftarrow{\Box}$, $\dot{\Box}$, $\dot{\overleftarrow{\Box}}$, $\tilde{\Box}$ and $\tilde{\overleftarrow{\Box}}$. On the basis of these properties we can observe some constructive parallels. Preference alting offers a great deal of fairness and allows propagation of external priorities over events.

The operator $\Box$ is idempotent, commutative and associative.

$$(a \to A) \Box (a \to A) = (a \to A) \qquad \text{<idempotent>}$$

$$(a \to A) \Box (b \to B) = (b \to B) \Box (a \to A) \qquad \text{<commutative>}$$

$$(a \to A) \Box \big((b \to B) \Box (c \to C)\big) = \big((a \to A) \Box (b \to B)\big) \Box (c \to C) \qquad \text{<associative>}$$

Operator $\overleftarrow{\Box}$ is idempotent and associative, but not commutative.

$$(a \to A) \overleftarrow{\Box} (a \to A) = (a \to A) \qquad \text{<idempotent>}$$

$$(a \to A) \overleftarrow{\Box} (b \to B) \neq (b \to B) \overleftarrow{\Box} (a \to A) \qquad \text{<not commutative>}$$

$$(a \to A) \overleftarrow{\Box} \big((b \to B) \overleftarrow{\Box} (c \to C)\big) = \big((a \to A) \overleftarrow{\Box} (b \to B)\big) \overleftarrow{\Box} (c \to C) \qquad \text{<associative>}$$

Although the operator $\Box$ is commutative it is not surprising that operator $\overleftarrow{\Box}$ is not commutative, because $\overleftarrow{\Box}$ is uni-directive and $\Box$ is bi-directive.

Operator $\dot{\Box}$ is idempotent, associative and partially commutative.

$$(a \to A) \dot{\Box} (a \to A) = (a \to A) \qquad \text{<idempotent>}$$

$$(a \to A) \dot{\Box} (b \to B) \approx (b \to B) \dot{\Box} (a \to A) \qquad \text{<partially commutative>}$$

$$(a \to A) \dot{\Box} \big((b \to B) \dot{\Box} (c \to C)\big) = \big((a \to A) \dot{\Box} (b \to B)\big) \dot{\Box} (c \to C) \qquad \text{<associative>}$$

Operator $\dot{\Box}$ is partially commutative. If both events a and b are ready then $(a \to A) \dot{\Box} (b \to B)$ will initially select process $(a \to A)$ and $(b \to B) \dot{\Box} (a \to A)$ will initially select $(b \to B)$. This is because by the fact that the search for ready guards starts at 0. The $\dot{\Box}$ is not entirely equal to $\Box$ with respect to this property.

Operator $\dot{\Box}$ is idempotent and associative, but not commutative.

$$(a \to A)\dot{\Box}(a \to A) = (a \to A) \quad \text{<idempotent>}$$

$$(a \to A)\dot{\Box}(b \to B) \neq (b \to B)\dot{\Box}(a \to A) \quad \text{<not commutative>}$$

$$(a \to A)\dot{\Box}\big((b \to B)\dot{\Box}(c \to C)\big) = \big((a \to A)\dot{\Box}(b \to B)\big)\dot{\Box}(c \to C) \quad \text{<associative>}$$

Operator $\dot{\Box}$ is not commutative. If both events *a* and *b* are ready then the left-side process will be selected. This is equivalent to $\overleftarrow{\Box}$.

Operator $\tilde{\Box}$ is idempotent, commutative and associative, and thus its properties are equal to the $\Box$ operator.

$$(a \to A)\tilde{\Box}(a \to A) = (a \to A) \quad \text{<idempotent>}$$

$$(a \to A)\tilde{\Box}(b \to B) = (b \to B)\tilde{\Box}(a \to A) \quad \text{<commutative>}$$

$$(a \to A)\tilde{\Box}\big((b \to B)\tilde{\Box}(c \to C)\big) = \big((a \to A)\tilde{\Box}(b \to B)\big)\tilde{\Box}(c \to C) \quad \text{<associative>}$$

Operator $\tilde{\dot{\Box}}$ is idempotent, associative and partially commutative.

$$(a \to A)\tilde{\dot{\Box}}(a \to A) = (a \to A) \quad \text{<idempotent>}$$

$$(a \to A)\tilde{\dot{\Box}}(b \to B) = (b \to B)\tilde{\dot{\Box}}(a \to A) \text{ for } pri(a) \neq pri(b) \quad \text{<commutative>}$$

$$(a \to A)\tilde{\dot{\Box}}(b \to B) \neq (b \to B)\tilde{\dot{\Box}}(a \to A) \text{ for } pri(a) = pri(b) \quad \text{<not commutative>}$$

$$(a \to A)\tilde{\dot{\Box}}\big((b \to B)\tilde{\dot{\Box}}(c \to C)\big) = \big((a \to A)\tilde{\dot{\Box}}(b \to B)\big)\tilde{\dot{\Box}}(c \to C) \quad \text{<associative>}$$

If *pri*(*a*) = *pri*(*b*) then the preference choice operator shows similarities with the resolute choice operator:

$$\big\{(a \to A)\tilde{\Box}(b \to B) \mid pri(a) = pri(b)\big\} \overset{stat}{\Leftrightarrow} (a \to A)\dot{\Box}(b \to B)$$

$$\big\{(a \to A)\tilde{\dot{\Box}}(b \to B) \mid pri(a) = pri(b)\big\} \Leftrightarrow (a \to A)\dot{\Box}(b \to B)$$

The first is based on a *cyclic-indexing* policy. The left hand-side and right hand-side are statistically equivalent, but can differ in the first choice.

The second is based on a first-come-first-served policy, which left hand-side and right hand-side are equivalent.

If $pri(a) \neq pri(b)$ then the preference symmetric choice operator shows similarities with the preference asymmetric choice operator:

$$\{(a \to A)\tilde{\Box}(b \to B) \mid pri(a) \neq pri(b)\} \Leftrightarrow$$
$$\{(a \to A)\tilde{\tilde{\Box}}(b \to B) \mid pri(a) \neq pri(b)\} \Leftrightarrow$$
$$\{(b \to B)\tilde{\tilde{\Box}}(a \to A) \mid pri(a) \neq pri(b)\}$$

Any difference in priority, e.g. $pri(a) < pri(b)$, turns the preference asymmetric choice operator into a specific resolute asymmetric choice operator:

$$\{(a \to A)\tilde{\tilde{\Box}}(b \to B) \mid pri(a) < pri(b)\} \Leftrightarrow$$
$$\{(b \to B)\tilde{\tilde{\Box}}(a \to A) \mid pri(a) < pri(b)\} \Leftrightarrow$$
$$(b \to B)\dot{\Box}(a \to A) = (b \to B)$$

when both a and b are offered.

APPENDIX F

Solving priority conflicts with buffered channels

F.1 Introduction

Communication with unbuffered data channels between higher-priority processes and lower-priority processes can cause priority inversion problems. The effect is that a higher-priority process, which gets blocked on a data channel while waiting for a lower-priority process to read from the data channel, will be pulled down to the priority of the lower-priority process. This is in disagreement with the prioritized parallel relationship. The result is a performance penalty that can cause the higher-priority process not to meet its deadline. This problem is due to a priority conflict in the design.

In Section 512.5.0, a technique is described for finding priority conflicts in CSP diagrams. In this appendix, this technique is used to show that a design that is not priority conflict-free, as a result of a data channel, can become priority conflict-free by using a buffered data channel. The technique is applicable for complex patterns.

F.2 Buffered data channels solve priority conflicts

Figure F-1a shows two communicating processes P_1 and P_2 via rendezvous data channel c. Process P_1 is executed at a higher priority than P_2 and process P_3 is executed with a priority somewhere in the middle. We assume that process P_3 will repeatedly wait on its channel otherwise P_2 will never be executed and this model makes no sense. Process P_3 can hold P_2 from executing and process P_1 will not be served by P_2. It is likely that process P_1 can not meet its deadline due to P_3 claiming the CPU. This priority inversion problem can be solved when a sub-sampling buffered process is used that decouples P_1 from P_2. Such a buffer process is depicted in Figure F-1b. A sub-sampling buffered data channel can replace the buffer process *Buf* and channels *a* and *b* which makes the diagram as simple as in Figure F-1b. This example is described in Section 5.4.

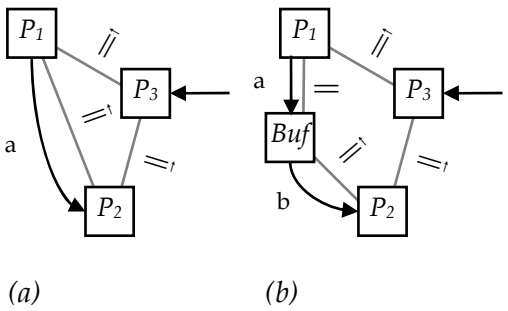


Figure F-1 *Priority conflict cause and solution:*
(a) Priority inversion problem caused by channel a,
(b) Solution via a buffer process..

In Figure F-1b, the data channels are labelled *a* and *b*, which also identify their communication events. The communication event *a* and *b* are alternating or in sequence, but they cannot happen at the same time. When *a* and *b* are alternating then the priority conflict should be check on engagement in *a* and in *b*. The buffer process *Buf* should not be full

otherwise it will be blocking P_1. We assume that *Buf* describes a sub-sampling buffer that can never become full.

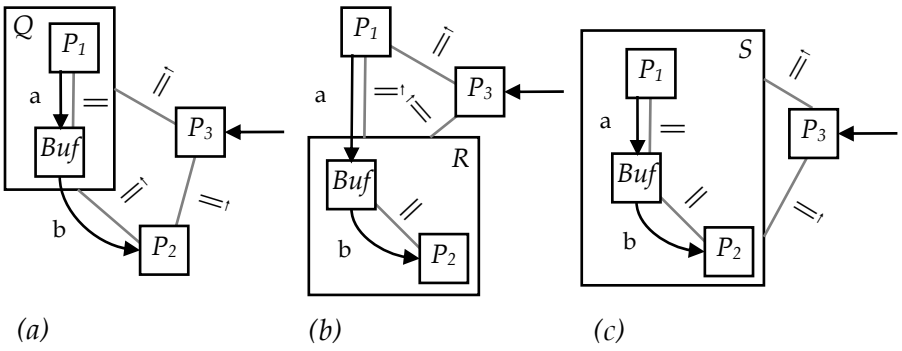


Figure F-2 *Proof of priority conflict-free using buffered communication:*
(a) communication process on channel a shows that the model is priority conflict-free
(b) communication process on channel b shows that the model is priority conflict-free
(c) communication process on a and b (in sequence) shows that the model is not priority conflict-free.

Figure F-2a shows the engagement in *a*. Processes P_1 and *Buf* form communication process *Q* at the instance of communication. Event *b* cannot occur at the same instance in time as event *a* and thus priority inversion problem does not apply on *b*. Channel *b* does not block P_1 and so *b* does not require buffering. We can omit *b*. All unequally-prioritized parallel relationships do not point in one direction of a cycle. Therefore, we conclude that this scenario is priority conflict-free.

Figure F-2b shows the engagement in *b*. Processes *Buf* and P_2 form communication process *R*. Similarly, event *a* cannot happen at the same instance of time as event *b*. At least one unequally-prioritized parallel relationship points in the opposite direction on the cycle. Thus, this scenario is priority conflict-free.

In case the buffer is a FIFO type of buffer and reaches its full state then the buffer will be blocking P_1 and the priority inversion problem rises. In this case, the scenario of communication events is sequential: first *b* then

a. This sequential communication comprises a single communication process. Figure F-2c shows that this causes a priority conflict since the unequally-prioritized parallel relationships point in one direction on the cycle between S and P_3.

Therefore it is important that the buffer does not reach the full state. Overwriting (or sub-sampling) is an important property of the buffered data channel to avoid the priority inversion problem.

In case the channel is directed from P_2 to P_1 a similar proof can be given as described above and a super-sampling buffered channel is essential.

APPENDIX G

Compositional analysis rule

G.1 Introduction

In this appendix, a *compositional analysis rule* is described that is useful for analyzing compositional CSP constructs, such as

- determining operators on hidden interrelationships derived from user-specified paths of relationships,
- writing ambiguous or unambiguous algebraic expressions,
- detecting specification conflicts.

The compositional analysis rule applies to triangular cycles in compositional diagrams.

G.2 Triangular cycles

A *triangular cycle* consists of three processes that are completely connected, e.g. every process is connected to each other process. Every pair of processes is connected by either a user-specified interrelationships or by hidden interrelationships.

For example, three processes on a user-specified path are part of a triangular cycle that is closed by a hidden interrelationship between the outer processes. See Figure G-1. The black lines are user-specified

interrelationships and the grey line is the visualized hidden interrelationship. The operator on the hidden interrelationship needs to be determined in order to make the composition unambiguous.

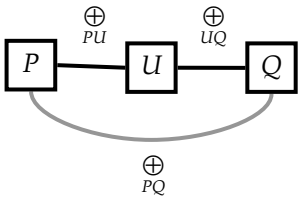


Figure G-1 *Triangular cycle with one hidden interrelationship.*

Let operator $\underset{PQ}{\oplus}$ represent a binary CSP operator between P and Q and $\overline{\underset{PQ}{\oplus}}$ its complement.

For each pair $\left(\underset{PQ}{\oplus}, \overline{\underset{PQ}{\oplus}}\right)$ applies

$$\left(\underset{PQ}{\oplus}, \overline{\underset{PQ}{\oplus}}\right) \in \left\{(\leftarrow,\rightarrow),(\|,\|),\left(\overleftarrow{\|},\overrightarrow{\|}\right),(\Box,\Box),\left(\overleftarrow{\Box},\overrightarrow{\Box}\right),\left(\overrightarrow{\Delta},\overleftarrow{\Delta}\right)\right\}$$

Operators are distinguished by their identifier below the symbol. The operator being inverted holds the same identifier. This way, operators can be distinguished by their identifiers, as in Figure G-1.

These operators are directional commutative

$$P \underset{PQ}{\oplus} Q = Q \overline{\underset{PQ}{\oplus}} P$$

For example, $P \| Q = Q \| P$, $P \Box Q = Q \Box P$, $P \rightarrow Q = Q \leftarrow P$, $P \overleftarrow{\|} Q = Q \overrightarrow{\|} P$, $P \overleftarrow{\Box} Q = Q \overrightarrow{\Box} P$, and $P \overrightarrow{\Delta} Q = Q \overleftarrow{\Delta} P$.

The number of the maximal hidden interrelationships between processes on a path of processes connected by first-order interrelationships is

$$\frac{1}{2} n(n-3)+1 \quad with \quad n \geq 1$$

Triangular cycles are closed by these hidden interrelationships. Thus, the total number of triangular cycles in a CSP diagram of 3 or more processes is determined by the same formula.

The operators on the user-specified interrelationship determine whether the operator on the hidden interrelationship can be uniquely determined (derived) or whether it can be randomly selected. In the latter case the design is ambiguous; i.e. the tool can choose one of many operators that are valid (specification conflict-free). An ambiguous triangular cycle can be expressed by a set of algebraic expressions, or by a single ambiguous algebraic expression.

G.3 Compositional Analysis Rule

The compositional analysis rule applies to triangular cycles for which all operators are known, either specified by the user, derived, or randomly selected. In case an operator is a choice of one in a set of operators, this rule can determine which choices are valid in the process architecture. The rule returns an algebraic expression or it returns a specification conflict in case an algebraic expression does not exist.

The compositional analysis rule is specified as follows:

$$\textit{if} \quad \underset{PU}{\oplus} = \overline{\underset{UQ}{\oplus}} \quad \textit{and} \quad \underset{PQ}{\oplus} \in \left\{\rightarrow, \leftarrow, \|, \square, \vec{\square}, \bar{\square}, \vec{\|}, \bar{\|}, \vec{\Delta}, \bar{\Delta}\right\} \quad \textit{then} \quad \left(P \underset{PQ}{\oplus} Q\right) \underset{PU}{\oplus} U$$
$$(\textit{note} : \textit{ambiguous solution})$$

else

$$\textit{if} \quad \underset{PQ}{\oplus} \in \left\{\underset{PU}{\oplus}, \underset{UQ}{\oplus}\right\} \quad \textit{then}$$

$$\textit{if} \quad \underset{PU}{\oplus} = \underset{UQ}{\oplus} \quad \textit{then} \quad P \underset{PU}{\oplus} U \underset{UQ}{\oplus} Q$$
$$(\textit{note} : \textit{unambiguous solution})$$

else

$$\textit{if} \quad \underset{PQ}{\oplus} = \underset{PU}{\oplus} \quad \textit{then} \quad P \underset{PQ}{\oplus} \left(U \underset{UQ}{\oplus} Q\right)$$
$$(\textit{note} : \textit{unambiguous solution})$$

else

$$\textit{if} \quad \underset{PQ}{\oplus} = \underset{UQ}{\oplus} \quad \textit{then} \quad \left(P \underset{PU}{\oplus} U\right) \underset{PQ}{\oplus} Q$$
$$(\textit{note} : \textit{unambiguous solution})$$

else

not conflict-free

A compressed triangular cycle results in a compressed algebraic expression. The compressed algebraic expression can be expanded by substituting sub-processes with their algebraic expressions. Of course, no algebraic expression can be completed in case a sub-process is in a specification conflict. The specification conflict needs to be solved by selecting or specifying another valid operator that results in a specification conflict-free diagram.

APPENDIX H

Pass-by-reference versus pass-by-value

H.1 Pass-by-reference

Pass-by-reference is a default concept in object-oriented programming languages for passing objects between objects. This mechanism is fast and dynamic, but at the same time it can be unsafe when passing references between objects with multiple threads of control. Multiple writers and readers that are allowed to access shared objects at the same time and in an unsynchronized way can easily corrupt the content of the shared object due to a race hazard. Shared objects must be synchronized to prevent race hazards. These precautions illustrate that multithreading is *not* orthogonal to objects but is intertwined with objects.

The sender process releases its ownership of the object after passing the reference through the channel. The receiver process will become the new owner of the object. If necessary, the receiver can pass the reference to object back to the sender which again claims ownership.

Since Java supports aliases (multiple references to objects) the Java compiler does not check for ownerships. Thus, passing references in Java can cause unsafe situations and it is up to the programmer to apply the rule of ownership to guarantee safety. If Java had a notion of channel primitives then this information could be used by the compiler to check for ownership which would make concurrency in Java a lot safer.

A problem with sending references over channels is that this only works on shared memory systems and pass-by-value is required between distributed memory systems. Channels with a pass-by-reference mechanism must create a clone (at the receiver side) and the reference to that newly created object must be returned to the receiving process. Java supports cloning and the garbage collector deletes unreferenced objects. Cloning and garbage collection seem to be useful for business applications with lots of resources available, but they are time-consuming and non-deterministic which makes them unpopular for embedded real-time systems with limited resources and strict timing requirements.

H.2 Pass-by-value

Pass-by-value is default for occam channels and default for primitive data types in Java. The producer passes the data (content) of the message object instead of its reference. The data of the source object will be copied in the (pre-allocated) destination object at the receiver side. Each process has a copy to work with, without the overhead of synchronization. The ownership rule is implicit to the pass-by-value mechanism. The pass-by-value mechanism is identical for shared memory systems and for distributed memory systems. Objects can be efficiently reused without continuously creating and destroying objects. This mechanism works for programming languages without cloning and garbage collection as for C and C++.

A problem with pass-by-value on shared memory systems is that the overhead of communication can be high for objects that are larger than the size of the reference (pointer, hash-code). The problem is less significant when objects are small and communication is at a low frequency. Pass-by-reference can also be implemented with pass-by-value. In this case, the message object is a container holding a reference to another object. On communication the reference to the object will be copied. This inherits all advantages and disadvantages of pass-by-reference and requires the help of an object container. Deep copying (i.e.

recursively copying inner objects) can also be part of the mechanism, but then one must avoid cyclic references.

H.3 Message passing for control software

Control software favours deterministic behaviour, small packaged communication, and transparency between shared memory systems and distributed memory systems. Pass-by-value fulfils these requirements. As long as small objects are passed, the pass-by-value has advantages over pass-by-reference. On these grounds the pass-by-value mechanism and data channels are commonly used for control software.

The performance of pass-by-reference and pass-by-value not only depend on the size of message but also depend on the *overall performance* of the hardware and software architecture. The architecture determines how frequent objects are passed and how often memory must be allocated/destroyed or can be reused. In most CSP-based applications one does not share objects other than channel- and barrier-objects. Sharing objects between processes is rare. The alternative of sharing an object is that the object is part and under control of a server process that is connected with shared channels to its clients. The client processes synchronize on channels or barriers and follows a protocol of interaction with the server process. This is thread-safe and the server process provides a clear behavioural description.

Notation

$a \in X$	set membership, a is an element in set X
$X \subseteq Y$	X is a subset of Y (= $\forall a.a \in X \Rightarrow a \in Y$)
$\{a,b,c\}$	set with elements a, b, and c
$\{\},\varnothing$	the empty set
$X \cup Y$	union
$X \cap Y$	intersection
$X - Y$	difference (= $\{a \in X \mid x \notin Y\}$)
$\mathbb{N}$	natural numbers (= $\{0,1,2,...\}$)
$\mathbb{N} \setminus \{0\}$	natural numbers without 0 (= $\{1,2,...\}$)
$e \rightarrow P$	prefixing
$?x : A \rightarrow P$	prefix choice
$P \overset{\checkmark}{\rightarrow} Q$	single action transition
$P \oplus Q$	arbitrary composition
$P;Q$	sequential composition
$P \parallel Q$	synchronous parallel
$P \,{}_X\|\|_Y\, Q$	alphabetic parallel
$P \underset{X}{\parallel} Q$	generalized parallel
$P \mid\mid\mid Q$	interleaving

$P \vec{\|} Q$	prioritized synchronous parallel
$P \sqcap Q$	internal or nondeterministic choice
$P \Box Q$	external choice
$P \vec{\Box} Q$	unequally-prioritized external choice
$P \dot{\Box} Q$	resolute equally-prioritized external choice
$P \dot{\vec{\Box}} Q$	resolute unequally-prioritized external choice
$P \tilde{\Box} Q$	preference equally-prioritized external choice
$P \tilde{\vec{\Box}} Q$	preference unequally-prioritized external choice
$P \Delta_i Q$	interrupt composition on event i
$P \vec{\Delta} Q$	exception composition
$P \setminus X$	hiding
$P \lhd cond \rhd Q$	P if $cond$ is true else Q
$\langle\,\rangle$	the empty sequence
$\langle a_1,...,a_n \rangle$	trace of events containing a_1,..,a_n in that order
$s \frown \langle a \rangle$	concatenation of a to trace s
$(P,Q,\oplus)$	relationship with $\oplus$, from P to Q
$\left(P,Q,\overline{\oplus}\right)$	relationship with $\oplus$, from Q to P
$(P,Q,\oplus)_{\varnothing}$	relationship with $\oplus$, from P to Q, with P and Q being neighbours
$\underset{PQ}{\oplus}$	operator between P and Q, from P to Q
$\underset{PQ}{\overline{\oplus}}$	inverse operator between P and Q, from P to Q
Σ	alphabet of all communications
✓	termination event

$process.id : Type$	compound identifier label
$c?x$	input x from channel c
$c!y$	output y to channel c
$b*$	synchronize on barrier b
$P*$	infinite recursion
$\mu X.P;X$	recursion
$pri(a)$	function that returns the priority index of the thread with the highest priority engaging in communication event a.

Bibliography

4Links (2003). *SpaceWire PCI Interfaces*, at URL http://www.4links.co.uk/spacewire_pci.htm.

20-sim (2003). *Control Labs Products*, at URL http://www.20sim.com.

Analog Devices (2003). *Datasheet ADSP 21992*, Analog Devices, at URL http://www.analog.com/UploadedFiles/Data_Sheets/52993570ADSP-21992_0.pdf.

Arnold, K., J. Gosling and D. Holmes (2000). *The Java Programming Language Third Edition*, Addison-Wesley, San Francisco.

Awad, M., J. Kuusela and J. Ziegler (2002). *Octopus -- Object-oriented technology for real-time systems*, Nokia, at URL http://www-nrc.nokia.com/octopus/.

Balkema, W., P. Boer, E. Dertien, T. v. Engelen and G. v. Oort (1999). *Arty*, University of Twente, june 1999

Barnes, J. (1988). *Ada 95, Second Edition*, Addison Wesley.

Barrett, G. (1993). *occam 3 reference manual (March 31 1992 draft)*, INMOS, at URL http://wotug.ukc.ac.uk/parallel/occam/documentation/.

Barrett, G., M. Goldsmith, G. Jones and A. Kay (1988). *The meaning and implementation of PRI ALT in occam*, 9th occam User Group, Occam and the Transputer--Research and Applications, C. Askew, IOS Press, Southampton, pp. 37-46.

Beckett, D. (1994). *Biography Willam of Ockham*, University of Kent, at URL http://wotug.ukc.ac.uk/parallel/www/occam/occam-bio.html.

Beneder, T. (1998). *Arty*, University of Twente, july 1998

Bosch (2003). Robert Bosch GmbH, at URL http://www.can.bosch.com.

Brinch-Hansen, P. (1972). *Structured Multiprogramming*, Communications of the ACM, July 1972, Volume 15, Issue 7, pp. 574-578.

Brinch-Hansen, P. (1973). *Operating System Principles*, Prentice-Hall International, Englewood Cliffs, NJ.

Broenink, J. F. and G. H. Hilderink (2001). *A structured approach to embedded control systems implementation*, Proceedings of the IEEE International Conference on Control Applications, M. W. Spong, D. Repperger and J. M. I. Zannatha, Mexico City, Mexico, September 5-7, 2001, pp. 761-766.

Burns, A. (1987). *Occam's priority model and deadline scheduling*, 7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines, E. T. Muntean, LGI-IMAG, Grenoble.

Burns, A. and A. Wellings (1990). *Real-Time Systems and their Programming Languages*, International Computer Science Series, Addison-Wesley Publishing Company.

CMX-RTX (1998). *CMX-RTX -- Real-Time Multi-Tasking Operating System for microporcessors, microcontrollers and DSPs*, at URL http://www.cmx.com/.

Cornhill, D., L. Sha, J. Rajkumar and H. Tokunda (1978). *Limitations of Ada for real-time scheduling*, Proceedings of the International Workshop on Real Time Ada Issues, ACM Ada Letters, pp. 33-39.

Cronie, H. S., F. W. Hoeksema and C. H. Slump (2003). *A CSP-based Processing Architecture for a Flexible MIMO-OFDM testbed*, Communicating Process Architectures 2003, J. F. Broenink and G. H. Hilderink, IOS Press, University of Twente, Enschede, The Netherlands, 7-10 September 2003, Volume 61, pp. 225-234.

Davies, J. and S. Schneider (1995). *Real-Time CSP*

DEC (2003). *Alpha processor 21364*, at URL http://www.theregister.co.uk/content/archive/6610.html.

Dijkstra, E. W. (1965). *Solution of a problem in concurrent programming control*, Communications of the ACM, September 1965, Volume 8, Issue 9, p. 569.

Dijkstra, E. W. (1968a). *Cooperating sequential processes*, In Programming Languages, F. Genuyes, London Academic Press.

Douglass, B. P. (1999). *Doing Hard Timer: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Object Technology Series, Booch, Jacobson and Rumbauch, Addison Wesley Longman, Inc., Reading.

dSPACE (2002). *Real Time Interface*, at URL www.dspace.com.

Engelen, T. H. v. (2004). *CTC++ enhancements towards fault tolerance and RTAI*, University of Twente, August 2004, Report number 022CE2004.

Engelen, T. H. v. (2004). *CTC++ for the Arty Robot*, University of Twente, August 2004, Report number 024CE2004.

ESD (2003). *Controller Area Network (CAN)*, ESD gmdb, at URL http://www.esd-electronics.com/german/PDF-file/CAN/Englisch/intro-e.pdf.

FDR (2004). *FDR2*, Formal Systems Ltd., at URL http://www.fsel.com/.

FSMLabs (2002). *Real-Time Linux*, at URL http://www.fsmlabs.com/.

Gell-Mann, M. (1995). *What is complexity?* John Wiley and Sons Inc., Volume 1, at URL http://www.santafe.edu/sfi/People/mgm/complexity.html.

GNU (1996). *The GNU Project and the Free Software Foundation*, at URL http://www.gnu.org/.

Hatley, D. J. and I. A. Pribhai (1987). *Strategies for Real-Time System Specification*, Dorset House Publishing, New York, NY.

Hilderink, G. H. (2002). *A Graphical Modelling Language for Specifying Concurrency based on CSP*, IEE Proceedings Software, IEE, April 2003, Volume 150, pp. 108-120, '2' 2.

Hilderink, G. H., A. W. P. Bakkers and J. F. Broenink (2000). *A Distributed Real-Time Java System Based on CSP*, The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-2000), IEEE Computer Society, Newport Beach, California, March 15-17, 2000, pp. 400-407.

Hilderink, G. H. and J. F. Broenink (2003). *Sampling and Timing: a Task for the Environmental Proces*, Commmunicating Process Architectures 2003, J. F. Broenink and G. H. Hilderink, IOS Press, University of Twente, Enschede, 7-10 September 2003, Volume 61.

Hilderink, G. H., J. F. Broenink and A. W. P. Bakkers (1998). *A new Java thread model for concurrent programming of real-time systems*, Real-Time magazine, Q1 1998, '1' 1.

Hilderink, G. H., J. F. Broenink and A. W. P. Bakkers (1998). *Software design method for heterogenous embedded systems*, 17th Benelux Meeting, Mierlo, NL, 4-6 March 1998, pp. 183-183 (abstract).

Hilhorst, R. A., J. van Amerongen, P. Löhnberg and H. J. A. F. Tulleken (1994). *Supervisory control of mode-switch processes*, Automatica, Volume 30, Issue 8, pp. 1319-1331.

Hiroshi, S. (1997). *What is Occam's Razor*, at URL http://math.ucr.edu/home/baez/physics/General/occam.html.

Hoare, C. A. R. (1974). *Monitors: An Operating System Structuring Concepts.*, Communications sof the ACM, Volume 17, Issue 10, pp. 547-557.

Hoare, C. A. R. (1978). *Communicating Sequential Processes*, CACM, Volume 21, Issue 8, pp. 666-677.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*, Prentice-Hall, London, UK.

Inmos (1988). *occam 2 Reference Manual*, International Series in Computer Science, C. A. R. Hoare, Prentice Hall.

Intel (1996). *The Intel386 Processor Family*, at URL www.intel.com.

Isermann, R., J. Schaffnit and S. Sinsel (1999). *Hardware-in-the-loop simulation for the design and testing of engine-control systems*, Control Engineering Practice 7, 18 Augustus 1999, pp 643-653.

Ivimey-Cook, R. (1999). *Legacy of the Transputer*, WoTUG 22: Architectures, Languages and Techniques for Concurrent Systems, B. M. Cook, IOS Press, University of keele, UK, April 11-14, 1999.

Jones, G. (1987). *On Guards*, 7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines, E. T. Muntean, LGI-IMAG, Grenoble.

Jovanovic, D. S., G. H. Hilderink and J. F. Broenink (2001). *Integrated Design Tool for Embedded Control Systems*, Process 2001 Workshop, K. Karelse, Veldhoven, The Netherlands, October 18, 2001, pp. 121-126.

Jovanovic, D. S., G. H. Hilderink and J. F. Broenink (2002). *A Communicating Threads (CT) Case Study: JIWY*, Communicating Process Architecture 2002, J. S. Pascoe, P. H. Welch, R. J. Loader and V. S. Sunderam, IOS Press, University of Reading, UK, Volume 60, pp. 311-320.

Jovanovic, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004). *gCSP: A Graphical Tool for Designing CSP Systems*, Communicating Process Architectures 2004, I. R. East, J. M. R. Martin, D. Duce and M. Green, IOS Press, Oxford Brooks University, UK, 5-8 September 2004, Volume 62, pp. 233-251, at URL http://www.wotug.org/.

Kernigham, B. W. and D. M. Ritchie (1988). *The C Programming Language, Second Edition*, Prentice Hall, Inc.

KROC (1999). *The KroC home page*, at URL http://www.cs.ukc.ac.uk/porjects/ofa/kroc.

Labrosse, J. J. (1992). *uC/OS The Real-Time Kernel*, Publishers Group West, Emeryville, CA 94662.

Lahpor, G. J. (1998). *The design of a low-cost multi-processor system*, Control Engineering, University of Twente, Enschede.

Lammertink, T. (2003). *Joystick Controller for JIWY*, University of Twente, June 2003

Lau, F. C. M. and K. M. Shea (1988). *Mapping a Process Network onto a Processor Network*, 9th occam User Group, Occam and the Transputer--Research and Applications, C. Askew, IOS Press, Southampton, pp. 91-112.

Lauer, H. and E. Satterwaite (1979). *The impact of Mesa on system design*, Proceedings of the 4th International Conference on Software Engineering, IEEE, pp. 174-182.

Lawrence, A. E. (1998). *Extending CSP*, 21th World Occam and Transputer User Group Technical Meeting, Architectures, Languages and Patterns for Parallel and Distributed Applications, P. H. W. a. A. W. P. Bakkers, IOS Press, Canterbury, United Kingdom, 5-8 April 1998, Volume 52, pp. 111-131.

Lewis, B. and D. J. Berg (1996). *Thread Primer: A guide to multithreaded Programming*, Sun Microsystems Press, Mountain View, USA.

Maggee, J. and J. Kramer (1999). *Concurreny: state models & Java programs*, Worldwide series in computer science, John Wiley & Sons, New York, USA.

Martin, J. M. R. and S. A. Jassim (1997). *How to Design Deadlock-Free Networks Using CSP and Verification Tools -- A Tutorial Introduction*, Parallal Programming and Java -- WoTUG-20, A. Bakkers, IOS Press, Enschede, The Netherlands, 13-16 April 1997, Volume 50, pp. 326338.

Martin, J. M. R. and S. A. Jassim (1997). *A Tool for Proving Deadlock Freedom*, Parallel Programming and Java -- WoTUG-20, A. Bakkers, IOS Press, Enschede, The Netherlands, Volume 50, pp.1-16.

McColl, W. F. (1996). *Scalable Computing*, Computer Science Today: Recent Trends and Developments, J. v. Leeuwen, Springer Verlag, Volume number 1000 in Lecture notes in Computer Science, pp. 41-61.

Microsoft (2003). *Visual C# Developer Center*, at URL http://msdn.microsoft.com/vcsharp/.

Modderkolk, P. (2003). *Concurrent motor controller for Arty*, University of Twente, October 2003

National Instruments (2004). *Real Time Labview Module, Add-On Software for Designing Reliable, Deterministic Systems*, at URL http://www.labview.com/.

Nissanke, N. (1997). *Realtime Systems*, Prentice-Hall International Series in Computer Science, Prentice-Hall.

Orlic, B., H. Ferdinando and J. F. Broenink (2003). *CAN Fieldbus Communication in CSP-based CT library*, Proceedings of the 4th PROGRESS Symposium on Embedded Systems, NBC Nieuwegein, The Netherlands, October 22, 2003.

Page, I. (2001). *The Handel-C Language*, at URL at URL: http://www.celoxica.com/methodology/handelc.asp.

PCI-6024E, N. (2000). *National Instruments PCI-6024E Multifunction DAQ*, at URL http://www.ni.com.

ProBE (2003). *Process behavior Explorer -- User Manual*, Formal Systems Ltd., at URL http://www.fsel.com.

Ptolemy (2003). *The Ptolemy II Project*, at URL http://ptolemy.eecs.berkeley.edu/.

QNX (1998). *QNX Neutrino RTOS*, at URL http://www.qnx.com.

RadiSys (2002). *OS-9*, at URL http://www.radisys.com/.

Roscoe, A. W. (1987). *Routing messages through networks: an excercise in deadlock avoidance*, 7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines, E. T. Muntean, LGI-IMAG, Grenoble.

Roscoe, A. W. (1998). *The Theory and Practice of Concurrency*, Series in Computer Sciences, C. A. R. Hoare and R. Bird, Prentice-Hall.

RTAI (2002). *Realtime Linux Application Interface for Linux*, at URL http://www.aero.polimi.it/~rtai/.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991). *Object-Oriented Modeling and Design*, Prentice-Hall International Editions, Englewood Cliffs, N.J.

Sanvido, M. A. A. and V. Cechticky (2002). *Testing embedded control systems using hardware-in-the-loop simulation and temporal logic*, IFAC 15th Triennial Worl Congress, Barcelona, Spain, July 2002.

Schneider, S. (2000). *Concurrent and Real-time Systems*, Worldwide series in computer science, S. U. U. D. Barron and B. U. U. P. Wegner, John Wiley & Sons, Chichester, UK.

Selic, B., G. Gullekson and P. T. Ward (1994). *Real-Time Object-Oriented Modeling (ROOM)*, John Wiley & Sons, Inc.

Sha, L., R. Rajumar and J. Lehoczky (1990). *The Priority Inheritance Protocol: An Approach to Real-Time Synchronization*, IEEE Transactions on Software Engineering, Volume 39, Issue 9, pp. 1175-1185.

Silberschatz, A. and P. Galvin (1994). *Operating Systems Concepts*, Addison-Wesley, Reading, Massachusetts, USA.

Skeptic (2004). *Occam's Razor*, at URL http://www.skepdic.com/occam.html.

Smith, L. (2002). *JIWYNET*, University of Twente, August 2002

Smith, M. L., C. E. Hughes and K. W. Burke (2003). *The Denotational Semantics of View-Centric Reasoning*, Communicating Process Architectures 2003, J.F.Broenink and G.H.Hilderink, IOS Press, Enschede, Volume 61, pp. 91-96.

SpaceWire (2003). *Overview of SpaceWire*, at URL http://www.estec.esa.nl/tech/spacewire/.

SPoC (1998). *The Southampton Portable occam Compiler (SPoC)*, at URL http://gales.ecs.soton.ac.uk/software/spoc/.

Stanley-Marbell, P. (2003). *Inferno Programming with Limbo*, John Wiley & Sons.

Stephan, R. A. (2002). *Real-time Linux in Control Applications Area*, Control Engineering, University of Twente, Enschede.

Stroustrup, B. (2000). *The C++ Programming Language*, Special Edition, A. T. Labs, Addison Wesley, New Jersey.

Sun Microsystems, L. (2004). *Java 2 Platform, Second Edition 1.5.0, Software Development Kit 5.0*, at URL java.sun.com.

Texas Instruments (1996). *TMS320F240 DSP Controller*, Texas Intruments, at URL http://focus.ti.com/lit/ds/sprs042e/sprs042e.pdf.

Texas Instruments (1999). *TMS320F6711 DSP Controller*, Texas Intruments, at URL http://focus.ti.com/lit/ds/symlink/tms320c6711.pdf.

uC/OS (1998). *Micrium uC/OS-II -- The Real-Time Kernel*, at URL http://www.ucos-ii.com/.

UML (1998). *Unified Modeling Language 1.4*, OMG, at URL http://www.uml.org/.

van Amerongen, J. (2003). *Mechatronic design*, Mechatronics, 2003, Issue 13, 1045-1066.

van Amerongen, J. and P. C. Breedveld (2003). *Modelling of physical systems for the design and control of mechatronic systems*, Annual Reviews in Control 27, pp 87-117.

van Breemen, A. J. N. (2001). *Agent-based multi-controller systems*, Control Engineering, Twente University Press, Enschede, 232.

van Drunen, J. M. (2000). *Realization of link drivers implementing CSP-channels on 20-controller*, Control Laboratory

Electrical Engineering, University of Twente, Enschede.

Ward, P. T. and S. J. Mellor (1985). *Structured Development Techniques for Real-Time Systems*, Prentice-Hall, Englewood Cliffs, NJ, Volume 3 volumes.

Welch, P. H. (1989). *Graceful Termination -- Graceful Resetting*, Applying Transputer-Based Parallel Machines, Proceedings of OUG 10, Occam User Group, IOS Press, Enschede, Netherlands, April 1989, pp. 310-317.

Welch, P. H. (1996). *Wot, no chickens?* 25 September 1996, at URL http://wotug.ukc.ac.uk/parallel/groups/wotug/java/discussion/3.html.

Welch, P. H. and P. D. Austin (1999). *The JCSP home page*, at URL http://www.cs.ukc.ac.uk/projects/ofa/jcsp.

Welch, P. H. and A. Bakkers (1992). *In Parallel*, O. universiteit, Euroterm Maastricht, Heerlen.

Wijbrans, K. C. J. (1993). *Twente Hierarchical Embedded Systems Implementation by Simulation: a structured method of controller realization*, Electrical Engineering - Control Laboratory, University of Twente, The Netherlands, Enschede.

Wijbrans, K. C. J., J. v. Amerongen, A. W. P. Bakkers and J. F. Broenink (1993). *Twente Hierarchical Embedded Systems Implementation by Simulation: a structured approach to controller realisation on transputers*, Journal A, Volume 34, Issue 1, pp. 51-59.

WindRiver (2002). *VxWorks -- Real-time operating system*, at URL http://www.windriver.com/.

Yourdon, E. N. (1989). *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, NJ.

Yourdon, E. N. and L. L. Constantine (1979). *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ.

Curriculum vitae

Gerald Hilderink was born on September 23rd, 1968 in Haaksbergen, the Netherlands. In 1992, after successfully completing the study Technical Computer Science at the Hogeschool Enschede in Enschede, he started his study Informatics at the University of Twente in Enschede, the Netherlands. In 1997, he received his M.Sc. degree in Informatics in major Embedded Systems at this university.

During his Master's project at the laboratory of Control Engineering, he investigated the possibilities of applying the theory of *Communicating Sequential Processes* (CSP) to the programming language Java. This work resulted in a masters' thesis which illustrated the advantages of using CSP concepts over the Java thread model for building reliable and safe concurrent software in Java. This work resulted in a "Best Student Award" paper and an article in Real-Time Magazine 1998. This work has interested many people and has become part in lectures at the University of Twente and other universities all over the world.

In February 1997, he started his Ph.D. research at the laboratory of Control Engineering at the University of Twente. This research was based on his Masters' thesis project and aimed at the development of a sound and formal foundation for developing high-performance control software. He realized to bring the look and feel of CSP, occam, and transputer technology to heterogeneous computer systems and object-oriented programming languages. He developed the *Communicating Threads* (CT) library, which provides CSP concepts for the programming languages Java, C, and C++.

This research continued in a STW/PROGRESS (TES.5224) project which started in 2001. He worked at the project in a PostDoc position. The project applies CT and CSP diagrams to embedded control systems and software tool development. In September 2003, he organized together with Jan Broenink the Communicating Process Architectures 2003 conference in Enschede.

Gerald Hilderink has always been interested in embedded systems and problem solving. He recognizes the importance of the interaction between theory and practice in order to design reliable embedded systems.