1. Using formal methods for solving the software crisis is avoidable as much as using nuclear energy for solving the global energy crisis.

2. The best time to start debugging a program is before the first bug (error) is discovered.

   (page 160 of the thesis)

3. If one starts analyzing prevention of all possible errors along the way of creating something, the way will never be travelled.

4. Just as assembly languages nowadays are considered as low-level software, over a decade or two today’s (non-graphical) ‘higher-level languages’ will be viewed in the same way.

   (page 63 of the thesis)

5. By getting a comprehensive ‘undo’ feature, after all these years the modelling tools 20-sim and gCSP are loosing some of their genuine virtues to model irreversibility of the real-world phenomenon of making mistakes.

6. Benefits from the current technology of producing recycled paper for more-than-one-time-reading materials are doubtful: what do we save if using so much chemicals and energy to make it white again?

7. Hypertext is invention no. 1 of the information age.

8. Paper copies of scientific dissertations should soon become history; hence you are holding a relic in your hands!

9. Fear and serenity are the only two components of one’s mood; bad mood (i.e. negative feelings) can always be analysed in terms of fear.

   (Fear is a mind killer – Frank Herbert, “Dune”)

10. Although ‘PhD’ means Doctor of Philosophy, devising more than one profound wisdom per year is eligible to suspicion.
Designing dependable software: a CSP-based approach.

Dusko Jovanovic
Designing dependable process-oriented software

a CSP-based approach

Ph.D. thesis of Dusko Jovanovic

Dedicated in gratitude to the powers of meditation

At times I think, at times I am.
Paul Valéry
Graduation committee:

Chairman: prof.dr.ir. A.J. Mouthaan
Promotor:  prof.dr.ir. J. van Amerongen
Assist. promotor: dr.ir. J.F. Broenink
Opponents: prof.dr. H. Brinksma
           prof.dr. T. Krol
           prof.dr. A. van Deursen
           prof.dr. S. Turajlić, dipl.ing.

University of Twente, NL
University of Twente, NL
University of Twente, NL
University of Twente, NL
University of Twente, NL
University of Twente, NL
University of Belgrade, Serbia

University of Twente, Control Engineering Lab
and Drebbel Institute for Mechatronics

CTIT Ph.D.-thesis Series
Series number: 1381-3617
CTIT number: 06-82

This research was supported by the PROGRESS program of
the Technology Foundation STW, the Dutch organization
for Scientific Research NWO and the Dutch Ministry of
Economic Affairs under grant TES. 5224

© 2006 by D.S. Jovanovic
All rights reserved. No part of this work may be reproduced by print, photocopy
or any other means without permission from the author.

Cover design “Mindmap of technical creativity I”

Printed by Wöhrmann Print Service, Zutphen, NL

ISBN: 90-365-2334-6
DESIGNING DEPENDABLE PROCESS-ORIENTED SOFTWARE
A CSP-BASED APPROACH

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof.dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday 16th of March 2006 at 16.45 hours

by

Duško Jovanović

born on 6th of May 1975

in Obrenovac, Serbia
This dissertation is approved by the promotor prof.dr.ir. Job van Amerongen and the assistant promotor dr.ir. Jan F. Broenink.
Summary

This thesis advocates dependability as a crucial aspect of software quality. Process orientation, as it is defined in this thesis, concentrates on the notion of a process as a basic building component of a dataflow-centred software architecture. The dependability approach in the proposed variant of process orientation builds on a few specific strengths of the particular dataflow-centred architecture which is based on the principles of the CSP process algebra.

The CSP/CT process-oriented modelling and programming environment for control applications has been enriched in this work with various complementary instruments for raising dependability of concurrent software. In addition to the design methodology enhancement, the main deliverable is a graphical CASE tool, named gCSP, which facilitates modelling, visualizing and managing software models of evergrowing complexity. By manipulations of once developed models, the gCSP tool exploits the formal underpinning of the methodology to allow formal verification of the designs by automatically generating formal specification in the CSPm language. Efficient production and trusting the final outcome of the design—implementation code—is substantially increased by automatic code generation of C++ code compliant with the CTC++ implementation library for concurrent programming. In this thesis it is illustrated, worked out and shown on examples and mechatronic set-ups that the process-oriented CSP/CT framework is suitable for hosting various established dependability instruments: concurrent exception handling, N-version programming, logging, monitoring and several variants of watchdogs.

This thesis advocates: tool-based visual programming, investments of increasing computer capabilities in bearing overheads of dependability of complex software systems, separation of versatile software concerns at the modelling stage, and making software development an engineering discipline by predictability established on a mathematically-based development. This together is proposed for raising quality of (embedded) software in design time.
Contents

Part I Prerequisites and tooling ......................................................... 1

1 Introduction ................................................................................... 3

1.1 Motivation: dependability versus immaturity ............................ 4

1.2 Objective: dependable embedded systems ................................. 6

1.3 Embedded control systems ....................................................... 7

1.3.1 Embedded systems ................................................................. 8

1.3.2 Real-time systems ................................................................. 9

1.3.3 Control systems ................................................................. 10

1.4 Embedded software .................................................................... 12

1.4.1 Concurrency ................................................................. 12

1.4.2 Complexity ................................................................. 13

1.4.3 Real-time behaviour of software ......................................... 13

1.4.4 Formal modelling and verification ....................................... 14

1.4.5 An intermediate summary of the issues ................................. 15

1.5 Dependability ............................................................................. 16

1.5.1 A short survey of established safety and fault tolerance concepts ......................................................... 17

1.6 Design strategies ......................................................................... 20

1.6.1 Mechatronics ..................................................................... 22

1.6.2 Stepwise refinement .......................................................... 22

1.7 Process orientation and CSP/CT ............................................. 23

1.7.1 CSP as a modelling paradigm for concurrent process-oriented software ......................................................... 25

1.7.2 Dependability potentials of the CSP-based process orientation ......................................................... 27

1.8 Scope, contributions, case studies and outline of this thesis ......... 28

1.8.1 Scope of the thesis ............................................................. 29

1.8.2 Contributions of the thesis .................................................. 29

1.8.3 Case studies ................................................................. 30

1.8.4 Thesis outline ................................................................. 34

2 Key concepts, definitions and tools ........................................... 35

2.1 Notions and standards of software quality ................................. 35

2.1.1 ISO/IEC 9126, 14598 and 25000 standards ............................................. 36

2.1.2 IEC 61508 – Functional safety of E/E/PE safety-related systems ......................................................... 37

2.1.3 CMM – Capability Maturity Model ........................................... 38

2.1.4 Other ................................................................. 38
2.2 Software dependability........................................................ 39
  2.2.1 Errors, failures, faults and fault tolerance ......................... 40
  2.2.2 Error recovery versus error masking ................................ 45
  2.2.3 Exceptions, exception handling and atomic actions ............ 46

2.3 Real-time terminology ........................................................ 47

2.4 Concurrency and concurrency-specific phenomena ............ 49

2.5 CSP foundation and derivatives ........................................ 50
  2.5.1 CSP diagrams ................................................................. 50
  2.5.2 CSP libraries ............................................................... 51

2.6 Process orientation ............................................................ 53

2.7 Formal analysis .................................................................. 54
  2.7.1 Methods and tools for formal analysis ............................... 54
  2.7.2 CSPm .............................................................................. 55
  2.7.3 FDR and ProBE tools .................................................... 56

2.8 Embedded control systems .............................................. 58
  2.8.1 Embedded systems ....................................................... 58
  2.8.2 Control systems .......................................................... 58
  2.8.3 20-sim tool ................................................................. 59

2.9 Interdomain tooling coverage ........................................... 60
  2.9.1 gCSP as an interdomain bridge for embedded software ....... 60

3 Modelling CSP/CT architectures with the gCSP tool.......... 63

3.1 Graphical modelling languages and tools ......................... 63
  3.1.1 Requirements for the gCSP tool development ................. 64

3.2 The gCSP graphical language ........................................... 65
  3.2.1 Processes ................................................................. 69
  3.2.2 Communication relationships ..................................... 76
  3.2.3 Compositional relationships ....................................... 78
  3.2.4 Compositional hierarchies .......................................... 85
  3.2.5 C-tree and the CSP/CT modelling principles ................ 89

3.3 A practical example.......................................................... 91

3.4 The gCSP tool ................................................................ 97
  3.4.1 Tool menus and the toolbar ......................................... 99
  3.4.2 Graphical editor ......................................................... 99
  3.4.3 The C-tree ............................................................... 99

3.5 gCSP models of the case studies ..................................... 100
  3.5.1 JIWY ................................................................. 100
  3.5.2 Tripod ................................................................. 103

3.6 Conclusions ...................................................................... 106
  3.6.1 Directions for further development ............................... 106
Part II  Dependability instruments for process-oriented software .......................................................... 109

4  Automatic code generation and formal verification of CSP/CT software ......................................................... 111

4.1  Transforming abstract models into machine-readable forms ........................................................................ 112
  4.1.1 Formal analysis ......................................................................................................................................... 113
  4.1.2 Automatic generation of source code ..................................................................................................... 113

4.2  CSPm code generation and formal deadlock checking .................................................................................. 114
  4.2.1 CSPm code generation options ............................................................................................................. 120

4.3  Code generation of implementation source code ......................................................................................... 122
  4.3.1 Network builder and source code structure ......................................................................................... 122
  4.3.2 Low level refinement and custom (user-defined) code ......................................................................... 124
  4.3.3 Inclusion of 20-sim generated code ..................................................................................................... 125
  4.3.4 Hardware manipulation code .............................................................................................................. 128
  4.3.5 CTC++ code generation options .......................................................................................................... 128

4.4  Case study ................................................................................................................................................ 131

4.5  Conclusions .............................................................................................................................................. 136
  4.5.1 Summary of the design trajectory for generating formally verified CSP/CT software ......................... 136
  4.5.2 Directions for further development ...................................................................................................... 139

5  Exception handling mechanism for CSP/CT software .................................................................................. 141

5.1  Exception Handling Mechanisms .............................................................................................................. 142
  5.1.1 EHM history, state-of-the-art and state-of-the-practice overview ......................................................... 142
  5.1.2 EHM terminology and properties ....................................................................................................... 145
  5.1.3 EHM requirements .............................................................................................................................. 147

5.2  The EHM concept within CSP/CT, libraries support and the gCSP tool coverage ..................................... 149
  5.2.1 The gCSP tool support ......................................................................................................................... 152
  5.2.2 Exception handling support in the CT libraries .................................................................................... 155
  5.2.3 Abnormal (exceptional) termination of the CT constructs ................................................................. 158

5.3  Use of the EHM facilities .......................................................................................................................... 160

5.4  Case study ................................................................................................................................................ 166

5.5  Discussion ................................................................................................................................................ 173
  5.5.1 Properties of the prototyped EHM ......................................................................................................... 174
  5.5.2 Conclusions ......................................................................................................................................... 176
  5.5.3 Directions for further research and development .................................................................................. 177
6  Dependability design patterns for CSP/CT software  

6.1 On design patterns

6.2 Watchdog patterns
6.2.1 Liveness watchdogs
6.2.2 Real-time feasibility watchdogs
6.2.3 Integrity watchdogs

6.3 N-version programming
6.3.1 N-version programming in CT and gCSP
6.3.2 Example: robust adder

6.4 Logging and monitoring
6.4.1 Logging
6.4.2 Monitoring
6.4.3 Modelling access to the L/M coordinator in the gCSP tool
6.4.4 Example: monitored adder

6.5 Case study
6.5.1 Logging and monitoring on Tripod
6.5.2 N-version programming on Tripod
6.5.3 Watchdogs on Tripod

6.6 Conclusions and suggestions
6.6.1 Logging and monitoring
6.6.2 Watchdogs
6.6.3 N-version programming
6.6.4 Directions for further development

Part III  Reflections and details

7  Wrapping up the big picture

7.1 Conclusions
7.1.1 Contributions revisited
7.1.2 Error coverage and complementarity of the proposed dependability techniques
7.1.3 Benefits of programming dependability in terms of concurrency
7.1.4 Benefits for designing (dependable) software in the CSP-based process-oriented way
7.1.5 Why use CSP/CT in making embedded systems
7.1.6 What sits in the way

7.2 Recommendations for further research
7.2.1 The tool, graphical language and code generators
7.2.2 Exception handling
7.2.3 Dependability design patterns
7.2.4 Distributiveness for the future

7.3 Closing
Appendices ........................................................................229
Appendix A Some implementation details of the CSP/CTC++
extection handling mechanism ...............................................229
Appendix B Atomic actions in CSP/CT – an outline ..............233
Appendix C Some implementation details of the watchdog
mechanism ...........................................................................237
Appendix D CTC++ code generation and templates for 20-sim....239
Samenvatting .....................................................................241
Sažetak .............................................................................243
Acknowledgements ............................................................245
About the author ...............................................................249
References .........................................................................251
Part I  Prerequisites and tooling

Chapter 1  Introduction

Chapter 2  Key concepts, definitions and tools

Chapter 3  Modelling CSP/CT architectures with the gCSP tool
1 Introduction

We need to make the phrase “software engineer” mean something.
Until we have professional standards,
reasonably standardised educational requirements,
and a professional identity,
we have no right to use the phrase, “Software Engineering”.
David Lorge Parnas (“Software Aging”, 1994)

The term “software crisis” has been coined some thirty years ago by Dijkstra (1972, p.238), after the term “software engineering” had been introduced in 1968 on the first NATO Software Engineering Conference (Naur and Randell, 1969). From that moment on, the named crisis has never ceased (Gibbs, 1994) – it just has transformed as the abilities of the computer hardware transformed (increased), and consequently expectations of the users. Today it is expected that electronic artificial intelligence gets embedded in virtually any domain of everyday’s physical activity. The emergent knowledge society is rooted in the ubiquitous proliferation of the computer-based surroundings.

The motivation and the conclusion of this work in short are: modern (“knowledge”, “post-industrial”, “information”) society increasingly depends on computers and their software. It is crucial to make them dependable. Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable (Avižienis et al., 2004, p.13). The more software is implicated in all walks of real life, the more its structure has to reflect the nature of this real world. One of the characteristics of functioning of the physical world is concurrency. In this work, process orientation as a software architecting paradigm is taken as basis for developing dependable concurrent software. The term “process orientation” in this thesis pertains to variants of the dataflow-driven software design, where a process represents the basic building block component of software functionality, thus different from the meaning of process of system/service development/delivery, i.e. an evolutionary change, restructuring and improvement in organizational sciences (Forsberg, 1998).

As, according to Moore's law (Moore, 1965), transistor density in integrated circuits doubles approximately every 18 months, the market and, accordingly, the industry expect a growth of the computer systems performance at equal pace, by putting consequently more and more functionality in software.

This thesis advocates that, besides improved software functionality and performance of embedded computers, a piece of the growing computing power must be devoted to the dependability aspects, by viewing them as primary design objectives.
1.1 Motivation: dependability versus immaturity

Ubiquitous computing, as the omnipresent penetration of computers is termed by Xerox PARC (Weiser, 1991), or pervasive computing by IBM (Ark and Selker, 1999), stems from spectacular advancements in the micro- and nanoelectronics according to Moore’s law, which has been valid over forty years. In order to attain a full benefit of the revolutionary miniaturization and corresponding increase of computing power, the hardware progress has to be proportionally matched by the software production technology. However, that is not the case: the demands for harnessing the available hardware power are not followed by the mastery of crafting adequate software solutions. The lost balance between the progress of the hardware and software technology causes virtually all “hi-tech” projects to experience tremendous delays, budgets overruns and unreliability – symptoms of the software crisis. Under the market pressure, the picture worsens taking into account premature enforcing total computerization of safety-critical systems.

Many industrial projects deprive of deploying extensive dependability mechanisms in order to avoid related overheads. This thesis advocates that it must be accepted that a proportional part of the constantly increasing computing power has to be used to afford higher levels of safety and fault tolerance. Otherwise, the post-industrial society would leap into a hazardous adventure with uncountable consequences, which have already emerged numerous and frightening. Whole books, articles and a myriad of Internet resources have been published about the dark side of the premature introduction of computers into many technical systems under profit/domination pressure (Leveson, 1995; Neumann, 2005; Goldstein, 2005). Recognized experts in the field contend infancy of the software production: (Parnas, 1994; Brooks, 1995; Martin, 1996). Nevertheless, embedded systems are rapidly penetrating vital public and personal technical systems, making the embedded software quality safety-critical.

But also the profit drive itself is jeopardized, since embedded systems quickly become a crucial business-critical aspect. Namely, embedded software reliability is an issue for the competitive position of a company at the market. Consumer electronics (mobile phones, handheld computers, modern television systems, white goods), office equipment (mailflow systems, printers, high-throughput photocopiers, “all-in-one”s), medical and graphical scanners and the like are non-safety-critical products, but their reliability is a commercial asset. It goes without saying that the public, got used to all kinds of “intelligent” gadgets, becomes more and more sensitive to the slightest malfunctioning and performance hick-up’s of the “embedded products”.

Ubiquitous networked computing nodes, named “electronic dust”, are shaping everyday’s environments into so-called smart surroundings (Smart Surroundings project, 2005), lending themselves for the infrastructure of ambient intelligence (Aarts and Marzano, 2003). Smart surroundings are characterized by high topological reconfigurability, (wireless) ad-hoc networking, concurrency and customization.
Those are the requirements imposed to the information technology. But what is the next major paradigm shift in software production that may empower these gigantic-scale intelligent systems?

First of all, we should admit that the state-of-the-practice level of software production is hardly to be termed “engineering”, but development at best, if not craftsmanship (McBreen, 2001) or art in many cases. In order to recognize a creative activity as engineering, it has to have certain qualities, as formally rigorous design, quantified quality assessment and predictability (Dijkstra, 2001; Wang, 2002; Selic and Motus, 2003). Metrics of software production are not widely established, quality of the software is not predictable in design time and is mainly guaranteed by testing (Willcock et al., 2005) – but as Dijkstra famously observed, “program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence” (1972, p.864). In short, formalized (and even more: mathematically rigorous) reasoning in the software development process in industrial practice is largely missing, quite opposite to recognized engineering disciplines, as civil engineering, avionics, electronics, control or mechanical design.

This thesis raises yet another voice in promoting the (embedded) software production to an engineering discipline. It proposes certain approaches for improving software quality in design phase through the process-oriented paradigm for concurrent systems. It follows a practical orientation, in a sense that it addresses certain design issues of embedded systems directly, therefore for industrial settings: the main aim of the reported work was a tool-supported framework for designing dependable concurrent software systems with special provisions for (but not limited to) embedded control systems.

Dependability, as a term signifying a couple of software qualities as indicated in Figure 1-1, is proclaimed in (Avižienis et al., 2004) as a holistic measure of amount of trust in computer-supported systems. Embedded software is to be built with having all the quoted qualities in mind, with, as this thesis motivates, an emphasis on reliability and safety.
1.2 Objective: dependable embedded systems

99% of the worldwide produced microprocessors are used in embedded applications (Burns and Wellings, 2001, p.1). Strict concise definitions of embedded systems are presented in section 2.8, p.58. However, before proceeding further, it is worth trying to clarify a bit more precisely what the term “embedded system” means. Essentially, it pertains to a computer system that is a part of a bigger physical system and is responsible for supporting functionality of that integral system, which is not (solely) an information service. Moreover, embedded computers are often invisible, such that the users are not aware of their presence (like in elevators, watches, microwave ovens, washing machines, weapons, cars and so forth).

Having highlighted immaturity of the software production process, it is obvious that with the proliferation of ambient intelligence modern society actually gets surrounded by an unreliable, and maybe worse, unsafe environment! Software reliability prediction and measurement is not yet a well-established discipline (Burns and Wellings, 2001, p.127). This statement is not a surprise taking into account figures that illustrate how much trouble the software industry has to release operational systems in the first place. 15% - 25% of software defects are delivered to customers, 40% - 50% of total development costs are wasted on avoidable rework (McGibbon, 1999); 55% of large distributed systems projects cost more than expected, 68% overrun their schedules, and 88% require redesign (Galin, 2004); 70% of the software products do not deliver the contracted functionality (Verhulst, 2005). It has been known in advance that between 5 and 15 percent of the estimated US$10^12 worldwide investments in IT in 2005 would be abandoned “before or shortly after delivery as hopelessly inadequate” (Charette, 2005).

Innovations in software technology are less quickly recognized and introduced in industrial projects than hardware advancements. The reasons are, among others, lack of software integration means (as standards for interfacing components) and quantification of software properties. It should be borne in mind that software artefacts do not comply with physical limitations characteristic for other branches of engineering, since they are product of thought—put differently: “in software anything is possible”—(Leveson, 1995, section 2.3). Consequently, the confidence in innovations is won much harder and slower than in hardware industry. For instance, the dependability framework of atomic actions (Lomet, 1977), one of the superior architectures for fault-tolerant programming in concurrent environments, after almost thirty years of existence is still scarcely used as a standard solution.

Then where are the solutions seen? To cope with development of increasingly complex software, many CASE tools have appeared along with new (constantly improving, but also constantly superseding) design methodologies pleading to manage the problematic complexity. Industry is permanently looking for ways to decrease a product’s development cycle (“time-to-market”) and increase production efficiency, and (by the way?) quality. Is this enough to make the desperately needed technological breakthrough?
The greatest problem lays in contradicting interests of the economical
benefit and the quality benefit. This can be made obvious on a hypothetical
eexample of introducing a new, say superior, programming language. Namely,
many complaints are made against the dominating C/C++ infrastructure, but
proposals and (academic) projects on new languages have to count in
advance on a very low probability of any wider use. (Java is not an exception
here, since its success mainly comes from its great resemblance to C++ and
much more because of its innovative portability solutions than its novel
linguistic qualities). Namely, many project managers argue that a great share
of a popular language implying a huge knowledge base will compensate the
lack of methodological background.

It is widely admitted that applications of formal methods in software
design hold the greatest promise. This statement holds already more than
two decades; still the penetration of formal analysis, design and verification is
not a common practice. The application and tools support for this technology
still requires too high impacts on traditional software production
infrastructure, which prefers much more something like a “press-the-button”
solution. However, many share an opinion that the next revolutionary
breakthrough is expected in using formal methods for proving software
correctness (Design Tools project, 2001-2005a).

This thesis presents a combination of various approaches to raising
dependability of (concurrent) software—including formal verification—
particularly suitable for dataflow-oriented modelling, which in turn finds an
excellent architecture in process orientation. The quality of the approach is
evaluated by contributions to each of the dependability attributes presented
in Figure 1-1 and consequently by provision of a satisfactory error coverage
(as concluded in Chapter 7, p.216).

1.3 Embedded control systems

Being carried out in the Control Engineering group, this research found a
fruitful pilot-application domain in concurrent implementations of embedded
control systems: this application area is considered particularly interesting
for development and demonstration of software dependability instruments. In
this section the principles of embedded control systems are illustrated. The
evolution treated starts with the two basic ingredients: a controlling system
(computer) and the controlled physical object (the appliance, often named
plant or process). In Figure 1-2 the two parts are sketched isolated.
1 Introduction

Figure 1-2 Primordial components of intelligent (smart) systems

1.3.1 Embedded systems

The shape of an embedding-enabled computer (embeddable or embedded computer in the remainder) on the left in Figure 1-3, deviating from the regular rectangular form from Figure 1-2, is to suggest that embeddable computers often are not usable in a conventional (desktop PC) way – they are dedicated to integration in a bigger system, and only in that configuration they are useful. In turn, often a system designed to embed a computer to support providing the desired functionality is no longer autonomous (“Embedding appliance” in Figure 1-3, following the notions of embedding system from (Eggermont, 2002, p.116)). Only together functionality of the parts contributes to the behaviour of the integral system (Figure 1-4).

Figure 1-3 Constitutive components of an integral smart system

The left configuration in Figure 1-4 relates to the applications where the embedded computer delivers, among other functionalities, also the user interface. It can, however, be well completely invisible from the outside (the righthand side configuration).

Figure 1-4 Integral service-delivering (smart) system with an embedded computer
The widely spread term “embedded system” is often ambiguously used, introducing confusion whether it refers to the computer part of a considered system or the system as a whole. For instance, in (Grehan et al., 1998, p.3) the following sentences suggest that the ensemble is called “embedded system”: “Your programmable microwave is an embedded system. Your VCR is an embedded system. Your TV remote is an embedded system. And if your TV is programmable, it’s embedded, too.” Even in this text page 4, the term “embedded products” alluded to the integral system as a whole.

However, in the scientific and technical literature in the field the term “embedded system”, as also a proper English interpretation suggests, relates just to the computer system which is embedded into an operational ensemble. In absence of an established term for the “integral-system-supported-by-embedded-computer-system”, this ambiguity may be considered useful and non-harmful. Nevertheless, in the context of this text a more precise naming and the position of the software subsystem is required, and therefore the “embedded system” will designate only the computer system embedded in the operational, service-delivering integral system, which will be in the remainder referred to as smart system.

Another interesting terminology phenomenon emerging in the literature with respect to the naming conventions considers “embedded systems” a synonym for “real-time systems” (Grehan et al., 1998, book’s front cover; Burns and Wellings, 2001, p.1; Cooling, 2003, p.12). Although this is incorrect, as it will be clarified later, it is interesting to see why it is so eagerly used.

1.3.2 Real-time systems

Software for many computer-supported systems is constructed barely with its qualitative features (services) in mind, provided that by the implementation itself the services will be delivered to the user within reasonable time. In other words, in its construction the timely response is not explicitly taken into account. Indeed, for applications of many kinds the focus is on data-processing correctness only. For some other systems however the timeliness of the response to external stimuli is important or even crucial. This other group is referred to as real-time systems.

In the literature (Burns and Wellings, 2001, p.435), time-criticalness is ranged from interactive (systems without specified deadlines which strive merely to “adequate response times”), via firm and soft, to hard real-time systems (where missing deadlines represents system’s failure). Colloquially all real-time systems except hard real-time are called soft. This class is also often referred to as embedded data systems, where the relevant behaviour of the appliance can completely be described by waiting times between subsequent commands from the software (Broenink and Hilderink, 2001).

Many real-time systems are combinations of “hard” and “soft” components. Parts that include humans in the controlflow chain are at the best soft real-time. Output to a human operator, as displaying a piece of information of the system status, is an example. (Even in the case of a message like “Reactor core meltdown in progress…”). But parts responsible
for reacting on certain events, whether internal (as reacting on the core meltdown indications) or from humans (as for instance eject command of a pilot of a jet fighter) are hard real-time.

An inherently hard real-time class of systems are digital closed-loop control systems. Therefore embedded control systems are always hard real-time systems.

### 1.3.3 Control systems

Control systems are taken as example of real-time systems in many textbooks specialized in real-time and dependable software development (Anderson and Lee, 1981; Burns and Wellings, 2001; Cooling, 2003) because in digital control dealing with timing is an explicit functional requirement.

To give a proper overview of applications of computers in control, it must be mentioned that control theory and practice started developing in a very different shape than it is nowadays. Namely, while modern control systems involve extensive use of electronic digital computers, classical control theory was based on considerations of the controlled object as well as the controlling components as continuous (analogue) systems. Nowadays analogue components devoted to governing operations of controlled objects are exceptional (for example, a bimetal in an electric iron or water boiler, keeping temperature in a certain range).

The bridge between classical “analogue” procedures for designing laws for operating controlled objects and “digital” implementation of these laws is established by the theory of discrete-time (or sampled) systems (Åström and Wittenmark, 1997), i.e. discretization. Many “modern” procedures are developed to perform design in the discrete domain directly. This gives control systems their hard real-time aspect: regardless if control laws are directly designed in the discrete domain or firstly in continuous domain and than discretized (becoming actually algorithms), their execution on digital computers is coupled to the sampling frequency. The inverse of the sampling frequency—sampling period—represents a time interval within which all relevant input variables have to be sampled, fed to the control algorithms and output as manipulating (steering) instances.

In the scope of this thesis, digital controllers will be discussed exclusively. Almost all industrially interesting control systems nowadays comprise a digital computer. (Hence also new irons—as soon as they will go on-line according to the ambient intelligence visions—will have an “Internet-enabled” microprocessor inside). The controlling computer has to be properly interfaced with the controlled object in both directions (input/output), having measurements of the state of the control object as inputs (I), and influencing that state towards the wanted state by outputs (O).
1 Introduction

Hence there is a direct resemblance with the general configuration of an embedded system from Figure 1-3, as depicted in Figure 1-5; however, for some applications (as safety- or mission-critical), due to unreliability of computer systems, there is a requirement that the controlled object should be able to operate autonomously or instructed by a human operator. In normal operational mode, the computer is coupled to the controlled object as in Figure 1-6. The necessary interaction is usually arranged as a loop. Thick lines in Figure 1-6 represent interaction of the control system (computer) and the plant. Dashed lines indicate dynamical responses to the steering signal and the feedback signal of the plant and the controller respectively.

Besides having an inherent real-time quality, the control systems are in particular a challenging application target, since they are:

- structurally and functionally concurrent,
- often safety-critical,
- as often being used in huge industrial systems, they are business-critical, thus with a high demand on reliability.

Concurrency means simultaneousness of activities within a system. The simultaneousness is an inherent property of control systems, since they consist of sensors, actuators and controllers that naturally operate in parallel. Hence the corresponding software components should treat them in a similar manner.
1.4 Embedded software

Concerning design of embedded systems, the focus of this thesis is on software for embedded computers, and building embedded software is a lot different from development of standard desktop applications. A mix of unprecedented development difficulties and stringent production requirements creates a specific set of challenges to reconcile often contradictory demands as:

- versatility of the underlying hardware and portability issues,
- for mass products, low cost with minimal hardware capabilities,
- difficulties with testing in realistic exploitation conditions,
- immanent concurrency,
- ultimate reactivity,
- real-time behaviour (often hard real-time),
- high dependability,
- (often) low energy consumption – battery-backed systems,
- (sometimes) hostile operation environments,
- managing consequent complexity.

A few of the most intriguing (and at the same time rather general) embedded software development challenges are commented in the sequel.

1.4.1 Concurrency

By its nature and according to the name, software embedded in appliances that are situated in all kinds of places in the concurrent world has concurrency as a crucial characteristic. For achieving an ultimately requested reactivity, embedded (control) systems consist of a multitude of components that operate simultaneously. Therefore, in the design of the supporting software systems, the accent is on addressing this inherent concurrency explicitly. The main benefit of founding a software development paradigm on a concurrency-aware ground is the ability of capturing this kind of simultaneousness in the most natural way.

It is a general intention in modern software development to make software models resemble real world problems at hand. These notions have directed evolution of the software development from structured techniques of 1970’s towards newer technologies as intentional, extreme and contractual programming paradigms and object-, agent-, aspect- and, as put forward in this approach, process-orientation. Also the newest programming languages, as Ada and Java, provide language constructions for addressing the development of concurrent software directly. “The term concurrent indicates potential parallelism. Concurrent programming languages thus enable the programmer to express logically parallel activities without regard to their implementation” (Burns and Wellings, 2001, p.180).

Besides the concurrency-specific pathological problems like deadlocks and livelocks, parallel programs due to the intrinsic
simultaneousness are significantly more difficult to verify than serial ones. Summarized, challenges of dealing with concurrent software by the classical software development paradigm are qualified as:

- hard to reason about simultaneous activities,
- hard to model,
- looking at the dependability, according to (Anderson and Lee, 1981, p.149/150), parallelism exacerbates the difficulties of damage assessment since it can increase the ease with which damage may spread through a system,
- parallel software is being harder tested than sequential (Design Tools project, 2001-2005a).

However, handling the concurrency explicitly in software development:

- provides a high fidelity model of the functioning of real world problems, thus reducing the complexity of the design,
- allows simpler compositibility of software components as building blocks, featuring intuitive extendibility and reusability,
- boosts throughput and reactiveness of a design,
- transparently bridges the gap of distributing interleaved (timeshared) activities from one processing node on physically concurrent—possibly heterogeneous—processing entities.

1.4.2 Complexity

Hilderink (2005a, p.14) defines complexity as “the amount of thought it takes a person to grasp a problem and/or to develop a solution to that problem”. According to (Evans and Marciniak, 1987), it is a the degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures. Perhaps the simplest definition comes from (IEEE, 1990a) as “the degree to which a system or component has a design or implementation that is difficult to understand and verify”.

Complexity of modern systems in everyday use is for industry the primary driver for seeking better software development methods. Complexity comes from numerous interactions, often unstructured and undocumented, between a myriad of components, some not enough tested, some legacy from in-house or external projects. As highlighted by Wijbrans (1993), a software design methodology pleading to help managing complexity has to embrace these three capabilities: abstraction, partitioning, and hierarchy.

1.4.3 Real-time behaviour of software

“A common misconception is that real-time systems are equivalent to high speed computations. The important issue is that a real-time system should execute at a speed that matches, and makes it possible to fulfil the timing
requirements of the surrounding (embedding) system. In most cases this of course means that the execution speed is very important. It is, however, not this issue that makes real-time systems different” (Wittenmark et al., 2002, p.73). The most important problems of real-time design are:

- specifying temporal requirements (timeouts, deadlines, delays, frequencies),
- predictability of a design with respect to temporal specification.

The requirement of predicting temporal behaviour of an embedded computer system is one of hardest, if not ultimately the hardest development problem. In combination with requirements for distributiveness, heterogeneity, fault tolerance and portability of embedded software, guaranteeing dynamical properties and reactivity gets even more difficult. Therefore, in designing embedded software for systems that fall into the category of hard real-time and mission-critical, traditionally the worst-case design principle is followed, which yields oversized, and consequently overpriced products.

### 1.4.4 Formal modelling and verification

There are many arguments that the lack of software quality comes from ad-hoc modelling and design approaches that give up structured architecting and iterative (round-trip) design under time-to-market pressures. Due to these effects some authors dub the common way of software development “cottage industry” (Andriole, 1995; Cooling, 2003). This means, the software development can be in many phases attributed as chaotic. Opposite to chaotic stands formalized design. But what in fact makes design of software formal?

Use of some software methodology and/or a CASE tool does not imply that the design is formal, because the arbitrariness in interpreting design artefacts is often present. Many proposals of numerous software design methods claim a formal background of their methods, but in this text only those formalisms with strictly defined syntax and semantics are deemed formal. Actually, the following capability makes a design process formal: to model a system in such a way that the model can be questioned against a certain condition so that an unambiguous answer on fulfilment of the condition can be obtained. Or put differently: formally specified models can be unambiguously, authoritatively and automatically checked against certain properties.

The way to attain a necessary formalism level of the software specifications is deployment of formal methods (for an overview see section 2.6). However, the large abandonment of formal methods in industrial practice is already famous. The main reasons for low acceptance of formal modelling and design in industrial practice (Hall, 1990; Knight et al., 2001; Broadfoot and Hopcroft, 2003; Sharpe, 2004) hold because the formal methods are:
incompatible with existing design methods,
• requiring expert (mathematically involved) knowledge – thus expensive,
• incomprehensible for non-expert stakeholders, not easily mapping to specifications in natural language or graphical representations,
• not straightforward in expressing either design or assertion criteria,
• focused on the system as an isolated entity,
• not supported with easy-to-master and -use tools,
• poorly scalable, in the case of the dominating model checking procedures caused by thorough examination of all possible system states and tools' limitations to cope with the so-called "state-space explosion",
• rigid when changing system requirements.

But only formal methods are giving firm guarantees on certain vital software qualities, and therefore are superior over any other kind of verification (code reviews, testing or simulation). However, formal methods are not a silver bullet in the software verification. Formal checking is useful to assess software reliability on conceptual abstract level (deadlocks, livelocks, nondeterminism) and some substantial qualities, as liveness and safety of event sequences. Still, at the abstraction level where the formal methods operate, some potential dangers are not visible – non-reliable resources for example (Gibbs, 1994). Other verification means have to complement the checked architectures in order to protect a system's integrity further. Simulations as a qualitative verification give irreplaceable interactive feedback to the system designers. As formal methods verify a system model, not the actual system, complementary techniques as functional testing are needed to find implementation errors, (Katoen, 2004, p.31). Moreover, some things can never be proved and also people make mistakes in the proofs of those things that can be proved (Hall, 1990, p.12).

A few fresh research approaches leverage the power of applying formal modelling and verification beyond treating the developed system only. For example, to formally model the environment of the system (Brinksma et al., 2005). Or to apply the formal methods in the complementary testing technology – the test specifications are simpler than the system itself and by nature more formalizable than specification of the system under development (Huima, 2005).

1.4.5 An intermediate summary of the issues

Difficulties of dealing with embedded control software are multifold, because it has all the problems of development of systems that are:

• embedded
• concurrent
• reactive
• real-time
• critical
• and in light of combination of the previous, inevitably complex.

The challenge this research faced was devising a tool-supported methodology to explicitly and as much as possible independently address each of the peculiarities of the embedded control software, with accent on dependability.

1.5 Dependability

Dependability is the system property that integrates various important software quality attributes, as presented in Figures 1-1 and 1-7. Dependability of a computing system is the ability to deliver service that can justifiably be trusted (Laprie, 1985, 1995). It is notable that the given overview of dependability attributes (Figure 1-1) does not include security. A security level indicates a system’s immunity on intentional (malicious) attacks. This thesis focuses is on all unintended violations of trustworthiness of a software system – the objective is rectifying insufficiencies in development of dependable concurrent software. The term “security” will be used in a very narrow sense only for those properties of programming languages that prevent harmful (but again unintentional) consequences of unawareness or ignorance of some design issues. Avižienis et al. (2004) promote the attribute of confidentiality in addition to the quality attributes defined under dependability to cover also security issues. Figure 1-7 renders how security and dependability share some of the concerns.

Key areas of elaboration in this thesis are safety and reliability. Safety refers to a system’s, as well as its environment’s, healthiness preservation during specified operational conditions. Reliability is a measure of the “up-time” of a system (Douglass, 2003). Despite important differences between the notions of safety and reliability, they are often incorrectly used as synonyms. If a safe system fails, its failing must not have undesired consequences either for the system or its environment. However, the reliability of a system does not depend on what happens after the system fails – what counts is only how often and how long are the system’s service outages; the reliability of the system remains the same whether the system fails safely or not safely. A safe
system may fail frequently as long as it does not cause accidents or losses (Douglass, 2003), while an arbitrary rate of fails is in a direct contradiction with the notions of reliability. However, reliability and safety are often tightly correlated. Typical examples are systems without a safe state, for instance airplanes, where safety is a function of reliability of the flight systems.

In this thesis the reliability issues are mainly addressed through fault tolerance techniques. Elaborated safety and fault tolerance measures prescribe application of several procedures, mechanisms and design patterns for process-oriented development of software. The proposed patterns and mechanism are not “heavy” in the sense that they do not require restructuring an initial process-oriented design, like for instance introduction of fault tolerance based on atomic actions would require.

1.5.1 A short survey of established safety and fault tolerance concepts

Historically, first safety-dedicated redundant hardware has been used as blocking (shutting down to safe states) measures in safety-critical parts of complex designs. For price reasons (note a common, but increasingly arguable assumption that software development is cheap), there was a trend to put as much functionality in software as possible. However, due to emerged complexity of software systems, recently a balance is sought in hardware-software co-design approaches. Therefore, some functional blocks that could be implemented in software stay (or are being redesigned) in hardware due to performance and simplicity reasons – hardware is still easier to parallelize than software.

As for hardware defensive components, it is also characteristic for software means that they always introduce redundancy in design. Redundancy is the key for rising dependability in the system (Douglass, 2003). In any fault tolerance approach one will find redundancy in one way or another, static or dynamic or both. Also, some advanced design techniques may be seen as “redundant” compared with the bare development of systems “that run”. Let us start this overview with these “redundant” software development approaches.

Hazard Analysis

In order to combat possible safety threats (by preparing a system to react accordingly in accident situations), it is necessary to understand those threats. Methods for anticipating risky situations rely on:

- analysing scenarios – as Failure Modes and Effects Analysis (FMEA), Failure Modes, Effects and Criticality Analysis (FMECA), Fault Hazard Analysis (FHA),
- causes and consequences as causal branches – Fault Tree Analysis (FTA), Event Tree Analysis (ETA), Cause-Consequence Analysis (CCA), State Machine Hazard Analysis (SMHA),
• various risk assessments, acceptance analyses and special Hazard Analyses, as: Preliminary (PHA), (sub)System or Software (SHA), Operational (OHA), Hazards and Operability Analysis (HAZOP).

Many of these techniques, in detail discussed in (Leveson, 1995), are developed in the visual form and are supported by specialized as well as rather general drawing tools, as for example MS Visio (for instance Fault Tree Analysis, Audit and Cause and Effect diagrams).

**Formal verification**

Verification methods, as model checking and theorem proving, are believed to be the next great breakthrough in the software development technology. Application of formal verification methods is already established for all high risk systems. However, due to immaturity of tools and a need for mathematically-involved trainings, formal methods find a lot of resistance in introduction into everyday practice.

**Model-driven design (MDD) and architecture (MDA)**

MDA is an approach to using models in software development (Miller and Mukerji, 2003), initiated by the Object Management Group, the same body that promoted the UML (OMG, 2001). The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification. The central idea here is that models, created using modelling languages such as UML, should be the principal artefacts of software development instead of computer programs (Selic and Motus, 2003).

In fact, MDD is promoted by its proponents as just another evolutionary step in the development of the software field. “The magic of software automation from models is truly just another level of compilation” (Miller and Mukerji, 2003).

**Automatic code generation (ACG)**

The potential behind software models and the maturation of automatic model translation techniques has increased the interest in model-oriented development methods. The final goal of software modelling is automatic transformation from model to computer programs. Once a satisfactory model is constructed, automatic mechanistic processes can generate the corresponding computer programs (Selic and Motus, 2003).

The two principal reasons for making a software modelling tool capable of generating source code out of (graphical) software models is elimination of error-prone manual transformation of software models to implementation code and addressing deterioration of an architecture during the software maintenance phases (Parnas, 1994; Van Gurp and Bosch, 2002). Having these gains in mind, it is clear that an automatic code generation mechanism is an effective means in covering a broad class of implementation design errors.
Safety and reliability design patterns

Design patterns result from proven maturity of concepts for building software. They are statically redundant, because of employing components that stay in operation regardless errors occur or not. They reflect the notions of software reuse, which arose from the struggle against building software systems from scratch again and again. Many functionalities are commonly required by various software applications - while mechanisms for solving them tremendously differ in flexibility, genericness, scalability, extensibility and so on. Over the years, certain successfully applied mechanisms converged to versions with well-balanced properties. Those are recognized by competent software analysts and designers as software design patterns (Gamma et al., 1995; Douglass, 2003).

Various specific design patterns are devoted to observing a system in order to indicate and/or record suspect trends that may endanger system’s integrity. While general monitoring components observe states (conditions, variables) by value, watchdogs are established as specialized for monitoring temporal disruptions in a (sub)system’s functions. Many systems make use of logs and audit trails to perform post mortem analysis in order to improve system’s dependability by learning from ensued incidents or simply optimize the systems’ functionality or trace transient malfunction that do not leave material evidence. System’s logs are useful both for on-line as well as off-line analyses.

Checksums, autodiagnostics and functional assertions

Many other techniques, less structured than design patterns, are often used for backing system’s integrity. They often involve, in addition to functional redundancy also information redundancy. Examples are error-detection means (as parity bits and CRC checks) and self-corrective data structures, as Hamming code.

Recovery blocks

Recovery blocks are one of the first structured fault tolerance techniques based on notions of dynamic redundancy. Assumption of using recovery blocks is that result of each critical software component, before transmitted to the rest of a system is subject to the acceptance test (Horning et al., 1974). Acceptance tests are a kind of postconditions that a component must fulfil, otherwise the state of the system is restored to the state where the execution of the critical component started, and instead of that component the recovery block is being executed as an alternative. Recovery block’s result are also subject to the acceptance test, until the test is passed or there are no more redundant blocks to go (in which case a system failure is declared). Since the redundant recovery blocks are activated only when the acceptance test fails (thus after an error detection), this dependability approach is dynamic. Moreover, it is an example of backward error recovery, since the state of the system before an error manifestation is restored before the fault tolerating procedure commences.
Atomic actions

Atomic actions is a design pattern that builds on dynamic redundancy provided by recovery blocks (Lomet, 1977). It is a pattern that addresses error recovery in interdependent concurrent flows of execution. The idea is to synchronise concurrent components on entering critical regions where mutual collaboration takes place, so propagation of errors (information smuggling, (Kim, 1982)) may occur. Concurrent components are allowed to communicate with others registered as participants in the mutual action, but not with the outside world – this is the reason the collaborative action is called atomic. Before exiting the action, all participating components have to pass acceptance tests. All of them have prepared recovery blocks in case that some of them fail the tests. Often there is a need to distribute the reason of failure to all participating components in order to spawn a proper recovery block for all components. Point of synchronisation on leaving the atomic action can be understood as barrier synchronisation (Arenstorf and Jordan, 1989).

Exception handling

Exception handling is a dynamic redundancy mechanism that allows system architects to distribute dedicated corrective or alternative code components at appropriate places within the software architecture to maximize effectiveness of error recovery. Therefore, it successfully covers broad classes of anticipated intermittent errors and effects of environmental failures within a software system. The most important feature expected from an exception handling mechanism (EHM) is separation of nominal execution code and part of the code for treating exceptional situations.

Exception handling is primarily a forward error recovery mechanism, since it has the potential to try to tolerate faults without rolling back the system to the last known healthy state. However, it is shown in early works on concurrent exception handling schemes (Campbell and Randell, 1986) that it can be very well used to implement backward error recovery schemes, as recovery blocks and atomic actions. Anderson and Lee (1981) base their theory of fault tolerance on existence of an appropriate EHM.

Of all listed redundancy approaches, either in the development process or executable systems, this thesis integrates some aspects of model-driven development and code generation (in Chapters 3 and 4), formal verification (Chapter 4), dynamic redundancy by exception handing (Chapter 5) and a few architectural static design patterns (Chapter 6).

1.6 Design strategies

It has become already obvious that the keyword of the contemporary technological advancements and production is: integration. Integration of services, technologies, disciplines; considering and designing multidisciplinary systems in a synergetic manner, where the waterfall approach is out of question, for at least two reasons. Firstly, market demands
for seamless integration of system’s components from very different domains (of which many “non-classical”, as psychology or ergonomics) causes slight changes in one of them making substantial influences on the others. Secondly, it is a wish to have at the designing/modelling time a capability to optimize the system as a whole, balancing responsibilities of the overall functionality over the very different system parts.

Therefore, a design paradigm ideal is an interactive general modelling environment. The main obstacles to such one design environment are design discontinuities among different domains and lack of glueing notions and languages in between.

The Context of the driving research project put this thesis in an interplay among dealing with mechanical systems whose movements are to be controlled with a high precision, where accurate models of the dynamics of the machinery as well as mediating transducers have to be coupled tightly to the controlling intelligence – software. This context has been known as mechatronics for almost four decades (Mori, 1969). According to the backbone-idea of this research, as in Figure 1-8, the focus is on implementation of the control laws in embedded software, considering the realization actuality and the overall physical context at all times, as symbolically represented in Figure 1-9.
Two remarkable design concepts of the greatest importance for understanding the orientation of the research approach of this thesis are briefly introduced in the following two sections.

### 1.6.1 Mechatronics

Emergence of the mechatronic approach in designing sophisticated mechanical systems aims at eliminating discontinuities in the design trajectory of those systems, which are characterized by high precision (as telesurgery systems), high throughput (high speed printers) and the like. Mechatronic design is defined in (Van Amerongen, 2002; Van Amerongen and Breedveld, 2003) as an integrated and optimal design of a mechanical system and its embedded control system.

Apparent gaps in traditional design trajectory of computer-supported mechanical systems were long ago identified – see (Wijbrans, 1993). That research had the same roots as the one reported in this thesis and its predecessor (Hilderink, 2005a) and had been carried out in the same environment, the Control Lab of the University of Twente, though based on different technologies in all design aspects of mechatronic systems. Wijbrans (1993) relied of Hatley-Pirbhai (Hatley and Pirbhai, 1998) structured software analysis and design methodology and transputers/occam technology for implementation. However, in common with this thesis approach, the design philosophy of dataflow-driven system reasoning had been backed by the mathematical formalism of Communicating Sequential Processes – CSP – (Hoare, 1978). Wijbrans aligned the technologies of that time to support gradual evolution of models from abstract conceptual level to detailed implementation optimizations, as discussed in the following item. He combated the design discontinuities—calling them gaps at boundaries of different traditional engineering fields—by putting forward proper models as the central issue.

### 1.6.2 Stepwise refinement

“Modern system design begins in problem domains, usually with considerable informality, and ends in computer domains in completely formalized languages for programmers and users” (Mills, 1988). Transformations between these two abstraction extremes are being done in many ways, varying over companies and projects. The projects of interest are multidisciplinary and consist of numerous design phases.

Selic and Motus (2003) state that “in contrast to models in most other engineering disciplines, software models have a unique and quite remarkable advantage: they can be directly translated into implementations using computer-based automation. Thus, the error-prone discontinuities encountered in other forms of engineering disciplines when switching from a model to actual construction can be avoided. This means that we are able to start with simplified models and then gradually evolve them into the final products. Models that can be evolved in this way have to be fully formal and,
consequently, have the added major advantage that they are suitable for formal analysis."

A gradual and integral approach in multidisciplinary and/or complex projects is a subject of both industrial and academic research for a couple of decades. An ideal of stepwise refinement from higher towards lower levels of abstraction along the evolution of an engineered system has been described in (Wirth, 1971) and further elaborated in an early book of Dijkstra (1976).

Stepwise refinement is all about controlled and structured development of complex systems from the top abstract level to the low level of details with having one eye on the overall picture of structure and functionality of the complete system at hand. In mechatronic design, stepwise refinement supports the separate development of parts of the system and later integration of these parts, like in an idealized basic building-block approach. Support for separate development enables design of a system by a team of engineers, rapid prototyping of specific (problematic) parts of the system, and an evolutionary approach (Wijbrans, 1993): “In this approach, the system is no longer developed sequentially in separate parts, rather it is treated as a whole. Design decisions influencing system behaviour may change the process, the control algorithm or the sensors and actuators”. Stepwise refinement means that the total model (from a physical system to be controlled to control laws implemented as efficient concurrent control computer code) will gradually change from a basic functional or conceptual model towards a detailed model from which the code for the control-computer system can straightforwardly be generated and downloaded to the target platform (Jovanovic et al., 2003).

It is notable that Selic and Motus (2003) advocate formal approaches as necessary for flawless migration from one abstraction level to another. Many design approaches boast with supporting the stepwise refinement, although only formal theories (more precisely: process algebras) clearly articulate exploitation of this concept. The CCS-based formalisms (see section 2.7.1 on page 54) put forward bisimulations, while the CSP theory names the technique directly (Roscoe, 1997, p.46/47), as a generalization of the CSP concept of checking plausibility of gradual transitive refinements by series of specifications towards the final implementation. This is but one reason to consider the CSP foundation as a sound background for developing complex systems. Many other are discussed in the next item.

1.7 Process orientation and CSP/CT

The term process orientation in the sense of software architecting paradigm in this thesis pertains to variants of the dataflow-driven software analysis and design (DeMarco, 1978; Ward and Mellor, 1986). The term is known in organizational science and business (Forsberg, 1998) in a much more general context, with focusing on process as an organized activity, “a series of actions or operations conducing to an end” (Merriam-Webster, 2005); in the narrow terms of architecting software, it has not been officially defined, although legitimately used in many publications on concurrent programming, for
instance (Xu et al., 1995; Romanovsky and Sandén, 2001; Welch, 2002). Deriving from the CSP meaning of process as a behavioural entity, a process-oriented architecture favours restricted “channelled” communication among processes as fundamental functional entities, assuming arbitrary concurrency among them.

Processes are autonomous, strongly self-contained entities with well-defined interfaces. Processes communicate with each other by message passing over channels. These qualities resemble separation of concerns fostered by component-based software architectures. On the other side, processes and channels are often implemented as objects, exploiting useful reusability concepts of inheritance and polymorphism of object orientation (Booch, 1990). However, the most important difference comparing to object-orientation is the far more articulated (restricted) way of interaction among functional entities (processes), which features simpler analysis of both data- and controlflow within the architecture. By this, several anomalies detected in object orientation are eradicated, as aliasing, for instance (Locke, 2001).

Moreover, the concept of a process has traditionally developed different notions with respect to behavioural specification as a software entity. “Objects structure data and code while processes structure behaviour. Unlike objects, processes embrace observable properties of a concurrent program, such as reactivity, timeliness, responsiveness, priorities, and performance” (Hilderink, 2005a, p.43). However, one cannot rely on the traditional notions when interpreting a design paradigm. Therefore it must have a rigorous, preferably formal, semantic background with a clear meaning of all the elements constituting the paradigm vocabulary. This section introduces a semantics founded on algebraic grounds, based on the concept of denotational behaviours and events from CSP, which gives an excellent foundation for reasoning on reactive and concurrent systems.

Even without a mathematical semantics, it can be suggested that notions of process orientation yield the following architectural benefits to building quality software:

- Simple architecture,
- Communication model,
- Reusability,
- Maintainability and extendibility.

A process-oriented architecture has no more abstract ingredients than processes, channels and compositional constructs ruling the execution policy of processes. The communication model is message passing, a superior communication model with respect to the information smuggling threats (Jalote and Campbell, 1984, p.348). Processes, as components, are loosely coupled through well-defined interfaces to the rest of the system – thus being highly reusable. Although with known difficulties for interface extensions of an isolated process in some implementations (Locke, 2001), a process-oriented system is easy maintainable (thanks to clear separation of concerns among processes) and extendable (for the same reasons, new functionalities are added by extra processes without substantial interference to implementation of an already existing system).
1.7.1 CSP as a modelling paradigm for concurrent process-oriented software

The model of concurrency is not defined within the general concept of process orientation. A certain semantics has to be adopted in order to have a completely specified way of dealing with simultaneousness when building concurrent process-oriented software. It is well known that the principle concern with concurrency does not come from sole simultaneousness of the software entities, but from the sum of the simultaneousness and interaction. Therefore, it is desired that the chosen semantics explicitly supports also the synchronisation and communication issues.

Communicating Sequential Processes – CSP – process algebra (Hoare, 1978, 1985; Roscoe, 1997) is one of four major concepts pertaining to a formal basis of concurrent programming. The other three are: Calculus of Communicating Systems – CCS – (Milner, 1980, 1989), Algebra of Communicating Processes – ACP – (Bergstra and Klop, 1984; Baeten and Weijland, 1990) and Petri Nets (Peterson, 1981). Out of all four, CSP is the closest to parallel programming languages (Olszewski, 1993). CSP gives a clear and simple model of concurrency together with a synchronous model of communication. This is not a surprise, since exactly this combination was target of Hoare’s seminal paper from 1978 on concurrent software and CSP.

CSP can be described as “a collection of mathematical models and reasoning methods” on concurrent systems, with developed operational, denotational and algebraic semantics (Roscoe, 1997, p.149). As a model of concurrency and communication, CSP algebra perfectly lends itself to underpin the process-oriented software architectures. It provides a powerful mathematical notation for reasoning about concurrent systems in general, together with a rich supporting theory, a machine-readable subset called CSPm for producing verification scripts and verification tools (Martin, 1996; Scattergood, 1997; Formal Systems, 2004).

The CSP notions strongly reinforce the already advocated virtues of process orientation, by adding the following instruments to conquer complex simultaneousness and reactivity:

- a formal model of concurrency based on compositional operators,
- a synchronous communication model based on events,
- a model of reactivity (deriving from the previous),
- composibility of building blocks.

The order of execution among interacting processes is unambiguously specified by grouping processes within hierarchical construct-ruled compositions. Scheduling policy comes from interplay of the execution compositions (defined statically in design time) and interaction events between the system and its environment as well as within the system itself. By denoting an occurrence in time and space, the abstraction of events addresses the nature we want to have explicitly specified in the design of embedded systems. Notion of events in CSP provides the model of reactivity.
On the practical side, Ada’s synchronous concurrency is CSP-based, while the transputer (Ivimey-Cook, 1999)—a revolutionary hardware architecture for distributed systems—has been programmed by the pure CSP implementation language occam (INMOS, 1988). In occam a program consists only of processes, considering compositional constructs also as processes. To support more control on reactivity in designs, occam extends untimed semantics of CSP by including timers and primitives operating upon timers. Moreover, alternative and parallel constructs stemming from the CSP operators are available in prioritized versions, called prialternative and pparallel. Occam records many successful applications thanks to its simplicity and security (in the sense of strict consistency checks by the compiler). However, due to inability to compete with multifunctional microprocessors that could address a much wider market, production of the transputer ceased in 1996. The architecture is still actively used in specialized multimedia processors (Stevens, 2005), but not programmable by occam. The experiences with programming with occam were so positive and enlightening that a few universities continued providing occam API in libraries for mainstream programming languages, as the Communicating Threads – CT – variant developed at the University of Twente (Hilderink et al., 2000; Orlic and Broenink, 2003; Hilderink, 2005a). Similar libraries origin from the University of Kent (Moores, 1999; Welch, 2002; Brown and Welch, 2003), which also continues improving the occam compiler with advanced features (Welch and Wood, 1996), making it an increasingly promising alternative to the dominating implementation languages.

Throughout this text the process-oriented software design methodology, conceived in (Hilderink, 2005a) and extended in this work is referred to as the CSP/CT framework. This name is to give a specific meaning to applications of the abstract concepts of CSP in the scope of this thesis. In the first place, the CSP/CT framework refers to a design environment for building process-oriented software, which consists of modelling, verification and implementation instruments. The resulting software systems are being specified by models based on CSP principles and then implemented by the CT libraries. Secondly, the “/CT” suffix can be understood as it implies certain restrictions of the CSP language to the application area of the CT programming domain. Actually, a subset of CSP is used for reasoning on software design issues within this framework. This affects the tooling support for general modelling of CSP systems – only a limited CSPm support is provided, that suffices proving certain software properties of communication patterns, as deadlock freedom. Finally, in the remainder, occurrences of “CSP/CT” design paradigm always imply its process-oriented nature.

The CT implementation of a process-oriented design backed by the CSP model complements a list of desired features for developing embedded (control) systems by supporting:

- Execution platforms heterogeneity,
- Design portability,
- Design distributiveness,
- Performance,
- Real-time facilities.
CT libraries, coming in variants implemented in Java, C and C++, address different implementation needs for various platforms. CT libraries implement their own kernel independent of operating system primitives, and are thus portable to any platform supporting development with Java and C/C++. By having captured all hardware dependence in the implementation of channels and a small, clearly identified part of the CT kernel, a complete process architecture design can be straightforwardly ported from platform to platform, including distributed nodes. By following occam’s addition of priorities, some performance issues—as for instance priority inversion (Hilderink et al., 2000)—are dealt with. Explicit timed sampling, crucial for the control applications, is in the CT handled also by channels (Hilderink and Broenink, 2003).

1.7.2 Dependability potentials of the CSP-based process orientation

Numerous benefits for dependability of the CSP/CT software draw from its process orientation in general and the CSP formal background in particular.

First of all, the key to dependable design is conceptual simplicity and power to manage inevitable complexity. The greatest power of the process orientation is separation of responsibilities of potentially parallel executions taken care of by the processes and interaction (i.e. communication and therefore synchronization) in the channels. Secondly, a sound and simple foundation in place is greatly contributing to the essential strength: ease of understanding by designers, preferably supported by smart visualization of the designs. The CSP/CT framework presented in this text adopts the principles of graphical modelling promoted in (Hilderink, 2005a). Finally, for managing complexity, the whole approach adheres to the principles of abstraction, partitioning and hierarchy motivated in (Wijbrans, 1993).

Simple extendibility of an initial design allows nonobtrusively adding reinforcements of safety and fault tolerance (reliability) as layers in the software structure. The presented methodology strongly emphasizes this potential, which makes it possible to separate concerns of creating the prime functionality of software and increasing its dependability attributes.

The natural interaction modus within the process orientation is message-passing. For sake of generality, shared memory interactions can be modelled by message passing, but for safety-critical systems message passing is just what is required (Requirement on Interference-Freeness in ISO/IEC 61508 standard for functional safety). When looking at the interaction among software entities in object-oriented designs for example, notable there is a rather liberal flow of information through and among objects. This is not a favourable property for high-integrity systems, where possible error propagation should be strictly confined.

The flow of control, which is basically dataflow-driven in the multithreading model of concurrency, like used in object orientation, is actually not object-oriented: the objects are not entities that control the order of execution, since the control flow goes from one to another, following delegation of data processing. Active classes being execution management
authorities, as promoted in the UML, are heavily dependent on the behaviour of the other classes, their delegates. During growth of a system in design time and possible dynamic reconfigurations in run-time, the control obviously slips out of hands unless the active classes know a lot about the other entities in the system. Such a tendency however breaches the principles of encapsulation. A CSP process simply fulfils interaction contracts with its environment adhering to the causality of dataflow coming through input channel ports and going out through output channel ports. Processes are not aware of their identity nor of the identity of the outside world, nor of compositions with the rest of the system and neither the location of the process itself. Being described by mathematical rigor, an ensemble of CSP processes can be verified to be deterministic.

This leads to the last, but perhaps the greatest argument for superiority of the CSP-based process oriented design paradigm for dependable concurrent software: a transparent mathematical specification directly liable to formal verification. This methodology contributes to the initiative for deploying formal methods in software design by providing a bridge from domain-specific CAD tools via CASE tools specialized for software development to standard model checking tools.

The strengths of the process-orientation are also obstacles to its wider use. It imposes a disciplined and restrictive way of design, which defers arriving to first prototypes—though much more dependable—for benefit of detailed modelling and verification. The resulting implementation is less efficient, both with respect to performance (due to sophisticated constructions) and memory footprint. However, the assumption of this research and development is that the execution platforms are becoming faster and richer, either just by higher efficiency in using silicon or by hardware parallelisation. (Which is, by the way, a way easier with using process oriented software solutions).

1.8 Scope, contributions, case studies and outline of this thesis

The research and development results reported in this thesis are part of a broader project of integrated design of embedded control systems and embedding mechanical (robotic) devices, the STW/PROGRESS project “Design framework for heterogeneous real-time embedded control systems”, which lasted from May 2001 to October 2005, (Design Tools project, 2001-2005b). The ultimate goal of the project was establishing the methodology and a toolchain to support a design-discontinuities-free trajectory for developing mechatronic products, according to Figure 1-8, a starting milestone of this project.
1.8.1 Scope of the thesis

Let us be clear right upfront what is not the focus of this thesis. The focus is neither formal methods per se nor the real-time aspects of the proposed concurrent framework; the favourable formal checking properties are taken for granted by use of the external high quality formal checker FDR, while the real-time properties of the framework are drawn from the properties of the CT libraries (Hilderink, 2005a). The same goes for dependability of distributed (control) systems: applicability of the proposed dependability instruments is "distributive" as much as the CSP/CT paradigm is, which is commented in section 2.5.

The primary scope of this thesis is improvement of the CSP/CT software quality with respect to its dependability in mechatronic applications. The other aspect is bridging the gap among distant useful concepts for concurrent implementation of embedded control software: process orientation, CSP, graphical modelling, domain specific CAD tools, model checking and standard dependability techniques.

The thesis elaborates on the following dependability techniques:
1. Automatic code generation,
2. Formal model checking,
3. Dynamic fault tolerance – exception handling mechanism,
4. Static fault tolerance mechanisms in the form of design patterns.

1.8.2 Contributions of the thesis

A substantial part of the research carried out in the research project and reported in this thesis contributes to the dependability of concurrent software by:
1. Developing a graphical CASE tool – gCSP – capable of:
   - graphical modelling concurrent process-oriented software based on the CSP/CT framework
   - code generation from graphical models of designs' formal specification and automatic production of executable code,
2. Exercising the use of the graphical language and the gCSP tool, hence formulating dependability techniques and overall process-oriented design methodology within the CSP/CT framework,
3. Establishing the tool chain with two external specialized tools: demonstrating feasibility for formal checking of the graphical models with FDR and inclusion of the control code generated by 20-sim,
4. Extending the CTC++ library by fault tolerance and safety mechanisms,
5. Demonstrating the application of the developed concepts on two robotic case studies.

The application scope of the thesis has been initially targeted at embedded control systems (striped region in Figure 1-10), although the results are applicable to embedded software in general (greyed in Figure
1 Introduction

1-10). However, certain classes of systems that would benefit most of applying the process-oriented technology can be identified. Applications of the proposed methodology cause inevitable trade-offs of sacrificing performance for benefit of dependability. The targeted class of systems that justify making this kind of trade-offs by applying the CSP/CT dependability techniques are embedded systems with high demands on safety and reliability whose architecture consists of numerous simultaneous agents depending on mutual interaction of causal dataflows. In other words, process orientation best fits into those problem domains whose topology resembles dataflow diagramming. The applicability of the developed design procedures and patterns are demonstrated on a typical class of such ensembles: control systems.

[Image: Application areas of CSP/CT]

Figure 1-10 Application areas of CSP/CT

All examples and case study results reported in this thesis are demonstrable as executable software. This thesis aims at increasing dependability, and in that respect, the quality of software in design time.

1.8.3 Case studies

In the project three robotic case studies were set up, of which in this thesis two are elaborated in detail (JIWY and Tripod). The first basic one – LINIX – is described in (Jovanovic et al., 2003).

Complexity and functionality of the two set-ups is not trivial, in the sense that the control software has to implement typical multimode control regimes for robotic applications: calibration and alignment in the start up phase, servo control (position in both cases) and proper homing before shutting down.
The camera-system called JIWY – besides a sound name, the abbreviation “JIWY” has a meaning only locally in the project/lab environment – is shown in Figure 1-11. It is a mechatronic set-up for orienting a device within a certain spatial angle. It has two rotational joints, therefore characterized as a 2DOF (two degrees of freedom) device. The operational vertical angle is 165° and the operating horizontal angle is 120°. The maximum ranges are limited by mechanical (“hard”) end stops that prevent full swings such that the wires cannot be twisted or damaged. The angles between the hard end stops are respectively 300° and 150°. Each joint is equipped with one DC motor (actuator) and one incremental encoder (position sensor). The wires between these elements and the I/O-interface are bundled in one cable together with a watchdog signal lead. The watchdog signal is used for detecting whether the cable is damaged/disconnected (see section 6.2.3 Integrity watchdog).

The set-up is primarily controlled by an X-Y analogue joystick. Software extensions allow the user to control JIWY by viewing pictures from remote via a network connection (Smith, 2002; Ros, 2004). The I/O interface consists of analogue amplifiers for steering the motor and National Instruments PCI
6024E I/O card in the controlling personal computer (Figure 1-12). The 20-sim model of the all constitutive parts is used for developing control laws and generating control algorithms as parts (“control code”) of the CT-programmed JIWY control software. gCSP models that specify functionality of all control modes are elaborated in section 3.5.1 on page 100.

Figure 1-12 JIWY top level model in 20-sim

Tripod

Tripod is another mechatronic set-up, more complex and much more powerful than JIWY, designed for demonstrating advanced learning control strategies (De Kruijf, 2004). It is a positioning system with 3DOF (three degrees of freedom) thanks to three linear motors that drive a platform (Figure 1-13a and b). A pair of rods connects to each linear motor and end-effector platform. Due to the mechanically constrained movements of the rods, the platform cannot rotate but only translate, being always kept in the horizontal position (Figure 1-13c and d).

Experimenting with a set-up as powerful as Tripod bears much higher hazards than experimenting with JIWY. Therefore for Tripod, 20-sim code generation is used also for creating a C++ simulation model of Tripod’s dynamics. The simulation model has been used for developing the Tripod control software and prototyping and demonstrating some dependability software layers. It has been actually used in the context of Hardware-in-the-Loop – HiL simulations, (Isermann et al., 1999). This concept allows concurrent engineering of control systems when some parts of the system are not available, but replaced by competent simulation models. Therefore the existing parts (hardware) are placed in the virtual closed control loop. Besides facilitating concurrent engineering, this is also an economically beneficial approach to optimize control in systems where service outage means big financial losses or high risks for damaging the system under control. It goes without saying that it is not only important for the system safety, but also in eliminating high environmental risks (Jovanovic, 2001).

The most important safety requirement for Tripod is preservation of the working range of the platform. In (Eglence, 2003, p.12), the safe
cylindrical operational space is characterized by the radius of 170mm and height of 234mm. The only allowed excursion of the platform beyond this space is reaching the bottom of the cylinder by going up from the lowest, off-powered position and coming to it at shut down. In the shutting down mode the safety issue is a low speed when settling down. Reference source for the servo mode of Tripod are motion profiles (“paths”) for the three axes stored in numerical files. These files are loaded in memory before engaging in the servo position mode.

Figure 1-13 Tripod set-up, courtesy of Bas de Kruif (2004)
1.8.4 Thesis outline

The thesis consists of three parts. The first part, divided in three chapters, establishes background and motivation of this research (current Chapter 1 Introduction), the language used in the second part through relevant definitions, conceptual foundation and tooling support (Chapter 2 Key concepts, definitions and tools), complemented by the CASE tool developed in this research (Chapter 3 Modelling CSP/CT architectures with the gCSP tool).

Having prepared the conceptual and tooling background, the second part develops the dependability instruments which are subject of three chapters. The first two dependability techniques, subject of Chapter 4 Automatic code generation and formal verification of CSP/CT software, deal with covering development errors before a system is deployed. Use of dynamic redundancy to cover intermittent internal and external errors once the system is deployed is elaborated in Chapter 5 Exception handling mechanism for CSP/CT software, which proposes a way to handle exceptions in concurrent software. Chapter 6 Dependability design patterns for CSP/CT software, deals with complementing the previous dependability techniques by process-oriented application of several traditional static redundancy dependability measures.

The third part, consisting of the Chapter 7 Wrapping up the big picture, literature references chapter and four appendices concludes this work and provides details for a complete appreciation of the core chapters 3, 4, 5 and 6.

In the text **boldface** characters are used when mentioning a key term for the first time; *italic* characters are an invitation to focus the reader’s attention.
2 Key concepts, definitions and tools

If everything is perfect, language is useless. This is true for animals.
If animals don’t speak, it’s because everything’s perfect for them.
If one day they start to speak, it will be because the world has lost a certain sort of perfection.
Jean Baudrillard

This chapter sets out this thesis’ language by defining precise meanings of the key terms used in the remainder. It illustrates common concepts in the field of designing embedded (control) systems and dependable software, and reviews tool support for a few of the most important concepts. For the most part, definitions and notions are taken from the established references in respective fields, hence this chapter can be useful as a map for a deeper study into these disciplines.

Let us start with a central activity of this study which, according to many prominent authors, actually does not exist yet at the level the name suggests, and that is software building as an engineering discipline. Hopefully this thesis makes a small step to help establish what is defined by:

Software engineering is the profession that creates and maintains software applications by applying technologies and practices from computer science, project management, engineering, application domains, and other fields. (Wikipedia, 2005)

2.1 Notions and standards of software quality

This thesis advocates dependability as a crucial aspect of software quality. Talking to different stakeholders in a software project, one may see “software quality” recognized in different aspects: simplicity, safety, performance, security, predictability, reliability, extendibility, price, documentation etc. “A manager may be more interested in the overall quality rather than in a specific quality characteristic, and for this reason will need to assign weights, reflecting also business requirements, to the individual characteristics. The manager may also need to balance the quality improvement with management criteria such as schedule delay or budget overrun, due to the wish to optimise quality within limited cost, human resources and timeframe” (EAGLES, 1995).

Efforts of increasing software quality are usually focused on requirements engineering, disciplined software design processes and use of
CASE tools, special architectures and design patterns for critical parts of a design, choice of appropriate languages and development environments, code inspection and reviews, exhaustive testing and, where possible, simulations. In some highly mission- and safety-critical system exhaustive use of formal methods and formal verification is enforced by regulations.

A general definition of (technical) quality in the ISO/IEC 8402 standard is given as "the totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs". The definition adopted for this text emphasizes both formalized requirements and informally expressed expectations:

**Software quality** is:

1. The degree to which a system, component, or process meets specified requirements.
2. The degree to which a system, component, or process meets customer or user needs or expectations. (IEEE, 1990b, p.60)

The general definition of quality confirms that software quality can not be defined only as software without (or little, irrelevant, tolerable, reasonable...) errors. The software quality specification has to be defined accurate and detailed, by using a quality model, formalized by standards. It cannot be overstressed that the quality software requirements and the resulting specification are prerequisites to come in a position to reason about the software quality.

### 2.1.1 ISO/IEC 9126, 14598 and 25000 standards


ISO/IEC 9126 defines a quality model which is applicable to every kind of software. It defines six product quality characteristics and in an annex provides a suggestion of quality subcharacteristics, (ESSI-SCOPE project, 1997), Figure 2-1.
ISO/IEC 9126 is concerned primarily with the definition of quality characteristics to be used in the evaluation of software products. Compared with the dependability attributes from Figure 1-1 (page 5), it is notable that the main missing part in an intersection of the focus of this thesis and the ISO/IEC 9126 is explicit handling of safety.

2.1.2 IEC 61508 – Functional safety of E/E/PE safety-related systems

The main standard dealing with safety for electrical/electronic/programmable electronic (E/E/PE) devices is IEC 61508 (previously IEC 1508), first time published in 1998, being further on in a “maintenance phase” with an update expected in March 2006 (ISO/IEC, 2002a; IEE, 2005). The third part of the standard (IEC 61508-3 “Software requirements”) gives directly guidance for increasing Safety Integrity Level (SIL) of software for safety-critical systems. The standard sets out the requirements for ensuring that systems are designed, implemented, operated and maintained to provide the required safety integrity level. The standard specifies a process that can be followed by all links in the software supply chain so that information about the system can be communicated using common terminology and system parameters. SILs define criticality of system components with respect to the risks involved in the system application, with SIL4 being assigned to the highest risks.
2.1.3 CMM – Capability Maturity Model

Another, not (yet) standardized but perhaps the most widely referred software quality model is the Capability Maturity Model (CMM), proposed as a measure and guidance for quality of software development by the Software Engineering Institute (SEI) at Carnegie-Mellon University. The focus of the CMM is on management and organization of software production projects. The rationale for stressing improvement of the project management aspect of building software is justified by the statistics that the principal cause of failure in so many software projects is negligence of the problems complexity and importance of the first phases in big software projects: requirements analysis, formulation of specifications and proper responsibilities distribution over development teams.

CMM defines five levels of the software development process maturity, with level 1 as the lowest (without any recognisable structure in the software development) and 5 as the highest. The levels are respectively named Chaotic (or Initial), Repeatable, Defined, Managed and Optimising. “As of January (2005), nearly 2000 (US) government and commercial organizations had voluntarily reported CMM levels. Over half acknowledged being at either level 1 or 2, 30% were at level 3, and only 17% had reached level 4 or 5” (Charette, 2005).

For advancing from one to another capability maturity level, an organization has to comply to a number of quality assurance indicators. Since not all of the requirements are found equally important for each software community, a more flexible CMM – “Integration” model appeared later (SEI, 2005).

2.1.4 Other

There are many other indications and measures in use to assess or specify trustworthiness of software. Some of them are based on more-or-less rule of thumb indications (as “99% confidence level” in (Knight et al., 1985)), while there are also more theoretical approaches, some of them standardised. Notable are the following two related (or comparable) to the CMM model.

ISO 9000 standard series

ISO 9000 series of standards are a widely accepted norm which specifies requirements for Quality Management System (ISO, 1987), and in that sense it is usually compared with the CMM. The initial version appeared in 1987, followed by revisions in 1994 and 2000, with an intention to reflect more closely practical needs. ISO 9001 is intended for use in any organization which designs, develops, manufactures, installs and/or services any product or provides any form of service, and therefore applicable in the software industry. However, it is seen by practitioners as too bureaucratic, and therefore less favoured in comparison to CMM.
Cleanroom Software Engineering

Cleanroom Software Engineering is an implementation of the CMM
c MODEL (Linger et al., 1996). Cleanroom software engineering is a software
development and certification process based on theoretical foundations in
mathematical function theory and applied statistics. A principal objective of
the Cleanroom process is development of software that exhibits zero failures
in use. The Cleanroom name is borrowed from hardware cleanrooms, with
their emphasis on rigorous engineering methods and focus on defect
prevention rather than defect removal. The CMM and the Cleanroom
processes are highly compatible and mutually supportive. The focus of the
CMM is on management and organization; the focus of Cleanroom is on
technology and its implementation in engineering processes.

Cleanroom provides means of translating the informal requirement
specification into a formal specification. However, Cleanroom leaves
undefined which formal techniques to use. A fruitful combination of the
Cleanroom and the CSP algebra is elaborated in (Broadfoot and Hopcroft,
2004).

2.2 Software dependability

Among the five dependability attributes (see Figure 1-1 on page 5), this thesis
focuses primarily on safety and reliability. Safety is critical for societal
acceptability of embedded systems, reliability for an economical viability.
Practically, dependability of a system is the ability to avoid service failures
that are more frequent and more severe than is acceptable.

Most references in this section come from the recently published
founding document in the area of dependability and security (Avižienis et al.,
2004). The other older definitions are used only if more precise or concise.

**Dependability** of a system is that property of the system which allows
reliance to be justifiably placed in the service it delivers. (Laprie, 1995)

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

**Reliability** of a system is taken to be a measure of the success with
which a system conforms to some authoritative specification of its
behaviour. (Randell, B. et al., 1978, p.125)

The measure of success is preferably expressed in some probabilistic metrics,
usually MTBF (Mean Time Between Failures) or MTBR (Mean Time Between
Repair). Services of a reliable system are available with a high probability over
time. The specification defines only the external states of the system, the
operations that can be applied to the system, the results of these operations,
and the transitions between external states caused by these operations, the internal states being inaccessible from outside the system (Randell et al., 1978). Reliability of software components is much harder to quantify than reliability of hardware components (Burns and Wellings, 2001, p.127). However, quantification of the software reliability is still more tractable than quantification of the software safety. Therefore, stratification of software components with respect to the safety levels is usually carried out by using actually the reliability measures. For instance, SIL4 safety level means that the probability of failure on a component’s service demand falls in between $10^{-4}$ and $10^{-5}$, which is interpreted as “one safety-critical error acceptable in more than ten thousand years”.

Availability is defined in similar terms as the percentage of time for which the system will conform to its specification. Literally, it is a system’s readiness for service:

**Availability** of a system is the probability that the system will be functioning correctly at any given time. (Storey, 1996)

The remaining two dependability attributes are integrity and maintainability.

**Integrity** is absence of improper system alterations. (Avižienis et al., 2004)

**Maintainability** is ability to undergo modifications and repairs. (Avižienis et al., 2004)

### 2.2.1 Errors, failures, faults and fault tolerance

An *external state* which is not specified in the behaviour of the system is regarded as a failure of the system (Burns and Wellings, 2001).

**Failure** is the behaviour of a system that deviates from that which is specified. (Randell, B. et al., 1978, p.125)

Failures, defined in terms of external behaviour (specification of external states), result from problems internal to the system which eventually manifest themselves in the system’s external behaviour. During a transition from one external state to another, the system may pass through a number of internal states. The term “error” is used to designate that part of the state which is “incorrect” (Randell et al., 1978). An *internal state* which is not specified is called an error and the component which produced the illegal state transition is said to be faulty (Burns and Wellings, 2001).

**Error** is an indication of occurrence of an unspecified internal state of a system. (Randell et al., 1978)
**Fault** in a system is a defective value in the state of a component or in the design of a system. (Anderson and Lee, 1981, p.58)

A causal sequence of the defined terms can hence be diagrammatically expressed as the sequence

```
Fault → Error → Failure.
```

Upon an incorrect behaviour of an internal component of a system (fault), the system may display incorrect external behaviour (failure). Errors (one or more) are internal manifestations of the fault. Note that a component fault from the perspective of a system is considered a failure of the component as a subsystem.

An erroneous state is an internal state such that there exist circumstances (within the specification of the use of the system) in which further processing, by the normal (nominal) algorithms of the system, will lead to a failure (Randell et al., 1978). On basis of the defined meanings of failure, error and fault, fault tolerance, as a system property, can be defined as follows.

**Fault tolerance** is a system’s property to function reliably despite the effects of faults during normal processing. (after (Campbell and Randell, 1986))

A system can be designed to be fault-tolerant by incorporating additional components and abnormal (exceptional) algorithms which attempt to ensure that occurrences of erroneous states do not result in later system failures (Randell et al., 1978). Therefore fault tolerance is sometimes defined as an approach that enables a system to continue functioning even in the presence of faults (Anderson and Lee, 1981); fault tolerance emerges as a consequence of use of any techniques that, based on detection of errors, prevents system failures caused by faults of system components.

Requirements and therefore specification of a system may insist on complete masking of any faults, i.e. requiring the same external behaviour of the system regardless the faults in certain subsystems occur or not; these are rather stringent and sometimes unrealistic specifications—**full fault tolerance**—(Burns and Wellings, 2001).

**Full fault tolerance** is the ability of a system to continue delivering its services invariably whether errors occur or not.

More refined specifications (if permissible) would comprise adjusted requirements for system services under presence of various faults. It should be always defined what behaviour is acceptable before the system restores the common operation mode after an error occurrence or what are
permissible levels of service before maintenance can take place – often referred to as graceful degradation of the system services or fail soft (Burns and Wellings, 2001). The least stringent requirements are put on fail safe systems – the system maintains its integrity while accepting a temporary halt in its operation (Burns and Wellings, 2001).

Graceful degradation is the level of fault tolerance where the system despite the presence of a fault continues delivering the most critical services, while discarding any other less important functionality.

Fail-safe systems, upon an error manifestation, aim at urgent aborting the operation after performing minimal functionality of placing the system into a safe state (if one is defined).

Another approach, closer to the demand of absolutely reliable (failureless) systems is fault avoidance (or prevention). It is based on preventing faults entering the system; clearly, it is criticized as impractical and insufficient (Randell et al., 1978). Even after testing and systematic fault removal (Anderson and Lee, 1981) that ideally may lead to virtually faultless software, hardware components will eventually fail, making fault tolerance techniques indispensable for designing reliable systems.

In order to structure fault tolerance techniques, one has to be aware what kind of errors (faults) a system may end up with. In literature, faults are classified in many ways (for instance in (Randell, B. et al., 1978, p.127; Burns and Wellings, 2001, p.103; Avižienis et al., 2004)), and regardless of which classification is adhered to, the definitions in each of them are orthogonal and inevitably overlapping. From the perspective of constructing a reliable software component, the faults may be considered external (in the environment of a program execution, like memory or sensor failure, or failure of another software component) or internal in the component itself – as a deadlock or an unwantedly infinite loop which polls a signal that never comes (Cooling, 2003, p.55). Internal errors are almost always design errors, in (Avižienis et al., 2004) referred to as development errors. Most external failures from the point of view of a system are regarded as environmental faults (Avižienis et al., 2004), and the corresponding errors are therefore called environmental.

Development errors are indicators of all incurred faults during a system development, called development faults. (Avižienis et al., 2004, p.15)

Environmental errors are erroneous software system/component states caused by faults external to the system/component at hand during its use. They encompass physical faults (fault classes that affect hardware) and interaction faults (that include all external non-hardware faults). (Avižienis et al., 2004, p.15 and 17)
Some development faults affect the system unconditionally, in each run (like the dull infinite loop or a dead sensor), and in that sense they are permanent, and may be detected statically (by system inspections, i.e. by software code reviews, syntax checking in compilation time or formal verification). However, more troublesome are the errors that are detectable only in the application run-time, like loops that may become infinite under certain circumstances (stemming from faulty algorithmic logic) or dereferencing a pointer that became null. While a compiler cannot check this kind of dynamic (in some cases called transient) errors, formal checkers using construction of exhaustive state models for some classes of faults can (of course on account of long computation time). A perhaps more intuitive (in terms of the common use of English) division to permanent and transient error are in (Avižienis et al., 2004) mapped to solid and intermittent errors for sake of comprehensiveness, according to the following scheme.

- **Solid errors** are those errors whose activation is reproducible. (Avižienis et al., 2004, p. 21)

- **Intermittent errors** are those elusive errors whose activation conditions depend on complex combinations of internal state and external requests (Avižienis et al., 2004, p.21/22) or transient errors that appear at a particular time, remains in the system for some period and then disappear (Burns and Wellings, 2001, p.103).

Software designers may be aware of some errors originating from certain faults and pinpoint exactly a suspected region in the design or code, while some errors may be impossible or impractical to identify or detect. From the perspective of using techniques for tolerating faults, a very important division of errors is into anticipated and unanticipated (Anderson and Lee, 1981, p.5). It may be argued that any error of a system may be anticipated ("eventually, an error will occur"), but anticipated errors refer to those that arise with a substantial probability and can be clearly identified (cause and location), described and reacted upon properly. Maxion and Olszewski (2000) present a thorough discussion on astonishing incapabilities of human cognition in a comprehensive anticipation of exceptional situations in technical systems. It is in close relation to fault forecasting as a dependability approach.
(un)Anticipated errors are errors of whose source, place, activation scenario, behaviour, consequences or frequency the designer is (un)aware at design time.

The classification of errors in the three aspects (source, perseverance and predictability) adhered to in this thesis is schematically expressed in Figure 2-2. This classification is not exhaustive (compared with (Avižienis et al., 2004)), though in the scope of this thesis relevant for illustrating complementarity of instruments for increasing dependability in process-oriented architectures.

![Figure 2-2 A few classifications of software errors](image)

Note that the defined concepts in the original literature appear as a classification of faults, rather than errors. In this thesis however the focus is on using the information of a fault occurrence, which is error. For simplicity, the naming of faults is here used for naming the corresponding errors.

The difficulty of classifying software internal errors can be illustrated with an example of an (unwanted) infinite loop. Such a loop is almost always a development error (unless it is caused by an irregular external stimulus, when it is environmental). Easier to detect (thus rare in practice) are explicit infinite loops (being thus solid faults), while often they are intermittent (arising in certain conditions) and in that sense dynamic. As being predominantly development faults, they are unanticipated. However, in the case a repetitive response to an external stimulus is implemented as a conditional loop they might be considered anticipated, as long as the stimulus is justifiably suspected to possibly become unintentionally periodic.
2.2.2 Error recovery versus error masking

Any fault tolerance technique is based on introducing redundant information (processing) and/or redundant components into the system aimed to be fault-tolerant. For software fault tolerance, this redundancy in the code is responsible to implement and coordinate corrective or alternative components to the system’s normal mode of operation.

Corrective redundancy is activated upon an error detection (in that sense being dynamic), while alternative redundant components can be used either statically or dynamically.

**Static (masking) redundancy** is redundancy used to mask or hide the effects of faults in a component. (Randell *et al.*, 1978)

With **dynamic redundancy**, the redundant components only come into operation when an error has been detected. (Burns and Wellings, 2001)

Static redundancy fault tolerance techniques are considered heavyweight and expensive, because the redundant components involved remain in use, and in the same fixed relationship, whether or not any errors are detected (Randell *et al.*, 1978). As already stated, they mask the error and do not attempt recovering it – static redundant components are alternative to the primary operational component, not corrective. Consequently, static redundancy is used in highly available fault-tolerant systems (inevitably in those that conform to full fault tolerance).

Oppositely, any dynamic redundancy technique consists of four phases (Anderson and Lee, 1981):

1. Error detection,
2. Damage confinement and assessment,
3. Error recovery,
4. Fault treatment and continued service.

Since dynamic redundancy is activated upon an error detection, it is aimed to accomplish the **error recovery**. It might be done either by corrective software components or with alternative components hoped to provide acceptable results. The alternative components, likewise those deployed in static redundancy, may accomplish error recovery by hiding the effects of errors, but due to overhead of activating the alternatives, the temporal behaviour is changed, thus by definition the errors are not completely masked.

**Backward error recovery** corrects the system state by restoring the system to a state which occurred prior to the manifestation of the fault. (Jalote and Campbell, 1984)
Backward error recovery involves first of all backing up one or more of the processes of a system to a previous state which is hoped to be error-free, before attempting to continue further operation of the system or subsystem. This technique is thus in sharp contrast to forward error recovery, which is based on attempting to make further use of the state which has just been found to be in error (Randell et al., 1978).

**Forward error recovery** aims to identify the error and, based on that knowledge, correct the system state containing the error. (Best and Cristian, 1981)

Both techniques aim to place the system in a state from which processing can proceed and failure can be averted (Randell et al., 1978).

Many systems maintain an audit trail so that system activity can be (manually) certified if complying with legal and/or accounting regulations as well as for online recovery. Restoration of a recovery point using an audit trail is achieved by processing the events recorded in the trail in reverse order, successively changing the state of the system so as to undo the effects of each event (Anderson and Lee, 1981, p.187).

**Log (audit trail)** is the historical record of activity. (Anderson and Lee, 1981, p.187)

### 2.2.3 Exceptions, exception handling and atomic actions

An exception is an indication that something out of the ordinary has occurred which must be brought to the attention of the program which raised it (Anderson and Lee, 1981, p.77/78). Measures that are provided within the program for dealing with an exception are termed the **handler** for that exception, and the signalling of an exception will result in the handler for that exception being invoked (Anderson and Lee, 1981, p.79).

“A concurrent and potentially parallel nature of the execution of the processes may introduce ambiguity in the choice of fault tolerance measures to handle a particular exception condition” (Campbell and Randell, 1986). When before beginning of an error recovery several processes have thrown exceptions, these are called **simultaneous** (or **concurrent**) exceptions. These exceptions may indicate the same fault observed by different processes, but may represent also different faults (independent or possibly caused causally). Different combinations of simultaneously raised exceptions may require quite different exceptional operations in the system. A classical example is a safety system in a building with a gas installation. One set of measures is prescribed for an occurrence of gas leakage – an exceptional situation in use...
of the gas facilities. On the other hand, a standard procedure may be prescribed in case of fire, which is also considered as an exception in the normal building use. However, it may be wrong to treat these two exceptional situations occurring simultaneously as if they happen in isolation. Obviously a third approach has to be applied if the fire danger coincides the gas leakage, which situation may happen in case of an earthquake.

Determining a proper exceptional operation in the presence of simultaneous exceptions is called resolution of concurrently raised exceptions (Campbell and Randell, 1986) or concerted exception handling (Issarny, 2001). It is based on exception hierarchy, a structure that permits determination of the most convenient exceptional operation under given exceptional conditions. In (Campbell and Randell, 1986) a structure of the exception tree is advocated as suitable for resolving such complex situations: “if several exceptions are concurrently raised, the exception used to activate the fault tolerance measures is the exception that is the root of the smallest subtree containing all of the exceptions” (Campbell and Randell, 1986, p.819).

Availability of an exception handling mechanism (EHM) is in (Anderson and Lee, 1981) taken as a prerequisite for establishing the framework of atomic actions, already described as a design pattern for preventing propagation of erroneous information outside a set of processes engaged in a collaborative operation. Atomic actions are the principal classical error confinement architectural pattern for dynamic error recovery in concurrent systems.

Atomic action is the activity of a group of components if there are no interactions between that group and the rest of the system for the duration of the activity (Anderson and Lee, 1981).

### 2.3 Real-time terminology

Any classification of terminology with respect to temporal behaviour of the systems usually starts with making difference between soft and hard real-time systems. But first it should be clarified why the notion of “time” in a system’s behaviour is so often augmented with the adjective “real”. An excellent observation (Burns and Wellings, 2001, p.411) clarifies that: “The term ‘real’ is used to draw a distinction with the computer’s time. It is real because it is external.”

Real-time systems are informally described as those with temporal constraints; those that react to the inputs within time intervals dictated by the environment. In theory and practice, the most frequently met term that captures “temporal constraint” is deadline:

A deadline is a given time limit within a program has to satisfy a request for service or else system failure is likely to ensue. (Anderson and Lee, 1981, p.273).
Systems where a deadline can be missed occasionally, i.e. the service can occasionally be delivered late are termed soft real-time systems. (In this discussion on temporal behavioural aspects it is assumed that a service is always logically correct). For hard real-time systems a late delivery (beyond the specified deadline), although logically correct, is considered as failure.

A majority of mass-produced embedded systems are not to be attributed as real-time since their temporal behaviour is a consequence of its design, not a requirement (or at the best a non-functional requirement) – for example retrieval of the list of missed calls upon switching on a mobile phone. While the predominant part of telecommunication embedded applications are usually referred to as soft real-time systems, it is difficult, if not impossible, to find any example of closed-loop digital control systems that are not hard real-time. Hence, since this thesis concerns software applications of (embedded) control systems, the term “real-time” refers, if other is not explicitly stated, to hard real-time systems. Quantitative reasoning on the difference of soft, hard and non-real-time system is expressed in terms of the utility functions, Figure 2-3. The utility of service of a non real-time system does not have defined deadline, and the decrease of the service utility is subjective. For soft real-time systems, the utility of the service delivered after the deadline decreases, but still exists. Service of a hard real-time system delivered after deadline has no value.

![Figure 2-3 Utility functions of systems which are:](image)

- a) non-real-time
- b) soft-real-time
- c) hard-real-time
Real-time systems are systems designed with temporal behaviour as one of the functional requirements.

Soft real-time systems tolerate occasional late service delivery, i.e. occasional missing of prescribed deadlines.

Hard real-time systems are those real-time systems where an excess of the deadline for delivering services expected from a system is considered a failure.

2.4 Concurrency and concurrency-specific phenomena

Concurrency studies simultaneousness and causality of interacting activities within a reactive system.

In the scope of the CSP modelling paradigm, 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 (Hilderink, 2005a).

Some phenomena, unknown in sequential executions, are manifested in concurrent systems due to simultaneousness of activities that interact with each other. The most notorious of all is deadlock:

Deadlock is state of a concurrent system when no component can make any progress, generally because each is waiting for communication with others. (Roscoe, 1997, p.3)

A deadlocked process is in CSP modelled as a STOP process – a process that does not engage in any event; as opposite, a process that accepts any event and subsequently terminates is a CSP SKIP process. Deadlock certainly causes a subsystem (and possibly the whole system) to stop functioning.

For the phenomenon when a component does not get access to a system resource, the term starvation is used. However, this term is used rather for the situation where the system is not deadlocked, but due to some (usually scheduling) omissions in the design, some software components do not get a system resource for a long time or never.

Livelock (divergence) exists when a set of processes gets into an infinite sequence of communications entirely with each other that cannot be interrupted; once in such a state, the process can refuse all further communication from the outside world. (Welch, 1999)
The livelock condition is also known as divergence or internal chatter, and is modelled in CSP by the div process.

**Race hazards** is a phenomenon that a state of a resource (for instance a variable value) after a completed (part of a) program depends on the order of access of program-defined concurrent entities to that resource.

### 2.5 CSP foundation and derivatives

**CSP – Communicating Sequential Processes** is a notation for describing concurrent systems (i.e., ones where there is more than one process existing at a time) whose component processes interact with each other by communication. (Roscoe, 1997, p.1)

CSP is a calculus for studying processes which interact with each other and their environment by means of communication (Roscoe, 1997, p.8). CSP offers to the process-oriented design of concurrent systems a fundamental architectural vocabulary of building blocks: processes for capturing functional software components, synchronous (waiting rendezvous) channels for interprocess communication and operators for composing order of execution among the processes. In total, the order of processes’ advancements is determined by:

- synchronization on communication events on channels,
- compositional operators that arrange processes in higher execution constructs.

Fundamental CSP operators are sequential, alternative and a few parallel operators. Two processes related with a sequential operator execute one after termination of another (sequentially composed processes). Among processes composed by an alternative operator just one is being chosen for execution, depending on a certain condition or/and event (alternatively composed processes). If not specified by one of the former compositions, processes by default advance simultaneously, and depending on whether and how the synchronization on communication events is specified, a few variants exist (synchronous, alphabetized or interleaving parallel composed processes).

#### 2.5.1 CSP diagrams

The CSP diagrams is a graphical notation for CSP, proposed by Hilderink (2002; 2003; 2005a). It gives a graphical means in describing communication and composition aspects of CSP-based architectures, by introducing a graphical vocabulary for the basic CSP entities. It extends the set of three basic CSP operators in terms of compositional constructs, with prioritized variants for parallel and alternative—like in occam and (Lawrence, 2001)—
and an exception construct, (Table 2-1); the watchdog construct is a contribution of the dependability extension of this thesis.

Table 2-1 Constructs in gCSP

<table>
<thead>
<tr>
<th>Parallel</th>
<th>Priparallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternative</td>
<td>Prialternative</td>
</tr>
<tr>
<td>Sequential</td>
<td>Exception</td>
</tr>
</tbody>
</table>

Chapter 3 gives a complete overview of the version of the language implemented in the gCSP tool and the mapping of these constructs to CSPm expressions.

2.5.2 CSP libraries

CSP libraries provide an occam programming look&feel for mainstream programming languages (Java and C/C++). Communicating Threads (CT) libraries from the University of Twente are therefore called CTJ, CTC and CTC++ (Hilderink et al., 2000; Orlic and Broenink, 2003; Hilderink, 2005a). Similar libraries, fostered by the Kent University, are called JCSP (Welch, 2002), CCSP (Moores, 1999) and C++CSP (Brown and Welch, 2003).

The CT libraries represent a process-oriented implementation framework providing high-level concurrency design patterns based on CSP. These design patterns—compositional constructs and compatible communication primitives—encapsulate multithreading from a user’s point of view. In fact, the libraries add to an underlying object-oriented infrastructure process-oriented concepts useful for reasoning about concurrency and real-time behaviour. The libraries, with an integrated real-time kernel independent of operating system (OS) scheduling mechanisms, facilitate easier portability of a concurrent design. Being independent of an OS, the CT-based designs are portable with minimal adjustments to non OS-supported (“bare metal”) embeddable processors. Having process execution management as a part of the application, the real-time behaviour is more likely preserved when ported to another platform. The libraries implement also prioritized variants of the parallel and alternative constructs, introduced already in occam and later formally described in CSPP (Lawrence, 2001). For CTC and CTC++ (used for real-time embedded applications) the memory footprint and the scheduling overheads are low (Hilderink et al., 2003).

The CSP compositional hierarchy affects directly the architecture of CT programs. The CT programs are static compositions of constructs and processes (connected by channels). Part of the program code that implements such a composition is called a network builder. The compositional hierarchy
directly affects the exception propagation in the proposed exception handling mechanism, see Chapter 5.

The following three paragraphs summarize the most important facts about CT channels, processes and constructs. For a thorough treatment the reader is referred to (Hilderink, 2005a). This thesis contributions base on the first Twente-CSP-libraries, Communicating Threads (CT) developed in the period 1998-2005. Next generation are being developed in two branches: Communicating Processes – CP (Hilderink, 2005b) and Simultaneous Interactive Processes – SIP (Orlic, 2002-2006). Nevertheless, the following concepts will hold in any of them.

**CT channels** are passive objects that implement write and read methods, used by producer and consumer (or server and client, etc.) processes respectively for rendezvous data communication. This means that parallel composed processes are synchronized (and scheduled) on channel communication. A channel can carry various types of data in principle in both directions. Besides the two basic methods for putting in and taking out the data from channels, they implement also a few methods to support the concept of channel poisoning (Welch, 1989) (in this thesis referred to also as channel suspension). Namely, when a channel is suspended (with an exception), it refuses to respond to writings or readings. In the implemented exception handling mechanism these refusals raise exceptions that are used for commencing simultaneous error handling in parallel processes. The methods for using this functionality are suspend, isSuspended and rehabilitate. Channels can be legitimately suspended by exception handling processes (see Chapter 6) and the monitoring component (Chapter 6). Channels can be shared, but basic channels are one-to-one.

**CT processes** are (active) objects being constructed with references to the channels (or variables) connected to them. This means that a process does know neither identity nor location of other processes (this is a strong feature when distributing a CT design). The activity of a process is performed in its private workspace and is encapsulated in the run method. Processes may interact with their environments only through their communication interfaces. Interface to a channel is usually read- or write-only, making the channel communication unidirectional. A process itself can be composed of other process and constructs (than a process becomes a complex or parent process). Child processes may communicate with each other through internal channels or with the outside world through the interface of their parent.

**CT constructs**, as implementation of the CSP(P) operators—sequential, (pri)alternative and (pri)parallel—are also implemented and treated as processes; although, they do not have channel interfaces: their children, processes composed in the constructs, are immediately connected to the channels (unless a construct is a child of a complex process). This holds also for the CT exception and watchdog constructs added later. CT constructs, together with the synchronization on channels are responsible for scheduling their child processes. The sequential construct starts a child process after
termination of a predecessor child process according to the order of processes in the declaration list of the construct. The sequential construct terminates after the last process in the list is terminated. The alternative construct runs one of its child processes that can accomplish a channel communication. Child processes are connected to the incoming channels through guards – as soon as a guard has a party on the other side willing to rendezvous, the associated process is run and after its termination the alternative construct terminates as well. If more than one guard is ready to communicate, the construct schedules one of the processes in random (theoretically). The parallel construct allows interleaving of their children and terminates when all children are terminated. Prioritized versions of the alternative and parallel constructs schedule their child processes according to the order these are listed in the construct declaration.

The **CT kernel** provides the low level means of supporting scheduling ruled by the compositional constructs and channels. It possesses a small number of other most important OS primitives, as an idle process which occupies the processor when all other processes are inactive – usually blocked on internal or external communication.

CT implementation holds the following promises:
- Low memory footprint,
- OS independence,
- Clear separation of hardware dependant parts of a design,
- Transparent portability (hardware independence),
- Transparent distribution,
- Real-time facilities (prioritizing, timing in channels),
- Loose coupling of processes as basic building components,
- Composibility of bigger systems with proven subcomponents.

### 2.6 Process orientation

Although used already in many places, an abstract definition of a process read as:

**Process** is a set of interrelated resources and activities that transform inputs into outputs. *(ISO 8402, 1994)*

This definition is applicable both in contexts of business and techn(olog)ical processes as in the context of treating a process as a basic building blocks for the process-oriented software architecture. The latter is the case for this thesis:
Process orientation is a software development paradigm with processes as architectural and functional units interacting with each other only by message passing through channels, and with a defined model of concurrency among processes' executions.

2.7 Formal analysis

Being applied on abstract models of systems, formal methods are clearly a fault prevention (avoidance) approach (Anderson and Lee, 1981, p.4).

2.7.1 Methods and tools for formal analysis

Benefits from formal analysis of a model of a (concurrent) system are well known (Katoen, 2004). While testing procedures—no matter how carefully designed—intentionally target only anticipated sources of malfunctioning, an exhaustive checking is possible only by formal verification.

An abundance of different forms of formal methods and languages is subject of the research worldwide. However, for industrial it is necessary that the different approaches converge and yield a consistent suite of the most successful, tool supported formal methods applicable to different problem domains. Such trends are emerging (Aceto and Gordon, 2005).

For the theory of concurrent programming, process algebras have been applied with the greatest success – especially CSP, CCS/\pi-calculus, and combinations of these two, notably the internationally standardized LOTOS (ISO, 1989). Outside process algebras, the most popular formal methods are timed automata (Alur and Dill, 1994) (together with the widely used tool UPPAAL (Larsen et al., 1997)), Petri nets (Peterson, 1981), Z specification language (Spivey, 1989) standardized in 2002 (ISO/IEC, 2002b), also standardised VDM (ISO/IEC, 1996), the B-method (Abrial, 1996), and the specification language PROMELA (a PROcess MEta LAnguage) together with the perhaps the most mature and mostly used formal checker Spin (Holzmann, 2004). In the CSP family, there are derivatives that extend the CSP modelling capacities in various directions of interest: TimedCSP (Schneider, 2000) for embracing timing analysis in CSP models and CSPP (Lawrence, 2001) which allows reasoning of different levels of priorities in the CSP architectures.

As a mathematical, algebraic notation, the CSP language can be manipulated only by humans. While still usable for manual modelling systems with a few tens of processes, formal analysis of the behavioural properties of the models even of such modest size is beyond the capabilities of the human mind. Computer tools support for dealing with models larger than just toy-examples is necessary. CSP surpasses the other process algebras by its tools support.

The two most known tools for analyzing CSP models, FDR and ProBE, are products of Formal Systems Ltd. FDR (Failures-Divergence Refinement) is a commercial model-checking tool for state machines (Formal
Systems, 2003). It is built on operational semantics (state machine representation) of CSP models. ProBE is a much smaller tool that animates CSP event trace models.

### 2.7.2 CSPm

The CSP models must be represented in a machine-readable form in order to be processed by the verification tools. The machine-readable version of CSP, CSPm (Scattergood, 1997), is a subset of CSP that can be textually (ASCII) coded in scripts loadable by FDR and ProBE. A small example of how a producer-consumer model in CSPm would look like is given in this listing:

```plaintext
datatype theType = someValue | anotherValue

channel ch1 : theType
channel ch2 : theType

Producer12 = ch1!someValue -> ch2!anotherValue -> Producer12
Consumer12 = ch1?aVariable -> ch2?bVariable -> Consumer12
Consumer21 = ch2?bVariable -> ch1?aVariable -> Consumer21

SystemDF = Producer12 || {ch1, ch2} || Consumer12
SystemDC = Producer12 || {ch1, ch2} || Consumer21
```

The interpretation of the first three lines is quite obvious: a datatype `theType` consist of values (constants) `someValue` and `anotherValue`, and the channels `ch1` and `ch2` can carry that type.

Process communication patterns in CSP and CSPm are basically specified by operators for writing (“!”) a value to a channel and reading (“?”) a value from a channel to a variable (which needs not be declared). Thus, `Producer12` attempts writing the constant `someValue` to `ch1`, `anotherValue` to `ch2` and then repeats, while `Consumer12` attempts to read a value first from `ch1` and store it in the variable `aVariable`, then attempts to read a value from `ch2` and store it in `bVariable`; then repeats itself. `Consumer21` does the same, but the reading is in a reverse order. Writing and reading to/from channels represent communication events. A sequence of communication events, always ending with quoting a process name, is composed by the prefixing operator “->” that chains an event to a process (composing together a new process which engages in that event and for the rest behaves as the process – in the CSP terminology: the event guards the process). One way of expressing that a process repeats infinitely in recursion is putting the name of the process at the end of its own sequence. Prefixing cannot be used for composing processes in sequence (for that, the sequential operator “;” is used).

The two last lines in the script specify two synchronized parallel compositions (that are legitimate processes in their own). Each of them in fact
represents a parallel model whose analysis is desired. Such a right-hand side expression as a top level composition in a model is called **network builder** (in CSPm this time). The expressions of SystemDF and SystemDC are both network builders describing two different models (that share the same Producer12 specification).

Parallel process SystemDF is build by the first network builder out of two (parallel composed) processes, Producer12 and Consumer12. "|| { | ch1, ch2 | } ||" is one of the CSPm parallel operators, so-called **shared parallel**. It means that Producer12 and Consumer12, advancing in parallel, must synchronize on each event on the channels ch1 and ch2. The same applies for SystemDC that is built out of Producer12 and Consumer21.

As specified in this example script, SystemDF is deadlock-free ("DF") while in SystemDC a deadlock condition ("DC") occurs. The reason is simple: communication patterns of Producer12 and Consumer12 are compatible, while those of Producer12 and Consumer21 are not.

### 2.7.3 FDR and ProBE tools

The pathological conditions as deadlocks and livelocks can be captured by both ProBE and FDR, but quite differently. ProBE is an interpreter of CSPm scripts, and thus requires progressing through the event sequences (**traces**) until processes are terminated (what would be never a case with recursive processes) or no events are agreed upon by running, not yet terminated, processes – which means a deadlock. Obviously, using ProBE does not make sense for an exhaustive checking on deadlock conditions. It is rather used to understand a trace that leads to a deadlock occurrence already indicated by FDR. Two screens opened by ProBE for exploring the two network builders are presented in Figures 2-4 and 2-5. From Figure 2-4 it can be observed that ProBE represents a pair of communication attempts in processes (like ch1!someValue from Producer12 and ch1?aVariable from Consumer12) as one communication event observable (or "offered") by the environment, indicating only the written value (ch1.someValue).
In this particular case, SystemDC deadlocks immediately, which is indicated by an empty event pane in Figure 2-5. An experienced ProBE user can spot in the trace pane in Figure 2-4 a repeating pattern (on the top and at the bottom) – this may lead to a conclusion that SystemDF infinitely repeats without deadlock.

The power of detecting or “proving” deadlock freedom by ProBE is limited to the problem size like in the presented script. For an exhaustive analysis of more interesting systems the FDR is used. For the previous script, the FDR window is shown in Figure 2-6.

At the bottom all processes defined in the script are listed. From that list one may chose processes interesting for checking against deadlock. In the upper list it is shown that FDR finds all sequential process (first three in the list) deadlock free – in front of them a tick sign is displayed. It is also shown how a deadlock condition is indicated (a cross sign in front of SystemDC). Besides the indication, FDR may report the event trace that leads to a deadlocked situation, as well as an interactive debug engine for understanding the erroneous situation. Using these facilities for debugging a CSPm specification requires practice and good understanding of the CSPm modelling techniques. As the tabs at the top of the window suggest, FDR is also capable of analyzing CSPm models against the properties other than deadlock, such as livelock, determinism and refinement relations between different processes (Formal Systems, 2003).

![Figure 2-6 FDR window](image-url)
2.8 Embedded control systems

2.8.1 Embedded systems

An **embedded system** refers to dedicated computing equipment vital for the functionality of a surrounding service-delivering system approached by the user to use the non-computing service of the smart system as an ensemble.

The definition suggests that the user accesses a smart system that integrates a computer, being unaware of the presence of computer-supported information processing inside. It also suggests a high system specificity.

An **embedded control system** is a computer system used for the closed-loop control of a physical system in a predetermined environment. (Lent, 1989)

2.8.2 Control systems

Control systems are typical examples of smart systems: the average user is not interested in the inputs and outputs of the embedded control computer (controller), but only in operating and getting results of the control systems as a whole. In the scope of this thesis, *digital* controllers will be discussed exclusively.

A **control algorithm** is a coded control law with the aim of influencing the behaviour of a controlled object (plant or appliance) to achieve certain goals. (Van Amerongen, 2005)

A **digital controller** is an ensemble of control algorithm(s) and processing hardware that produces outputs based on sampled inputs within a predetermined time interval, referred to as sampling period.

A **closed-loop digital control system** is a system consisting of a controlled object, a digital controller with assigned sampling period, sensors, actuators, and interfaces. (after Van Amerongen, 2005)

Among control software developers the word “controller” is often used to refer to the coded algorithm only (i.e. control computer code), while in the control-dedicated computer industry the term is often used for the computing/sampling hardware only.
An **appliance** is a non-autonomous system that requires to be steered and whose internal state can be completely or partially observed.

A vast volume of the control theory and practice literature refers to the controlled object as **process**, stemming from chemical technology processes, historically the first problems area that successfully deployed digital control. In the scope of this text the word **process** is reserved for the software entity as a basic building component in process orientation.

**Control code** is part of the control computer code that implements control laws.

**Control software** consists of control code and all other necessary software components that make it fully operational code of a control computer.

Control code represents seldom more than 20% of the overall control software code.

### 2.8.3 20-sim tool

20-sim is a modelling and simulation tool for dynamical systems developed by ControlLab Products B.V. (CLP, 2002), a spin-off company of the Control Engineering group of the University of Twente. It is a standard MS Windows application consisting of several integrated modules that support designing mechatronic systems in many aspects.

Users’ inputs to the modelling module can be performed by means of one or more 20-sim Editors: bond-graphs (Breedveld, 2004), block diagrams, iconic diagrams, equations or by importing external models. Additional editors specialized for designing specific linear systems directly address the control aspect of mechatronic design: Filter Editor, Linear System Editor, Controller Design Editor. The tool complies well with the demand of offering a time-efficient and elaborate feedback to the user on the modelling/design decisions. By means of a flexible simulation module and visualization modules as animated graphs and 3D animations of a modelled object, the 20-sim Simulator allows user-appealing verification of built models.

In version 3.1 (December 2000), 20-sim introduced a step forward in coverage of the design cycle of a mechatronic product towards software implementation of control laws: automatic code generation for submodels. Stemming from the internal simulation model, generation of C-code in a few variants (stand-alone ANSI-C code, ANSI-C function, Simulink S-function) is available. Equally important to these PC (x86) targeted variants is the opportunity to extend the C-code generation module to generate code for virtually any processing platform with a C compiler and standard C libraries. In short, the principles of automatic code generation are as follows. A model of a subsystem for which code is to be generated is firstly transformed into a
form convenient for numerical computation. This is done anyway for simulation purposes. The code generation implemented in 20-sim use so-called template files and keywords. Template files give a wanted structure of source code, according to the code practices of a targeted platform. Keywords are placeholders for implementation instance of functional entities defined within a dynamical model.

2.9 Interdomain tooling coverage

In the advent of domination of interdisciplinary technologies, it is imperative to search for common notions and languages among disparate design cultures, mindsets, concepts and paradigms. However, inventing a common paradigm to be a mediating infrastructure for multidisciplinary design itself is not enough to make a superior design paradigm a success, unless supported by appropriate tools.

In the field of embedded software, initiatives to marry modelling of the problem domain and production of software that is to drive the artefacts from the problem domain are pursued since long. Moreover, matching the structure of the problem domain to the software architecture is modus vivendi of the dominating object orientation.

On the other hand, reliability of embedded software is an absolutely crucial property for its existence and socio-financial acceptance. Therefore, an ultimate necessity for verifiability of the embedded software designs is obvious. A cornerstone of sound verification is deployment of formal methods. Therefore the most successful design paradigms for developing reliable software inherently encompass formal specifications in one or another form.

However, the design paradigms and toolkits that cover all the three worlds of domain-specific design, software architecting and formal verification are remarkably scant. In the control engineering area the only one known to the author is described in (Cavalcanti et al., 2005), an approach based on use of the Simulink package (Mathworks Inc., 2005) and the Circus refinement calculus which represents a combination of CSP and Z. This approach is strong in connecting notations of block diagrams common in control engineering domain and formal analyses means, but is lacking explicit CASE functionalities for architecting control software, emphasized in the next item.

2.9.1 gCSP as an interdomain bridge for embedded software

CSP-based process orientation has been recognized and worked out in the software development of the 1990’s in the work of Wijbrans (1993). However, a just established toolchain reported in that work was soon outdated by disappearance of transputers and obsolescence of the software design methodologies and CASE tools adhered to in the 1990’s. Moreover, the great promise of exploiting the CSP potential for formal verification was not included in the proposed design paradigm.
Developing a mechatronic system starts with development of a dynamical model of a controlled mechanical system and a controlling structure in some of the domain CAD tools (20-sim in the context of this thesis and the project). Implementation of the controllers in embedded software and its deployment (realization) in hardware should follow in an evolutionary way. The toolchain resulted from this research, depicted in Figure 2-7, finds a common dataflow ground and a direct mapping between the control domain language (block diagrams) and process oriented software architecture (represented by CSP diagrams, i.e. gCSP models). Moreover, thanks to the features of our main deliverable, the gCSP tool, a straightforward link from the process architecture and high-end verification tool, the FDR model checker, has been established.

![Figure 2-7 The toolchain among engineering domains and tools covering mechatronic design](image)

Establishment of the toolchain and the capabilities of modelling dependable software with the gCSP tool are subjects of the core Chapters 3, 4, 5 and 6. Overviews of the integral dependability approach are provided in (Jovanovic and Broenink, 2005, 2006). An overall workflow performed by the toolchain is depicted in Figure 2-8.

In the context of this thesis, 20-sim plays the role of the control CAD tool; as model checker, FDR is used. The quality of the output from this workflow—generated control software—with respect to the error prevention is guaranteed by simulations of the dynamical model made in 20-sim, which then (suppose reliably) generates the control code, and by formal verification of the concurrent CT software that provides other functionalities of the control software, produced by the gCSP tool. Figure 2-8 suggests (in stage “Final graphical model”) that no change is allowed in a graphical model used for generating implementation version after it passes formal verification.
The incoming design artefacts (the control code) from control domain tools make this workflow control-software-specific. By neglecting the lefthand side of the diagram, the scheme is generally applicable for dataflow-driven formally verified software development.

Figure 2-8 The CSP/CT control software development workflow
3 Modelling CSP/CT architectures with the gCSP tool

A first quality aspect of a software design paradigm is the ability of modelling the designs. Graphical (visual) notations are proven to be best acceptable not only by the software developers, but all project stakeholders – (Cooling, 2003; Miller and Mukerji, 2003; Muller, 2004). On the other hand, quality assurance of a software system benefits significantly from a formal background of the modelling notation. The highlights are summarized in section 3.1.

The modelling paradigm for CSP-based concurrent software in form of a graphical language is first proposed in (Hilderink, 2002), and further elaborated in (Hilderink, 2005a) as CSP diagrams. Section 3.2 presents a subset of that language extended towards practical control applications; further, the formal underpinning behind the graphical vocabulary in machine-readable notation CSPm is specified. Section 3.3 clarifies use of the language for modelling an elementary closed loop – the basic control structure. Section 3.4 gives a short overview of the gCSP tool as a CASE tool. In section 3.5 top levels of the gCSP models for the two case studies are explained. Section 3.6 concludes the presentation of the tooling support of the developed design methodology.

3.1 Graphical modelling languages and tools

A trend of migrating from textual programming towards more abstract graphical means is apparent: as assembly programming nowadays is considered low-level software design (which was considered quite differently thirty years ago), over a decade or two today’s “high-level languages” may be viewed the same way (Miller and Mukerji, 2003).

Since long researchers and practitioners have been trying to provide software design methodologies and paradigms as companions to programming languages. Designing software barely at a language level is proven to lead to unreliable and unmaintainable code for any, but utterly simple applications. Many of the proposed and used software design methods are graphical: Ward-Mellor (Ward and Mellor, 1986), state charts (Harel,
Despite intensive applications of CSP in several engineering areas (communication protocols, concurrent programming, integrated circuits design), in all these fields some ad hoc graphical notations have been used, not even standardized within one application field. Hoare himself (1985, p.54 and 148) used a rudimentary visualization of processes and communication events (connection diagrams), however emphasizing impracticality of such primitive diagrams. The connection diagrams resemble to some extent hardware schemes (visualizing each event by a line, analogue to physical signal lines), which can be hampering when modelling software-intensive systems with elaborate interactions among processes. Similarly, in the literature that deals with applications of CSP various forms of visualizing CSP networks can be met, without an attempt for a comprehensive notation that could address a unified cross-disciplinary modelling approach.

Certainly, without a consensus on a well thought and standardized graphical notation, building graphical design tools for modelling CSP-based architectures was not feasible. On one hand, this has been recognized by many CSP researchers as a substantial difficulty in disseminating and teaching CSP as a design philosophy. On the other hand, it is a fact that nowadays any novel design paradigm has a small chance for a broader acceptance if not tool-supported. The CSP diagrams notation (Hilderink, 2002, 2003, 2005a) is acknowledged to be a first attempt of yielding a generally applicable graphical design language for CSP-based architectures (McDermott, 2005). The gCSP tool extends a subset of the CSP diagrams proposal towards practical applicability in designing CSP-based software implementable by the CT libraries. The graphical notation is formally underpinned with the standard CSPm formal description language. Automatic generation of CTC++ code and formal analysis facilitated by generation of CSPm code is subject of the next chapter.

3.1.1 Requirements for the gCSP tool development

The most elaborate standard for describing software graphically – UML – is described as “a graphical language for visualizing, specifying, constructing and documenting the artefacts of a software-intensive system”. The same goes for the general idea of gCSP. In short, the purpose of the language and the tool can be described as supporting building concurrent software based on...
the CSP algebra principles. In order to meet this goal, the development of the tool started with the following set of requirements. The tool should:

1. allow modelling of concurrent systems using the CSP diagrams notation,
2. preserve notions of the CSP theory and its peculiarities, but bring it closer to implementation needs,
3. support means for managing complex CSP models – allowing hierarchical organisation by containment ("part-of") relations among parent (complex) and child (leaf or also complex) processes,
4. allow the expression of communication and compositional patterns of process networks, the latter not only in terms of an extended set of CSP constructs, but also in terms of binary compositional relationships (as defined in the CSP diagrams proposal) and an occam-like compositional hierarchy,
5. transform software graphical models to different types of human- and machine-readable code,
6. allow semantic and integrity checks of the specified models,
7. allow visualization also in the domains of (formal) analysis and other relevant CSP model processing,
8. generate CSP networks suitable for incorporating operational code derived from other tools – for instance one-shot processes from 20-sim.

3.2 The gCSP graphical language

The extended subset of the CSP diagrams language implemented in the gCSP tool and used in this thesis is referred to as the gCSP graphical language; similarly, the models of CSP/CT software that contain (extended) CSP diagrams and the compositional tree-shaped view are called gCSP models.

The remainder of this section presents the gCSP graphical vocabulary, where applicable the C-tree representation and formal specification of the graphical elements. All graphical elements within the gCSP graphical language are collected in Table 3-1.
Table 3-1 gCSP graphical elements and their CSP abstractions

<table>
<thead>
<tr>
<th>gCSP symbols</th>
<th>CSP abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Process</td>
</tr>
<tr>
<td>Exception handling process</td>
<td>Process</td>
</tr>
<tr>
<td>Emergency watchdog process</td>
<td>Process</td>
</tr>
<tr>
<td>Rendezvous channel</td>
<td>Channel</td>
</tr>
<tr>
<td>Unsynchronized channel</td>
<td>Variable</td>
</tr>
<tr>
<td>Input and output ports</td>
<td>implicit</td>
</tr>
<tr>
<td>Channels’ joint</td>
<td>implicit</td>
</tr>
<tr>
<td>Primitive reader</td>
<td>“?:” operator</td>
</tr>
<tr>
<td>Primitive writer</td>
<td>“!:” operator</td>
</tr>
<tr>
<td>Primitive repeater</td>
<td>Closest are “µ” recursion and “***” infinite repetition operators</td>
</tr>
<tr>
<td>Custom code block</td>
<td>none (does not influence CSPm script)</td>
</tr>
</tbody>
</table>
20-sim code block

Reader linkdriver

```
LD_READER1
```

Id_READER1:Double

"?:" operator

Writer linkdriver

```
LD_WRITER1
```

Id_WRITER1:Double

"!:" operator

Logging/monitoring (L/M) linkdriver

```
LD_L/M1
```

Id_L/M1:Double

none (does not influence CSPm script)

Watchdog hit linkdriver

Set_Watchdog1

Channel

Watchdog set linkdriver

Hit_Watchdog1

none (does not influence CSPm script)

Watchdog remove linkdriver

Remove_Watchdog1

none (does not influence CSPm script)

Sequential relationship

";" and ";->" operators

Alternative relationship

external choice "[\]" operator

Prior alternative relationship

same as previous (CSPm does not support priorities)

Parallel relationship

interleaving "| |" or shared parallel "[ | | ]" operator
Priparallel relationship

same as previous
(CSPm does not support priorities)

Exception relationship

closest is "/" interrupt operator

Watchdog relationship

closest is "/" interrupt operator

CSP diagrams define **compositional view** and **communication view** into a graphical model. The basic building blocks for software functionality are processes. Data exchange is abstracted in **communication relationships** – channels.

![Figure 3-1 Processes and channels in the communication view](image)

The communication aspect of a CSP/CT architecture is represented in the communication view (Figure 3-1). The concurrency among processes is specified by **compositional relationships** and represented in the compositional view of a CSP diagram (Figure 3-2). The compositional aspect in gCSP is presented also by a hierarchical tree of compositional relationships—the **compositional tree (C-tree)**—Figure 3-4. The graph view into a CSP model with combined compositional and communication views is called the **hybrid CSP view** (Figure 3-3).

![Figure 3-2 Parallel composed processes in the compositional view](image)

![Figure 3-3 Communicating parallel composed processes in hybrid CSP view](image)

![Figure 3-4 The C-tree corresp. to the comp. view from Figure 3-2](image)
3.2.1 Processes

All processes in gCSP are divided into two main groups: primitive (leaf) processes and ordinary (complex) processes. Primitive processes cannot be parents, i.e. be composed of other processes. They are also subject to other constraints (only input or output channels or no channel interfaces at all). Primitive processes are variables, primitive communication and repetition processes, primitive entities for marking hardware access (hardware linkdrivers), linkdrivers for accessing the watchdoggng (watchdog linkdrivers) and logging/monitoring CT components (L/M linkdrivers), and code blocks for low-level processing specification. Exception handling processes and watchdog emergency processes, although ordinary processes, are given distinct graphical shapes. Remember that CSP constructs are also processes.

Complex or ordinary processes

In the gCSP language, primitive processes, being leaves of a design's compositional hierarchy, are contained by complex processes, basic (ordinary) structural entities. An ordinary process can contain also other ordinary processes. Ordinary (complex) processes are depicted as rectangles, with the default name inside, as in Figure 3-5. By default, gCSP names ordinary processes with a leading uppercase letter.

A complex process and its children are related by a containment relationship (also referred to as part-of or nesting). The ability of a complex process to contain other processes is important for partitioning complex graphical models. Process1 in Figure 3-6 encapsulates a subnetwork, consisting of two processes, shown in Figure 3-7. Figure 3-7a shows how the communication interface of a process is modelled in gCSP. Following the 20-sim convention, input ports are depicted as filled small squares, while outputs are empty squares ("to resemble the letter 'o', as in 'output').

In the C-tree (Figures 3-6b and 3-7b), processes with composed internals get a small square icon in the tree branch, with a "-" for expanded or "+" for collapsed representation of the internals. In this way the tree indicates if a complex process is already refined or just provisionally outlined. Constructs are always compositionally determined, hence the containment icon ("+" or "-" ) is always present in front of a construct icon. Note that each complex process in fact encapsulates a construct, called the top-construct for the given complex process. The kind of the top construct determines the kind of process (sequential, (pri)parallel, (pri)alternative, or with applied dependability techniques exception- or watchdog-guarded).
In CSPm processes are represented by an identifier expressed by a compositional formula. For the composition in Figure 3-7 (more precisely, 3-7b) the composition is expressed by

\[
\text{Process1} = \text{Process11} ||| \text{Process12} \tag{1}
\]

“\(|||\)” is the CSP interleaving parallel operator. Justification of using unprioritized operators and details on specifying parallel composed processes is subject of section 3.2.3.

**Exception handling processes**

Figure 3-8 shows the shape of an exception handling process. The exception handling mechanism (and issues on its formalisation) in the CSP/CT framework is subject of Chapter 5.
a customized shape better reveals an exception handling layer over an initial software composition.

**Emergency watchdog process**

A slightly distinct shape is assigned also to the process that is scheduled upon a watchdog timeout in a program (Figure 3-9).

![Figure 3-9 Emergency watchdog process](image)

For an empty emergency watchdog process, supposedly named `WatchdogTimeoutHandling`, the tool generates a CSPm specification of the following form:

\[
\text{WatchdogTimeoutHandling} = \begin{cases} 
\text{watchdog1timeout} &\rightarrow \text{STOP} \\
\text{watchdog2timeout} &\rightarrow \text{STOP} 
\end{cases}
\]

One emergency process reacts on all watchdogs that can be specified in the system. This example expression assumes two watchdogs: `Watchdog1` and `Watchdog2`. If the `WatchdogTimeoutHandling` process was refined, its specifications of reaction to timeouts of different watchdogs would refine the `STOP` processes. On specifying watchdogs see paragraph on watchdog linkdrivers on page 74. The used CSPm choice `"[|]"` operator is explained on page 81.

**Primitive communication processes**

Default names of the primitive processes are in uppercase letters, so as for primitive reader and writer in Figures 3-10 and 3-11; however, variables are depicted with all lowercase letters.

The primitive reader and writer model communication events. Both primitive readers and writers are crucially important in the CSP-based process-oriented design since they capture the points of data communication (hence possible synchronization) among processes. The execution order of processes in a CSP architecture is ruled by compositional relationships and communication among processes. Precise designation of points of interprocess synchronization is therefore substantial for the analysis of a program execution. Moreover, the primitive communication processes couple channels (used for communication among complex processes – external data) with variables (holding internal – intraprocess – data).
Figure 3-10 Primitive reader and a variable

Variables are presented only as named labels. On creation, each variable is associated with a type and is initialized. The type of the variable can be optionally shown next to the variable name. Variables are closely related to unsynchronised channels, called var-channels, explained on page 76.

A primitive reader process intermediates between the channel interface of a process and internal variables (Figure 3-10). It is presented as an encircled “?” symbol, stemming from the CSP notation for reading from a channel to a variable. The channel interface of a reader is restricted to an input channel and one output variable-channel only. The CSPm expression corresponding to Figure 3-10 is

\[ \text{READER1} = \text{dataChannel1}?\text{var} \] (3)

A primitive writer does just the opposite of the reader: it outputs internal data of a complex process (captured by variables) to an output channel (Figure 3-11). Following the CSPm channel writing operator “!” it is depicted as a circle with an exclamation mark. The channel interface of a writer is restricted to one input variable-channel and one output channel only. The CSPm notation accepts only constants to be explicitly written to a channel (by using the “!” operator). gCSP therefore employs the following notation for the algebraic representation of a primitive writer

\[ \text{WRITER1} = \text{dataChannel1}!\text{var_val} \] (4)

For each variable whose values are output to a channel, the value itself is represented by appending “_val” to the variable name.

**Primitive repeater process**

Figure 3-12 The repetitive construction with the repeater process

Figure 3-13 The repetitive construction in the C-tree
For specifying repetitive execution of a process (and also of constructs), the primitive repeater is used, depicted by an encircled star (\( \bullet \)) symbol – Figure 3-12. The repetition of a process is always modelled by the sequential composition of a repeater process and the process being repeated; the order in the sequence determines while-do (preconditions) or repeat-until (postcondition) nature of loops. In the C-tree sequential compositions with repetition processes are optimized in a form of a repetition \( \bullet \)-construct (Figure 3-13). It corresponds to the occam WHILE construct.

Following the representation of repetition as a sequential composition and the idea of the \( \bullet \) operator, a straightforward CSPm representation is

\[
\text{RECEPTION1} = \text{PROCESS1} \ ; \ \text{RECEPTION1} \quad (5)
\]

Although satisfactory for modelling the repeater process, this is not a quite common expression of infinite repetitive compositions at the low-level process specification in CSPm – following this representation, the repetitive reader from Figure 3-14 would be described as

\[
\text{RECEPTION1} = \text{READER1} \ ; \ \text{RECEPTION1}
\]
\[
\text{READER1} = \text{DataChannel1}?\text{var}
\]

while a CSPm practitioner would rather code that recursively, using the CSP operator for prefixing (\( \rightarrow \)), as

\[
\text{RECEPTION1} = \text{dataChannel1}?\text{var} \rightarrow \text{RECEPTION1} \quad (6)
\]

This customization of the sequential construct is possible by manipulation of the C-tree (about sequential relationship see section 3.2.3).

![Figure 3-14 A typical use of the repetition: repetitive reading from a channel to an internal variable](image)

**Hardware reader and writer linkdrivers**

A concept similar to that of the primitive reader and writer is adopted for communication between the software and hardware devices. Communication with hardware through channels is implemented in CT by linkdrivers (Hilderink, 2005a, p.151).
An output linkdriver is modelled as a reader that reads a value from a channel and converts it to a form acceptable for the actual hardware device. Since the reader linkdriver reads from a channel, although it models an output port from the software, it is depicted as an inverted reader “?” icon (Figure 3-15). The CSPm description of the reader linkdriver justifies use of the “?” symbol:

$$LD_{READER1} = ld\_READER1?ldVar -> LD_{READER1}$$  \(7\)

The channel for communication with the hardware is by convention named the same as the linkdriver (with leading lowercase letters though). The fictive variable that a reader linkdriver reads to is named ldVar for all linkdrivers. Note that a linkdriver, capturing a hardware component that behaves as a process simultaneous with the software, needs not be modelled explicitly as a repetitive process: all linkdrivers are assumed as infinitely repetitive.

The same holds for a writer linkdriver (Figure 3-16) that reads from hardware and writes to a channel.

$$LD\_WRITER1 = ld\_WRITER1!ld\_val -> LD\_WRITER1$$ \(8\)

In principle, a linkdriver and a channel connected to it are considered one; that is reflected in the names of channels. However, the convention holds that in gCSP names of processes begin with an uppercase, while names of channels begin with a lowercase letter. Following this convention is necessary for generating proper CSPm code, because names of linkdrivers have to be different of the names of events/channels used to model interaction with hardware. The type of the data that a linkdriver handles is suggested by the type of the linkdriver channel. The fictive value that a writer linkdriver inputs into software is named ld_val for all writer linkdrivers.

**Watchdog linkdrivers**

There are three linkdrivers manipulating the watchdog component. In order to let a watchdog be checking liveness of a part of a CSP/CT design, the watchdog component has to be allocated (set). During its function, the watchdogged part of a design has to signal (hit) the assigned watchdog, and deallocate it (remove it) after use. These operations are modelled by the three watchdog linkdrivers, Table 3-2.
3 Modelling CSP/CT architectures with the gCSP tool

Table 3-2 Watchdog linkdrivers

<table>
<thead>
<tr>
<th>Watchdog</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set_Watchdog</td>
<td>Set watchdog</td>
</tr>
<tr>
<td>Hit_Watchdog</td>
<td>Hit watchdog</td>
</tr>
<tr>
<td>Remove_Watchdog</td>
<td>Remove watchdog</td>
</tr>
</tbody>
</table>

Each watchdog specified in a model is reflected in the CSPm specification by creating a channel whose name reflects the name of the watchdog, so as for a watchdog named WatchdogPID1, the CSPm script would include:

channel watchdogPID1timeout

Logging/monitoring linkdriver

Logging and monitoring facilities are elaborated in section 6.4. Access to the L/M component is modelled by the L/M linkdriver in Figure 3-17.

The logging/monitoring layer does interfere with neither composition nor communication among the CSP/CT processes. Therefore, presence of the L/M linkdrivers does not influence the CSPm specifications.

Code blocks

Between data being read from input channels and written to output channels, usually within a process data processing takes place. The actual data processing is the lowest level specification of a process and is captured by code blocks. Code blocks do not have rendezvous (synchronous) channel interfaces. They operate only upon local variables. Access of a code block to variables is specified by variable-channel symbols, like in Figures 3-18 and 3-19.

There are two kinds of code blocks available in gCSP, rendered as rounded rectangles. Figure 3-18 depicts a custom code block, while in Figure 3-19 the
shape of a 20-sim code block is given. The use and difference of the code blocks is relevant for automatic code generation, which is elaborated in the next chapter.

Since the internal processing does not influence the communication patterns of processes—it is forbidden accessing the channels from the code block bodies—code blocks do not have a CSPm description. The presence of code blocks in a model does not influence the generation of CSPm code.

### 3.2.2 Communication relationships

Both sorts of the communication relationships – channels – have already been displayed in several figures so far: rendezvous channels represented as filled-arrowlines and variable-channels (shorter var-channels) represented with open-arrowlines, in (Hilderink, 2005a, p.66) originally introduced as state communication relationships. Different kinds of processes in different execution compositions can communicate with both or just one kind of communication relationships. These combinations in fact determine process communication interfaces. Further, the communication interfaces are determined by the types of data carried by the channels: boolean, byte, character, double (default), float, integers, object or reference.

#### Rendezvous channels

Rendezvous channels implement message-passing synchronous communication. These can be used between parallel composed processes. The CSP (synchronous) channels allow bidirectional communication; however, since CT processes access channels through unidirectional ports, CT channels are unidirectional, which is indicated by an arrow (Figure 3-20).

![Figure 3-20 Rendezvous channel of the Object type](image)

The gCSP tool conventionally starts channel names with lowercase letters. The label of a channel indicates the type of data the channel can carry.

Using a rendezvous channel between sequentially or alternatively composed processes instantly causes a deadlock-like situation: processes engage in a rendezvous which cannot succeed due to the compositional constraint – the two processes can never be alive at the same moment. Upon detection of such a situation the tool reports the design error and suggests turning the rendezvous channel into an unsynchronised variable – variable-channel.

#### Var-channels (unsynchronized variables)

The CT libraries facilitate communication between sequentially or alternatively composed processes by the ChannelVar objects – variables with
the channel interfaces: read and write methods. The same functionality is sometimes performed by ordinary variables (whose use should be restricted for intraprocess scope – among primitive processes and code blocks) or by helper processes running in parallel facilitating asynchronous communication among processes (existence of additional processes of course deteriorates performance).

Unsynchronized variables used between parallel composed processes are source of race hazards. Therefore, potentially dangerous var-channels between parallel composed processes are detected by the tool; the user gets warned and suggested to change them to rendezvous channels.

At the level of modelling with gCSP, variables and var-channels (depicted as in Figure 3-21) should be considered synonyms. In the code generation the tool may decide to optimize var-channels to ordinary variables.

A var-channel has defined an initial value. The associated variable is visible (accessible) within the scope of the containing process.

**Sharing/joining channels**

Joining channels is a notational means for graphical modelling of the concept of shared channels. Channels can be connected to more than two processes. Depending on the orientation, a channel may connect one producer with several consumers (one-to-any channel), several producers with one consumer (any-to-one channel) and multiple producers to multiple consumers (any-to-any channels). In case of a rendezvous channel, the synchronism principle applies (Hilderink, 2005a, p.63). For var-channels, if used properly as it has been described, sharing channels means barely using the same communication means (variable). The graphical technique of joining is performed by connecting channels to the small filled circle called joint, the same symbol used in (Hilderink, 2005a, p.64) for implicit server (delta) process.

In Table 3-3 the one-to-any, any-to-one and any-to-any rendezvous configurations as modelled in gCSP are shown. The same solutions apply to the var-channels too.
### 3.2.3 Compositional relationships

As described before, the order of execution among processes is determined by the compositional constructs and synchronization on rendezvous channels. Constructs compose *groups* of processes in certain execution patterns, while basic (non-shared) channels relate *pairs* of processes. CSP diagrams extend this "binarity" also to the compositional aspect by introducing binary compositional relationships. Compositional relationships allow building compositional CSP hierarchies bottom-up, by superimposing execution patterns on the dataflow model expressed by communication relationships. This complements a top-down approach when the designer starts building a network with complex processes and constructs.

Compositional relationships are graphically represented by named lines adorned with symbols of the CSP/CT constructs (Table 2-1, page 51). The user may choose for a more pronounced differentiation between compositional and communication relationships by specifying thickness of the lines. The names of the compositional relationships can be displayed on the diagrams, but they rarely are. On one hand, the reason is reducing the information complexity of a diagram. On the other hand, for naming the compositions as *groups* of processes, names of the constructs are far more relevant.
Sequential compositions

A sequential relationship is asymmetric – the pair of related processes is ordered. Both the compositional view (Figure 3-22) and the C-tree (Figure 3-23) uniquely reflect this order. When more processes are chained with sequential relationships, the C-tree orders them transitively. The resulting construction (process) is composed by the CSP sequential operator “;”.

\[ \text{Seq1} = \text{Process1} ; \text{Process2} \]  

The sequential relationship declares processes Process1 and Process2 as sequentially composed.

The CSP sequential operator “;” does not give enough graphical emphasis on the asymmetry of the sequential relationship. The arrow symbol for the sequential construct and relationship suits this purpose better. Although inarguably much more intuitive, it is derived from another CSP operator – prefixing – that is applicable to sequencing an event with a process (when the event may be considered as a trigger to the process – as in Dijkstra’s concept of guarded operations (Dijkstra, 1975)). Its use for the basic producer-consumer specification in the CSPm script (on page 55) and for expressing recursive repetition (page 73) reveals its machine readable notation: “–>”.

Allowing this operator in the CSPm description of gCSP models provides scripts that are simpler (shorter) and much more common to CSPm practitioners.

gCSP does not introduce a new relationship for prefixing. The C-tree allows, if applicable, turning a sequential construct into a prefixing construct. This optimization is particularly useful at the level of primitive processes (being leaf processes of a complex process), as in Figure 3-24.

A default composition of the sequentially composed primitive reader and writer yields the C-tree representation in Figure 3-25. A literal (blunt) CSPm translation of this composition would be
Here the primitive processes are explicitly named and composed by the sequential operator. The gCSP tool by default does one simplification when generating gCSP code for primitive processes as events ("in line" code generation of the primitive processes' descriptions):

```
Seq1 = inChannel?var -> SKIP ; outChannel!var_val -> SKIP
```

However, an experienced CSPm user would find this simplified notation uncommon too. It should be rather

```
Seq1 = inChannel?var -> outChannel!var_val -> SKIP
```

Getting an output exactly like this is achieved by an optimisation taken by turning the sequential construct into the prefixing variant, resulting in the C-tree in Figure 3-26.

This feature is particularly useful for specifying endless repetitions of a process' internal composition. It is already mentioned that repetitions are realized as customized sequential compositions. Therefore, having the previous composition endlessly repeated as specified in Figures 3-27 and 3-28:

```
Process1 = inChannel?var -> outChannel!var_val -> Process1
```

gives the following CSPm specification
Note that the CSPm engine optimizes away both Seq1 and REPETITION1, yielding a common CSPm code in the listing above. As it will be thoroughly discussed at the end of this section, the small circle ("bubble") with index 1 in Figure 3-27 is a means of expressing boundaries of constructs in CSP diagrams. Here it designates that the REPETITION1 process acts on the sequential composition of READER1 and WRITER1 as a group.

**Alternative and prialternative compositions**

The alternative composition of processes allows expressing the choice of different executions (encapsulated by alternatively composed processes) based on various criteria; therefore, the alternative composition may be specified in a variety of ways. Alternative processes can be chosen on basis of channel readiness for an event, which can be coupled with a logical condition; but the condition may rule activation of an alternative in its own. Also, after waiting for some specified time, an alternative may be chosen on basis of the expired timeout.

The principal mechanism of activating alternatives on basis of events is called guarding: an event guards a process. In the CT libraries special guard objects mediate between a channel and a process. In this thesis the concept of guarding is expanded also to the exception handling and watchdogging. Therefore, guards used in communication-event-driven choice of processes is further on referred to as comm-guards. Another type of guards used for the alternative composition, which help expressing that the choice of processes does not depend on the channel readiness, but only on the logical condition or the timeout are called skip-guards.

The graphical language (as well as the implementation library) includes prioritized version of the alternative (and parallel) construct, originally not formalized in CSP. The notions of priority, important for dealing with temporal behaviour of software, are introduced in occam, and later formalized in CSPP (Lawrence, 2001). Since CSPm provides a machine-readable form of algebraic CSP, the prioritized operators are not defined. This does not prevent formal analysis of the CSP diagrams (CT programs) with prioritized constructs: since CSP formal checking is not concerned with the temporal analysis, gCSP treats all prioritized operators as non-prioritized, thus the priority information is not contained in the CSPm scripts.

Similarly to the sequential construct, the prioritized alternative ("prialternative") construct is asymmetric. The arrow points to the higher priority process. In Figure 3-29 a prioritized alternative composition with (only) comm-guards is presented.
In the CSPm an alternative composition is represented by using the CSP external choice operator “[]”:

\[
\text{PriAlt1} = \text{inChannel1} \rightarrow \text{Process1} [\text{inChannel2} \rightarrow \text{Process2}]
\]

(11)

In this expression the processes are preceded by the events names (which coincide with the channel names), without the read “?” operators and names of the variables the channel message is stored to – the option of generating the communication events in this minimalistic way is documented in section 4.2.1 on page 120 in the next chapter.

Illustration of using comm-guards and logical conditions is combined with presentation of the alternative constructions (unprioritized) in Figures 3-31 and 3-32. The alternative construct \text{Alt1} behaves as \text{Process1} if a communication event is performed on \text{dataChannel1} and the binary condition \text{bin} is true. If data are communicated over \text{dataChannel2} and the condition \text{bin} is false, \text{Alt1} behaves as \text{Process2}. The \text{Alt1} process waits to make the decision until one of the two communication events occur. The waiting time can be limited by using timeout guards. About all alternative construct facilities the interested reader can find more details in (Hilderink, 2005a, p.84). The order of the processes in the C-tree is arbitrary, determined by the tool.

CSPm code for condition-comm-guarded processes looks like

\[
\text{Alt1} = \text{bin==true & inChannel1} \rightarrow \text{Process1} [\text{inChannel2} \rightarrow \text{Process2}]
\]

(12)

A special case when all comm-guards are degraded to skip-guards means that the choice of alternatively composed processes depends only on the logical conditions (Figure 3-33).
For this case gCSP provides an if-then-else construction of CSPm:

\[
\text{Alt1} = \text{if } \text{bin}==\text{true} \text{ then Process1 else if } \text{bin}==\text{false} \text{ then Process2 else STOP} \tag{13}
\]

This form of the alternative construction is useful to make choice which branch of an algorithm should continue executing. (Note that this is not possible to accomplish with a code block if the alternative execution should communicate on a latter point, since channel communication is forbidden to be used inside the code blocks).

**Parallel and priparallel compositions**

CSP defines few variants of parallel executions, for whether or not the parallel composed processes synchronize and depending on which event alphabet. The language of CSP diagrams abstracts away from these variants, providing one graphical symbol for all parallel operators of CSP. The gCSP tool however determines on the grounds of the communication layer which CSPm operator to use for formalization of the parallel – as specified in this paragraph. The resulting CSPm expression is the same for a priparallel as for a parallel construction.

The priparallel compositional relationship (Figure 3-34) is asymmetric: the arrow points to the higher priority process. Thus, the order of processes in the C-tree (Figure 3-35) is determined.

The parallel relationship is symmetric. The order of the processes under a parallel construct in the C-tree is irrelevant and is determined by the tool (Figure 3-36).
Assume that in Figure 3-37 a hybrid CSP view corresponding to the C-tree in Figure 3-36 is presented, thus with no communication (channel) between processes Process1 and Process2. That means that the two processes are running in parallel and do not synchronize. They interleave.

![Figure 3-37 Interleaving parallel composed processes](image)

This is described by an *interleaving* construct in CSPm:

\[
\text{Par}_1 = \text{Process}_1 \parallel \parallel \text{Process}_2
\]  

(14)

That is not the case for parallel composed processes in Figure 3-38, communicating over channel *dataChannel1*.

![Figure 3-38 Communicating parallel composed processes](image)

The tool identifies channels (if any) between (pri)parallel composed processes and generates the *shared parallel* CSPm construct accordingly:

\[
\text{Par}_1 = \text{Process}_1 || \{ \text{||} \text{dataChannel1} \} || \text{Process}_2
\]  

(15)

**Exception composition**

A process may handle exceptions occurred in another process. This relation is captured by an exception relationship, presented in Figure 3-39. The arrow above the triangle points to the exception handling process (this is important if the modeller chooses not to use an oval representation for an exception handler, but an ordinary rectangle).

![Figure 3-39 Exception relationship](image)

The C-tree in Figure 3-40 presents the exception construct. It associates an exception handling process with an ordinary process or a construct. The
ordinary process (or construct) combined with an exception handling process under the exception construct is referred to as exception-guarded process (construct).

The exception construct is not formalized in CSP, hence the gCSP tool in the current version does not generate a CSPm specification of the exception construction. This issue is revisited in Chapter 5.

**Watchdog composition**

Another form of guarding constructs or processes (and whole networks possibly encapsulated within) on specific kinds of malfunctions—as elaborated in section 6.2 Watchdog patterns—is captured by the watchdog relationship and construct, Figures 3-41 and 3-42.

![Watchdog relationship and construct](image)

Process2 is called the watchdog-emergency process for Process1. Process1 has to include a watchdog linkdriver in order to instantiate a watchdog and let Process2 be activated if the watchdog timeout expires.

The CSPm specification of the watchdog relationship makes use the CSP interrupt operator "/\":

\[
WD1 = \text{Process1} /\langle \text{Process2} \rangle
\]  

(16)

**3.2.4 Compositional hierarchies**

A principal issue in building a design methodology and a supporting tool for software systems is managing complexity of problems of practical ("industrial") size. In a growing CSP network partitioning in hierarchies is indispensable. There are two possibilities to establish the partitioning. The first is a compositional structure provided by constructs. The inherent compositional hierarchy is giving structure to the programs. The other possibility is intrinsically offered by using complex processes – they partition a model (and correspondingly a CT program) into self-contained functional as well as architectural units.

Both means have their advantages and disadvantages. On one hand, a compositional hierarchy in an arbitrarily large flat model can be perfectly established by grouping processes connected with the same sort of compositional relationships into corresponding CSP constructs. Depending on the size of such one flat model, viewing a compositional structure of a logical unit gives a good overview of the functionality. The problem arises when such a flat model grows, containing many logical units that should be separated. And if a practical-sized model is shown in just one CSP diagram,
having all processes, channels and compositional relationships on one hierarchical level, it becomes unmanageable, intractable and after all undisplayable on any possible screen.

The other extreme is using complex processes to capture every construct in a model separately. It has been already observed that every process has a top construct, hence it is possible to encapsulate unnecessary redundancy (to partition a model by processes bluntly following the compositional structure, already captured by grouping into constructs), it fragments the model. Each hierarchical level would contain just one compositional group. The overview of logical functionality is therefore obscured, if not lost at all. Too many processes introduce a serious memory and execution time overhead as well.

It is clear that the modelling paradigm and the gCSP tool should allow the modeller to find a proper trade-off between partitioning a model into logical units with an understandable complexity and a good functional overview. That means that the user may build an arbitrarily complex composition of constructs, and then choose which part of the system to encapsulate into complex processes.

Visualisation of constructs (in fact, the compositional groups) in a CSP diagram is not trivial. A construct is scattered over a group of processes connected by homogeneous relationships and does not have a simple graphical representation itself. Boundaries of a construct are not easily visualised. Moreover, using constructs for structuring a CSP network does not solve the partitioning of the model.

A solution for imposing compositional grouping on a process network in the CSP diagrams is proposed in the form of parenthesizing (Hilderink, 2005a). This notation introduces markings on compositional relationships that imply boundaries of a construct scope; however, this notation does not solve the problem of explicit visualization (and naming) of constructs in a compositional CSP diagram. Therefore in gCSP, the CSP diagrams are complemented with respect to the compositional structure by two means. One operates on the graph infrastructure of CSP diagrams (called boxed-grouping notation), while the other is a separate tree-like view into the compositional hierarchy, called the C-tree (compositional tree, resembling Abstract Syntax Tree known from the theory of compilers). As an example, let us suppose that Process1 from Figure 3-6 (with internals as in Figure 3-7) has been exploded (flattened) resulting in the network in Figure 3-43. It is
now ambiguous whether the parallel composition of Process11 and Process12 runs in parallel with Process4 or the parallel composition of Process11 and Process4 is prioritized over Process12. Note also other similar ambiguities in this diagram of just five processes. The parenthesizing notation may resolve this ambiguity in one of the following ways: as in Figure 3-44 or, for instance, like in Figure 3-45.

The explanation of the meaning of the grouping by parenthesizing bubbles is easiest by graphical means, either by spatial marking of the compositional groups at the diagram or by the C-tree. Figure 3-46 shows the boxed-grouping notation for Figure 3-44 by an unambiguous grouping of the processes Process11 and Process12 into a parallel composition further parallel composed with Process2, Process3 and Process4. An even more informative composition into named constructs is given in the C-tree, Figure 3-47. Figure 3-46 also gives the interpretation of the indices of the parenthesizing bubbles: they are determined by the number of crossings of any relationship and all grouping rectangles (“boxes”) necessary for an unambiguous determination of a composition into a construct. Situations that require higher indices are exemplified in the next section.

The interpretation for the parenthesized (“bubbled”) network in Figure 3-45 is given in Figures 3-48 and 3-49. Here the process Process12 is composed in lower parallel priority with respect to the parallel composition of all other processes. Figure 3-48 demonstrates the main disadvantage of the boxed-grouping notation: embracing some constructs by the grouping rectangles may entail significant topological reconfigurations, causing an inefficient use of the display surface. Note that the reconfiguration in Figure 3-48 is not quite proper: Process12 should have been moved more away from the denoted group, in order to place the parallel symbol out of the rectangle. That would make the diagram even less compact.
The boxed-grouping notation also allows indicating names of constructs within a CSP diagram – each box corresponds to a construct in the C-tree. However, the presented variant of the boxed notation has a mediating role between a fully constructs-oriented compositional hierarchy in the graph-view and the parenthesizing notation. If an additional small sacrifice of the displaying surface is allowed to be made, the presented variant of the boxed-grouping notation evolves to perhaps the most intuitive way to organize compositional hierarchies in the graph-view, which is shown in Figure 3-50. In these “non-intersecting” forms of the boxed grouping (graphs a and b corresponding to Figures 3-46 and 3-48), the boxes representing constructs entirely visually embrace their child process. As with the previous boxed notation, boxes directly correspond to the constructs in the C-tree.
It can be concluded that the parenthesizing notation has an advantage of keeping the CSP diagrams compact. The other advantage is flexible experimenting with various grouping configurations, because a compositional structure can be modified just by modifying indices of the bubbles (though it is not that easy to assure absence of compositional conflicts). The boxed-grouping notation proves more intuitive in expressing compositional hierarchies in the graph-view (the CSP diagrams) into a gCSP model. The disadvantage is an inefficient use of displaying surface.

However, using only constructs for structuring a CSP network does not solve the partitioning of the model into manageable chunks. A proper balance has to be found between introducing distinct hierarchical levels (by partitioning a model in subprocesses) and compositional (sub)networks at each hierarchical level.

It is also a conclusion that for reasoning on the compositional hierarchy the C-tree is a superior representation, both with respect to informativeness and efficiency. Although lacking the communication information, it possesses some other qualities that are elaborated in the next section.

### 3.2.5 C-tree and the CSP/CT modelling principles

The C-tree structure was conceived in earlier prototypes of tools for specifying CSP/CT software (Volkerink et al., 2000; Hendriks, 2001). It is implemented in gCSP because of the following reasons:

- Improving overview and navigability through a model,
- Providing the glue logic of a bottom-up and top-down approach in composing a model,
- Intermediating between levels of abstraction of the CSP diagrams and the CT-compliant source code,
- Influencing certain code generation alternatives and optimizations,
- Storing some hardware-specific parts of the source code.
Thus far the C-tree has been used extensively to clarify the compositional as well as containment hierarchies in the graphical CSP/CT designs. It gives a clear overview of composing subnetworks into constructs and encapsulating them in parent processes. The tree structure resembles much the way an occam programmer views the structure of a program. It naturally matches the CSP/CT compositional hierarchy. Besides giving a comprehensive and efficient overview of the design, the C-tree also improves significantly the navigability through a model. Section 3.4 on the gCSP user interface presents the means of navigating through the complex processes (going in and out), which is a sequential way of browsing the system hierarchy levels without a good overview of the current location in the hierarchy. In the tree a direct way of finding the wanted part of the design is both faster and easier.

When writing an occam program, the programmer has to reason in advance what the compositional structure of the program will be; first constructs need to be in place providing a structure, then their children (processes and other constructs) fill the structure. This is clearly a top-down manner of constructing a (CSP) system. Note also that at the moment processes are programmed, the design is fixed – there are no easy ways to modify the structure other than recoding. On the other hand, the idea of introducing compositional information also in the form of binary relationships was inspired by a possibility to create isolated parts of a design and then compose them in some way (bottom-up). However, in order to check the properties of a system and eventually create source code, a model has to be unambiguously and definitely determined (by the modeller or perhaps autonomously by the tool).

Building a CSP/CT model starts with making a dataflow diagram with processes and communication relationships. Processes get refined with complex subprocesses and leaf processes – communication primitives. By default, all processes specified in a gCSP model are parallel composed. During building the communication model, the designer starts realising at places alternative or sequential patterns of execution among some processes or groups of processes. Thereby drawing compositional relationships takes place: to explicitly visualize parallelism in a convenient way, and to impose sequential or by-choice executions where desired. All this before the system as a whole is considered. It is clear than in this stage the model as a whole is fragmented and possesses much of compositional ambiguity; however, the designer has a great freedom to treat the processes and groups of processes as true building blocks. At one moment, the modeller wants to see the big picture.

The current version of the gCSP tool cannot construct the C-tree automatically out of a partially (or completely) specified CSP diagram. The algorithm of checking the consistency of placing and indexing parenthesizing bubbles is estimated as too complicated for this stage of the tool development. However, the C-tree assists the designer to unambiguously arrange the compositional structure of the model. Only compositionally conflict-free groups of processes can be transformed in the C-tree constructs. Section 3.4 clarifies that the part of the gCSP user interface called C-tree does not consist of the tree-structure only: two additional compartments list all compositional relationship that are not associated with a construct (thus,
unresolved compositional relationships) and all processes not grouped in a construct (loose processes). The C-tree indicates if a process internals are composed in constructs or not (the icon "+"/"-" is present in front of a process icon or not). Hence the C-tree indicates the degree of the completeness of a model.

The code generation in general is more reliable and much faster when based on a hierarchical tree information than on a graph structure. The C-tree is in fact an intermediate domain between the levels of abstraction of the CSP diagrams and the source code. The strong resemblance between the C-tree structure and basic structure of a CT program will be shown in Chapter 4. Some code generation issues (specifying prefixing for CSPm) code are dealt with through the C-tree.

A few of the figures in this chapter have shown the C-tree root, the icon named Model. This is a suitable model element for storing some general information about the system that is modelled. For instance, some hardware specific code parts, hard to be modelled in a general graphical way, are stored here (like the ways of initializing hardware device drivers), for details see section 4.3.4 in the next chapter.

### 3.3 A practical example

In Figure 3-51 a basic closed loop control system, modelled in 20-sim, is shown.

![Figure 3-51: Closed control loop](image)

It consists of three functional blocks:

1. **PlantDyn** is a controlled object characterized by its dynamical behaviour. In this case it is a first order system in the s-domain described by:

   \[
   \text{state}(s) = \frac{1}{s + 0.8} \text{steering}(s) \quad (17)
   \]

2. **LoopCon** (loop controller) takes care to keep the measured variable (MV, signal state in this case) in proportion with the required set point (SP, here reference) signal. The simplest proportional controller (P-controller) with proportional gain \( K = 10 \) is used in this example:

   \[
   \text{steering}(s) = K(\text{reference}(s) - \text{state}(s)) \quad (18)
   \]
Coupled with PlantDyn, the controller influences the controlled state variable by manipulating the steering values.

3. SeqCon (Sequence controller) in general is a control system component governing separate control loops to maintain values of local variables (state in this case) according to a prescribed higher system-level sequence of activities. Having a system reduced to one control loop, the role of this component is reduced to a set point generator providing the controller with a desired reference profile.

Figure 3-52 shows variations of the steering and state values for the reference being a unity step signal at $t=1$.

A systematic transformation of a block diagram to a gCSP model starts by mapping the functional blocks to processes and signals to channels. Therefore, this is a straightforward 1-to-1 mapping. Translated in a dataflow (communication) view in gCSP (Figure 3-53), the names of signals from 20-sim model coincide with names of channels among the three processes corresponding to the three functional blocks from Figure 3-51.
In this particular example the plant model directly corresponding to the PlantDyn functional block from Figure 3-51 is encapsulated within the augmented InitPlantDynProcess process. While a model with the direct mapping is used in the next chapter (page 115), the augmentation is present in this chapter for two reasons. The main reason is demonstration of a multi-level hierarchy. Secondly, it was the wish to start with a working model: it will be clear in the next chapter that the internal composition of the InitPlantDynProcess is one way to solve a deadlock problem inherent to 1-to-1 mapping which is the subject of the next chapter.

The default parallel composition of processes is quite natural in this case, since it can be assumed that Figure 3-51 models three independent components that naturally operate simultaneously. Therefore, the hybrid CSP view looks like in Figure 3-54.

![Figure 3-54 Hybrid CSP view of the closed control loop example](image)

Examples of explicit ordering the execution sequentially are given at the level of each process’ internal specification (Figures 3-55, 3-56, 3-57 and 3-58). To precisely specify the order of inputting, processing and outputting the variables in the system components, sequential relationships are used. As explained in the previous section, sequential relationships in combination with primitive “-process indicate (endless) repetitions of the all three processes’ implementations. This is a typical pattern in embedded systems in general and also when using computational algorithms for CSP/CT processes out of 20-sim models. Namely, the 20-sim simulation engine takes care of giving the processing algorithms of the building blocks a proper repetitive execution model. When extracted from a 20-sim model, these algorithms are “one-shot”, which means that they produce outputs on basis of inputs for one sampling period. In order to let the model execute over successive sampling periods, a proper execution framework is provided by using a parallel composition among the processes that are, in turn, internally composed in sequences with the “-repetitions.

The SeqConProcess process (Figure 3-55) implements outputting set point values to the reference channel. The written instances are values of the variable step that is manipulated by the SetPoint code block. The upper sequential relationship indicates the activation order of the two elements (first the code block SetPoint, then the writer WRITER_SP). The lower sequential relationship used in combination with the REPEAT_SeqCon “-process indicates repetition of the upper sequence. The true value of the repetition condition implies that this repetition is endless.

Similarly, the sequence of the readers READER_Reference and READER_Feedback, the code block ControlLaw, and the
WRITER_Steering writer is being endlessly repeated by the REPEAT_LoopCon process (Figure 3-56). READER_Reference stores data read from the channel reference to the variable SP, while READER_Feedback does so with state and MV. The values of the variable P, produced by the code block ControlLaw, are written to the channel steering by WRITER_Steering.

The internal structure of the process InitPlantDynProcess illustrates a deeper nested process hierarchy. The first level contains a sequential composition of a primitive writer and a process capturing the functionality of the 20-sim submodel PlantDyn. This sequential composition specifies that an initial value of the state of the plant is output to the channel state before the first calculation of the dynamical response of the plant is initiated.

Figure 3-55 SeqConProcess internals

Figure 3-56 LoopConProcess internals

Figure 3-57 InitPlantDynProcess internals

Figure 3-58 PlantDynProcess internals
This sequence in fact borrows a common solution for solving algebraic loops that can occur in dynamical models with feedback, as in a topology in Figure 3-51. For another combination of transfer functions of the LoopCon and InitPlantDynProcess blocks, it would be necessary to define an initial value for an integrating element that would resolve the causality problem which block can produce the output first in simulating the loop. The problems (leading to deadlock) in a simplified gCSP model are discussed on this example in the next chapter.

Internals of the process PlantDynProcess are depicted in Figure 3-58. First the values from the steering channel are read to the variable $u$ by READER.u, than the value of the $x$ variable is calculated by PlantDynamics, and consequently that value is output to the state channel by WRITER.x. The C-tree in Figure 3-59 gives a complete overview of the compositional structure of the model.

![Figure 3-59 The C-tree of the control loop example](image-url)
For comparison of the overview capabilities, the model at hand is flattened in Figure 3-60. Thus, the containment hierarchy is removed. Therefore, the issue of managing the parenthesizing grouping arises. Figure 3-61 provides the corresponding boxed-grouping. The rule of indexing the bubbles is easy to verify.

Figure 3-60 Compositional view to internals of the processes from Figure 3-54 brought to the top hierarchical level ("exploded" view)

Figure 3-61 Boxed-grouping interpretation of parenthesizing indices from Figure 3-60
The absence of the containment hierarchy reduces the size of the C-tree. It is apparent from Figure 3-62 that partitioning in constructs alone keeps a tractable structure of the model (compare with the C-tree in Figure 3-59).

Figure 3-62 C-tree of the exploded view from Figures 3-60 and 3-61

3.4 The gCSP tool

This section briefly introduces the elements of the gCSP tool user interface and features relevant for the scope of this thesis. For detailed information on using the tool the reader is referred to (Design Tools project, 2001-2005b; Jovanovic et al., 2006a).

It is a standard windowed SDI (Single Document Interface) application. The gCSP tool is programmed in Java, permitting availability on different platforms. For a proper running, JVM 1.4 or higher should be installed. The format of model files is coded in XML (with .gcsp extension), while the tool outputs are also other file formats (CT libraries source files, occam, CSPm or graphical formats). The main tool screen is shown in Figure 3-63.
The two dominating design areas are the left pane with the C-tree and the right pane with the CSP diagrams editor. The most left toolbar belongs also to the CSP diagrams editor, further on called graphical editor, or G-editor. Hence, the main design entries are the G-editor and the C-tree. A pane for issuing textual feedback to the user is positioned below the G-editor. The pane above the G-editor contains an overview of an actual process interface. All these panes can be optionally shown or hidden.
3.4.1 Tool menus and the toolbar

Besides the standard tools of a PC application (opening/saving models, printing the models, cut/copy/paste, undo/redo), the gCSP toolbar is extended with several domain-specific buttons (Table 3-4).

<table>
<thead>
<tr>
<th>Toolbar elements for managing gCSP models</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon] Retrieved and saving the parts of a model contained in a complex process (submodels)</td>
</tr>
<tr>
<td>![Icon] Navigating through the containment hierarchy of graphical models in the G-editor, going out and in complex processes</td>
</tr>
<tr>
<td>![Icon] Toggling the boxed-grouping layer</td>
</tr>
<tr>
<td>![Icon] Toggling the exception handling layer</td>
</tr>
<tr>
<td>![Icon] Toggling the watchdogging layer</td>
</tr>
<tr>
<td>![Icon] Toggling the logging/monitoring layer</td>
</tr>
<tr>
<td>![Icon] Shortcuts for the code generation engines</td>
</tr>
</tbody>
</table>

3.4.2 Graphical editor

While the compositional and communication views are clearly suitable for focusing on one of the corresponding architectural aspects, in the design phase it is also handy to have an overview of both compositional and communication relationships at the same time - the hybrid CSP view. However, by opting for displaying all model details, the hybrid CSP view quickly gets cluttered and unreadable. Therefore, displaying the names of all relationship is useful only on separate views.

For facilitating management of various aspects of the models, besides different views on the CSP diagrams, the tool introduces also modelling layers, which can be superimposed on the diagrams or hidden. In the current version, the tool prototypes four layers: exception handling, watchdogging, logging/monitoring and the layer for the boxed-grouping notation, whose functionalities will be clarified in the upcoming chapters.

3.4.3 The C-tree

Figure 3-63 displays the C-tree part of the gCSP interface, in totality with the Loose Processes and the Unresolved Relationships compartments.
Unresolvable relationships are the compositional relationships between processes that are not yet composed into constructs (loose processes). These compartments concern the elements of an underspecified compositional structure at a certain hierarchical level (submodel). For a compositionally determined submodel, the Unresolved Relationships compartment must be empty, while the Loose Processes contains only one process: the top construct of the submodel.

3.5 gCSP models of the case studies

This section presents gCSP models of the two robotic case studies (introduced in Chapter 1) in a role of functional context diagrams.

3.5.1 JIWY

JIWY operates in three modes (Figure 3-64). The main mode implements a position servo control regime, where the horizontal and vertical axes follow the reference signal from the analogue X-Y joystick. This JIWY functionality is captured by the Servo process. In order to enter the servo mode from the central position with respect to both axes, an alignment mode, encapsulated in the Calibration process precedes the servo mode. After termination of the servo mode (in an arbitrary position), the Homing process takes over, driving the axes to the central (safe) position. The termination of the servo mode can be caused by pressing the joystick buttons or upon an exceptional event. Interaction with the hardware (joystick, encoders, motors), for sake of simplicity, is not presented on diagrams in this chapter. More operational details are described in the following chapters. Some peculiarities of handling discrete joystick events (buttons) can be found in (Hilderink, 2005a, p.249).

The communication aspect in Figure 3-64 specifies that determination of the extreme positions of both axes is the task of the Calibration process. These values are passed to the Servo process by the variables leftMax_H, rightMax_H (for the horizontal axis) and rightMax_V and leftMax_V (for the vertical axis). In the servo mode the central position, determined from the extreme values, represents the zero reference level. This position, for both horizontal and vertical axis is passed to the Homing process through variables center_H and center_V respectively. The Homing process drives the axes to this position.
The C-tree in Figure 3-65 shows two containment hierarchical levels. The processes on one level lower are further refined by leaf processes. Figures 3-66, 3-67 and 3-68 show hybrid CSP views at the first containment level. Both from the tree and the hybrid CSP views the symmetry of the software components for horizontal and vertical axes is apparent.

The internals of the process Calibration in Figure 3-66 show the execution flow of the calibration mode. Both axes go through sequences of exploring the extreme left positions, restoring at the initial position and exploring the extreme right positions. The extreme values are stored in local variables and communicated to the upper channels through the communication interfaces. The last processes (H_Home, V_Home) in the sequences place the axes in the central positions on basis of the extreme encoder values. After that the Calibration process terminates. The parallel composition of the alignment sequences for horizontal and vertical axes allows the calibration of the axes to be performed simultaneously.

The servo mode (Figure 3-67) consists of three processes: parallel composed Horizontal and Vertical controllers and a motor driver process that validates the values steered to the motors (SanityCheck). This operational mode will be given a lot of attention in the following two chapters. For the homing mode processes H_Home and V_Home from the Calibration process are reused (Figure 3-68). The values of center_H and center_V variables are calculated by the Servo process and given over to the Homing process after termination of the servo mode. Upon receiving these values, the
two homing processes drive the axes in parallel to the central positions. After that the JIWV software terminates.

**Figure 3-66** Internals of the **Calibration** process

**Figure 3-67** Internals of the **Servo** process

**Figure 3-68** Internals of the **Homing** process
3.5.2 Tripod

The Tripod robot also operates in the three modes, for alignment, servo control and homing. The last two are at the top-level Tripod model (Figure 3-69) encapsulated in one complex process, named ServoController. The alignment and calibration mode are implemented within the StartupController process. The third process in Figure 3-69 is a process that transforms the steering values commanded by the control modes to three-phase signal for Tripod’s linear motors. It runs in parallel with all modes and submodes.

For brevity, in this section only the top and the first level in the containment hierarchy are shown. Internals in the deeper hierarchy are shown where needed later in the text. For Tripod, Figure 3-69 shows also interfacing with the hardware. There are three encoders, mapped to channels with different names for different working modes, therefore presented as nine writer linkdrivers. The tenth Period linkdriver represents time sampling.

The internals of the StartupController are displayed in Figure 3-70. Four subphases precede the servo position/shut-down mode, organized as in Figure 3-71. This mode is in fact a state machine, with the four processes modelling four states of the machine in the servo mode. Transitions among the alternatively composed states is managed by the process ModeSwitcher. The shutting-down mode, being also a servo position mode, is encapsulated together with the other servo submodes. Finally, Figure 3-72 shows the commutation of all three AC phases of Tripod’s linear motors.
Figure 3-70 Alignment/calibration sequence in the start-up controller

Figure 3-71 Mode switching state machine in the servo controller
Figure 3-72 Internals of the Commutator process
3.6 Conclusions

The gCSP tool supports the CSP-based modelling paradigm that combines the graphical notation and formal specification of process-oriented architectures. Moreover, it allows automatic code generation of complete executable software. Hence, the two main functionalities of the tool are graphical modelling, presented in this chapter, and code generation, elaborated in the next chapter.

The major part of the graphical language implemented in gCSP stems from the proposal of CSP diagrams (Hilderink, 2005a). Several extensions to this language encompass some elements important for practical applicability, including dependability provisions. CSP diagrams as a graph representation of the CSP/CT designs are complemented with the compositional tree to form together the gCSP model.

In order to improve handling complexity of concurrent software due to the number of aspects that influence ways a CSP architecture should be modelled and managed, besides the different views into a CSP diagram, the tool prototypes also design layers. The layers are optionally displayed on top of the two CSP views in the G-editor, allowing the user to balance the level of information in graphical representation of a model at hand. To that end, the tool introduces a drawing means for visualizing construct boundaries in the diagrams (boxed-grouping) and an optional distinctive shape for the exception handling processes and watchdog emergency processes. These two orthogonal dependability mechanisms, as well as logging and others presented in later chapters, are recognized as the typical cross-cutting concerns in aspect-orientation software development paradigm (Filman et al., 2005). The core chapters of this thesis demonstrate how these aspects can be treated separately at the modelling level.

Next to the boxed-grouping notation, the tool functionality is substantially influenced by the introduction of the C-tree compositional structure along the CSP diagram views. Similar trees were only visualization means in the previous prototypes of the CSP/CT modelling tools, believed to become obsolete by the implementation of the CSP diagrams. However, already in early stages of the tool development it was realized that the quality and efficiency of the tool would benefit a lot from a useful combination of the CSP diagrams and the tree-like hierarchical structure complementing the graphs. Shortly, the C-tree facilitates the transformation from the CSP diagrams to the machine readable forms of the models, provides much more efficient means for navigating and viewing the model structure and indicates the level of the compositional determination of a gCSP model in total. The full appreciation the reader will get in part II of this text.

3.6.1 Directions for further development

With the complexity of practical problems, it is hardly possible that a novel engineering discipline, even though promising, may merit attention of the industrial users if not accompanied by a supporting tool of a sufficient
sophistication. The feedback from the industrial partners of the project (Design Tools project, 2001-2005a) on the research results with respect to the design methodology and the implementation tools were encouraging, especially concerning the possibilities of having the CSP designs formally verified. Both the design trajectory and the tool quality will experience major benefits from a broader exploitation by the academic and industrial users. In order to make the first experiences to the target groups even more appealing, the most imperative improvements are listed here:

- Automating creation of the C-tree out of a model’s compositional hierarchy expressed by the boxed notation; further, transforming the C-tree into the parenthesizing (“bubble”) compositional grouping as an alternative way of presentation. These transformations practically would allow generating the code directly from the CSP diagrams.

- For marrying the intuitive value of the boxed-grouped notation with compactness of the parenthesized demarcation within compositional hierarchies, it would be necessary to embed some sophistication in the way the gCSP tool handles the boxes. In fact, the tool should not stick to this simple geometric boundary, but encompass child processes tighter within flexible shapes.

- The current graphical specification for nested alternative constructs awkwardly handles comm-guards, which are attached to the processes and not to the compositional relationships. A separated graphical notation for the comm-guards would be recommended.

- Modelling synchronisation on barriers by aggregating some of the existing CSPm features and implementing them in the CSP would be the next task in the research and development of the CSP/CT dependability arsenal (on their potential use within the N-version programming see section 6.6.4 on page 210).

- The development requirement 7 from page 64—visualization of the deadlock and other imaginable formal analysis—is not implemented in the current tool version. To facilitate this, the G-editor should be enhanced with a capability of controlled flattening the model structure, which would build on the existing feature of imploding part of network into complex processes and exploding complex processes by bringing their internal to the higher hierarchical level.

- The ProBE tool demonstrates the way the event advancement in time of a CSP network can be interpreted (mimicked). This animation is based on sequences of events (traces). With a more pronounced support for process alphabets (events), a possibility of simulating CSP diagrams becomes feasible. This would contribute significantly to understanding CSP models and learning the CSP principles.
Part II  Dependability instruments for process-oriented software

Chapter 4  Automatic code generation and formal verification of CSP/CT software

Chapter 5  Exception handling mechanism for CSP/CT software

Chapter 6  Dependability design patterns for CSP/CT software
4 Automatic code generation and formal verification of CSP/CT software

The standpoint of the modern software development doctrine is that more than a half of the development time should be spent in software modelling (Booch et al., 1999). The rest goes to all other stages: verification, implementation, testing, optimization.

This ratio is hardly sustainable if the software development stages following the modelling process are not supported by the development methodology and tools. Reliance on manual transformations of a complex model to machine-readable source code contradicts the requirement of the predominant modelling stage. Manual coding is known as highly error-prone and expensive. Therefore, an automatic mechanism of transforming software models to program executables is not anymore a preference, but a necessity.

In order to keep manual interventions in the produced machine-readable outcomes from the modelling tools to a minimum, the modelling paradigm and notation have to enable detailed software specification. Managing the modelling process with low-level source code refinements is far from trivial. The notation and supporting tools face the problem of bridging the conceptual distance of abstract (human-comprehensible) and concrete (machine-readable) system specification.

The mechanism of automatic code generation (ACG) in gCSP is used to address two important aspects of software quality. The first is obtaining a machine-readable formal specification from CSP diagrams ready for formal verification (FV) in the CSP model checking tools. The second one is
producing source code from the graphical specification that can be compiled to executable programs without manual interventions. For an intermediate presentation of the compositional hierarchy contained in a CSP diagram model the C-tree is used. Producing one or other machine-readable presentations does not require any change in a gCSP model. The two transformations are produced from a single model, by choosing different menu commands of the tool.

Following a logical order in software development, after specifying a program model, the specification is formally verified. Upon possible corrections and/or refinements, the source code gets generated and the software can be further tested in its executable form. That order is followed also in this chapter. Section 4.1 gives a brief overview of state of the practice of formal analysis and automatic code generation. In section 4.2 the mechanism for generating a formal specification of a gCSP model and interpretation of deadlock conditions is described. Model transformations into CTC++ code is subject of section 4.3. Section 4.4 exercises both code generation mechanisms on the JIWY case study. A summary of code generation techniques and their uses concludes this chapter in section 4.5.

4.1 Transforming abstract models into machine-readable forms

As commented in the previous chapter, commonly used modelling tools in research and industry are specialized either for formal analysis or code modelling and generation. In the upcoming sections the capability of the gCSP tool to generate code both for model checking and implementation out of a same graphical model will be elaborated.

If not in the direction of formal specification on basis of graphical notations, there are significant advancements with respect to automatic generation of deployable code out of graphical software descriptions. The leading UML-based CASE tools (Rational Rose, Rhapsody) possess highly customisable code generation engines for modern mainstream languages. Nowadays, generation of source code in the mainstream languages is not only a feature of CASE tools – also CAD tools in domains other than software development transform domain-specific dynamical models into executable or source code. For control applications, the examples are 20-sim, the Matlab/Simulink Real-time suite (Mathworks Inc., 2005) and LabVIEW Real-Time (National Instruments Inc., 2005).

The major problem to overcome in automatic code generation is the distance between highly abstract graphical models (often in different engineering fields) and the target programming language. To allow for a smooth refinement of the design artefacts at different levels of abstraction and in different engineering disciplines, a design trajectory has to go through several domains. Spanning distant system specifications is often done by multiple transformations through gradual refinement stages. An exhaustive overview of interdomain transformation approaches is given in (Milićev, 2001). The last transformation to machine-readable source code is usually
performed by automatic code generation techniques (although similar
techniques can be deployed also for higher transformations).

4.1.1 Formal analysis

Benefits from formal analysis of a model of a (concurrent) system are well
known; however, cases of effective use of formal methods in industrial
practice are exceptional, and usually pursued when regulated by the law.
Reasons for a rare use of formal methods in industrial practice are already
discussed in section 1.4.4. However, when used, the common approach
consists of three major steps:

1. The problem at hand has to be modelled by a chosen formalism in a
way that captures relevant behaviour. The problems to be faced are
the choice of the formal notation (language) and proper application on
the problem peculiarities.
2. Articulating assertions of the model properties of interest. It is not
enough to know how to model the problem and what properties to
search for, but how to ask for them. Principal kinds of assertions to
be found in the majority of formal methods deal with liveness,
deadlock, livelock, safety and determinism properties.
3. Finally, the practitioner has to fire the formal checking algorithm and
to interpret the obtained result. The common problem is an
exponential growth of time required for formal checking with respect
to the size of the model (i.e. growth of operational number of states,
popularly called “state-space explosion”). The other problem is
mapping the obtained result back into the model in order to
understand and correct possible modelling omissions.

Different formal methods exhibit different levels of success in the listed
analysis phases. The common conclusion is that a higher level of automation
and tool support for all analysis stages are required for industrial
applications, as well as a clear understanding what are the limitations of
each advocated formal approach.

The capacity of checking graphical CSP/CT models formally by help of
gCSP is illustrated with verifying deadlock-freedom. Some design errors, not
identifiable by the FDR model checker, that can lead to deadlocks are
discovered by the tool itself.

4.1.2 Automatic generation of source code

There are two principal motivations for making the modelling tools capable of
generating source/executable code out of (graphical) software models:

- to eliminate manual transformation from modelled behaviour to
  computer code (for instance, models of controllers to control code).
  The manual coding of abstract models is proven to be lengthy (i.e.
  expensive) and error-prone.
• although it is claimed that “software does not deteriorate with use”, it is also true that the structure of the software actually deteriorates with maintenance (“software aging”, (Belady and Lehman, 1976; Parnas, 1994; Van Gurp and Bosch, 2002)). A carefully designed code generation engine generates well structured programs regardless how complex the software systems grow.

A serious problem of automatic code generation is the minimization of manual interventions in producing executables. This is not such a problem with respect to the code providing the software architecture (structure) as it is when the design comes to the low-level code specification, mostly with respect to dealing with hardware and bespoke processing algorithms. The actual processing algorithms are defined by the user. In tools the user-defined code (or the code that can hardly be specified on the high abstract level of graphical modelling) is placed in the overall code structure in two ways:

1. The user edits the source files once generated by the tool. Manipulating the source code is allowed only in the sections marked by the tool as editable. During regeneration of the code these code sections are preserved. The user has to work always on the same set of source files (because the manually entered code is not part of the model). The advantage of this approach is that it enables flexible interventions in the source code. The disadvantage is that the user is allowed to interfere beyond the model; the model does not hold complete program specification.

2. The low-level code is managed within the graphical tool. Thus, the model contains all the information to reproduce the complete code. The imperfection of this approach is inconsistency with the prime intention to design system graphically: graphical design proves problematic for entering for example the hardware initialization/release and bespoke algorithms. For parts of the system of this kind the design must resort to low-level code editors within the tool.

The development of the gCSP tool is putting forward the second approach, in order to maintain models self-contained and reusable.

4.2 CSPm code generation and formal deadlock checking

To illustrate the use of generated CSPm formal description for checking properties of gCSP models, we will use the 20-sim block diagram of the closed loop straightforwardly translated to the gCSP model in Figure 4-1. Internals of the processes are visible on the flattened version in Figure 4-3.
The CSPm script stemming from the C-tree of this example (Figure 4-2) is presented in the following listing.

```
datatype Double = step_val | P_val | x_val

channel steering : Double
channel reference : Double
channel state : Double

ParClosedLoop = LoopConProcess || (SeqConProcess || PlantDynProcess)

SeqConProcess = reference!step_val -> SeqConProcess
PlantDynProcess = steering?u -> state!x_val -> PlantDynProcess
```

Analyzed both with ProBE and FDR, all processes exhibit a deadlock-free behaviour, except the network builder (ParClosedLoop process)! This means that the specified system deadlocks!
The deadlock manifests itself due to communicational incompatibility of the `LoopConProcess` and `PlantDynProcess`:

\[
\text{LoopConProcess} = \text{reference?SP} \rightarrow \text{state?MV} \rightarrow \text{steering!P_val} \rightarrow \text{LoopConProcess}
\]

\[
\text{PlantDynProcess} = \text{steering?u} \rightarrow \text{state!x_val} \rightarrow \text{PlantDynProcess}
\]

The emphasized expressions reveal the conflict: `LoopConProcess` tries to input the feedback value from the channel `state`, and then output the control instance to the `steering` channel. `PlantDynProcess` attempts just the opposite: first reading from `steering`, and than writing to `state`. The two processes do not agree to engage in any of these events (channels) due to the clash of the order of readings and writings. The network hangs – does not start the cyclic calculations.

The deadlock occurrence in such a configuration can be also interpreted in the graphical way (Figure 4-3). Note the cycle (a closed oriented path) composed by the sequential relationships in the processes `LoopConProcess` and `PlantDynProcess` and the channels between them. The deadlock condition is indicated by the uniform orientation of the sequential relationship along the designated closed path (the orientation of the channels does not matter).

This example illustrates that a blunt parallelization of entities found in other domains can yield unsound systems even for regular models from those domains. The original 20-sim model in Figure 3-51 which is successfully simulated, yields control code which composed within an identical structural topology produces erroneous control software. In most cases some conditions from the original domain get lost along the interdomain translation, or certain
assumptions about the translation fail to hold. As commented on page 95, for certain classes of transfer functions or choice of discrete implementations, dynamical models may contain algebraic loop, sorted out by the 20-sim simulation engine. However, that information is lost in the translation from the block diagram to the gCSP model, which only later reappears as the deadlock condition. This proves necessity of asserting healthiness of translated models.

In this particular case, there are two solutions for eliminating the deadlock condition. One has been already seen in the previous chapter: the controlled object was represented by a complex process which, besides the PlantDynProcess process, contained a primitive writer that initiated the cyclic circulation of values along the closed path – as having defined initial value for an integrator in a loop of a dynamical model that resolves an algebraic loop.

The other solution is suggested by the uniformity of orientation of the closed path (cycle) in Figure 4-3. Breaking the orientation of the cycle leads to a deadlock-free network. In order to accomplish that, the communication order in one of the involved processes has to be altered. This modification may not be easily possible (without more elaborate modification of the algorithms) in all cases. Nonetheless, using the simplicity of the PlantDynProcess, in this example a simple modification is possible. Drawing from the already discussed solution, the key is making the PlantDynProcess output a feedback instance to the state channel first, and when the first steering instance becomes available on the steering channel, engage in the first calculation of the dynamical response of the controlled plant. As it will be seen later (the code of the PlantDynamics code block in 4.3.2), the first sample from the state channel that the process LoopConProcess gets is always zero, since the initial value of the state variable $x$ in PlantDynProcess is zero. Therefore the calculation correctness, by outputting the variable $x$ first and than engaging in the steering input and dynamics calculation, in this case is not violated. This modification is reflected in the composition of PlantDynProcess (Figure 4-4) and consequently to the C-tree (Figure 4-5).
The “problematic” lines in the initial CSP script after this update reflect the harmonized communication patterns (in the script segment below), which is also visualized in Figure 4-6.

\[
\begin{align*}
\text{LoopConProcess} &= \text{reference?SP} \rightarrow \text{state?MV} \rightarrow \text{steering!P_val} \rightarrow \text{LoopConProcess} \\
\text{PlantDynProcess} &= \text{state!x_val} \rightarrow \text{steering?u} \rightarrow \text{PlantDynProcess}
\end{align*}
\]

Figure 4-6  Exploded control loop model with the deadlock-free cycle
This example illustrates how presence of a deadlock condition is detected. Still, tracing down the exact cause of it is not straightforward. Interpreting the FDR output itself requires substantial skills grounded on solid insight in the CSPm notation and the CSP notions. In order to provide a more tractable output from FDR for problems of practical size, manipulating the CSPm scripts manually may be necessary as well. One of the future research and development efforts may be directed in a mechanism for feeding the FDR outputs back to the visual presentation in gCSP.

Although useful for interpreting the deadlock phenomenon for a particular class of CSP networks, the presence of the oriented closed path along the sequential relationships and (synchronous) channel is a necessary, but not a sufficient condition. For instance, the closed oriented path exists also in the hybrid CSP diagram in the variant of this example from the previous chapter, although not causing a deadlock. This is easily proved by running an FDR analysis or generating, compiling and executing the program (which works properly). Informally, the indication of the deadlock condition by means of the closed oriented path makes sense in a network of processes whose internals consist of sequential compositions. Well known recipes for preventing deadlock conditions (Welch, 1987; Martin, 1996) actually amount to elimination of sequential compositions whenever possible, in forms of I/O-SEQ and I/O-PAR design patterns (Welch, 1987). This approach will be demonstrated on the case study later in this chapter (section 4.4). Still, the visualization of the deadlock conditions in the forms of closed oriented paths has its educational significance. To research and exploit it readily, the "exploding" feature for gCSP is proposed: automatic flattening hierarchical models.
4.2.1 CSPm code generation options

The CSPm code generated by the gCSP tool can be customized with respect to a few details. The Tools > Code Generation > Preferences dialog with selected “CSPm Preferences” tab is shown in Figure 4-7.

![Figure 4-7 Code generation preferences dialog](image)

Choice of the options influences the code generation as follows:

1. By default, the warnings on some irregularities in a gCSP model that affect the CSPm code generation are being reported in the gCSP Warning Message pane. Examples are: primitive communication processes not connected to a channel or to a variable, parallel composed processes connected by a var-channel, sequentially or alternatively composed processes connected by a rendezvous channel or ordinary processes composed with the prefixing construct in the C-tree.

2. As already noted, the tool automatically performs some CSPm code reductions (optimizations) in order to produce the CSPm scripts simpler and compliant with the common notation. If the “Generate with building blocks” option is activated, all primitive processes are explicitly declared in the script; consequently, the prefixing optimization is overruled.

3. It has already been noted in section 2.7.3 (page 56), ProBE does not make distinction between usage of “?” or “!” in communication events. It takes into account only events on the channels, without matching writers and readers. The explicit declaration of reading and writing operators is used by default in order to keep the detailed information in the CSPm scripts and improve its readability. However, for the deadlock analysis this level of detail does not matter, so the generated code can be even further simplified by ticking the “Generate events only” option in the code generation dialog. The
example script would be even simpler, as illustrated in the following listing.

```plaintext
channel state
cchannel steering
channel reference

ParClosedLoop = LoopConProcess || (reference, steering, state) ||
(SeqConProcess ||| PlantDynProcess)

SeqConProcess = reference -> SeqConProcess

LoopConProcess = reference -> state -> steering -> LoopConProcess

PlantDynProcess = state -> steering -> PlantDynProcess
```
4.3 Code generation of implementation source code

The gCSP source code generation engine is developed for CTC++ code in order to prototype the design trajectory applicable for practical case studies. Subsequently the occam generator is added (Groothuis et al., 2005). The CTC and CTJ engines are planned for a later development stage.

4.3.1 Network builder and source code structure

The gCSP tool generates C++ source files in a directory named after the model at hand. A simple organization of the source code is implemented, where each file contains a C++ class for every process in the model. Since every process, represented in run-time by an object, has a corresponding class, each class is used to instantiate only one object. This “one process – one class – one object” simplification and its limitations, as well as detailed information on the generated source files, are presented in Appendix D.

To illustrate the mapping between the graphical model structure and CTC++ network builder, a general example in Figure 4-8 is constructed. It is followed by the CTC++ code generated out of it.

```c++
/** Auto Generated - gCSP **/

//include’s section: include all necessary CTC++ headers and process headers

int main (void) {

   //-- Channel Allocations
   Channel<double> *ch1 = new Channel<double>{};
   Channel<double> *ch3 = new Channel<double>{};

   //--
```
Channel<double> *ch6 = new Channel<double>();
Channel<double> *ch2 = new Channel<double>();
Channel<double> *ch5 = new Channel<double>();
Channel<double> *ch4 = new Channel<double>();

//-- Process Allocations
SeqP1 *SeqP1_1 = new SeqP1(ch1);
SeqP2 *SeqP2_1 = new SeqP2(ch2, ch4);
ParP1 *ParP1_1 = new ParP1(ch1, ch3);
ParP2 *ParP2_1 = new ParP2(ch3, ch5, ch4);
AltP1 *AltP1_1 = new AltP1(ch2);
AltP2 *AltP2_1 = new AltP2(ch6);
ParP3 *ParP3_1 = new ParP3(ch6, ch5);

Guard *Guard_AltP1_1 = new Guard(AltP1_1);
Guard *Guard_AltP2_1 = new Guard(AltP2_1);

//-- Network Builder
Alternative *Alt1 = new Alternative{
    Guard_AltP2_1,
    Guard_AltP1_1,
    NULL};
Parallel *Par1 = new Parallel{
    ParP3_1,
    ParP2_1,
    ParP1_1,
    NULL};
Sequential *Seq1 = new Sequential{
    SeqP1_1,
    SeqP2_1,
    NULL};
Parallel *Par2 = new Parallel{
    Alt1,
    Par1,
    Seq1,
    NULL};

Par2->run();

//delete’s section: delete all dynamically allocated objects
return 0;
For the control loop example (Figures 4-1 and 4-2), the network builder is somewhat simpler:

```c
/** Auto Generated - gCSP **/

//include's section: include all necessary CTC++ headers and process headers

int main (void) {
    
    //-- Channel Allocations
    Channel<double> *steering = new Channel<double>();
    Channel<double> *state = new Channel<double>();
    Channel<double> *reference = new Channel<double>();

    //-- Process Allocations
    SeqConProcess *SeqConProcess_1 = new SeqConProcess(reference);
    LoopConProcess *LoopConProcess_1 = new LoopConProcess(steering, state, reference);
    PlantProcess *PlantProcess_1 = new PlantProcess(steering, state);

    //-- Network Builder
    Parallel *ParClosedLoop = new Parallel(
        LoopConProcess_1,
        SeqConProcess_1,
        PlantProcess_1,
        NULL);

    ParClosedLoop->run();

    //delete's section: delete all dynamically allocated objects

    return 0;
}
```

### 4.3.2 Low level refinement and custom (user-defined) code

Network builders for subnetworks (subprocesses) in a CSP/CT program hierarchy look very similar to the presented top-level network builders. The gCSP tool consistently generates the network consisting of constructs, ordinary processes, channels and channel interfaces. However, at the lowest level the actual processing and data exchange takes place. For the communication, primitive processes for data input/output are used as building blocks from the CTC++ library – the source code for them need not be generated. Primitive repetition processes are usually generated in form of the loops within the network builders. In some specific, not so common
situations, separate .cpp’s named after the repetition process may be created that contain an appropriate loop for repeating single processes or networks.

The actual processing algorithms are defined by the user. As it has been already discussed in section 4.1.2, gCSP contains complete models, including the low-level source code contained within code blocks. For the deadlock free composition on Figure 4-6 the Euler simulation with a simulation timestep of 1 second would be coded like in the following listings.

SetPoint:

\[
\text{step} = 1 ;
\]

ControlLaw:

\[
P = K \ast (\text{SP} - \text{MV}) ;
\]

PlantDynamics:

\[
\begin{align*}
// \text{calculate the plant model} \\
\text{dx} &= a \ast \text{x} + b \ast u ; \\
// \text{Euler integration} \\
\text{x} &= \text{x} + \text{dx} ; // \text{hk = 1}
\end{align*}
\]

Custom code blocks are useful for arbitrary user-defined code. However, under a reasonable assumption that satisfactory (or even sophisticated) dynamical models are worked out in 20-sim, manual coding of the models as in the previous listings represent an error-prone design discontinuity. 20-sim is capable of generating C/C++ code for the dynamical submodels that exactly reflect simulation models developed in the design phase that precedes composition of the control software. The next section describes a method of including (sequential) 20-sim generated code in the gCSP models and consequently creation of control software within the CSP/CT concurrent framework.

### 4.3.3 Inclusion of 20-sim generated code

Figure 4-9 illustrates refinement of a gCSP model with the low-level algorithms, actually a subworkflow that finds place in the final block of the general CSP/CT workflow from Figure 2-8 on page 62 (“Control software generation”). As the figure suggests, when having available 20-sim-generated control code, there is no need for manual coding of the control laws. For combining 20-sim generated source code of dynamical submodels and a gCSP generated concurrent framework, code generation engines of both parties are extended to interface with each other.
4 Automatic code generation and formal verification of CSPCT software

The 20-sim code generation is customizable by the user who can provide source code templates with a number of predefined keywords to be replaced with actual model entities by 20-sim. For using 20-sim code with gCSP generated code compliant with the CTC++ library, 20-sim code generation templates are improved as elaborated in Appendix D. Including C++ code of 20-sim submodels in the gCSP models is performed by 20-sim code blocks.

Using 20-sim code blocks

Interface on the side of the code generated from a 20-sim functional block (submodel) is contained in the header of the .info file named after the submodel. For the P-controller from Figure 3-51 (page 91) this information is shown in the following listing excerpt. The script relates the \(x\) and \(y\) vectors (used for transferring the calculation variables to and from the dynamic equations in the 20-sim generated code) with the symbolic variables names used at the model level.

```plaintext
INPUTS & OUTPUTS INDICES

*** INPUTS ***
xx_V[1] = u[0];  /* MV */
xx_V[0] = u[1];  /* SP */

*** OUTPUTS ***
y[0] = xx_V[2];  /* P */
```
For each 20-sim dynamical submodel whose code is intended to be used in a CTC++ program, one 20-sim code block is inserted in the gCSP model. The name of a 20-sim code block must match the name of the submodel from 20-sim. To each port of a 20-sim submodel corresponds a variable in the gCSP model, whose name matches the name of the submodel's port. The way the variables are connected to the code blocks corresponds to the orientation of the ports (input or output).

In the deadlock free composition from Figure 4-6 all custom code blocks are replaced with 20-sim code blocks (Figure 4-10). Based on the number, names and orientation of the variables connected to the code blocks, each code block maintains lists of input and output vectors \( \mathbf{u} \) and \( \mathbf{y} \) and candidate variables. The dialog for the LoopCon code block shows the connection of the vector components and the variables (Figure 4-11) – forming the code inclusion interface on the gCSP side. For assigning gCSP model variables to input and output vectors the shown .info file is used.
4.3.4 Hardware manipulation code

Generally, for capturing hardware access points linkdrivers are used. They refer to the device drivers initialized at the beginning of a program execution. Parts of the drivers are residing in the dynamic memory, hence need to be deallocated at the end of the program execution. This portions of the program code for manipulation of hardware resources are specified in the Model icon, the root of the C-tree. The specification dialog is shown in Figure 4-12.

![Figure 4-12 The specification dialog for the Model icon in the C-tree](image)

4.3.5 CTC++ code generation options

Default code generation settings yield the following process body for the LoopConProcess.cpp generated for LoopConProcess from Figure 4-3. The model with custom code block is used for simpler understanding of the generated code. (The code is presented in two parts, with leaving out some less interesting lines in between).

```cpp
class ControlLaw : public Process {
    public:
        ControlLaw(double& SP, double& P, double& MV, double& K) :
            SP(SP), P(P), MV(MV), K(K){};

        void run() {  // calculate the steering
            P = K * (SP - MV) ;
        }
    
    private:
        double& SP;
        double& P;
        double& MV;
        double& K;
};
```
LoopConProcess::LoopConProcess(ChannelIn<double> *reference,
ChannelOut<double> *steering, ChannelIn<double> *state) {
    this->DataChannel13 = reference;
    this->DataChannel12 = steering;
    this->DataChannel14 = state;

    K = 10;

    //-- Process Allocations
    reader_Reference_1 = new Reader<double>(DataChannel13, &SP);
    reader_Feedback_1 = new Reader<double>(DataChannel14, &MV);
    writer_Steering_1 = new Writer<double>(DataChannel12, &P);
    ControlLaw_1 = new ControlLaw(SP, P, MV, K);

    //-- Network Builder
    Seq_LoopCon = new Sequential{
        reader_Reference_1,
        reader_Feedback_1,
        ControlLaw_1,
        writer_Steering_1,
        NULL};
}

//-- Run method
void LoopConProcess::run() {
    while(true) {
        Seq_LoopCon->run();
    }
}

//-- Destructor's

ControlLaw_1 (in the first part) represents the controller dynamics coded in
an implicit process. If a 20-sim code block LoopCon (Figure 4-10) had been
used instead of the custom code block ControlLaw, the similar implicit
process would have been generated named XXLoopCon, having “XX” relate to
“20” – following a standard 20-sim code generation convention.

Second part of the listing reveals that the sequential construct
Seq_LoopCon consist of four processes: a code block (the implicit process),
one writer and two reader building blocks. In sequential execution patterns
the overhead of having ordinary processes (and the sequential construct) can be optimized away; that is the consequence of turning the sequential construct into a prefixing one in the C-tree. The result of this optimization is shown in the following listing. Most of the dynamically allocated object in the former listing does not appear here (sections for process allocations and network builder are empty).

```cpp
LoopConProcess::LoopConProcess(ChannelIn<double> *reference,
                             ChannelOut<double> *steering, ChannelIn<double> *state) {
    this->DataChannel13 = reference;
    this->DataChannel12 = steering;
    this->DataChannel14 = state;

    K = 1;

    //-- Process Allocations

    //-- Network Builder
}

//-- Run method
void LoopConProcess::run() {
    while(true) {
        DataChannel13->read(&SP);
        DataChannel14->read(&MV);
        // calculate the steering
        P = K * (SP - MV) ;
        DataChannel12->write(&P);
    }
}
```

Through a CTC++ tab in the Tools > Code Generation > Preferences dialog, the user may force using the primitive writers and readers as processes—building blocks—when generating the CTC++ code. In that case, compared with the previous listing, the primitive processes would be allocated and reading and writing from/to channels would be performed by running these processes.

For testing programs and logging data communicated over channel, one would have to add to the model additional code blocks only for outputting variables values to the screen. In order to avoid polluting the model with this kind of debugging means, it is made optional that primitive
writers output to the screen the values they are about to output to the channel along with a user-defined message. The readers may also print to the screen the value of the variable they have just entered from a channel.

### 4.4 Case study

The described procedure of generating deadlock-free CTC++ programs will be demonstrated on the JIWY servo operational mode. The gCSP model for the Servo process from Figure 3-67 is refined with interaction with the hardware, while the channels for mode transition (extreme left and right and centre position) are omitted – Figure 4-13a.

![Figure 4-13 JIWY Servo process with link-drivers](image)
An “exploded” model in Figure 4-14, detailed as the C-tree in Figure 4-13b, reveals graphically the internals of the processes. The cores of controller processes are 20-sim code blocks implementing previously developed position controllers in 20-sim. The order of activating the readers, code blocks and writers is initially indicated by the sequential relationships. A sanity check process implements reading the steering instances calculated by the controllers, verification of their values sanity and forwarding (limited) values to the motors. In this configuration $\text{SanityCheck}$ handles the steering values produced by the horizontal controller before checking the values for the vertical axis. The indicated cycle in Figure 4-14 does not cause a deadlock condition, since this closed path is not uniformly oriented. For verifying deadlock-freedom the following formal specification is used.

```plaintext
datatype Double = ld_val | output_val | corr_val | vOut_val | hOut_val

channel joyV : Double
channel joyH : Double
channel dacH : Double
channel ver_san : Double
channel encH : Double
channel hor_ver : Double
channel dacV : Double
channel encV : Double
channel hor_san : Double

ParServo = JoyV || joyV || (Vertical ||| encV, hor_ver, ver_san ||| (EncV ||| (Horizontal ||| encH, joyH, hor_san ||| (EncH ||| (JoyH ||| (SanityCheck ||| dacH, dacV ||| (DACH ||| DACV))))))))

EncH = encH!ld_val -> EncH
JoyH = joyH!ld_val -> JoyH
EncV = encV!ld_val -> EncV
JoyV = joyV!ld_val -> JoyV
DACH = dacH!ldVar -> DACH
DACV = dacV!ldVar -> DACV

Horizontal = encH!position -> joyH!in -> hor_san!output_val -> hor_ver!corr_val -> Horizontal
Vertical = encV!position -> joyV!in -> hor_ver!corr -> ver_san!output_val -> Vertical

SanityCheck = Seq3 ; SanityCheck
Seq3 = Seq4 ; Seq5
Seq4 = hor_san?h -> dacH!hOut_val -> SKIP
Seq5 = ver_san?v -> dacV!vOut_val -> SKIP
```
Figure 4-14 Exploded JIWy deadlock-free model
The generated CSPm script makes clear the benefit of having the tools creating the formal specification – only the network builder for this simple example takes three lines. The communication pattern is hard to read, let alone how difficult it is to manually code it the first time right. After the successful formal verification of the CSPm script, the result can be validated by generating, compiling and executing the CTC++ code. The JIWY follows the joystick commands smoothly.

The power of the formal analysis can be demonstrated by making a seemingly minor modification. Suppose that for some reason, e.g. during the software maintenance, the order in which SanityCheck treats the axes is altered (Figure 4-15) – first the steering of the vertical axis is checked, and then of the horizontal. By bringing all processes to the same hierarchical level the closed oriented path can be relatively simply revealed. However, for complex systems this visual approach can be intractable, since the deadlocked loop could spread over many processes and different hierarchical levels. The conflicting statements in the script are shown in the following snippet.

```


SanityCheck = Seq5 ; SanityCheck
Seq3 = Seq5 ; Seq4
Seq4 = hor_san?h -> dacH!hOut_val -> SKIP
Seq5 = ver_san?v -> dacV!vOut_val -> SKIP
```

The process Horizontal attempts writing to the channel hor_san before writing to the hor_ver channel. In order to have Vertical running, the channel hor_ver has to be read prior to writing to channel ver_san. The problem is that SanityCheck expects to read first the ver_san channel, and than hor_san. Therefore the three processes are mutually deadlocked.

One may try to find a solution to this deadlock condition in a way similar to that taken in the previous example. However, it may appear to be more complicated in practical problems, where the order of inputting and consequently taking variable values in calculation is important for the calculation correctness. On the other hand, turning back to the old calculation order in SanityCheck is pointless, since it is assumed that modification of the original configuration has been done for a reason.

Deadlock-free design patterns for certain classes of CSP networks have been studied in detail (Welch, 1987; Martin, 1996). Under certain circumstances, it has been proved that having all inputs and all outputs in parallel guarantees deadlock freedom. Welch (1987) postulates two deadlock-free design patterns, called I/O-PAR and I/O-SEQ.
Figure 4-15 Exploded JIYW model with indicated deadlock cycle
The first pattern forces all inputs and outputs to be accomplished before the calculation of the next instance is performed. The I/O-SEQ allows the inputs to be all performed in parallel before the calculation, and than all outputs also in parallel afterwards. Because in control algorithms all inputs must be sampled before the steering values output (calculating the steering signals usually comes in between, except in some jitter-elimination control schemes), the I/O-SEQ pattern is applicable (Figure 4-16).

4.5 Conclusions

The two main functionalities of the gCSP tool are graphical modelling of CSP-based designs and transformation of the graphical models to machine-readable forms via automatic code generation. The tool’s code generation features address two principal aims:

1. Formal checking of the CSP/CT graphical models by CSPm specification scripts,
2. Implementation of the CSP/CT models by source code compliant with the CTC++ library (or the KRoC compiler).

An important property is that both outputs are created from one graphical model without any modifications. The next item summarizes the procedure.

4.5.1 Summary of the design trajectory for generating formally verified CSP/CT software

The gCSP CASE tool establishes a chain between application-specific tools (like 20-sim for design and implementation of control laws) and CSP tools operating upon the standard CSPm specifications (as FDR and ProBE). The design trajectory is briefly summarized in five steps:

1. Creating a dataflow diagram of the overall software architecture – starting with a kind of context diagram of the software functionality that captures main responsibilities of the prospective software components in CSP/CT processes.
2. Refinement of the processes, possibly in different ways. (If 20-sim is used, those processes that implement dynamical models should be worked out first, by using the 20-sim code blocks defined in gCSP). Concurrently with the development of the domain-specific parts of a design, the communication and compositional infrastructure around those parts can be specified.
3. Generating CSPm formal specification and checking against interesting properties. (The 20-sim code blocks are transparent for performing formal checks on the model). If no errors are found, the trajectory continues with step 5.
Figure 4-16 I/O-SEQ pattern applied to both controller processes
4. In case FDR discovers errors in performed checks, the user has to interpret the failure report. In this chapter the interpretation for certain classes of system has been illustrated. After locating the cause of failure, the user has to modify the design as in step 2.

5. The model with checked all properties of interest is ready for transformation into compileable source code. After compilation and linking, the executable code can be tested on the target platform.

The main contribution of the methodology and the tool development is spanning the conceptual distance between graphical/dynamical models for physical systems and concurrent software architecture on one side and the formal specification of the software architecture at the other side. This is performed by mapping block diagrams from 20-sim into CSP diagrams and further the CSP diagrams into the selected subset of the CSPm language. This mapping allows exploitation of the high quality formal checking procedure performed by FDR.

Automatic code generation of the complete CTC++ source eliminates manual coding of the graphical/dynamical models towards the implementation language (CTC++ in this case). The resemblance of the software structure and the originating graphical model features efficient code review and alleviates the maintainability of the implemented software solution. gCSP models are self-contained, in the sense that they contain a complete specification of the compileable deliverables, including the low-level algorithms. The described mechanism of including arbitrary processing algorithms provides an environment for giving sequential legacy code a process-oriented concurrent framework. The presented examples show that blunt parallelization of sequential code may be dangerous. It is comparable with "objectification" of a non-object-oriented code: it can make it less understandable without gaining anything. With parallelization, it can be even worse: the code may stop working, i.e. it can easily deadlock! Therefore, along applying the mechanistics, integrating a system in a robust concurrent ensemble has to be guided by a methodological rigor, where certain design patterns (as the illustrated I/O-SEQ) play the main role.

Referring to the error classification, it can be concluded that automatic generation of source code represents an important procedure for eliminating development errors due to manual coding. On the other hand, formal checking has a broader spectrum of error prevention. It helps discovering and eliminating design errors in the system architecture itself; the source and cause of this kind of errors is in principle unknown, and in that sense the formal methods are a powerful tool for coping with unanticipated errors. The errors are usually reoccurring in each system run (solid errors), but also can effect the system in less easy reproducible scenarios (intermittent errors). The subsequent chapters elaborate other measures for complementing the error coverage in the CSP/CT software.
4.5.2 Directions for further development

The methods of automated obtaining verified and understandable real-time concurrent software are recognized as necessary with the increasing complexity of embedded software of today and tomorrow. Following steps in improving the qualities of the CSP/CT design paradigm illustrated in this chapter would aim at:

- Experimenting with formal analysis of other gCSP model properties on basis of prototyped CSPm code generation which are already supported by FDR: livelocks, determinism, specification refinements. The result of this kind of experiment would lead to further refinements of the CSPm code generation and wider applicability of the existing result in formal analysis of CSP-based systems.

- Supporting also code generation for CTC and CTJ libraries. Or even more general: using the principle of code generation templates from 20-sim to support various targets – any reasonable implementation platform.

- “One process – one class – one object” restriction referred to in section 4.3.1 pertains to a lack of support for managing refinements of process functionalities through class inheritance relations. For each process defined in a gCSP model, a C++ class is created, derived from the CTC++ Process class. The current implementation instantiates one object for every process, which means that from a custom process class just one object gets instantiated. This is also a reusability problem: processes with the same names and slight modification overwrite each other, depending on the order the tool generates the corresponding classes. The idea that gCSP processes with the same name all refer to one class and hold only specificities for particular process-objects has been already investigated and cued for implementation.

- The continuity in the tool link between 20-sim and gCSP has been emphasized in a few aspects. The gCSP models inherit topology from 20-sim block diagrams; the dynamical models in form of the C/C++ code generated from 20-sim are included in the CTC++ concurrent programs generated by gCSP. However, in this tool link some stages are performed by user interventions. By a closer integration (import) of 20-sim models within gCSP the tool chain can be strengthened. To this end, a review and comparison of structures of the 20-sim and gCSP models and modelling paradigms would be a first step to take.
Exception handling is considered “the most powerful software fault tolerance mechanism” (Romanovsky, 2001). An exception handling mechanism (EHM) is an indispensable feature in the design of languages for programming reliable systems. In their seminal book on fault tolerance, Anderson and Lee (1981, p.77) state that “exceptions and facilities for exception handling form the basis of the framework suggested in this book for the implementation of fault tolerance.”

Practical results during the research history of thirty years appeared as sophisticated EHMs in modern mainstream languages for programming mission- and live-critical systems, like C++, Java and Ada. Not without irony, still the most popular language for embedded applications, C, lacks an incorporated EHM; therefore, usefulness of that language “in the structured programming of reliable systems is clearly limited by such an omission” (Burns and Wellings, 2001, p.164). C caters for error handling facilities in conjunction with an underlying (preferably a POSIX-based) operating system —thus in a platform dependent way—while EHMs of C++, Java and Ada contain disparate language-specific peculiarities, although with a lot of similarities.

The EHM conceived in (Hilderink, 2005a) and worked out in this chapter is lifted to a methodological level of designing process-oriented software, specified independently of any language that may be used for implementing CSP principles. History, related work and taxonomy of
Exception handling mechanism are subject of section 5.1. Section 5.2 describes the way the EHM proposed for the CSP/CT framework is supported by the gCSP tool and the CTC++ library. Use of the mechanism is in this chapter demonstrated on a number of examples (in section 5.3) and on the JIWY case study (in section 5.4). Uses in other contexts and examples and on the other set-up are illustrated in the next chapter. The results of this chapter are discussed and concluded in section 5.5. A compact version of this chapter is provided in (Jovanovic et al., 2006b).

5.1 Exception Handling Mechanisms

An EHM allows system architects to distribute dedicated corrective or alternative code components at places within software compositions that maximize effectiveness of error recovery.

The quality of reliable software equally depends on its “normal” mode of operation as on its provisions to cope with “abnormal” conditions that may arise during software execution. In the remainder of this thesis the ordinary mode, called also normal (Xu et al., 2000; Buhr and Mok, 2000), common case (Zilles et al., 1999) or error-free (Burns and Wellings, 2001), will be referred to as nominal, following (Vardanega, 2003); the flow of control upon an exception occurrence will be named exceptional. In safe and highly available systems quality requirements for handling exceptional situations are often more stringent than for the nominal parts of the code. In this kind of systems the amount of code for detecting and handling exceptions may surpass in factors the nominal operation code. According to Cristian (1995, p.81/82), “in operational computer software systems often more than two thirds of the code is devoted to detecting and handling exceptions”.

Principles of EHM are based on provision of separate code segments or components to which the execution flow is transferred upon an error occurring in nominal execution. An exception represents an error abstraction. Code segments or components that attempt error recovery (exception handling) are called exception handlers. The main virtue of this structure for treating exceptional conditions in software execution is a clear separation between nominal program flow and parts of software dedicated to correcting errors.

5.1.1 EHM history, state-of-the-art and state-of-the-practice overview

The beginning of the research into handling exceptional occurrences in software is marked by the seminal paper of Goodenough (1975). This paper deals with capturing issues of software exceptions (calling them exception conditions), exception declaration, propagation and handling, some EHM requirements and giving an overview of the techniques used for exception conditions handling by that time. In the decade that followed the research
5 Exception handling mechanism for CSP/CT software

was focused mainly on refining EHM models included in today's mainstream languages.

With a growing demand of data processing parallelization—for boosting computational throughput and facilitating distribution—research of exception handling in concurrent architectures has been established in mid 1980s (Jalote and Campbell, 1984; Campbell and Randell, 1986; Jalote and Campbell, 1986). Because of alterations of a program's execution flow due to exceptional operations, EHM models additionally complicate understanding of concurrent software. In (Campbell and Randell, 1986) issues of exception handling in sequential systems are contrasted with those in concurrent systems, especially the problems of concurrently raised exceptions resolution and simultaneous error recovery (see section 2.2.3). To cope with this phenomenon, in that study the concept of exception hierarchy is introduced. Since then the leading group in the field at the University of Newcastle upon Tyne headed by Brian Randell (also a pioneer in structured system-wide fault tolerance concepts (Randell, 1975)), proposes a series of improvements and modifications. Later the problem of handling concurrently raised exceptions is proposed in the form of coordinated actions (“CA” actions), still leaving unresolved some implementation-related issues (Xu et al., 2000); this paper, together with a selection of eight other the most relevant articles in the field formed the recent major overview: a special IEEE issue in 2000 (Perry et al., 2000), followed by the retrospective in (Romanovsky et al., 2001).

Since the hardest problem is posed by analysis of execution flow in concurrent software with exceptions, it is worth mentioning a few studies giving mathematical (formal) semantics of EHM. A comprehensive overview of the EHM terminology being underpinned by set theory provided in (Cristian, 1995); before that, formal algebraic semantics (interestingly: all by using CSP) appeared in (Dix, 1983; Jalote and Campbell, 1984; Banatre and Issarny, 1992). Related to that, in the 1980s CSP has been recognized also suitable for formal description (and prospectively formal analysis) of exception handling used in atomic actions (Jalote and Campbell, 1986), although criticized as weak for addressing practical implementation issues (Romanovsky, 1997). A CSP algebraic treatment of the EHM prototyped in this thesis is proposed in (Hilderink, 2005a).
Despite the favourable properties in articulating error handling and the fact that EHM s are the only highly structured fault tolerance concept directly supported at the level of languages, they are not so readily used in mission- or life-critical systems as it may be expected. Lack of tractable methods for testing or, even more desired, formal verification of programs with exception handling is to be blamed for the hesitant use of this powerful concept. As clearly stated in (Cristian, 1995), “since exceptions are expected to occur rarely, the exception handling code of a system is in general the least documented, tested, and understood part. Most of the design faults existing in a system seem to be located in the code that handles exceptional situations.” In this publication it is reported that about 65% of the system failures are due to design faults in exception handling and recovery algorithms. One of the most (in)famous space-mission catastrophes for which the embedded software is blamed, the Ariane 5 crash (Lions, 1996), was due to an unhandled overflow exception in Ada. The contact with industrial partners of this project, (Design Tools project, 2001-2005a), confirmed that for mission-critical applications the industry refrains from using exception handling facilities because of exactly the same reasons reported ten years ago.

It has already been indicated that modern concurrent languages, although providing instruments for concurrent programming (Java, Ada) or even giving certain support for process-orientation (Ada), must be complemented with a proper design concept for engineering complex well-understood concurrent architectures. By looking at the concurrent-specific exception handling facilities of these languages, it can be stated that the languages’ support is even weaker (Xu et al., 2000). “Ada has very elaborate features for handling concurrency, but exception handling is basically sequential” (Romanovsky and Kienzle, 2001). Although Ada supports propagating an exception raised during rendezvous communication (the exception is delivered to both the producer and the consumer participating in rendezvous), exceptions cannot be passed out of the class handlers. However, Ada does allow exception handlers to be called in several concurrent tasks when an exception has been raised in one of them. Java inherited a mature EHM from C++ – a sequential language. The only thing where Java’s EHM shows some intention to help drawing attention on an exceptions that causes termination of one thread to the other threads in the same thread group is leaving the exception to be considered at the level of the group (if threads were grouped together). It turns out however that this mechanism alone could not address a concerted exception handling among the threads in a program. The semantics of handling concurrently raised exceptions and simultaneous error recovery in multiple parallel processes, as well as concurrency among exception handlers need to be provided by a higher concurrent (EHM-aware) abstraction layer, preferably language-independent.

Nevertheless, exception handling facilities are so needed that their abandonment is out of question. Only their further evolution is to be expected. Existing EHM s provide the ultimate error recovery mechanism in sequential programming. They are also essentially important for implementing atomic actions, the principal architectural pattern for dynamic error recovery in concurrent systems (Randell, 1975; Campbell and Randell, 1986; Romanovsky, 1997; Burns, 1999; Xu et al., 2000).
5.1.2 EHM terminology and properties

Although considered at the first place as the most powerful software fault tolerance mechanism (Campbell and Randell, 1986; Romanovsky, 2001), an EHM is also a safety mechanism. This chapter illustrates its both natures. The developed EHM is demonstrated as a:

- **Fault tolerance mechanism**, because after receiving exception(s), an exception handler provides continuation of a program execution by attempting the same algorithm (after the damage diagnosis and treatment) or by deploying alternative algorithm(s) – see Example 3 in 5.3 and Example 2 in section 5.4.
- **Safety mechanism** because, upon an exception occurrence, it can be used to establish an emergency operation mode or to drive the system to a safe state (as in Example 1 in section 5.4).

There is a number of properties that determine position of a particular exception handling mechanism in a general EHM taxonomy.

**Dynamic redundancy**
EHM is the most used dynamic redundancy approach to fault tolerance: redundant (handling) code executes only when an error is detected. In the EHM terminology, an exception represents occurrence of an fault, while detection of the error causes throwing (raising, signalling) the exception. Error recovery is done by exception handling, which starts with catching the exception (in common EHMs catching and handling are used as synonyms).

**Forward and backward error recovery**
EHM is by definition a forward error recovery mechanism, since upon an error occurrence the flow of execution is transferred to exception handlers that, based on the type of exception, attempt to remedy the faulty state of the system and continue providing services. As a clean mechanism for substituting unsuccessful executions with alternatives, the mechanism is suitable for implementing backward error recovery as well (Cristian, 1982) – a simple scheme for recovery blocks based on exceptions is given in (Campbell and Randell, 1986).

As being inherently a forward error recovery mechanism, its effectiveness strongly depends on strict identification of the error occurred. That means that an EHM effectively deals with anticipated errors only. For each error anticipated in a system, proper handlers have to be prepared. Exception handlers implement error correction (or mitigating) procedures that are specific for certain anticipated faulty states. As discussed in Chapter 2, the assumption of anticipating all possible errors that may occur in a system represents the main disadvantage of this technique. For a study on low success of anticipating possible exceptions in even simple programs see (Maxion and Olszewski, 2000).
Flow of execution – handling models

Depending on the flow of execution between the nominal and exceptional operation of software (in presence of an exception), the so-called handling models (Buhr and Mok, 2000; Burns and Wellings, 2001) can be divided in five groups: termination, resumption, hybrid, retrying and nonlocal transfer. The first two gained initially equal attention, but practice made the termination model prevail (Cristian, 1995) – it is adopted in the most used languages, as C++, Java and Ada. In all these languages the context (a code block or function) in which an exception has been raised does not continue its execution after the exception is handled (what would be the case in the resumption model). This means that it is a responsibility of the handler to take care of the service to be completed like it would have happened if the nominal sequence had not raised the exception. Thus, the termination model does not mean that upon an exception occurrence the application terminates – only the initial flow of control gets terminated without a possibility to be resumed.

Exception propagation

After being thrown, an exception propagates to the place it can be eventually caught (and handled). A crucial mechanism of an exception handling facility is its propagation mechanism, which determines how to find a proper exception handler for the type of exception that has been thrown. Exception propagation always follows a hierarchical path, and in the concrete languages different choices are made (Buhr and Mok, 2000; Xu et al., 2000): dynamically along the function call chain or object creation chain or statically along the lexical hierarchy (Knudsen, 1984). The exception propagation mechanism is crucial in understanding the execution flow in presence of exceptions and its complexity directly influences acceptance of the concept in practice.

Scope of a handler

Depending on the mechanism of exception propagation, a handler may be eligible for handling exceptions from various parts of the program code. The scope of a handler can be defined as the totality of these parts. If it is possible to unambiguously determine the origin of each exception raised in a program, the scope of a handler is determined by the set of all exceptions it is prepared to handle.

Default handlers

When along a propagation path of an exception of a certain type no handler is found that is prepared to handle that type, the program terminates abruptly. The exception is than called unhandled. This situation can be mitigated by default (universal) exception handlers (Goodenough, 1975; Cristian, 1995) that may catch any type of exception; however, default exception handlers usually cannot do any better than indicating the occurrence of an unhandled exception by issuing proper warnings to the environment (Campbell and Randell, 1986, p.819). When an EHM is used in implementation of a backward error recovery scheme, unexpected errors are transformed into default error conditions – an exception occurrence implies failure of the
acceptance test, without specific information on the cause of the failure. In that case, a recovery block is practically a default exception handler. More frequently, default handlers help when a poorly documented third-party component is used that throws exceptions which are not declared. The concept of universal exception handlers is supported by Java and C++.

**Declaring exceptions**

Exceptions that a component may throw form a part of its behaviour and its interface. Exception lists, attached in Java (compulsorily) and C++ (optionally) to methods that can throw exceptions, are part of a method’s behavioural description. Java compilers check whether a method throws only declared exceptions (except some run-time exceptions, called unchecked), therefore Java EHM supports the consistency of the coverage against unhandled exceptions. This feature however hampers software extensibility with respect to consistent and exhaustive exception coverage, and is considered restrictive (Stroustrup, 1994; Buhr and Mok, 2000), potentially limiting reusability.

**Exception derivation**

Buhr and Mok (2000) state that exception refinement through derivation (inheritance in object-oriented languages) supports a more flexible programming style: a programmer can choose to handle an exception at different degrees of specificity along the derivation hierarchy. This means that handlers for an exception derived from a more general one can handle specificity for that exception type only; after that, the handler can raise the exception again in order to propagate it to a more general handler for the rest of the handling. Besides the flexibility and readability of the design, this feature also facilitates reusability.

### 5.1.3 EHM requirements

**General requirements**

The following list combines some general criteria for evaluating quality and completeness of an EHM. According to (Campbell and Randell, 1986; Milčev, 1995; Buhr and Mok, 2000; Burns and Wellings, 2001), a high-quality EHM should:

1. be simple to understand and use,
2. provide a clear separation of the nominal program code and the code intended for handling possible exceptions,
3. prevent an incomplete operation from continuing,
4. allow exceptions to contain all information about an error occurrence that may be useful for a proper handling, i.e. the recovery action,
5. be flexible to allow adding, changing and refining exceptions,
6. allow execution overheads of exception handling code only in the presence of exceptions – exception handling burdens on the nominal execution flow have to be minimized,
7. allow a uniform treatment of exceptions raised by the program and the execution environment,
8. impose declaring exceptions that a component may raise,
9. allow nesting exception handling facilities.

Concurrency-specific requirements

While mechanisms competent for coping with exceptional occurrences in sequential programs appear mature for standard use (at least in fulfillment of the aforementioned requirements), there is not yet a clear consensus on models for concurrent exception handling; however, a few properties that are supposed to be a minimum are notably recognizable in the literature (Campbell and Randell, 1986; Issarny, 2001). A concurrent exception handling mechanism should ensure that:

10. upon an exception occurrence in processes communicating within a parallel execution with other processes, all processes involved in the parallel execution get informed which exception has occurred,
11. all participating processes simultaneously enter the recovery activities specific for the exception occurred,
12. in case of concurrent exceptional occurrences in different parallel composed processes, a handler is chosen that treats the compound exceptional situation rather than isolated exceptions.

The difficulty with concerting error recovery in parallel composed processes is posed by the fact that an exception occurrence in one process is an asynchronous event with respect to other processes. The most efficient mechanism for notifying all participants in a parallel composition with an emergency situation would be enforcing exceptional termination of all parallel composed processes (similar to Ada ATC – Asynchronous Transfer of Control or asynchronous notification in Real-time Java). However, this solution would interfere with the scheduling/dispatching mechanism in the core of CT. Moreover, a higher risk of corrupting processes’ states by an asynchronous abortion would put more responsibility on the programmer (therefore in Ada Ravenscar Profile, (Burns, 1999), for high-integrity systems ATC is disabled).

For CSP architectures an indirect mechanism, called graceful termination (Welch, 1989), is proposed. It is based on the **channel poisoning**: sending a poison along channels as a mechanism for terminating (or resetting) an occam network of processes. Processes that receive the poison spread it further via all the channels they are connected to. Eventually all processes interconnected via channels will receive the poison token and terminate. The method can be used for implementing the termination model of compositional constructs. In the CSP/CT framework this approach is slightly modified as proposed in (Hilderink, 2005a): instead of passing the poison via the channels, the idea is to poison (invalidate) the channels. A further extension of this concept, crucial for meeting the stated requirements, in the CTC++ EHM is described on pages 151 and 157.
5.2 The EHM concept within CSP/CT, libraries support and the gCSP tool coverage

An unfavourable property of the practical EHM is that they are defined as a feature of a particular language, hence having different semantics. The EHM elaborated in this chapter is to address the concurrency-specific issues and therefore it is aimed to be used at the level of processes in a process-oriented concurrent environment (contrasted to levels of statements, methods, classes, or objects (Xu et al., 2000, p.3), in the mainstream languages). The mechanism is designed to be implementable with the same semantics in different languages. It is prototyped in C++ (like an extension of the CTC++ library), with use of macros that are supported in the CTC version as well.

This mechanism distinguishes roles of exception handling processes, crucial for the exception propagation mechanism, and exception handlers, which constitute error handling code (Figure 5-1). Exception handling processes contain exception handlers coded in the gCSP code blocks. Exception handling processes, together with other process-level CSP/CT entities, like ordinary processes, compositional constructs and channels facilitate propagation of exceptions. An exception handler is a part of exception handling code prepared to handle a particular type of exception (and possibly derived exceptions). Handlers are not aware of the process-level propagation mechanism. By using the exception catching statements they receive exceptions from the process-level for handling, and by using the exception throwing statements they can rethrow partly handled or unhandled exceptions.

In the proposed framework, exception handling code is supposed to be usually found in the code blocks of exception handling processes, but a lot of it can be also found in code blocks of ordinary processes – for instance if a process encapsulates a complex algorithm that is originally developed with using some (if any) native exception handling facilities in the used implementation language. Therefore, in implementation of ordinary processes, exception handlers are useful to allow reuse of legacy code without modifying the original code. As long as the use of the native EHM of the language is confined to internal use within a process, it does not clash with the EHM on the process-level. Practically, this means that these exceptions (in Ada terminology called internal), if not converted to the process-level exception sets, must all be handled within the process. However, as a last resort, a process should submit all unhandled internal exceptions to the process-level EHM complying with the process-level exception handling mechanism.

An exception handling process is attached via the exception construct (Hilderink, 2005a, p.171)—in the CTC++ library called ExceptionCatch—to the nominal execution captured by ordinary processes/constructs. The exception handling process guards the process (construct), which is then referred to as exception-guarded process. Conceptually, the prototyped mechanism follows the CSP process-oriented paradigm: both nominal operation and exceptional operation are encapsulated in processes, the compositibility of the design is preserved by
combining these processes by a construct. This construction is conceived by Hoare already in (1973), by capturing the behaviour when a process Q2 takes over after process Q1 signals a failure as

Q1 otherwise Q2.

Hoare refines this notation in (1985) by introducing interrupt (\(\Delta_i\)) and catastrophe (\(\Xi\)) operators. Hilderink finds in (2005a, p.91) implementation of these operators prohibitive with respect to the resources allocation and proposes a simpler exception operator (\(\Delta_u\)).

Practically, use of the mechanism follows the logic of the try/throw/catch clause from C++ and Java EHMs. In this way, a concept familiar to the broad programmers' community is resembled, hoping to make the mechanism easier acceptable. The three keywords are implemented as macros at the level of the CT library, allowing experimentation with the CT C version library and yet marking the points of attention if the EHM is ported to other languages. TRY marks a guarded activity in a process or a channel, upon a detected error THROW throws an exception set, while CATCH can be found in the code of the ExceptionCatch construct. Definition of the EHM macros within the CTC++ library follow as

```
#define TRY try
#define THROW(a) throw (a)
#define CATCH(a) catch (ExceptionSet *a)
#define ENDTRY
```

ENDTRY is defined in the CTC version for a proper cleaning up. For details, see (Van Engelen, 2004).

An important concept in this mechanism with a couple of favourable implications is that of a collection of exceptions – exception set. In the prototyped mechanism exception sets are thrown, not isolated exceptions. Immediately after creation, an exception (instance of the Exception class or its derivatives) becomes an item in an ExceptionSet object; raising an exception itself has no implementation other than just an exception object creation, but is used in this text as short for “an exceptional occurrence”. Further more, for brevity, in this text exceptional occurrences will be named usually as “throwing exceptions”, bearing in mind the described automatism: handling processes receive exception sets and let the handlers handle contained single exceptions. After an exception is raised in a process or in a channel, an exception set that contains the exception is thrown by the processes and collected by the parent constructs. Also, a handling process does not catch exceptions (i.e. exception sets), it rather (only) receives and delegate them to the handlers. The exception construct catches an exception set and forwards it to the associated exception handling process.
In the graphical notation as implemented in gCSP, Figure 5-2 associates the exception handling process \texttt{ExcHandler1} to the exception-guarded process \texttt{Process1} with the exception relationship, following basically Hoare’s “otherwise” principle.

For this EHM, the termination model is adopted, being recognized as much simpler and more intuitive than the others, as already explained. After a process raises an exception (i.e. throws an exception set) it consequently terminates and the associated handling process(es) are executed. It is not possible to continue the exceptionally terminated process. For a repetitive process, however, if the exception handling is successful, the framework is flexible enough to model (and implement) restarting the guarded process, and continuing the repetitive execution (see Example 1b in section 5.3). For some possibilities to extend broader the resumption facilities see (Jovanovic et al., 2005).

The mechanism of raising exceptions, i.e. throwing exception sets in processes and channels is essentially the same. Since channels are passive objects, exception throwing happens in their \texttt{read} or \texttt{write} methods executed (called) by processes, thus in the context of processes. The exceptions are propagated identically as they are actually thrown in a process’ \texttt{run} method.

A separate class for any particular exception can be derived from the \texttt{Exception} class. Similarly to the Java/C++ EHM’s, the user of the mechanism may put in the exception object all information useful for proper handling. The channels should also define appropriate exceptions to be thrown when a channel code detects an exceptional condition. However, when suspended, i.e. poisoned (by an exception handler, with an exception – called the poisoning exception), channels throw the poisoning exception further on. In this way, an exception occurrence propagates through the network of communicating processes, so they can all take proper corrective actions by collaborative exception handling. Upon testing the state of a channel (whether it is suspended or not), a null value (meaning “not suspended”) or a copy of the poisoning exception is returned. Calling the \texttt{rehabilitate} method turns a suspended channel into the normal state. It is the responsibility of the channel programmer to take care that, upon occurrence of an exception condition, channels throw exceptions to both rendezvous parties (i.e. in both \texttt{write} and \texttt{read} methods).

The use of the described EHM is illustrated by several examples in sections 5.3 and 5.4, but first the modelling principles and some highlights on the implementation and the CSP/CT semantics are elaborated in the following three sections.
5.2.1 The gCSP tool support

Besides editing graphical models of the exception handling layer and (partial) code generation based on the level of a model's detail, the gCSP tool complements some consistency aspects of the prototyped exception handling model. In order to avoid implementing some important features in a way specific for a chosen implementation language or introducing too much generic overhead, the features of declaring exceptions or tracing unhandled exceptions are delegated to the model processing by the tool.

A simple model in Figures 5-3 and 5-4 is used to illustrate the tool's support in modelling exception handling layers in the gCSP notation.

![Figure 5-3 EHM layer in the G-editor](image1)
![Figure 5-4 EHM layer in the C-tree](image2)

Process P parallel composed with process Q is guarded by the handling process P_Handler (that is hence also part of the parallel composition Par1), while the parallel construct Par1 itself is guarded by the handler Par1_Handler.

**Network builder**

For the model in Figures 5-3 and 5-4, the tool generates a pair of .cpp and .h files for each process plus the global network builder in the gCSPmain.cpp source file and one header file with definitions of all exceptions used in the program. For this simple model the network builder is as follows:
5 Exception handling mechanism for CSP/CT software

```c
//-- CSP include files
#include "csp/lang/include/ExceptionCatch.h"
#include "csp/lang/include/ExceptionHandler.h"
#include "csp/lang/include/Parallel.h"

//-- Include process header files
#include "include/Q.h"
#include "include/P.h"
#include "include/P_Handler.h"
#include "include/Par1_Handler.h"

int main (void) {
    //--- Process Allocations
    Q *Q_1 = new Q();
    P *P_1 = new P();
    ExceptionHandler *P_Handler_1 = new P_Handler();
    ExceptionHandler *Par1_Handler_1 = new Par1_Handler();

    //--- Network Builder
    ExceptionCatch *ExC1 = new ExceptionCatch(
        P_1,
        P_Handler_1
    );

    Parallel *Par1 = new Parallel(
        Q_1,
        ExC1,
        NULL);

    ExceptionCatch *ExC2 = new ExceptionCatch(
        Par1,
        Par1_Handler_1
    );

    ExC2->run();

    //delete’s (process and construct objects deallocation...)
    return 0;
}
```

Some observations on the implementation of the exception construct in the CTC++ library are in section 5.2.2.
Exceptions declarations

One tab of a process Properties dialog serves declaring exceptions that can be raised in the process and not (completely) handled internally. The name of the exception specifies its type. An exception can be declared as a kind of already declared exception, which specifies then a derivation relationship between them. A property of an exception is also a list of arguments to be assigned in creation of the exception instance. The list of exceptions that can be raised in the process is appended with the list of exceptions that may be thrown by all channels connected to that process.

Hence, in the specification of a channel a list of exceptions that the channel can throw should be declared as well. Channels connected to an exception-guarded process may be poisoned by the exception handling process; thus, all exceptions declared in processes connected by the channel are also in the declaration of that channel.

Based on all exceptions declared in a model, the tool maintains a global exception list. Derivation (inheritance) of exceptions is reflected in the constructors of exception objects. The include section in the .h file of each process that declared exceptions is appended accordingly by including exception headers. The corresponding THROW statements in processes’ .cpp files are automatically generated (as commented code) – serving as reminders and shortcuts when coding error detection points. Some other potentials of the global exception list are discussed in section 5.5.

Refining a handling process’ scope

The tool can determine the scope of a handler, i.e. all exceptions that may end up in an exception set received by the particular handler process. However, a handler process need not support handling of all these exceptions. In a handler’s Properties list the modeller may choose which of the possible exceptions the handler intercepts. Since a handler may handle an exception partially or unsuccessfully, the intercepted exceptions may be rethrown. Those exceptions, together with unhandled and moreover newly introduced by the handler form the exception declaration of the handling process. Unhandled exceptions, if any, can be detected by the tool.

Exception handling layer

Visualization of the exception coverage in the gCSP tool is present both in the G-editor and the C-tree. In order to make this protective layer as transparent as possible, besides a distinct shape for exception handling processes, it is possible to hide all handling processes by toggling off the layer in the G-editor (the toggling button is shown in section 3.4.1 on gCSP user interface, page 99).

Another feature makes it easy to create exception-guarded constructions upon processes and constructs in an existing CSP/CT network. In order to replace a process (or construct) with an exception-protected construction, the user can choose “Add exception guarding” from a pop-up menu both in the tree and the graphical editor. This option automatically places the chosen process/construct within an exception construct and assigns an empty exception handling process.
5 Exception handling mechanism for CSP/CT software

5.2.2 Exception handling support in the CT libraries

This section presents some implementation aspects of the discussed exception handling mechanism in the CTC++ library using channel poisoning, as implemented in (Van Engelen, 2004). Some architectural details are provided in Appendix A.

Exceptions, types of exceptions and exception derivation

The Exception class is a base class for deriving exceptions anticipated in the system. The type of an exception instance (encapsulated in the class ExceptionType) carries implicit information about the cause of the exception (yet the parameters of the exception constructor may carry additional information). Exception handlers must be able to tell exception objects one from another. In order to provide the required functionality on the level of libraries for different languages, a generic type checking facility based on linked lists is prototyped. The linked list is actually used to enable an exception handler to detect (and perhaps handle) derivatives of an exception the handler is programmed for. This mechanism is used to structure gradual and reusable exception handling. Methods isOfType and isDerivedFrom of the class ExceptionType serve these purposes. About the use of the methods see on exception handling processes later.

Exception set

ExceptionSet is a class that can contain multiple Exception instances in a linked list. This collection of exceptions is introduced in the EHM because of the following reasons:

- the parallel construct collects exceptions thrown by child processes (unhandled or rethrown by local handlers) and throws them higher up,
- the alternative construct may abnormally terminate after more then one comm-guards throws exceptions. Those are collected in an exception set,
- an ensemble of exceptions can have a meaning different than a sum of separate exception occurrences (therefore an exception set allows construction of an exceptions hierarchy, see section 2.2.3 on page 46.)

Immediately after an exception creation in a process, it becomes the (only) item in an ExceptionSet object that is thrown by the process. An ExceptionSet object is caught by ExceptionCatch construct and forwarded to the exception handling process, a child of the that ExceptionCatch construct (as in composition in Figure 5-2).

Exception construct

ExceptionCatch

This construct is added to the adopted occam set of constructs to support composibility of the prototyped EHM. It is named ExceptionCatch and
composes a process and an exception handling process (Figure 3-39 and Figure 3-40 on page 84). This construct catches an ExceptionSet and forwards it to the associated handling process. Note that all CT constructs are processes, which means that a construct can be also exception-guarded (including guarding exception constructs themselves, which allows nesting exception handling facilities).

Each exception construct is generated (coded) in this prescribed pattern:

```cpp
void ExceptionConstruct::run() {
    TRY {
        process->run();
    } CATCH (exceptionSet) {
        exceptionHandler->run(exceptionSet);
    } ENDTRY;
}
```

The guarded process is run first. When it terminates normally, the ExceptionCatch construct terminates normally as well. When the normal process terminates abnormally by throwing an ExceptionSet, the ExceptionCatch construct executes the exception handler. If the handling process terminates normally, the construct terminates normally. If however the handler terminates abnormally by throwing an ExceptionSet, the construct terminates abnormally.

For a constructed instance of a process (p1) guarded by instance of an exception handling process (eH1), the exception construction in CTC++ looks like:

```cpp
anExC = new ExceptionCatch(p1, eH1) ;
```

A difference with the construction of the other CTC++ constructs is that the number of arguments is restricted to two (a process and an exception handling process), therefore a NULL terminator in the list of the constructor arguments is not necessary.

**Exception handling process**

An exception handling process is a type of CT process with a type-specific run(ExceptionSet *) method. This method is called by an exception construct (as in the previous code excerpt).

An exception handler specific for certain exceptions iterates through the received ExceptionSet, handles the exceptions and, if it did not handle all exceptions in the set, throws the set with the remaining exceptions higher up. When an exception is handled, it is removed from the exception set and deleted by calling isHandled method, which returns the next exception from
the set. If handling of an exception is not supported, next method is called, to get the next exception in the set.

Using the isDerivedFrom method, a handler checks if the exception at hand is derived from a type that it can handle. To execute handling code written exactly for a specific exception, a selection that further examines the type of the exception using isDerivedFrom (for subtypes) and isOfType (for a specific type) can be nested. In this way an exception can be handled gradually. (For gradual handling by chaining exception handlers see Example 3a in section 5.3). A handler may handle an exception in a general way and then leave the exception in the set for further handling by the higher exception handler by omitting invocation of the isHandled method and calling the next method instead.

An exception handler, guarding one of communicating parallel processes, can detect the so-called concurrent exception occurrences (see section 2.2.3). This means that in a parallel system more than one exception has been raised before the first has been handled. In these situations handling exceptions one-by-one may be wrong. The first action that a handler in a parallel CT network does is suspending with the raised exception all channels connected to its guarded process. Actually, before suspending channels, each exception handler checks whether the channels are already suspended, and with which type of exception. If the types of exceptions are different, this means simultaneous exception occurrences. The handlers may synchronize before they start handling exceptions (by having channels or barriers among them); or in case of the simultaneous exceptions, rather than trying to handle concurrent exceptions, handlers may throw higher up sets with all different exceptions collected from all channels. The higher exception handler catches (overlapping) exception sets from all handlers and it is able to reconstruct the exception hierarchy. This is crucially important when programming error recovery in atomic actions (see Appendix B Atomic actions).

**Exception propagation and default exception handler**

From the moment an exception is created and placed in the exception set that is thrown by a process, it propagates through (parent) processes and constructs to the first ExceptionCatch construct in the compositional hierarchy. The exception handler contained by the ExceptionCatch construct may handle all exceptions from the exception set, or just a subset, and rethrow the rest. Also, it may handle some exceptions partly or create new exceptions. Thus, an exception propagates through the compositional hierarchy until a proper handler is found. The way of propagation is visualized in the compositional tree of the gCSP tool.

Exceptions also may propagate along the communication network, according to the channel poisoning wave. It may look that an exception wave may uncontrolled spread over multiple compositional levels and eventually over a whole network. However, if the nature of an exception is such that all surrounding processes, belonging to the same or another construct, are unable to safely and reliably continue their execution, putting the whole application into an exceptional mode of operation actually is wanted. If this is
not desired, the designer should take care of partitioning the system properly to confine the error – EHM is a tool for making this easier, but does not itself solve the issues of architecting high integrity and availability systems (a more conservative view within the same framework is presented in (Jovanovic et al., 2005)).

In conclusion, the propagation mechanism is simple, since it follows the visual compositional and communication structure of the CT concurrent design. In the mainstream languages an exception propagates backward along the chain of function invocations, thus dynamically. In the developed EHM an unhandled exception has to be explicitly rethrown in order to let it propagate higher up along the compositional hierarchy. Finding a proper handler in this mechanism is static, which has an advantage with respect to the real-time behaviour of the software: Buhr and Mok (2000, p.833) state that “less dynamic choice of a handler better suits a real-time environment”).

In this EHM, a default exception handler would be a handler associated with the top level construct in the network builder. It should be prepared to handle any exception in the system, i.e. objects of the Exception class. This actually means that the top level construct in a fault-tolerant CT program is always an ExceptionCatch construct whose exception handler handles all exceptions that may have propagated to the top from any part of the program. This handling should probably be in a generic way, but it is a means of preventing abrupt abortion of a possibly safety-critical execution.

**5.2.3 Abnormal (exceptional) termination of the CT constructs**

Hilderink (2005a, p.91) advocates that the introduced exception construct (operator) does not affect the semantics of the original CSP operators. However, it influences a CT program’s execution flow, ruled by the CSP/CT constructs. This section discusses the behaviour of the constructs in presence of exceptions and the necessary augmentations of the constructs’ implementation.

**Sequential construct**

If a process in a sequential composition terminates abnormally by raising an exception, consequently the sequential construct terminates abnormally without running any more processes.

The sequential construct is implemented as a thread in which its child processes are executed. When one of the sequential processes throws an exception set, the thread terminates, hence the sequential construct, and the exception set propagates to the closest exception construct in the compositional hierarchy. Therefore, supporting exception handling on the level of the CT libraries requires no change to the already existing sequential construct.
Alternative (and Prialternative) construct
The alternative construct makes a choice to run one of its child processes based on the readiness for rendezvous on the channels that are connected to the child processes. The observation of the parties engaging in rendezvous on the other side of channels is responsibility of the comm-guard objects.

Occurrences of exceptions in an alternative execution may happen in any phase of the following:
1. Initial checking of the guards. The guards can signal problems to the alternative construct. If a problem was found, the alternative terminates abnormally.
2. Waiting for one or more guards to become ready or signal a problem. If a problem was found, the alternative construct terminates abnormally.
3. Running the associated process. If the process terminates abnormally, the alternative construct will terminate abnormally. If a process terminates regularly, so does the construct.

When the alternative construct has checked all the guards (in phase 1), it checks if an ExceptionSet is created by the guards. In that case, it throws the ExceptionSet (which is a collection of Exceptions generated by guards or channels) higher up and terminates. If no exceptions occurred, the construct waits until one guard becomes ready. During this period, a channel can also raise an Exception and wake up the alternative construct. When the alternative construct wakes up, it checks again if its ExceptionSet is empty. If this is not the case, it throws the ExceptionSet higher up. Upon throwing an exception from one of the child processes, this construct behaves as the sequential construct.

For saving memory and execution time, an optimization is applied to ensure that the alternative construct does not contain an ExceptionSet when it is just created. Instead, it takes the first thrown ExceptionSet as its own.

Parallel (and Priparallel) construct
The behaviour of the parallel construct in nominal operation is that it terminates when all its child processes have terminated. This behaviour, as it is implemented in the described EHM, stays the same under abnormal termination of any of its child processes: even if some of the processes terminate abnormally, the parallel execution is not terminated until all other processes terminate (regularly or abnormally). After termination of all processes, the parallel construct collects possibly thrown ExceptionSets by child processes and places their exceptions in one ExceptionSet. If this ExceptionSet is empty when all child processes have terminated (so there were no exceptions raised by the child processes) the parallel construct terminates regularly.

This semantics of the parallel construct termination poses two problems: if the exception handler of the process that raised an exception cannot cope with it (or a handler does not exist at all), the exception cannot
be propagated to a higher handler until the parallel construct terminates, i.e. all other processes terminate. Moreover, if processes synchronize with an exceptionally terminated process, they get blocked on channels, and they will never terminate! The concept of channel suspension is used to ameliorate this problem. This solution is discussed in Example 2 in section 5.3.

For performance reasons, as with the alternative construct, the parallel construct contains no ExceptionSet when it starts. If an exception set is thrown, that one is used by the construct further on.

5.3 Use of the EHM facilities

The best time to start debugging a program is before the first bug is discovered. With an imperfection-aware mindset, along the process of modelling a system’s functionality the designer may anticipate a great deal of possible problems. Deferring the safety and fault tolerance analysis for later stages of design increases costs of extending the coverage against undesired conditions. The testing phase will in any case reveal a lot of residual design errors, regardless how thoroughly fore precautions were taken. The already mentioned study of Maxion and Olszewski (2000) demonstrated the difficulty of foreseeing all possible causes of errors in a system, even on the scale of one source file program.

For providing a comprehensive coverage against system failures, the designer should deploy a systematic safety and integrity analysis procedure, as elaborately prescribed in the literature, (Leveson, 1995) for example. For different groups of errors the EHM facilities can be used with greater or less success, directly or with additional software and hardware components.

A good deal of software (internal) faults is amenable to precise identification and relatively effective treatment (correction, alternative algorithm(s) or at least precise signalling). The examples are violating array boundaries, dereferencing a null pointer and so on. Treatment of software faults caused by computer failures is platform dependent, consequently the effectiveness of the error recovery as well (an observation is given in bullet 6 on page 175). The examples of these external faults are memory failure, file system corruption or a communication link failure. With respect to errors in the environment of an embedded computer system, for all those errors that do not manifest themselves uniquely in mapped software components, often additional hardware components have to be introduced in the system.

After a thorough analysis and localization of errors planned for the forward error recovery by the EHM facilities, CT processes and constructs that potentially raise exceptions are replaced with the exception-guarded augmentations. Each process becomes a child of an exception construct and is associated with a corresponding exception handling process. As already indicated in the description of the gCSP and CT support, potentially exception throwing code is embraced with a TRY block and extended with THROW statements, handlers are structured by the exception catching statements for receiving the anticipated exceptions.
Examples in this section illustrate the use of prototyped EHM for treating software errors. Use of the EHM for handling some external errors is illustrated in the next section on the JIWY robotic case study.

Examples 1 a) and b) show the basic Hoare’s “otherwise” behaviour. Example 2 a) highlights the recognized deadlock-like condition in parallel composition with exceptions. Example 2 b) illustrates addressing this and other imposed concurrency-specific EHM requirements. Examples 3 a) and b) demonstrate use of the mechanism for flexible fault tolerance.

**Example 1.** The following two models are minimal extensions of the basic CT exception handling construction from page 84. In both variants process Division divides a constant by a number read from the keyboard. If a zero is read, it throws an exception.

1a) Process1 from Figure 3-39 (page 84) is replaced by a sequence of the Division process and the primitive repeater (•-process) – see Figure 5-5a. This sequence is in principle repeated infinitely, since the repetition condition for the •-process is true. This means that the process Division reads the denominator from the keyboard until it is a zero. Figure 5-5b shows the compositional hierarchy of this elementary example. DivisionHandler guards the sequential construct composed of the process Division and the •-process.

When Division throws an exception it gets terminated and the intrinsic sequential construct as well. After execution of DivisionHandler, this program terminates.

![Diagram 5-5a](image1)

**Figure 5-5 Basic use of the EHM behaviour**

1b) A modification as depicted in Figure 5-6 allows retrying the process operation after an exceptional occurrence and (successful) handling. That is achieved by repeating the exception relationship by the •-process (note that the bubble indicates different grouping than in the previous example and that repeating is not unconditional any more – the initial value of the variable repeat is true, but can be changed by the exception handling process as indicated by the var-channel). The different execution pattern is apparent from the C-tree in Figure 5-6b. If handling of an exception is successful, this scheme permits retrying the operation in which the exception has occurred.
After termination of the handling process (which does not modify the \texttt{repeat} variable in case of a successful exception handling), the composition is started again. However, in the case of an unsuccessful handling, the handler may stop the program by turning the variable \texttt{repeat} to \texttt{false} or (re)throwing the exception or both.

Example 2. Throwing an exception in one of communicating parallel composed processes may lead to a deadlock condition. This may happen in two different ways (both are illustrated by Example 2a). The first is a programming omission: a rendezvous channel does not throw an exception to both communicating parties (\texttt{Keyboard} and \texttt{Division} in Figure 5-7). But the other one is more serious: a process that is exceptionally terminated before a rendezvous will keep the other party blocked on the channel forever. The solution to this problem is the channel suspension (poisoning) mechanism (elaborated in 2b).

2a) If reading the keyboard and the division by the read value is separated in two processes, it is natural to compose them in parallel and they have to communicate the read value over a channel (Figure 5-7).

The guarding against division by zero can be achieved either in the channel or in the processes. The channel may be programmed not to allow writing a
zero value to it. In the write method in that case an exception is raised (thrown when Keyboard attempts to write a zero to the channel). If the semantics of the rendezvous channels is correctly programmed, the exception is thrown to the both sides of the channel. Thus, when Division attempts reading from the channel, the same exception is thrown. If this is not the case, Division gets blocked on the channel forever!

A similar deadlock-like condition will inevitably happen if detection of a zero denominator is a responsibility of the communicating processes. Regardless which process exceptionally terminates by throwing an exception, the other process will eventually try to read or write to/from the channel and get blocked forever – the other party has terminated!

In order to eliminate this anomaly, handlers of parallel composed processes at the beginning of their execution should suspend all channels connected to the process that has raised the exception. In that way all parallel processes get notified about an exception occurrence and terminate by throwing the same exception. If a process is blocked on a channel that gets suspended, the channel wakes up the process and it terminates immediately by throwing the poisoning exception. This is however a favourable scenario: a major downside of this EHM is the situation when a channel gets suspended by one party in the rendezvous, while the other is not blocked on that channel, but engaged in a calculation or waiting on another channel – the termination of the other party is not immediate, but delayed. For more on this problem see bullet 11 on page 176.

2b) The mechanism of channel suspension is illustrated on the Commstime benchmarking example (Welch, 1996).

![Commstime example](image1)

![Channels suspension chain](image2)

The benchmark consists of four communicating processes composed in parallel (Figure 5-8). The processes cyclically communicate integer numbers
over the channels. Prefix initiates the first cycle of communication by outputting a zero (otherwise it just forwards the received value from Increment), Delta distributes the number written by Prefix to the Increment and TimeMeasurement processes. Increment increments the number obtained from Delta. After the circulated value reaches a certain limit, the processes terminate. The limit value is related to the number of channel activations and context switches, which is used by the process TimeMeasurement to benchmark different implementation platforms.

If the TimeMeasurement process signals an exception, MeasurementHandler suspends the channel coming to the TimeMeasurement process with the exception that has been thrown (the act of poisoning is marked “1” on Figure 5-9). On the next writing attempt to that channel by Delta, the channel throws the exception it is suspended with, so DeltaHandler gets informed on the thrown exception and suspends all channels connected to Delta (“2”) – for the mechanism’s simplicity, handlers suspend already suspended channels too. It depends on the scheduling if the same happens first with the Prefix(Handler) or Increment(Handler) – “3/4”. In this way all processes terminate and the handlers get control. After synchronization (the synchronization primitives—channels or barriers—are not depicted for the sake of simplicity), they all start handling the same exception.

Example 3. Two models in this example demonstrate general fault-tolerant schemes in CSP/CT. In Example 1b) a limited form of a fault-tolerant execution scheme was presented: after an exception occurrence the operation may be reattempted, but only with the same algorithm. In most dependability approaches that claim fault tolerance, one or more alternative algorithms (or more generally, resources) have to be activated.

3a) The composition in Figures 5-10 and 5-11 consists of an exception construct (ExC2) that contains another exception construct (ExC1).

If a primary algorithm (here represented as the process Prime) fails with raising an exception, the handler ExcHandling1 takes over. Besides confining the error (damage), it may implement the operation alternative to Prime. However, if that attempt is not successful, other alternatives can be
implemented by chaining handlers (in this example ExcHandling2 may do one more attempt).

Besides outlining a framework for a fault-tolerant execution scheme, this composition also captures some general EHM issues:

1. **Finding a proper exception handler**
   ExcC2 in this example is the closest higher exception construct that intercepts exceptions thrown by the handling process ExcHandling1. In a general case, construct Exc1 could be nested deeper in the compositional hierarchy under ExcC2. The level of ExcC2 could be a more suitable context for assessing and recovering from some errors detected in process Prime. (Re)raising exceptions in a handler allows using the mechanism of propagating exceptions along the compositional hierarchy for handling errors at the level that provides maximal effectiveness. Leaving unhandled some exception in an exception set will automatically cause their propagation to the place in the hierarchy where the effect of error recovery is the most effective.

2. **Partial (gradual) handling of an exception**
   In systems with an extensive exception handling support it is common to have a complex inheritance hierarchy of exceptions. Many exceptional situations can be handled in a common way after its specificities are solved. Therefore handling of an exception can be done in a few abstraction levels. A context-specific part is solved by the nearest exception handler which then rethrows the exception higher up, where it can be treated in a more general way.

3. **Handling exceptions raised in handlers (nested exception handlers)**
   As any other part of a program, an exception handler can also end up with an exception occurrence; especially if it is taken into account that exception handlers in principle try to remedy an irregular situation. Therefore, an EHM should be used consistently in all risk-prone software parts, hence in the handlers as well. Thus, composition from Figures 5-10 and 5-11 may model nesting of exception handling facilities: ExcHandler2 as being the handling process for ExcHandler1.

3b) One of the favourable properties of EHMs is their applicability in backward error recovery schemes, particularly for implementations of recovery blocks (Campbell and Randell, 1986). As in Example 3a), besides the primary algorithm, each process may have multiple alternative algorithms, which are in this scheme modelled as ordinary processes (rather than exception handling processes) in a natural CSP way: being composed in an alternative construct. The idea of the recovery blocks is in (Burns and Wellings, p.334/335), summarized as: “if any process fails its acceptance test, all processes have their state restored to that saved at the start of the conversation and they execute their alternative modules”.
5 Exception handling mechanism for CSP/CT software

5.4 Case study

Use of the prototyped EHM in a robotic context is demonstrated on the JIWY end-effector. The two examples emphasize the necessity of system-specific knowledge in order to precisely assess an error (exception) cause and consequences and take proper intervention in an efficient manner. Both the disadvantage of needing a good system-specific insight and the advantage of the specific error treatment are properties of the EHM as a forward error recovery mechanism.
Even for such a simple robotic system as JIWY is, the designer may end up with tens of problematic scenarios that benefit from using the EHM support. All software components mapped to physical components endangered with any identifiable failure (sensors, actuators, energetic and communication lines) may be guarded by exception handlers. Generally, environmental faults manifested as software exceptions in nominal functional processes make a subset of all external errors that can be satisfactorily treated by EHM provisions. That subset for the JIWY example comprises camera streaming errors, irregular joystick readings, axes position measuring out of the operational range, irregular motors’ steering. The last two are used as examples in this section. Detection of other hardware problems require introduction of additional software components (not modelled in the normal operational model configuration) and in most cases additional hardware (sensors). The examples are current or voltage overloads, motor overheats, hitting physical end stops or cable breaks. The last one is implemented by use of the prototyped EHM in frame of the integrity watchdog design pattern (section 6.2.3 on page 187).

The examples illustrate adding the EHM layer on the servo mode of the JIWY software. The safety issue of protecting motors against steering overloads is already addressed in the model of SanityCheck process presented in the previous chapter. In those models a simple limiting of the steering values is coded. Now the prevention of the excessive motor current values is done by identifying the irregular steering values as exceptions. The handling process in this example implements a safety measure of driving the system to a safe state. In the second example also a fault-tolerant behaviour is implemented. The excursion of the axes beyond the specified range is interpreted as control law inappropriateness and another control law is deployed.

The servo mode consists of the same components as presented in the previous chapter. Only the channel between Horizontal and Vertical controllers is dropped for simplicity and clarity of the channel suspension mechanism. The CSP diagrams of the servo mode are repeated in Figures 5-14 and 5-15.
Figure 5-14 Composition model of JIWY servo mode

Figure 5-15 JIWY servo mode without interaction between controllers
**Example 1.** Protecting motors of an excessive steering

An actuator overload may happen due to various causes. The cause may be external: the exploitation conditions may deviate from specified (wrong set point profile), or the actuated parts may be blocked due to an obstacle (*environmental* errors). But also due to software malfunctioning (*development* errors): a wrong parameter value in the controller, a controller design omission or a failure of the controller software module. Usually, the controllers comprise signal limiters before energizing the actuators. However, to be on the safe side, it is better not to assume that as a rule. Moreover, the failure of the controller module invalidates the role of the limiters as well.

The *SanityCheck* process from Figures 5-14 and 5-15 is augmented by composing it with the exception handling *OverloadHandler* in the ExC exception construct. This would be the minimal safeguarding configuration in this case. Upon detecting a prohibitive value on the channels *hor_san* or *ver_san* (Figure 5-15), *SanityCheck* throws the *OverloadException* exception (and consequently terminates), which is caught by *OverloadHandler* in the parallel construct that executes the servo control mode, channels connected to *SanityCheck* would be suspended by the *OverloadHandler*. This would cause termination of processes *Vertical* and *Horizontal*, which also throw the *OverloadException* exception.

In Figure 5-17 (communication view only) *Horizontal* and *Vertical* are also accompanied with exception handlers (*OverloadHandler_H* and *OverloadHandler_V*). This is not the minimal
safeguarding configuration, but is strongly recommended. In a minimal configuration exceptions thrown by Horizontal and Vertical would propagate out of the ParServo parallel construct to be caught by a top-level, default exception handler that may not exist always. In that case the control program would abort.

The other reason of having the handlers associated with the axes controllers is to allow the system to enter a safe mode upon compromising permitted motors steering range. Namely, it is up to the nature of the system and the design/safety requirements what functionality of the exception handlers has to be. The default behaviour of exception handlers is stopping the system in the state where an exception has occurred. But, stopping the control mode implies deenergizing the actuators, which does not always lead to the freeze of the physical set-up, but may end up in collapsing, due to gravity for example. This means that exception handlers sometimes have to implement special control modes for ensuring the safe state of the system. In this example, a safe homing regime is chosen. The communication diagram (Figure 5-17) shows that for this functionality exception handlers must be connected to the linkdrivers in order to steer the system to the safe state.

SafetyHoming H exception handling process requires access to the encoder readings (EncH linkdriver) and the actuator (DAcH linkdriver) in order to drive the horizontal axis to the initial position.

After an OverloadException from SanityCheck is thrown, OverloadHandling_SC handling process poisons channels hor_san and ver_san with copies of the OverloadException (marked “1” on the Figure 5-17). It depends on the scheduling of processes Horizontal and Vertical.
which one gets a copy of the exception from the channels first; in that order (the uncertainty is marked “2/3”) handlers OverloadHandling_H and OverloadHandling_V are scheduled consequently. They implement the safety homing for the horizontal and vertical axes respectively.

**Example 2. Soft endstops**

Composition and communication configuration for guarding the axes from escaping out of the defined working area is the same as for the motor overload protection from Figure 5-17. However, the minimal configuration would differ. The source of exceptions is not the SanityCheck process, but both Horizontal and Vertical controllers. Thus, two exception handlers would be the minimum. As explained in the previous example, it is recommended to have a handler for each process in the parallel composition.

Also the order of the channel suspension differs in this case and is more deterministic. For instance, in Figure 5-19, the Horizontal controller observes a reading from the encoder EncH that is out of boundaries. Horizontal terminates by throwing the SoftEndstopException exception. This exception is handled by EndstopHandling_H which suspends the hor_san channel. The suspension causes throwing the SoftEndstopException exception again after SanityCheck attempts to read from hor_san. Consequently, EndstopHandling_SC suspends both hor_san and ver_san channels, and eventually, upon exceptional termination of Vertical, EndstopHandling_V gets activated.

As discussed before, instead of barely stopping the further execution of the servo mode, handlers may implement a safety homing regime (as indicated by the connection of the handlers to the hardware drivers in Figure 5-19).

---

**Figure 5-19 Channel suspension chain on the SoftEndstopException**
A more interesting case for using the EHM arises in this context. Namely, the occurrence of crossing the boundary of the permitted working range need not (and usually should not) lead to a drastic solution as stopping or shutting down the system in a safe state. Namely, exceeding the “soft” working range is not always that serious incident. Moreover, in a system more than one set of safety boundaries may be defined, in order to take appropriate measures accordingly.

For instance, a non-stationary plant can be the subject to control. This means that some control adaptation strategy has to be deployed, for instance gain scheduling (Hilhorst et al., 1994). In the most simplified form, that means a transition from one control law to another upon an appropriate event. That event may be the act of throwing an exception.

On the JIWY set-up the boundary crossing event is used for such an experiment, Figure 5-20. The primary controller (Vertical) is designed with an exaggerated overshoot. Consequently, the effective working range is narrowed. In other words, even for having the set point within the permitted range, for certain sudden increments of the reference signal, the axes may pass the soft endstops. In this example (in Figure 5-20 only part of the model responsible for the vertical axes is shown), occurrence of the SoftEndstopException exception activates an alternative controller (Vertical1), making the system tolerate inadequacy of the primary controller. In fact, using the design pattern from example 3a), one may implement a collection of redundant controllers and a policy of activating them accordingly to the observed state of the system. Each of the chained (nested) exception handlers may contain an alternative controller (as
Vertical2 on Figure 5-20 is for Vertical1). Note that the handler implementing the redundant controller has to be connected to all link-drivers associated to the nominal working regime of the axis (joystick, encoder and the motor driver).

5.5 Discussion

"Exception handling and the provision of error recovery are extremely difficult in concurrent and distributed systems" (Xu et al., 2000, p.2). This undoubtedly valid statement justifies (hopefully) the relative complexity of the prototyped EHM support, both for modelling exception handling in the graphical design tool and implementation in the CTC++ library.

The main contributions of this development are:

- The prototyped EHM provides exception handling facilities for building safe and fault-tolerant concurrent software based on the CSP/CT environment. It gives a framework to deal with concurrency-specific phenomena, as resolution of concurrent exceptions and simultaneous (concerted) handling. While several CSP-based concurrent EHMs have been proposed, to the author’s knowledge, none was practically implemented until now.

- This EHM architecture abstracts away from using peculiar features of any particular programming language; still, it fulfils the established EHM quality criteria (stated in section 5.1.3t, and discussed against in this section).

- The gCSP tool features:
  - visualization that reflects orthogonality of the exception handling layer and nominal operation architecture,
  - establishing some consistency mechanisms to complement integrity of the EHM extensions to the CT library,
  - a prototype for automatic generation of EHM components.

- Demonstration of practical use of the developed exception handling facilities for various goals.

Compared with the starting point of this research into CSP/CT EHM recorded in (Van Engelen, 2004; Hilderink, 2005a), in this chapter contributions of combining the channel poisoning with raising exceptions and mechanism of detecting and handling simultaneous exceptions are elaborated.
5.5.1 Properties of the prototyped EHM

Here the fulfilment of the general requirements stated in section 5.1.3 is discussed.

1. **The mechanism has to be simple to understand and use.**
   This is of course a relative and subjective measure. Nonetheless, this EHM was designed to resemble the widely used EHM model, the try/throw/catch mechanism in C++/Java. Moreover, the exception propagation mechanism is doubtlessly simple thanks to the static compositional hierarchy of the CSP/CT architecture. The complication that channel suspension brings in is managed by the tool support. The gCSP global exception management naturally builds on simple graphical representation of the exception handling layer in a CSP/CT design. With maturing of the prototyped code generation, the outlines of purely graphical programming fault-tolerant CSP/CT software are emerging.

2. **The framework should provide a clear separation of the nominal program code flow and the code intended for handling possible exceptions.**
   A satisfactory separation of the nominal code of the EHM models in C++/Java in try blocks and catch statements is extended to the process-level of the CSP/CT architecture and visually emphasized in gCSP by having separate nominal processes and exception handling processes. Since gCSP generates separate source files for all processes defined in a model, the separation of concerns of nominal and exceptional operation in program code is accomplished even physically.

3. **Preventing continuation of an incomplete operation.**
   The termination model of this EHM implies fulfilment of this requirement. Upon rising exceptions, the guarded process is terminated and exception handling in the handler commences. It is the programmer's responsibility to fully use the error information that the thrown exception carries in order to exploit the entire potential of the EHM as a forward error recovery mechanism.

4. **Information about occurrence of an exception must contain all necessary data for a proper handling, i.e. recovery action.**
   Again a good property of the C++/Java EHM models is followed and extended: an exception class, derived from the base Exception class, may contain any information the programmer finds useful for a proper error recovery. An exception (constructor) may have parameters, which improve the readability at the place of the exception creation. The exception set concept allows dealing with issues of concurrently raised exceptions.

5. **Flexibility to allow adding, changing and refining exceptions.**
   This EHM follows the object-oriented paradigm of defining exception classes for different exceptional occurrences. Moreover, a generic mechanism for using the inheritance information for allowing gradual exception handling is implemented.
6. The execution overhead of exception handling code should be allowed only in the presence of exceptions – exception handling burdens on the error-free execution flow have to be minimized. Compared with the C++ EHM, it can be stated that in this respect the overhead introduced in the nominal code is the same. Additional overhead in an exceptional situation (due to instantiation of an exception set and switching the execution context to the exception handling process) is the price for allowing for handling exceptions in a concurrent framework.

7. The mechanism should allow the uniform treatment of exceptions detected both by the environment and by the program. This is an issue that this EHM cannot significantly improve on its abstract level – it is restricted by the implementation language and the underlying infrastructure (OS, hardware). This EHM can cope with this requirement as much as an implementation language can. A language-native EHM and run-time support can accomplish interception an exception caused by a hardware failure and make use of the THROW macro to transform the hardware error information to an exception set.

8. Declaring exceptions that a component may raise
The prototyped CT library implementation itself does not offer any provisions for declaring exceptions that can be raised in a process or compositions (constructs). Conceived as a conceptual mechanism, it does not use any language-specific support for checking exception declarations (at compile time). The exception consistency mechanism is a part of the exception management implemented in gCSP. By this separation of responsibilities the library usage is relieved from the burden of strict exception declarations (that is found restrictive by some authors, see page 147).

9. Nesting exception handling facilities
Campbell and Randell (1986, p.814), define an exception handler as “a component that may have its own context, exceptions, and exception handlers”. The exception handlers in this EHM are implemented as processes composed in the exception constructs, which are processes themselves and can be composed further in higher exception constructs. Example 3a) in section 5.3 shows how this requirement is covered by a generally applicable scheme.

10. Upon an exception occurrence in a process involved in a parallel execution with other processes, all participating processes need to be informed which exception has occurred.
Exception handlers of communicating, parallel composed processes are supposed to suspend (poison) with the thrown exception all channels connected to the guarded process (this information is present in the network builder, i.e. the graphical model within gCSP). Consequently, the suspended channels distribute the exception at hand to all processes engaged in the communication on suspended channels or already blocked on those channels. In that way all participating processes get notified of exceptional occurrences. It is assumed that the parallel composed processes are connected by channels, otherwise
proliferation (smuggling) of corrupted information is not a threat and the synchronized recovery is not an issue anymore.

11. **All participating processes need to be suspended (aborted if a termination model is applied, as here is) and enter the recovery activity specific for the exception occurred.**

Suppose that exception handlers suspend channels as described in the previous item, at rendezvous all processes get terminated due to the exception raised in suspended channels and the corresponding handlers take over. The main drawback of this framework is a prospective delay of commencing error recovery after an exception has occurred. If a process is blocked on a channel at the moment the channel gets suspended, the process is terminated immediately in favour of the associated handler. But, a process may be involved in a lengthy calculation before it reaches the rendezvous point and the associated handler starts the execution. None of the implementation languages of the CT libraries (C, C++, Java) allows for asynchronous transfer of control (ATC, like in Ada or Real-Time Java, (Burns and Wellings, 2001, p.339) and the CT is not preemptive in that sense. In order to preserve the cross-language support, this issue should be solved in the CT infrastructure in a generic way (Jovanovic et al., 2005).

12. **In case of simultaneous (concurrent) exceptional occurrences in different parallel composed processes, a handler should be chosen that treats the compound exceptional situation rather than isolated exceptions.**

In the described EHM, this happens for instance when exceptions occur in different participating processes before they reach the first rendezvous. If exceptions occur in more than one process, it is inevitable that one of the participant processes will try to inject an exception to a channel already suspended with another type of exception. If the types of exceptions are different, this means simultaneous exception occurrences. This kind of situations can be dealt with in a few ways. Before handling the exceptions, handling processes may equalize their exception sets so that all select the appropriate recovery actions or may all terminate by throwing their exception sets higher up. The higher exception handler is able to construct the exception hierarchy. This exception handler for compound exceptions is always connected to the parallel construct, not to the parallel composed processes themselves. In this exception handler process recovery code needs to be provided for all relevant combinations of exception that may appear in the exception set.

### 5.5.2 Conclusions

This chapter presented an exception handling mechanism for concurrent process-oriented software in line with the underlying CSP principles. The updates of the original CTC++ library preserve the composibility of the CSP design paradigm. Moreover, the gCSP tool preserves an initial, EHM-unaware design of nominal software operation. The architecture enhanced with the
exception handling layer keeps the same compositional hierarchy as the initial design.

The crucial issues of concurrent exception handling—propagation of exceptions among parallel composed processes, detection of concurrent exceptions and simultaneous error recovery—are addressed by extending the channel poisoning mechanism. Although capable for achieving the desired functional behaviour, this concept suffers from a few real-time temporal penalties. The temporal behaviour would benefit from a sort of asynchronous preemption facility, which requires a considerable update of the CT kernel (announced for the redesigned SIP edition of the implementation library).

The aid of a design tool support in designing complex, especially multiprocess systems is indispensable. The automation of maintaining an extensive exception coverage relieves the designer from a serious consistency burden. In this approach the implementation abstraction layer (the CTC++ library) is kept simpler on account of the sophistication that is delegated to the gCSP tool.

A distinctive graphical representation of the exception handling layer over CSP/CT architectures along with a simple exception propagation mechanism target the main difficulties reported against using EHM facilities, as “the least documented and understood part”. As being encapsulated in regular CSP/CT processes, the issue of testing exception handling code does not differ from testing any other process functionality in a design.

Suitability of the prototyped EHM with respect to implementation of atomic actions is discussed in Appendix B. The most promising announcement in the area of exception handling mechanisms—EHM in Ada 2005—was unfortunately insufficiently detailed at the time of writing to make a comparison with the proposal elaborated in this section.

5.5.3 Directions for further research and development

Enhancements of all the libraries, the tool and the design methodology may be recommended as the following:

Design methodology

The distinctive quality of a CSP-based design paradigm is amenability to formal analysis. Chapter 4 presented use of the CSPm code generation in the deadlock analysis. The formal treatment of the newly introduced exception operator is offered in (Hilderink, 2005a). The evaluation and simplification of this formalism through an extension of the CSPm code generation engine and perhaps further refinement of the EHM domain of the graphical language is promising. The potential of having exception handling in a concurrent, formally checkable environment would substantially increase the acceptability of the prototyped EHM.

The scope of this development was restricted to one-processor execution framework. Having in mind the distributiveness potential of the CSP/CT framework, a logical extension of the proposed EHM are distributed
designs. Experimentation in that direction would certainly reveal new points of attention.

**Implementation libraries**

This EHM prototype may undergo different degrees of optimization and/or enhancement in order to minimize potential temporal penalties.

The mechanism for creating and using exception inheritance is simple, but not efficient (both with respect to memory and time resources). The inheritance hierarchy is instantiated as a linked list of `ExceptionType` objects with every single exception. Browsing the inheritance hierarchy deploys slow string comparison procedures (Van Engelen, 2004). The gCSP tool already contains information on the exception inheritance hierarchy within a model. With a centralized administration of the inheritance hierarchy exception objects would not need haul their complete genealogy and furthermore the inheritance relationships would be faster determined in run-time. The efficiency of dealing with exception derivations may benefit from a CT component serving as an exception manager (Jovanovic et al., 2005).

**gCSP tool**

A comprehensive exception coverage in complex systems causes a large number of exception types, often related in different ways. On basis of the global exception list functionality, additional abilities may be proposed. The information of the exception inheritance hierarchy may be visualized using standard UML class diagrams. More challenging would be a creation of an editor for managing exception hierarchies for resolving concurrently raised exceptions. In (Campbell and Randell, 1986) a tree structure for such a hierarchy is recommended. It would be extremely interesting to explore relations between this tree structure with the inheritance hierarchy and the compositional tree structure of gCSP models.
Previous chapters dealt with techniques for preventing development errors in process-oriented architectures before a system is deployed, and with handling anticipated intermittent errors (exceptions) in run-time. Regardless how much attention is paid to prevent and eliminate architectural and implementation errors—through a careful design and thorough testing—residual development errors in any software system are inevitable. According to Anderson and Lee (1981, p.294), design faults are probably the most important category of unanticipated faults, and are a consequence of unmastered complexity in a system. This chapter considers minimising effects of solid unanticipated development errors that stay uncovered by previously elaborated techniques.

It has already been observed that any technique for increasing software dependability relies on introducing some form of redundancy into the (development process of) executable systems. For anticipated errors, the designer may rely on dynamic redundant software components that are invoked only upon detection of an error. In order to correct or mask unanticipated errors, the only resort is static redundancy – the price to pay is the permanent overhead, since static redundant components remain in use whether or not any errors occur.

This chapter deals with static redundancy in the form of architectural (and non-trivial) design patterns suitable for process-oriented architectures. “Non-trivial” means that simply cramming additional functional blocks to an
error-unaware software structure (for instance classical limiters in control systems, as process SanityCheck in Figures 5-14 and 5-15 on page 168) is not regarded a design pattern; use of those is trivial in any design paradigm and they change an original (error-unaware) structure. Rather here, certain approaches are considered that take advantage of the process orientation of the architecture and are orthogonal to the initial architectural topology. However, it will be shown that classical safety measures, as signal limiting, are very well attainable.

Design patterns as generalized solutions are introduced in section 6.1. Some of the design patterns elaborated in this chapter are preprogrammed in the CT library and can be activated at will, as it will be described for system load watchdogs in section 6.2; for the patterns that operate on modelling level (other watchdogs from section 6.2, N-version programming in section 6.3 and the logging/monitoring mechanism in section 6.4), principles and examples are provided. Unlike the previous chapters, due to a handful of different techniques, both gCSP support (modelling and code generation) and examples are decentralized all over the chapter sections. A retrospective of the proposed design patterns is illustrated by application on the Tripod case study in section 6.5, and together with directions for further research systematized in the concluding section 6.6.

6.1 On design patterns

Maturity of a design paradigm is evaluated considering establishment of certain design patterns within that paradigm. A design pattern is a generalized solution to a commonly occurring problem. According to Douglass (2003, p.xvii), the very best developers abstract the problems and their solutions into generalized approaches that have proved consistently effective.

This chapter presents application of a few selected classical static redundancy approaches enriched and adapted to the CSP/CT architecture. The selection criteria are:

- Suitability for a process-oriented architecture, in particular CSP/CT,
- Non-obtrusiveness (transparency) to an initial, error-unaware design – minimizing any increase in the model/system complexity,
- Simplicity, i.e. a wide recognition in industry.

The most popular static dependability design patterns are:

1. Replied subsystems – “hot” spares, Switch-to-Backup and Single Protected Channels (Douglass, 2003), Triple Modular Redundancy (TMR – (Anderson and Lee, 1981)) or N-version programming (Chen and Avižienis, 1978),
2. Timeout and watchdog mechanisms (Anderson and Lee, 1981; Perrin, 1999),
3. Monitoring and logging mechanisms (Anderson and Lee, 1981),
4. Run-time assertions and error detections (e.g. overflow, limiters or sanity checks) (Leveson, 1995),
5. Redundant data, as checksums, CRC, Hamming codes (Knight and Leveson, 1986),
6. Self-checking and auto-diagnostics software,
7. Alarms and switching to manual operation.

In the remainder, variants of the first three approaches are elaborated as dependability design patterns applied to the CSP/CT framework. The first three are architectural dependability instruments, while the others are functional techniques applicable in any architecture.

Seriousness of actions that the protective components in the design layers can initiate are ranked as **intervention levels**, ranging from the least invasive to more radical: logging to a file, activating warning functionality along continuing nominal operation, influencing (modifying) wrong values communicated along channels, poisoning channels, aborting execution by throwing an exception or switching control to emergency modes.

### 6.2 Watchdog patterns

Several authors regard the watchdog mechanism the simplest error detection check (Anderson and Lee, 1981, p.124), the “life ticks” as called by Douglass (2003, p.436). They are viewed as simplification of the replication design patterns, of which the N-version programming is presented in the next section, 6.3.

Historically, watchdogs as hardware circuitries have been initially used to support timing checks of software functioning (Perrin, 1999). The principal purpose of those watchdogs was detection of **software malfunctioning** as the microprocessor software does not “hit” the watchdog (actually resets the watchdog counter) regularly. If the “life tick” delays to arrive to the watchdog circuit (or arrives too early in some implementations), the circuit resets the processor in a good hope that after some acceptable time the system is up and running correctly again. Over time, the principles of this rough hardware dependability concept has been leveraged towards pure software solutions—(Anderson and Lee, 1981, p.123; Douglass, 2003, p.443)—and has been used not only for irregularities indication based on timing disruptions, but also in other forms (for system integrity concerns as presented later in this chapter).

There are two main realisation issues to be addressed when designing and using watchdogging mechanisms for increasing dependability of a system. First is the dependability of the watchdog itself. Namely, it is one trouble if it fails to detect an error; the greater trouble is posed if it erroneously reacts on non-existing errors. This is related to the second problem with this mechanism: since watchdogs are traditionally primitive mechanisms, it is a challenge to make it not simply shut down or reset the system, but really react as a fault-tolerant mechanism. Rephrased: what can be done upon perception of an error?

The problem of dependability of the watchdog itself is addressed by making it a **hardcore dependability** concern of an application (Anderson and
Lee, 1981, p.72) – meaning that “the critical components which support the fault tolerance activities of the systems must operate reliably if system failures are to be prevented. If the size, complexity and construction of such critical components are comparable with those of the system for which they are supporting fault tolerance, there will be cause for concern.” For high integrity systems in general, and in the CSP/CT environment in particular, good care has to be taken that the watchdog mechanism itself is simple and not affected by the error it is supposed to indicate.

The solution to the problem of adding to the reaction of a watchdog a fault-tolerant behaviour strongly depends on the flexibility of the software architecture to which the mechanism is applied as well as on the kind of problems one tries to solve with the watchdog. For making a watchdog a passive observer that only notifies irregularities to a log file, the implementation does not pose much problems – an example is logging CPU overloads in (paragraph 6.2.2). More challenging approaches deploy some of the more invasive intervention levels, as dealt with in sections 6.2.1 and 6.2.3.

This section illustrates achieving various protective effects, ranging from pure software implementations to involving hardware additions, achieved with specific modelling constructs and patterns or just by an appropriate combination of software-hardware. In the following three sections three different variants of the watchdog design pattern for the CT libraries are presented: the first two (6.2.1 and 6.2.2) are pure software solutions, with a gradually decreasing gCSP modelling support; in 6.2.3 a mechanism extended with hardware redundancy—being highly problem-specific—does not visually reflect presence of the watchdog layer at all. The first, liveness watchdog, as a general software solution is fully supported by the modelling tool. Section 6.2.2 elaborates a preprogrammed CT library extension for monitoring vital system parameters.

6.2.1 Liveness watchdogs

Watchdogs implemented in software are closely bound to the concept of timeout, employed to ensure that various facilities of the system are not lost or locked out by programs that fail to complete (Anderson and Lee, 1981, p.124). These kinds of checks are generally known as liveness checks.

The liveness watchdog mechanism can react passively or actively, over the full range of the intervention levels. A prerequisite of using the liveness watchdog pattern is provision of a fail-safe state of the system (Douglass, 2003). For allowing an active, but a safe fault tolerance role of the watchdogging layer in CSP/CT networks, a new construction—the watchdog construct and a corresponding compositional relationship—is introduced. It combines the nominal-operation network that realizes the intended functionality of an application and a redundant (emergency) network being activated upon a watchdog timeout event. In that respect the mechanism is reminiscent of the exception handling mechanism. However, there are substantial differences that motivate creating a separate construct. While exceptions are being thrown by processes stranded in an exceptional
situation, with the liveness watchdog mechanism transfer of control happens without having the culprit processes being aware of that. Furthermore, while the EHM allows a fine granularity of replacing parts of the network by alternative executions, CT liveness watchdog timeout events are not capable of pinpointing the troublesome processes. Namely, due to rendezvous blocking, a process blocked on a zapped process may have an associated watchdog with a timeout shorter than that of the malfunctioning process. Therefore, a watchdog timeout causes the whole nominal network to terminate and executes the alternative network to try to remedy the situation. It depends on the application to what extent the emergency network can confine and assess damage, recover the error and continue (gracefully degraded) service, which is often an emergency activity.

**Modelling the liveness watchdog in gCSP**

In Figures 6-1 and 6-2 the closed loop example from page 115 is extended by the liveness watchdogging layer. Presence of the watchdog dependability pattern reflects on both the graphical and tree editors. The compositional watchdog relationship and construct associate the main and the emergency network. The shape of the watchdogged processes (LoopConProcess in the figure) is adorned with a small shield-like symbol.

![Figure 6-1 Closed loop example with watchdog protection](image)

Setting (and by that starting) a liveness watchdog is done before entering the body of repetitive operations. In the generated code this is accomplished by invoking method `set` upon the watchdog object, the CT correspondent of the set watchdog linkdriver. The designer may choose to hit the watchdog before engaging in any suspected operation—writing/reading from channels or a processing code block—by inserting the hit watchdog linkdriver in sequence with the risky operations. The corresponding CT method is `hit`. More likely is a minimalistic approach, like in this example: placing only one watchdog hit in the repetitive algorithm. After completion of the repetitive set of operations the watchdog should be disengaged (which is modelled by the watchdog remove linkdriver, i.e. `remove` method). Setting, hitting and removing
linkdrivers (operations) can be placed at the same compositional hierarchy level, or, as in this example, be distributed at different places in a program. Care should be taken that every setting of a watchdog must be matched by a corresponding removal.

![Diagram](image)

**Figure 6-2** Hitting watchdog happens in `LoopConProcess`

One may argue that the liveness watchdog pattern heavily influences an initial design of a process, and looking from the compositional point of view that is true. To alleviate this impact, the gCSP tool handles the liveness-watchdog mechanism as a modelling layer that can be switched off and on – so the watchdogging additions can be removed from the views.

**CT implementation of the liveness watchdog**

As discussed, the main issue is to assure that the error in an application does not affect the watchdog layer as well. This implies that this layer should be very well independent of the rest of a CT program. If watchdog components had been implemented as ordinary CT processes, a culprit process that keeps a CT network locked out would have also blocked the watchdogging mechanism.

Since the watchdog mechanism operates on basis of temporal (mis)behaviour of the system, the timer used to program watchdog timeout is bound to the real time clock with a high granularity. Elapse of the timeout guard is in implementation associated to an interrupt – a classical solution from the time of hardware watchdogs (Perrin, 1999). In the interrupt routine different intervention levels can be attained. In the most radical case—transferring the flow of control irreversibly to the emergency process—the nominal network (its top construct) is removed from the scheduler queue. In that case, because the watchdogs would stay activated in the interrupt routine, it is the responsibility of the programmer to deactivate them in order to prevent explosion of alarms while the recovery operation is underway.
Modelling the watchdog timeout mechanism by the alternative composition and details on programming the timer interrupt to this end is elaborated in Appendix C.

It should be made clear that the applicability of purely software watchdogging mechanisms covers much less environmental failures (as power supply dips or electromagnetic interference) compared to the classical watchdog circuitries. Though the spectrum of the intervention levels of software implementation is much broader in the direction of fault tolerance, for highly safe designs the software watchdog has to be accompanied by the hardware variants.

6.2.2 Real-time feasibility watchdogs

When liveness of a system is secured, it makes sense to reason about meeting temporal (real-time) constraints of the system. This paragraph proposes two watchdogging patterns to assist run-time monitoring of certain temporal system indicators. The first is a simplification of the liveness watchdog which also operates at the modelling level as presented in the previous paragraph; the other is a preprogrammed CT implementation level mechanism.

Deadline watchdog

In this pattern a timely critical process sets the watchdog at the beginning of an operation, and removes it at the end. Hitting the watchdog is not necessary – Figure 6-3 reflects the simplicity of this pattern. Note that by applying this form of the watchdog pattern the internal specification a time-critical process can remain intact – the process is only put in sequence with the setting and removing watchdog linkdrivers.

If the dog-watched process does not finish (terminate) before the set deadline, the watchdog times out. It may then implement any intervention technique. Although this functionality can be achieved with set and remove methods, to improve the CT code readability, watchdogs can be turned to work in the “one-shot” regime to serve this design pattern – see (Huijgen, 2005, p.38/39).
If meeting the deadline of a specific algorithmic part of a program is dubious, this pattern can be applied also inside a process specification, like it is shown on the PlantDynamicsProcess from the closed loop example, Figure 6-4. Hereby it is suspected that calculating the plant dynamics may compromise a simulation timestep, therefore the duration of calculating the plant dynamics is monitored by the set-remove pair (and can also be measured in this way).

**Processor load indication and monitoring slack time margins**

This form of the watchdog layer operates at the low CT scheduling level, and therefore does not reflect on a gCSP graphical model visually; in the tool this option is switched on or off in the root of the C-tree (Model icon). It is a preprogrammed facility that at will may be enabled at the CT kernel level, according to the settings in the gCSP model.

The mechanism represents an augmentation of the idle process functionality of the CT scheduler. In the CT kernel the idle process possesses the processor when no other process is running (usually when all other processes wait for communication events, internally on channels or externally on linkdrivers). Therefore, in all cases when the system is not overloaded, the idle process will get some CPU time to waste: $t_{idle}$. On basis of this time an estimation about the system load can be made. In case of digital control systems, closed loops are bound to a certain sampling period ($T$). Therefore, two complementary indicators, processor load ($L$) and slack time margin ($M$), are given by the following formulae:

$$L = \frac{T - t_{idle}}{T}, \quad M = \frac{t_{idle}}{T}, \quad L + M = 1.$$

A processor load watchdog monitors the system load and, in a case the load exceeds a given threshold, the watchdog may deploy some intervention action. The least invasive is giving indications of the processor load $L$ in the
log file, like in an example of the log file of a system where the real-time feasibility margin was set to 80%:

```
"timestamp","name","value"
1382,"SystemLoad", 0.24
1392,"SystemLoad", 0.55
1402,"SystemLoad", 0.69
1412,"SystemLoad","Overload!"
1412,"SystemLoad", 0.83
1422,"SystemLoad","Overload!"
1422,"SystemLoad", 0.81
1432,"SystemLoad", 0.68
1442,"SystemLoad", 0.61
1452,"SystemLoad", 0.21
```

### 6.2.3 Integrity watchdog

Finally, a mechanism that encompasses also hardware redundancy is presented in this section. The companion software solution is accordingly highly system-specific. Therefore, instead of attempting to give a general scheme, a practical example is described, which combines the watchdogging principles with the exception handling mechanism.

**JIWY cables integrity**

In Chapter 5 the JIWY axes controllers are exception-guarded by handling processes against exceeding permitted angle ranges and excessive motor steering; in this example – Figure 6-5 – cable integrity violation exception (CableBreakException) is added. The handler for this exception is added to an enhanced exception handling process, called H_ExceptionHandler.

Operational signals for the horizontal axis (horizontal encoder, horizontal joystick axis, horizontal motor steering) all go through one cable. The watchdog wire is connected to a circuitry that upon the wire break causes a hardware interrupt mapped to the CableBreakException.

Figure 6-5 models the following functionality. Upon occurrence of CableBreakException exception, the handling process H_ExceptionHandler takes control. This process implements annulling the steering signal (in order to prevent uncontrolled motions upon reconnecting the cable) and choice of the user for the further operation. On pressing certain joystick buttons (note that joystick buttons wires are not connected to the JIWY set-up, but directly to the control computer), the user may choose to restart the JIWY software. Since restarting the calibration sequence is beyond the responsibility of the servo mode, the user choice to restart the system has to be communicated to a higher hierarchical level, through the restart port. Opting for program termination terminates the exception handling process, and consequently the servo control mode.
Obviously this is not a timeout-based mechanism. However, it can be very well combined with the deadline watchdog to restrict the outage of the JIWY service. This examples demonstrates a watchdog functionality that relies on additional hardware and the exception handling mechanism. Implementation details, that include use of a hardware interrupt line, are reported in (van Engelen, 2004, p.47). Van Engelen (2004) also describes the procedure of setting up the interrupt line and a way of writing the interrupt handling routine for the CT software which are the crucial ingredients of this design pattern.

6.3 N-version programming

Multiple (N) software component versions (functional replicas) bring in redundant algorithms derived from the same functional specification and developed independently—by different tools, languages, teams etc.—(Chen and Avižienis, 1978). If there is a disagreement in outcomes from different software component versions, an odd number N>2 of the functional components allows elimination of error influences by majority voting policy. The goal of providing multiple copies of a critical algorithm is to increase the design diversity in order to avoid negative influence of common mode development faults. Avižienis (1985) contends that N-version software can only be successful and successfully tolerate faults if the required design diversity is met.

The feasibility of avoiding common design omissions is criticized in (Knight et al., 1985; Leveson, 1986), as unable to provide 99% confidence level software, since it assumes that a program can be completely, consistently and unambiguously specified and that programs which have been developed independently will fail independently (Burns and Wellings, 2001, p.110). However, the practice proves (Hatton, 1997) that a 3-version
software is 5 to 9 times more reliable than a corresponding single-version high-quality design.

The basic principle of replicating software components is picturized in Figure 6-6. Version1 and Version2 are two independent implementations of the specification of a critical algorithm. They get the same inputs, process them simultaneously and produce sets of results, which, if both implementations for the given set of inputs function correctly, are correct (although not necessarily exactly the same). Having the result from Version2 can provide additional confidence about the result from Version1; a cacophony of results indicates presence of errors in one (or both) versions. The Comparator process running simultaneously with the versions checks the consistency (exactness or close similarity) of the versions' results at the end or possibly during the calculations.

![Figure 6-6 Replicating principle](image)

The described principle and the CSP/CT implementation offer a framework for any form of replicated (sub)systems: Protected Single Channel, Switch-to-backup (Douglass, 2003) or heterogeneous redundancy. (Homogeneous replication, as Triple modular redundancy – TMR – (Anderson and Lee, 1981, p.120) makes no sense for software components due to common mode development faults). In CSP/CT networks a critical process is being replaced by an error-tolerant one that contains multiple processes with the same functionality as the original plus coordination processes (for distributing inputs and comparing outputs).

Anderson and Lee (1981, p.276) give the Comparator process (calling it “driver program”) the following crucial responsibilities:

1. invoking each of the versions,
2. waiting for the versions to complete their execution,
3. comparing and acting upon the $N$ sets of results.

In the proposed scheme in Figure 6-6 the CSP/CT parallel execution inherently fulfills a lot of the “driver program” responsibilities. Versions are parallel composed processes that do not need to be invoked explicitly – as composed in parallel they start executing simultaneously as soon as they get inputs. Also there is no need to pay special attention on waiting versions to complete their executions – the parallel construct composing them will rule
that the comparator can compare results only when it gets data, being unblocked of rendezvous with all the versions. Anderson and Lee (1981) add a requirement that “the synchronization scheme has to allow for different execution times of the modules” which is also inherently satisfied by the semantics of the parallel construct in CSP/CT. Fulfilment of these requirements by the CSP/CT mechanisms is discussed in detail in the example in section 6.3.2.

The dependability of the concept depends on the comparator component – making it in this concept the hardcore of the scheme. Nonetheless, the most troublesome algorithmic challenge in N-version programming is construction of the comparator algorithm – and that is the problem of inexact voting when the versions cannot provide an exact output although all of them correctly implement the specification (think of accuracy of floating point calculations or solving equation with multiple solutions). Since the pattern in this text aims at providing only an architectural structure for deploying N-version algorithms according to the well known principle, the problem of inexact voting itself is not dealt with here. For a starting point for studying this issue the reader is referred to (Burns and Wellings, 2001, p.112).

It is worth noting that N-version, although being a legitimate fault tolerance approach, in case of an odd number of versions actually does not require an error detection algorithm in the comparator process! Namely, if there is an inconsistency among the results, the outlier can be just ignored and the output gets the value voted by majority. However, it is wise to provide a mechanism that will issue a warning that an error occurred. Also, for the system with only one replica provided (like in Figure 6-6), it is necessary that the comparator detects and reacts upon an error, since it cannot know which version is wrong. If an active reaction on the error occurrences is needed, similarly with the applications of watchdogs, the designer may choose how radical it can be, ranging from a record in a log file to throwing an exception.

The last elements of the proposed architecture, the server process, according to (Anderson and Lee, 1981, p.278), also deserves proper attention. As portrayed in the upcoming example, thanks to the CSP/CT implementation a proper synchronism with the other N-version components can be taken for granted.

### 6.3.1 N-version programming in CT and gCSP

The most common configuration for N-version programming, with \( N=3 \), is shown in Figure 6-7. The case with three versions is minimal for applying the majority voting principle. For mass-product applications \( N=3 \) is already an expensive overhead due to triplication of the development effort; for specific safety-critical applications, the number of versions can be much larger than three.
Figure 6-7 3-version programming scheme applied to a process $P$

The gCSP tool allows filling an empty process with a construction as in Figure 6-7. The user chooses in the G-editor or in the C-tree a critical process and specifies the number of versions as well as the number of input and output ports. The tool creates the internal N-version construction with an appropriate number of processes, a server and a comparator process as well as an appropriate number of distribution channels. Naming of automatically constructed processes and the constructs originates from the name of the critical process following a logical pattern, as shown in Figure 6-8 on the example of the SanityCheck process from the integrity watchdog scheme (Figure 6-5).

Figure 6-8 A created N-version scheme inside a custom process

Some aspects of using this concept in CSP/CT programs are elaborated in the following example and finally on the Tripod set-up.

### 6.3.2 Example: robust adder

In this example, besides deployment of the 3-version scheme from Figure 6-7, a combination with the exception handling mechanism is demonstrated. The
EHM-supported configuration is shown in Figure 6-9. The example assumes a hypothetical case of having adders microcoded as firmware (thus software) for inexpensive microcontrollers used, for instance, in a glass recycling pipeline. Supposedly an initial design dealt with adding small positive numbers summing up to five (bottles on an intake belt), and a 4-bit-adder has been used (the overflow occurs for sums greater than 7). Let us further suppose that later an 8-bit was added to reinforce the dependability of the system, and yet later a 16-bit-adder was added for upgrade to the 3-version configuration.

As it often happens in software development, in the course of upgrading a software-supported system, initial assumptions are being forgotten, and for extending the capacity of a system (in this case adding more intake belts for the recycling plant) the legacy components are used, not appropriately meeting the requirements of the new system. Clearly, in the new context, Adder_4b will be often giving erroneous results. Thanks to the other two versions of a greater capacity, the erroneous 4-bit calculation will be masked: two (Adder_8b and Adder_16b) of the three versions do agree on the result, which tolerates the oversight.

However, the applied scheme may not cope with errors of a different source. For instance, in case of erroneous readings of values well beyond the expected range from a malfunctioning input sensor that counts bottles, this error protection may be easily defeated. The erroneous input values would run Adder_8b, and maybe even Adder_16b in overflow. In that case, some erroneous input values would be masked while some other not, and even
worse, after a double overflow (from negative to positive numbers again), the error may even not be detected! For a mechanism for identifying this sort of malfunctioning see section 6.4 on logging and monitoring.

Use of the exception mechanism in case that the comparator process discovers inconsistencies in the result will be discussed after implementation of the constituent processes is clarified (Figures 6-10, 6-11, 6-12). The design of the constitutive parts is quite intuitive.

Figure 6-10 shows one of the versions that accepts two operands (in parallel) via the input ports and outputs the sum through the output port. The summation algorithms for 4-, 8- and 16-bit additions are captured by code blocks. The sequence is indefinitely repeated, so that after outputting the result for one set of inputs, the reader responsible for fetching the first operand immediately attempts to rendezvous in the next cycle.
Input ports of all versions are supplied with data by the server process, whose design is in Figure 6-11. As the readers in the version processes, here both readers and writers are parallel composed (complying with the I/O-SEQ pattern for avoiding deadlock, as presented in Chapter 4). The sequence between input parallel and output parallel secures that both operands are updated before they get distributed to the versions. After the parallel distribution, the whole composition may be repeated immediately, but an additional synchronisation may be needed in some cases (accomplished by the READ_synch primitive reader in this example). This will be discussed after the design of the comparator process is commented.

The I/O-SEQ pattern is partially applied in the design of the comparator process: only on inputting data in parallel. The sequence with the decision making code block and further to the output and synchronisation with the server (if applied by the WRITE_synch primitive writer) is crucial for the correctness of the framework. Namely, the pattern in this way rules that all versions deliver their results before the DecisionMaker algorithm runs.

In cases when it is wanted that a new cycle of calculation is initiated only after a positive verdict on the previous cycle, inserting the reader READ_synch in the server process and the writer WRITE_synch in the comparator processes in sequence with the rest of the composition achieves the wanted synchronisation. Namely, until WRITE_synch sends and acknowledgement to the server process, inputting of a new set of operands is not restarted (by the REP_Server repeater).

Having this extra channel between the comparator at the end of a cycle and the server at the beginning is also favourable for a simple termination of the framework in case of an exception occurrence somewhere in the framework. However, this synchronization is not necessary to have the N-version scheme work in general, so it is not automatically generated by the tool.

### 6.4 Logging and monitoring

Only measurable things can be controlled – reasoning about behaviour of an entity is possible only if the entity is observable. In order to have insight in the functioning of a software system, its behaviour has to be made observable. With embedded systems, results of the software activity are directly observable in physical manifestations, but waiting for physical effects to judge the reliability of embedded software is usually unacceptable. Also, in order to understand and improve the behaviour, one needs its records with a sufficient accuracy. In the context of successful error masking dependability instruments (as N-version programming), errors are happening in parts of the system, but being masked they are not observable, so not removable. Or, after a successful automated backward error recovery all evidence of faults in a system may be lost. With a logging/monitoring facility these kinds of errors would not go undetected – created logs can serve as trail files or audit logs (Anderson and Lee, 1981, p.11 and 243).
Having means of registrating various events (or states) is desired or explicitly required in many classes of technical systems. The ability of recording interesting events in software-supported systems is in this section covered by logging and monitoring (L/M) facilities. They are introduced to make it possible:

- to observe a system’s behaviour *on-line*:
  - as part of safety and fault tolerance mechanisms (Anderson and Lee, 1981, p.183),
  - as a debugging facility in experimenting with newly developed (embedded) software,
  - to perform adaptation of a system to environmental changes (specifically in the context of control systems for deployment of adaptive controllers);

- to analyse a system’s behaviour *off-line (post mortem)*:
  - for failure analysis,
  - for optimizing an overall performance of a plant at the highest – business – levels of large automated systems,
  - in legal issues, to determine security-related liability based on system logs as audit trails.

For a process-oriented architecture rooted in CSP principles, where the fundamental building blocks are processes with behaviour completely defined by the traces of events they communicate over channels, event registration mechanism has a particular potential. Observing the software behaviour is possible by *only knowing traffic on the channels*. This holds a promise of having yet another degree of the confidence in the CSP/CT network without intervening in an initial architecture nor modifying processes with stable implementations.

In the CSP/CT the functionality backing a network observation is captured by logging and monitoring components that are designed to be interchangeable, like illustrated in the class diagram in Figure 6-13.

![Figure 6-13 Inheritance relation of the logging and monitoring components](image_url)

In a context where differentiating specifics of the logging from monitoring is not important, the corresponding component will be referred to together as the **L/M coordinator** component.
6.4.1 Logging

The logging functionality is captured by a CT component that mediates between the CSP/CT communication layer and hardware recording media, as symbolically presented in Figure 6-14. This logging coordinator component is a typical unsynchronized shared resource (Cooling, 2003, p.252) of a write-only kind. The logging mechanism is actually an augmentation of the CSP/CT communication layer. Unlike the CSP/CT communication layer, it is chosen to be asynchronous (no rendezvous blocking in communication writing to the logging coordinator) and implicit (no channels to carry the data instances). This is because additional channels would be non-functional elements polluting an initial design and there is no need for formal analysis of this unilateral mechanism. This choice furthermore simplifies implementation of the logging coordinator as a global passive object and spares unnecessary context switches for better performance.

![Figure 6-14 Functional position of the logging coordinator component](image)

The principal responsibility of the logging coordinator is to attach time stamps to the data it receives from other components in a program and to store the record to a medium (RAM, permanent memory devices, network etc).
Probe channels

For yielding design transparency of the logging layer, a new concept of probe channels is introduced. In order to observe the behaviour of a CSP/CT network, the channels from an initial design that carry interesting data are being upgraded to probe channels. In this way the original topology remains intact and there are no changes in interfaces and implementation of processes.

In the generated code, instead of ordinary channels, there appear instances of the ProbeChannel class with an additional argument, which is the ID for marking the data in the log file coming from a probe channel. An example of creating a probe channel that carries integers reads as

\[
\text{Channel<int> } \ast \text{chML1 = new ProbeChannel<int>("chML1");}
\]

The principal mechanism of logging data communicated in the channels relies on sending data to the L/M coordinator before engaging in rendezvous, both in the read and write methods. Optionally, it is possible to log also messages that mark accomplishment of the rendezvous synchronisation—so upon a successful disengagement from rendezvous—this helps tracing the order of execution and inspect the scheduling policy.

Other L/M-aware components

For allowing registration of interesting events other than communication events, access to the global L/M coordinator is used often within ordinary processes, exception handing processes, the watchdogging layer and the like. Generally any component in the CSP/CT program can send data to the log file via the logging coordinator, provided it registered itself and obtained an ID. Therefore, appropriate methods of the L/M coordinator are provided to send data for logging.

Storing data

When data is received by the logging coordinator it is stored with some extra information: when the data was written to the coordinator and by whom it was written.

<table>
<thead>
<tr>
<th>timestamp</th>
<th>senderID</th>
<th>message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A message can be a value of the data communicated over the channel or a text message. In its initial design (Huijgen, 2005), values that can be stored are either numerical values (converted) in the format of double or predefined character strings. The log files are stored in a CSV (comma-separated-values) format as in the following log file example.

"timestamp","name","value"
3091930311342,"Process1",0.01
3091930311562,"Process2","Hello"
3091930311562,"Process3","world!"
The timestamps format originates from the CT timer implementation, in microseconds starting from the start-up of the computer. The current implementation of the CT timer allows also expressing time in hours, which can be adjusted to the absolute time.

### 6.4.2 Monitoring

The logging layer of the L/M coordinator passively records whatever is submitted to logging. Analysis of log files is a way easier with stored indications of particularly interesting data or trends. Moreover, as already pointed out, the qualitative behaviour of a CSP/CT network can be judged on basis of the channel communication. Therefore, observation of a suspicious behaviour, and acting upon it, may be very well used for early error confinement, assessment and possibly correction.

To create room for a more active use of the data available in run-time, a component closely coupled to the logging is introduced as a monitoring coordinator and symbolically presented in Figure 6-15 as a layer between the CSP/CT communication layer and the logging layer.

![Figure 6-15 Functional position of the monitoring coordinator component](image)

**Passive monitoring**

As an added value in the process of observing run-time behaviour of a network, monitoring is used to process and indicate interesting data values.
by inserting comments along with the data itself. Seen only as an extension to the logging functionality, it preprocesses the data sent for logging. For indicating errors or suspected trends in the communicated values, monitoring may need more than one source of information to assess the behaviour. The algorithms for assessing behaviour of a system are of course highly application specific.

**Active monitoring**

Letting the L/M coordinator not only preprocess data for logging, but also influence the communicated data or the execution flow (by inducing exceptions), the described mechanism becomes an active dependability pattern. In this way, without changing the structure of the system, some useful functionalities can be added, as limiting certain values, taking emergency actions and so on. For such one active role the infrastructure of the monitoring functionality is used, and for sake of not inventing yet another term, this mechanism exists under the hood of “monitoring”, though attributed “active”.

One example of the active use of the monitoring coordinator would be to achieve the same protective functionality of the SanityCheck process from Figures 4-13 and 4-14 (pages 131 and 133). Instead of having an extra process to check (and if necessary limit) the steering values to the output hardware linkdrivers, the same can be achieved by monitoring the linkdrivers (dAcH and dAcV) directly – Figure 6-16. Likewise, the adder, example from the previous section, is later in this section made safer by the monitoring mechanism.

![Figure 6-16 JIWY servo mode with the monitoring probe channels](image)
6.4.3 Modelling access to the L/M coordinator in the gCSP tool

There are two ways to depict access to the L/M coordinator(s) from the CT components. Channels carrying interesting information are turned into probe channels, while all the other component, typically variables, must be connected to the L/M linkdrivers.

As illustrated in Figure 6-16 monitored channels—probe channels—are depicted by a thicker, lightgrey line. The generated code for declarations of probe channels comply with that on page 197. For all probe channels in a model the tool generates this kind of definitions in the source code.

Access to the L/M facility from parts of code other then channels is modelled by using the L/M linkdrivers presented on page 75. The local variables that are sent to logging are connected to the linkdrivers by var-channels (see Figure 6-18). According to these assignments, corresponding methods calls are generated. L/M-aware components need to register themselves with the L/M coordinator.

The L/M layer has to be activated during configuration of the CT library. A dialog on the model root icon is reserved for activating the L/M layer in the model itself. Displaying the L/M layer (L/M linkdriver and the probe channel distinction) in the G-editor can be toggled, similarly to other gCSP layers.

6.4.4 Example: monitored adder

In the previous section it was shown how the reliability of counting disposed bottles in a recycling plant is increased by putting multiple adders (of different capacities) work in parallel within the N-version framework. Here the safety of the 1-version 8-bits adder is improved by using the L/M layer, deployed to indicate out-of-range input values or output overflows due to for example a noisy input (environmental failure) or/and an algorithmic mistake (development fault).

Both the input channels and the output channel of the adder are turned into probe channels, Figure 6-17. Probe channels implicitly send the data to the monitoring coordinator before communicating them further within the communication layer. The monitoring coordinator deploys an algorithm lighter then addition (otherwise the monitoring pattern would turn into 2-version software). input1 represents the number of bottles in the outlet pool and input2 represents the number of bottles on the intake belt. By knowing that the outlet pool is emptied when the number of bottles exceeds 60 and that the maximum capacity of the intake belt is 40, monitors of the input channels log warning messages when the input values fall out of the [0,60] and [0, 40] ranges respectively. The third monitor on overflow of the output channel alarms if the output value becomes negative – in case that the adder is improperly used in an increased capacity recycling plant.
Figure 6-17 Robust adder with probe channels

For getting the output pool jammed after one successful cycle, the log file may look like the following.

```
2691923564635,"input1",0.000000
2691923564643,"input2",21.000000
2691923564644,"output",21.000000
2691925913708,"input1",21.000000
2691925913713,"input2",19.000000
2691925913728,"output",40.000000
2691930319961,"input1",40.000000
2691930319968,"input2",18.000000
2691930319969,"output",58.000000
2691932003627,"input1",0.000000
2691932003633,"input2",26.000000
2691932003635,"output",26.000000
2691935593289,"input1",26.000000
2691935593291,"input2",25.000000
2691935593302,"output",51.000000
2691939103415,"input1",51.000000
2691939104009,"Adder input1 Monitor","The pool jammed!"
2691939104418,"input2",19.000000
2691939104439,"output",70.000000
26919392226159,"input1",70.000000
26919392228777,"Adder input1 Monitor","The pool jammed!"
26919392228786,"input2",31.000000
26919392229101,"output",101.000000
2691939231244,"input1",101.000000
2691939231274,"Adder input1 Monitor","The pool jammed!"
2692010033283,"input2",29.000000
2692010023298,"output",-126.000000
2692010025204,"Adder output Monitor","Output overflow!"
```

The same functionality can be achieved without using probe channels. The internal composition of the adder can be modified as in Figure 6-18. The internal variables holding the values read from or written to channels are sent to the L/M coordinator by the L/M linkdrivers.
6.5 Case study

Representatives from all design patterns elaborated in this chapter are applied on the Tripod set-up, either on the physical set-up or on its simulation model or both.

In this practical application it turned out that the order of applying the patterns was just opposite of the order they are presented in this chapter. To get a first impression of the functioning of the system, the L/M layer was used. Using it was also important for proper implementation and testing the N-versions of the controllers. Lastly, the crucial Commutator process has been extended with a liveness-watchdog for each axis. The top level Tripod software model with the applied patterns all together is shown in Figure 6-19. The presence of the watchdogging and the logging/monitoring layers are immediately observable; N-version implementation of the servo controller is applied deeper in the ControlSystem process.

6.5.1 Logging and monitoring on Tripod

The probe channels are used at the top model level, for channels \( z_1, z_2, z_3 \) (position feedback), \( z_{1v}, z_{2v}, z_{3v} \) (velocity feedback) and \( \text{force}_1, \text{force}_2, \text{force}_3 \) (steering) – Figure 6-19.

Logging is used to observe the default servo control mode before adding advanced dependability mechanisms, to test behaviour after added safety layers and error injections and to visualize the effect of monitoring in other operation modes. For instance, Figure 6-20 shows part of a log file taken in the alignment phase and visualized in the 20-sim simulator. Due to a calculation imprecision in an initially used reference path file, a slight
abuse of the working area (page 33) happens for the z-axis (the grey plot). Therefore, by using limiting via monitoring the movement has been flattened around 8 seconds.

Figure 6-19 Dependability-improved Tripod software: top level network

Figure 6-20 Log of the altitudinal and radial excursion of the platform, courtesy of Mark Huijgen (2005)
6.5.2 N-version programming on Tripod

The internal structure of ServoController is shown in Figure 3-71. The key process for the servo functionality is FollowPath, within which the motion profile files are read and worked out by the PID controllers, as in Figure 6-21. Initially there is one PID controller process for each axis.

By applying the 3-version scheme from Figure 6-7, each PID controller is internally implemented by three copies. (Making identical software copies makes sense only for the scheme demonstration!). The 3-version-schemes use the EHM layer. In the case one of the PID controllers has an error unmaskable by the 3-version scheme, a corresponding NWayExc handling process poisons incoming channels from the PathFromFile process. Therefore PathFromFile will also throw the exception. With PathFromFileExc handling process rethrowing the exception, the parallel construct terminates, so the higher level FollowPathExc handler on basis of the exception set after the parallel construct termination has chance to remedy the exceptional situation.

Figure 6-22 shows the result of an experiment on injecting errors in the PID controller 3-version scheme. Errors are of the kind that the outputs
of the versions clearly deviate from each other. At second 4 one such error is
activated in one of the versions in the controller of one axis. It is clearly
visible that steering of all axes continues smoothly, which means a complete
masking of the error. The false controller version continues functioning
incorrectly. At second 6 one more version in the 3-version controller starts
malfunctioning. The comparator processes of that axis reacts by throwing an
exception, and soon all controllers escape the servo position mode upon the
EHM layer activation. The safety homing is performed by three modes. After
returning to the zero level, holding the safe (zero) position is active till 7.5
seconds, after which motors deenergizing takes over.

Figure 6-22 Log of the activities of the 3-version Tripod controllers, courtesy of Mark Huijgen
(2005)

6.5.3 Watchdogs on Tripod

The top level network of the dependability-improved Tripod software in Figure
6-19 indicates presence of the liveness watchdogs in the Commutator
process. It is at the end of the dataflow of the Tripod software network, thus
all unexpected delays within the network will cause watchdog alarms in
Crummutator. The internal structure of the watchdoged Commutator is in
Figure 6-23.

Each axis commutation algorithm sets watchdogs after entering the
calculation loops. After each calculation watchdogs are hit, and just before
termination of the Commutator process all watchdogs would be removed. In
cases of a prolonged waiting for entering force steering from one of the input
channels, or outputting to the hardware or freezing of calculation algorithms,
one of the watchdogs’ timeouts expires. Subsequently WatchdogSafetyProcess in Figure 6-19
takes control over the whole system.
6.6 Conclusions and suggestions

This section summarizes the suitability of the chosen design patterns for the CSP/CT modelling/executive environment, the most important benefits contributed by the proposed realizations and their mutual complementarity. It also proposes points of attention for further development.

6.6.1 Logging and monitoring

Thanks to the fact that CSP processes (likewise whole networks) are defined only by the traces of events, the complete behaviour of a CSP/CT program can be observed only by observing the channel communication. Using this property, full information on behaviour is obtained by only replacing the ordinary channels by the probe channels augmentations. By this, adding the important aspect of monitoring to a CT program is possible without either changing functionality/implementation of processes or influencing the topology of the network.
In practical experiments, the logging and monitoring mechanisms proved interesting to be used as first dependability patterns in prototyping, deploying and testing of a system. The mechanism was used in the same way with the Tripod simulation model as on the robot itself. Actually, a recipe of using these patterns would suggest that a first testing version of a constructed CSP/CT software should have all channels as probe channels, regardless the program is control software or just a console application. An extensive use of the logging facility is demonstrated in (Huijgen, 2005), Chapter 5. In that work many other features of the L/M layer which are not applicable for real-time applications are also implemented, as dumping the log file from memory to a disk file or changing the size of the log file in run-time. Taking into consideration the real-time aspect of the L/M mechanism, only using logs in memory guarantees a predictable overhead. Writing to memory is real-time, but of course inconvenient for longer observations of the system.

### 6.6.2 Watchdogs

Some authors consider liveness-watchdog a lightweight protection with respect to the execution and code size overheads. It is also true that they are indispensable for handling the livelock- and deadlock-like run-time phenomena and also for the deadlocks and livelocks in the narrow (concurrency-specific) sense if the formal verification has not been used (or not used properly). On the other hand, for helping a zapped system out of a blockade, watchdogs imply the heaviest intervention level: aborting the nominal network in the locked state and giving control to an emergency network to try to remedy the situation. The watchdogging layer lacks the flexibility that the exception handling layer possesses: replacing only stranded processes with redundant processes. A deadlocked process unfortunately cannot throw an exception, as well as in the current CT implementation the watchdogs residing in the interrupt routine cannot pinpoint a culprit process causing a timeout expiry and induce an appropriate exception in the nominal network.

The other presented watchdogs not modelled at the graphical level, as the preprogrammed facility for monitoring processor load and slacktime margins, will be kept as useful in future library editions. Moreover, the mechanism makes a fruitful combination with the logging mechanism, by recording the processor run-time utilization. This is a potential that yet should be fully exploited in distributed designs.

Some authors define real-time systems as those where a missed deadline means failure of the system, so having the deadline watchdog abruptly aborting the system if a critical process misses its deadline—as the deadline watchdog presented in section 6.2.2 does—is just appropriate. However, making the deadline watchdog terminate the nominal network is not useful in the development phase of a hard real-time system, and for sure not wanted in many soft real-time systems. In these cases combining again the watchdog and the logging component gives the hard real-time system developers a guideline to discover the system's bottleneck on basis of the log
file records. In the exploitation of soft real-time systems, combining the watchdog and the monitoring component provides another elegant solution: monitoring gives enough space to see if the service outage is intermittent or permanent. Based on the observation within a certain tolerance time interval, the monitoring component can decide how to react according to the malfunctioning behaviour.

Finally, it has been demonstrated that under the hood of “watchdogging” many other techniques can be identified as useful for increasing fault tolerance of a system. The hardware redundancy in the JIHY cables is a representative of a hardware watchdogging element combined with the fault tolerance aspect of the CT exception handling mechanism coupled to the interrupt caused by breaking the watchdog wire.

6.6.3 N-version programming

Supporting the N-version programming principle is the heaviest static redundancy design pattern of all proposed in this chapter. However, while all the other patterns have to cause a substantial change in the program flow upon error detection, only the N-version programming can literally tolerate errors in a way that the execution of the program does not experience any difference. Combination with the logging functionality however does not let the errors stay undetected.

In the case of unmaskable errors, N-version programming collaborates elegantly with the exception handling facilities. Some improvements of the exception handling mechanism to serve the N-version scheme even better are worth further investigation (Romanovsky, 1999, 2000).

It should be noted that this chapter emphasized an elegant fit of the CSP/CT architecture and N-versioning implementation needs. For using the concept itself, one should be warned of algorithmic (design) details responsible for yielding the real benefit of the N-version programming, as sound requirements for the design diversity and peculiarities of inexact voting schemes.

All the presented patterns are to a good extent supported by the modelling means of the gCSP tools, both in the G-editor and the C-tree. The means of minimising obtrusiveness of the superimposed dependability mechanisms upon an initial design were stressed for each of the patterns, particularly the concept of design layers. This process-oriented concept alleviates “pollution” of a nominal design where modifications of the design and the model topology are necessary. Automatic code generation for the additional elements and mechanisms is less comprehensive than that for the dependability instruments in the previous chapters and is specified for further development.
6.6.4 Directions for further development

Logging and monitoring
In the current CT implementation, besides predefined text messages, all other data have to be converted to rational types in double precision before submitted for logging. For complex datatypes this is however not an option. Hence, an extension for storing the other datatypes deserves an implementation effort. Modelling access to multiple logs for different types of data or for different parts of a system would empower the prototyped CT support for multiple log files (Huijgen, 2005). It has been already mentioned that a solution for large log files with satisfactory real-time access is to be searched in the real-time networking support.

Manipulation of the L/M coordinator(s) is a part of the configuration engine of the implementation library. A tighter collaboration of the gCSP tool and the library configuration engine would be the key for a higher automation. Also, additions to the automatic code generation can straightforwardly relieve programmers from giving attention to proper inclusion of the L/M aspect to the source code files.

Watchdogs
For the liveness watchdog, unambiguous identification of a process that causes a timeout is the first issue calling for attention. This would help allowing the mechanism to use some of the moderate intervention levels rather than one of the extremes of passive logging at one end and abrupt termination of the nominal programs at the other end. A promising way of enabling graceful degradation regimes would be framing the liveness watchdogging mechanism within the exception handling facilities – ideally, this would replace use of a separate watchdog construct by using the exception construct. The other approach would base on allowing emergency networks to access the states of the abandoned nominal networks.

Appendix C offers a gCSP model of the CT watchdog. The subsystem of the watchdogs composed by an alternative construct, as modelled in Figure C-1b on page 237, should be implemented in exactly the same way by the implementation library constructs. Such implementation, as it holds for the modelling, would benefit from the CSP semantics precisely expressing the nature of the construction. However, since the CT constructs cannot be elegantly used for programming interrupt routines, another mechanism that allows using the library primitives for the liveness watchdogs should be devised. The proposed watchdog mechanism is implemented by plain C++ means, which only emulate the behaviour modelled by the given CSP diagram.

With watchdogs for system state monitoring it would be interesting to add observations of resources other than processor time (as unallocated memory for identifying memory leakage for example) and then be able to trace it in the log file. Running out of memory is possible even in absence of the memory leakage implementation faults, for example in case when a system has an extensive exception collection – a danger of exhausting all available memory is present if the exceptions occur frequently.
**N-version programming**

In definitions of the N-version programming (for instance (Chen and Avižienis, 1978)), a possibility of having crosschecks also *during* the execution of versions is highlighted. Allowing this sort of synchronisations with the primitive communication processes is clumsy, although not impossible (Huijgen, 2005). The synchronisation concept of barriers (Hilderink, 2005a) has a potential to directly addresses such a need. As already mentioned, slight adjustments in the exception handling mechanism would allow even more comprehensive possibilities of combining the N-version programming pattern with the exception handling facilities. Finally, multiple versions of algorithms certainly degrade the performance of an initial design; if not taken into account, it may imperil the real-time feasibility of software. It will be interesting to extend this design pattern by having the comparator process sensitive on timeout on incoming channels or prescribing use of deadline watchdogs.
Part III  Reflections and details

Chapter 7  Wrapping up the big picture

Appendix A  Some implementation details of the CSP/CTC++ exception handling mechanism

Appendix B  Atomic actions in CSP/CT – an outline

Appendix C  Some implementation details of the watchdog mechanism

Appendix D  CTC++ code generation and templates for 20-sim

References
Dependability

- Readiness for correct service
- Continuity of correct service
- Absence of catastrophic consequences on the user(s) and the environment
- Absence of improper system alterations
- Ability to undergo modifications and repairs

<table>
<thead>
<tr>
<th>Availability</th>
<th>Reliability</th>
<th>Safety</th>
<th>Integrity</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchdogs</td>
<td>Formal verification</td>
<td>Formal verification</td>
<td>Watchdogs</td>
<td>Code generation</td>
</tr>
<tr>
<td>N-versioning</td>
<td>Code generation</td>
<td>Exception handling</td>
<td>Logging</td>
<td>Logging</td>
</tr>
<tr>
<td>Monitoring</td>
<td>Exception handling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N-versioning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monitoring</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Watchdogs N-versioning Monitoring

Formal verification
Code generation
Exception handling
N-versioning Monitoring

Formal verification
Exception handling
Watchdogs Monitoring

Watchdogs Logging
Code generation Logging
7 Wrapping up the big picture

In the world of one-man programming, the programmer, like a mathematician, uses his or her creative insights to create mathematically sound programs. Team-based software engineering, on the other hand, is akin to putting many minds together to engineer a single mathematical construct. This is not an approach that has a record of success in the world of mathematics, yet we are by necessity building software in this manner.

Wei-Lung Wang ("Beware the engineering metaphor," 2002)

The main aim of the reported work was providing a tool-supported framework for designing dependable concurrent software with special provisions for (but not limited to) embedded control systems. Practical orientation of the dependability approaches in this thesis is reflected in providing tooling, modelling soundness and uniformity to solutions already used in industry.

This chapter provides a summary on suitability of the chosen design patterns for the CSP/CT modelling/executive environment, the most important benefits contributed by the proposed realizations of the patterns and mechanisms, their mutual complementarity and prospects for further improvements. Sections 7.1.1 and 7.2.2 summarize the contributions of the dependability instruments proposed in this work with respect to the error coverage they provide; sections 7.1.3 through 7.1.6 reflect on advantages and disadvantages of the process orientation as paradigm for developing dependable (embedded) software. Before a final conclusion on this work in section 7.3, recommendations for enhancing the technology as a whole are listed in section 7.2.

7.1 Conclusions

The dependability mechanisms elaborated in this work cover error prevention and error tolerance techniques. Chapters 3 and 4 concentrated on fault prevention; Chapters 5 and 6 presented fault tolerance instruments, dynamic and static respectively. The order of presentation of the dependability approaches implies the order they are applied in practical situations.

7.1.1 Contributions revisited

Dependability instruments for concurrent software elaborated in this thesis map to the dependability attributes (Avižienis et al., 2004) as the scheme on the left illustrates. According to the list of the contributions of this thesis as in section 1.8.2. on page 29, each of them is reviewed in the following five paragraphs.
Capabilities of the developed gCSP tool

The process-oriented CSP/CT modelling and programming framework has been enriched in this work with a CASE tool that helps model, visualize and manipulate software models in order to manage the inevitable complexity. Chapter 3 established understanding of concepts and features of the gCSP tool, basis for further development of the dependability approach of the three core chapters in part II. In Chapter 3 the complete graphical language is described along with its formal specification in machine-readable notation CSPm, directly liable to formal verification by the FDR model checker. By manipulating of the built models, the gCSP tool exploits the formal underpinning of the methodology to demonstrate feasibility of formal verification of graphical designs. Efficient production and trusting the final outcome of the design, the implementation code, is significantly increased by the code generator for CTC++-compliant code.

Special attention is paid to the features that support abstraction, partitioning and hierarchical organization of the graphical models, key to complexity management. The implemented compositional tree serves multiple purposes, as design overview and model navigation, design partitioning by containment hierarchy and handling concurrency by compositional hierarchy. Furthermore, this structure proved efficient in mediating between the low-level (machine-readable) implementation code and abstract (human-comprehensible) graphical models.

Graphical modelling principles of the CSP diagrams allowed introduction of the concept of design layers, a graphical consequence of the framework’s composibility. The idea of layers also uses the power of process orientation to allow orthogonality of design concerns such as:

- nominal functionality,
- safeguarding and redundancy mechanisms,
- multi-view representation of interest in different design stages and/or to different stakeholders.

The gCSP tool is yet another indicator of the advent of the tool-based programming paradigm. As compilers made a revolution with producing machine code out of higher level language designs, the tool-based paradigm shifts the production one level higher in such a way that now code in the higher languages is a subsidiary representation of the abstract graphical designs which are becoming the main, human-efficient design space.

Methodology for designing dependable process-oriented software

The software design principles adhered to in this work promote raising quality of software in design time, before there are any implementation artefacts to test, i.e. before the software is to be deployed. Mechatronic design paradigms experience design discontinuities along development trajectories, traditionally bridged by manual ad-hoc transformations of models from different design domains. This work contributes to eliminating the methodology- and tool-unsupported gaps in an integrated design trajectory for mechatronic systems by provisions of automatic code generation. Having the formalized graphical modelling in place, model checking of these graphical models has been made “press-the-buttons” away. Furthermore, the
implementation code ready for deployment and operational tests is produced from the very same model used for formal verification, without any manual intervention.

For coping with inevitable run-time errors, the CSP/CT framework proved hospitable for various dependability raising mechanisms: concurrent exception handling, N-version programming, logging, monitoring and several variants of watchdogs. This string of industry-exploited (“classical”) disparate dependability instruments has been brought under a uniform process-oriented, graphical semantics – a semantics of a layered software architecture, literally represented by graphical design layers in the gCSP tool. The primary criterion in choice and implementation of the protective redundancy was nonobtrusiveness of superimposed dependability layers to an initial (nominal) graphical design. Together with the high compositibility of the CSP/CT building blocks, the orthogonality of the framework facilitates maintainability and extensibility of the CSP/CT software. Besides providing and integrating new added value of dependability in the CSP/CT framework for embedded software, the methodology is facilitating integration of the specific domain tools, as summarized in the next item.

The interdomain tool chain

In multidisciplinary engineering disciplines, of which mechatronics is a remarkable example, besides an inherent complexity, factors of disparate cultures, mindsets, concepts and languages cause predominant problems. The discontinuities in design phases finding place at borders of different domains are best bridged by unified high-level modelling paradigms and integrated tooling environments.

We find that process orientation, based on dataflow modelling, serves well as a glue logic in mechatronic designs. For developing embedded software for these systems the gCSP tool finds its place in between two worlds. The first deals with modelling dynamical systems controlled by laws to be implemented in embedded software. A completely different one is inhabited by tools that verify the quality level of software specifications through for example model checking. Moreover, in between these two important engineering disciplines, gCSP adds the power of maintaining software production aspects by managing the software structure and provides the most interesting output for the practitioners: the compileable code. Only a few toolkits of such a wide span are known to date in this field (Cavalcanti et al., 2005). The toolchain promoted in this thesis has traced a path for further integration of the constitutive tools in one integrated design environment. The context of mechatronic design in the Control Engineering laboratory benefited of the gCSP as a CASE tool binding the 20-sim control laws implemented in embedded software with the FDR tool capable of formally verifying that software.

Extensions to the implementation library

An ultimate goal of this research project was to demonstrate the results in mechatronic applications. Demonstrating extensions of the CSP/CT methodology would not be possible without corresponding extensions of the implementation CT libraries (Van Engelen, 2004; Huijgen, 2005). It is
expected that the planned support for automatic code generation of all elements graphically modelable by the gCSP tool will lead to further polishing of the concrete implementations. Results of this research have speeded up initiatives for improving the initial CT library towards its successors (Orlic, 2002-2006; Hilderink, 2005b), holding a promise to simplify the dependability mechanisms and boost the performance further.

Two robotic case studies
The practical applications of the products of this thesis were two mechatronic demonstrators, JIWY and Tripod. Programming JIWY and Tripod represented an irreplaceable feedback to the design methodology and tools. Finally, besides being provided with a tool support, promotion of a new paradigm must be backed by tangible examples as well. A lucky circumstance of this research was its setting in robotic control problems, where major development steps can be quickly and rather obviously demonstrated by convincing physical phenomena; likewise, the flaws exhibit themselves also rather vividly.

7.1.2 Error coverage and complementarity of the proposed dependability techniques

This thesis proposes a combination of various approaches to dependability of concurrent software, found suitable for dataflow-centred process-oriented modelling. The comprehensiveness of the approach was being driven and evaluated by the error coverage, depicted in Figure 7-1.

Complementarity of the dependability design patterns elaborated in the previous chapter was discussed in section 6.6, page 206. In this item the complementary error coverage of all the proposed dependability instruments is interpreted according to the sketch in Figure 7-1.

1. Formal verification (FV) is applied upon mathematically rigorous specification of a design of a system, being attainable at early design stages. Design flaws discovered by formal verification are reproducible erroneous conditions, hence solid errors. With by the designer assisted modelling of the various abnormal phenomena, formal verification can cover also classes or errors which are normally considered intermittent. In scope of the origin of the errors, formal verification as illustrated in Chapter 4 covers development errors; however, modelling the environment of an embedded system is increasingly attracting research attention (Brinksma et al., 2005), giving a perspective that the same technology can combat also environmental errors. Finally, formal verification reveals design flaws whose source and place the designers are not explicitly aware of, thus being a measure against unanticipated errors.
2. Automatic code generation (ACG), founded in Chapters 3 and 4, is successful in preventing errors caused by manual transformations of abstract software models to implementation code. Therefore, ACG covers development errors of the solid nature, of which all are in principle unexpected, i.e. unanticipated.

3. Exception handling mechanism (EHM) from Chapter 5, as being primarily a forward error recovery means strongly depends on proper anticipation of errors’ source and place. As being a dynamic redundancy technique, it allows the designer to provide protective code against all predictable intermittent errors in any conceivable activation scenario, both caused by development faults and environmental failures. The last error coverage distinguishes the EHMs of all the other defensive techniques as the most successful in raising the system’s attention on unspecified behaviour of the environment.

4. Design patterns (DP) from Chapter 6, as static redundancy instruments, are expensive in covering all anticipated intermittent and solid errors; therefore, they target only those errors recognised as the most probable and difficult to combat by the previous techniques. They are clearly effective in alleviating or masking development errors, of which many are usually unanticipated.

It should be emphasized that the process orientation, besides lending itself conveniently for uniting these disparate techniques and mechanisms, in the first place possesses immanent qualities for clear and safe designs (as being motivated in section 1.7.2 on page 27).
A great deal of the modelling and implementation effort in this research went to integrating the exception handling mechanism into the dependability design patterns, which proved beneficial for effectiveness of the patterns. Liveness-watchdogs can compensate for (or complement) absence (or omissions) of deadlock detections by formal verification early in the error prevention phases or in system maintenance (when accurate models might not be available anymore). However, it should be kept in mind that timing checks are useful to reveal the presence of faults in a system but not their absence, and are used to supplement other checks in an operational system (Anderson and Lee, 1981, p.123; Douglass, 2003, p.415; Cooling, 2003, p.238). The logging and monitoring mechanisms are found useful to support confidence and diagnostic of both nominal and protective software functionality. N-version programming possesses a unique potential not only to completely mask development errors in a design, but also to help develop and test trustworthiness of alternative/improved software components by letting them run along with already approved components in realistic exploitation conditions. The logging/monitoring facilities are then used to keep track of behaviour of the newly introduced components behaviour.

Anderson and Lee (1981) emphasize that any kind of the reliability technique relies on a critical hardcore component that has to be genuinely reliable. For formal verification and automatic code generation the critical parts are the gCSP code generators, and the FDR model checker trustworthiness (for the formal verification). Dependability of the end result—the CT-compliant implementation code—depends on the cornerstone of this development: dependability of the CT libraries, here taken for granted. For the exception handling techniques, the EHM itself is the hardcore part. Reliability of the liveness- and deadlock-watchdogs depends on proper functioning of the time references in the system and on separation of the watchdog program (or a watchdog hardware device) from the CT program that may lockout. The hardcore component of the N-versioning schemes is the comparator process. Likewise, logging (and monitoring) mechanisms depend on reliable storing of data and accuracy of the time references.

Finally, it should be pointed out that for ultimately dependable systems hardware redundancy is irreplaceable. Irrespective how sophisticated and thorough software dependability coverage may be, for example EMI disruptions of the computer component(s) inevitably lead to the system malfunctioning. The CSP-based process orientation is a convenient design paradigm for extending software dependability techniques with hardware redundancy measures by its transparent distributiveness.

7.1.3 Benefits of programming dependability in terms of concurrency

Although humans realize and use concurrency, behaving and being concurrently, the aware process of thinking (seemingly resembling the processes of speaking) appears to be sequential and concentrated, though timeshared. And that was the way people tried to conceive artificial intelligence, hitting this unnatural barrier of sequentiality as soon as the
designs became but utterly simple. Once this bad habit, traditionally though in basic programming courses, is dropped, the concurrency inevitably arises intuitive. However, even if realized intuitive and useful, when programmed with sequential means (as multithreading), concurrency turns into a monster again. Hence a natural modelling and implementation environment is required to support the concurrency to truly benefit the designs.

Apart from the cognitive essence, in this work concurrency has been recognized particularly suitable to embrace redundancy, the key to reliability. In reactive systems, what embedded systems—operating simultaneously with their environment—inherently are, the companion dependability redundancy has to run in parallel with the nominal software components, looking after preservation of the specified state of the system. This should be accomplished as much as possible orthogonally to the intended nominal functionality of the software, as the concurrent design and execution permit us.

Still, the state of the mind, being predominantly sequential, has the difficulty to verify the concurrent activities. That was one of the motivations to provide an automated tool chain that ends in the model checking tools and that is the reason to propose even a higher level of visualisation to reveal the problems in complex concurrent designs.

### 7.1.4 Benefits for designing (dependable) software in the CSP-based process-oriented way

The natural concurrency from the problem domain reflecting on the design has to be structured within the software architecture in order to make it tractable for practical and reliable applications. The architecture in this work is provided by process orientation, formally based on the CSP model of concurrency. By finding the common language of processes, channels, events and compositional constructs between this particular process-oriented framework and the CSP algebra, the CSP/CT software framework offers conceptual means for supporting the favourable properties of designing systems with explicit concurrency.

**Managing complexity and dependability through a natural architecture**

In process orientation as defined in this text, software design is driven by dataflow and causality inherited from the problem domain of a particular application. In the control domain, notions of functional blocks stem from the era of analogue elements, representing pieces of logic (mechanical, hydraulic, electronic) that naturally operated simultaneously. Although today implemented in software, they are still via graphical means reasoned upon as simultaneous entities. Process orientation allows mapping from the domain topologies to the process-oriented software architecture by direct correspondence from blocks to processes and signals to channels, enriched by the CSP compositional notions granting intuitive parallelism, and where wanted, explicit specifications of sequences or execution by alternatives.

The dependability potential of the CSP-based process-oriented framework stems from a natural favorization of the message passing abstracted by the notion of events.
**Composibility for understanding, extendibility and reuse**

Loose coupling among processes in process orientation is achieved by message passing through channels. This is important for reducing complexity (by partitioning and hierarchy), reusability and maintainability. Processes’ behaviour is described only by incoming and outgoing events, which form well-defined interfaces – facilitating abstraction and reliability. Furthermore, this allows high composibility of building blocks and orthogonality in separating concerns of software components into layers.

**Performance and reactiveness through events and data locality**

Self-containment of processes is further facilitated by strong encapsulation referred to as locality (Welch 1989). According to Welch, it has to do also with boosting performance: ‘one of the reasons why transputer process scheduling is fast is that only local information need be examined; a process is associated with channels – not other processes; a process does not maintain pointers to whatever processes may be at the other ends of those channels – it does not know (and does not need to know) whether its neighbours are alive, dead, terminatable, siblings or even if they have ever existed’. These observations suggest that the locality of processing is the key to loose coupling, portability and performance. These features are granted by the CSP principles and treatment of events, which provide the model of reactiveness.

**Transparent distributiveness**

This property derives from previous qualities. Locality is the key to composibility, the ability to reason about building a software system just as hardware: from well-defined and verified building blocks aware only of the context of the immediate surrounding. As the hardware processing is physically concurrent and susceptible to distribution, the very same notions in software design would give also to the software such an ease of distributiveness. Separation of hardware independent and hardware dependent parts of design is the first requirement for straightforward distribution of a design from one node to multiple nodes. In order to attribute the distribution transparent, the design framework has to facilitate preserving the composition and communication structures intact. It can be stated that a formally described concurrent design, divided in well-defined processes interacting only through channels, is a handy subject to distribution.

**Straightforward verifiability**

Here listed the last as being an implicit quality, but perhaps the first in its power, the mathematical CSP underpinning gives the formal soundness to the proposed architecture. Moreover, it leverages a good experience in compositional programming known from occam to (traditionally regarded as) “advanced” design aspects as dependability. It was shown that CSPm descriptions modelled almost all graphical gCSP elements in a simple way. An exception is the exception handling mechanism whose formalization is under consideration. However, although yet to be formalized, this dependability instrument already greatly benefited from its CSP-inspired establishment. Power of an EHM is the error (exception) propagation to the right context for its effective handling, with simplicity and effectiveness being the most
relevant factors. Simplicity of understanding and controlling alterations of execution flow in a system full of simultaneous activities is conceptually and visually supported by the exception construct, alike all other CSP/CT constructs. Controlling and understanding exceptions propagation is supported by the CSP/occam tree-like compositional architecture implemented as the C-tree in the gCSP tool. The effectiveness is reflected on the capacity of the system to engage in the failure treatment in the proper context in a system-specific way. These advantages consequently substantially improved the applicability of all other dependability patterns.

7.1.5 Why use CSP/CT in making embedded systems

Drawing from the properties of designing with explicit concurrency in the CSP-based process-oriented way, we may conclude that the CSP/CT paradigm possesses the following qualities:

- Simple architecture,
- Simple and safe event-based communication model featuring message passing over channels,
- Formal model of concurrency,
- Event-driven model of reactiveness,
- Reusability, maintainability and extendibility.

Thanks to the carefully extended occam programming model supported by object-oriented concepts in the CT libraries, the CSP/CT designs are characterized also by:

- Transparent distributiveness,
- Design portability,
- Implementation platforms heterogeneity,
- Control over performance and real-time behaviour.

When advocating straightforward distributiveness and portability, the argument is that the concurrent behaviour has to be encapsulated in the application itself, without relying on assumptions of vendor-dependent operating systems' support. The CT library handles this by having its own portable kernel. Several applications demonstrated the capability of the CT library to run on bare metal platforms (Van Drunen, 2000; Cronie et al., 2003; Orlic et al., 2003). If executed on an OS-supported system, the CT kernel and processes claim one or a few highest priority real-time OS' tasks for speeding up development and porting by using I/O services provided by the OS. Scheduling is kept the responsibility of the CT kernel. When distributed over (heterogeneous) nodes—bare metal or OS-supported—on each of them the same kernel is executed, preserving an unchanged behaviour. Portability of the kernel is achieved by reprogramming clearly defined platform-dependant kernel code. The implementation of the processes is reused without any modifications. They still interact with the outside world only through channels. The channels are further responsible for adapting to
a new execution environment. The linkdrivers represent the clearly separated hardware-dependent parts of a design. Adapted to a new (distributed) environment, they are “plugged” in the channels of the original design providing the distributed processes with the same read/write rendezvous communication interface. For having the same compositional behaviour, the composition relations among processes have to be preserved as well, but that is an issue successfully addressed since long with occam on transputers, refreshed and extended for CSP/CT in (Orlic, 2002-2006).

Similar principles of self-containment benefit also the real-time behaviour of a ported/distributed application. In short, the application itself should be real-time at the first place (Hilderink, 2005a). This thesis does not contribute to designing/reasoning on the real-time behaviour of the CSP/CT design, although proposing some means of assessing it. More attention to these aspects are given in (Orlic, 2002-2006).

7.1.6 What sits in the way

Among countless proposals for changes in software development paradigms, just a very small fraction comes to life (especially in industrial practice). Once established software technologies are used as long as they can cope with the problems at hand (and often much longer). However, the volume and nature of contemporary challenges to be tackled apparently surpass the capabilities of the currently used software technologies.

Since businesses prefer evolutions to revolutions, a newly proposed paradigm should build on known ones. However, compromising newly devised concepts may lead to countereffects. Process orientation as illustrated in this text maintains bridges to earlier technologies as object and component orientation. However, the chosen pillars of those bridges stem from these popular technologies’ predecessors, believed to be abandoned, as structured analysis and design rooted in dataflow-driven reasoning. Nevertheless, several characteristics and assumptions of the proposed software development paradigm can be anticipated as potentially critical for an industrial appreciation.

A new mindset

At the technical level, the proposed software paradigm imposes locality, i.e. refraining from manipulations of global states (variables). Processes are only legitimate building blocks. Interaction among software units is only permitted through strictly defined interfaces of channel communication.

At the management level, time to come to the first testable prototypes is prolonged for sake of detailed modelling (specification). Although more tractable, efficient and shorter, the phases of testing, debugging, maintenance etc. are preceded with a relatively long period of modelling and verification. The proposed design cycle of predominant modelling without tangible/testable pilot versions and prototypes finds a lot of resistance in practice – an attitude clearly indicating a not-yet-an-engineering state of practice.
Restricted formalized design

Some of the previously quoted design notions can be seen as serious restrictions of the design process. Indeed, proponents of the formally managed software development point out a necessary discipline in restricting the means and techniques used in software development (Selic and Motus, 2003; Broadfoot, 2005). These distinguished authors advocate use of only those techniques liable to thorough verification. This is often a severe constraint in a programming field already full of constraints, and it may easily be in contradiction with demands of efficient implementation, low-level intimacy with hardware, low power and other optimizations which require special programming techniques. However, in many successful software communities internal norms with similar notions have been used since long. The proposed paradigm offers implicitly these proven concepts to a wider community.

Technological limitations

Although the strong point of the proposed formal modelling is the potential for large-scale verifiability, the same quality is on the other hand hindered by the limitations of the verification technology. The proposed means for formal verification—the mainstream of deploying formal methods—is model checking. Unlike the other ways of verification (as simulation or theorem proving), model checking suffers from state-space explosion, a consequence of exhaustive verification of all possible execution scenarios, which in turn is the major advantage over simulation and testing. The reason to believe that this verification technique has the future is the very same one for proposing the dependability through redundancy: confidence in increasing hardware capabilities, both with increasing density of computational power as with distribution.

7.2 Recommendations for further research

This research has been continuously opening avenues to several promising sequels, both in specific parts of the design trajectory as well as the integration. Proving feasibility of spanning different design domains encourages integration of more and more refinements into a comprehensive design approach. The most obvious further research can be proposed in the following directions:

- Following ideas of dependability patterns, development of architectural patterns based on the gCSP language, for instance specific behavioural patterns (as fair parallel scheduling or time sharing) with inclusion of the deadlock-freedom I/O-SEQ and I/O-PAR schemes,
- Incorporating explicit notion of time and verifying temporal behaviour of the CSP/CT software architecture,
- Development and formal specification/verification of the atomic actions concept which would follow a necessary formal specification of the developed exception handling mechanism,
• Smaller investigation portions focusing on optimization of parts of dependability mechanisms proposed in this thesis.

After having indicated the principal directions for completing and extending the CSP/CT design framework, the remainder draws attention to technical improvements of specific parts of the design framework and methodology. To be noted before all, as it is symbolically presented in Figure 7-1, a lot of the possible error sources are still to be covered.

### 7.2.1 The tool, graphical language and code generators

The principles of using gCSP for modelling and code generation were presented on numerous examples; however, for the sake of illustrating various aspects of the tool applications, none of the examples, with a modest exception of the Tripod software, was close to the complexity of industrial cases. Although it can be advocated that the tool is equipped with facilities to struggle with much more complex systems, only deployment for such design efforts can elicit the tool’s bottlenecks. The following aspects are proposed to make the results presented in Chapters 3 and 4 widely applicable.

**Unification, simplification and full formalization**

The idea of layered software modelling gave a strong confidence that the process orientation is a promising way to go about with complex embedded software. The gCSP tool combines the views (communication, compositional, hybrid, C-tree) with layers (compositional grouping, logging, watchdogging, exception handling). It is worth considering how these modelling means can be unified.

The graphical language grew elaborate, even without implementing all features originally proposed for the CSP diagrams. Further application of the language should be accompanied by assessments of the language succinctness versus expressibility. Some compressions have been already proposed, as reducing the watchdogging mechanism to an exception handling pattern.

The exception handling mechanism is the largest part of the proposed dependability package with no exercised formal background. On the other hand, not all potentials already embraced by generating CSPm specifications and the FDR model checker are elaborated in this thesis, for example livelock and determinism checks. Moreover, control systems are an excellent problem domain for a sound assessment and unification of a few temporal extensions of CSP. The sophistication of the liveness and deadline watchdogs and timeout guards for alternative constructions are the first drivers for experimenting with time support.

**Visualisation of quality indicators**

While the gCSP code generation is generally applicable for the large CSP/CT class of CSP models, the graphical deadlock interpretation as illustrated on a few simple examples cannot be used in that way for complex architectures
where a deadlock loop spreads over multiple hierarchical levels. Moreover, the deadlock paths illustrated in this text were visualised manually. It is necessary to explore the possibilities of feeding debugging information from FDR back to the graphical models. For visualising the deadlock traces, a mechanism more flexible than the currently used submodel flattening (explode/implode) feature should be constructed.

An important verification means is simulation. The bridge among the gCSP capabilities to visualize CSP designs and animation provided by ProBE has yet to be established. Integration of the two tools would offer the powerful concept of executable specifications. Thanks to the graphical mapping from events to channels, the already built graphical infrastructure would add to designing with the gCSP tool an unparalleled visual feedback in prototyping CSP-based systems, an intuitive indicator of the conceptual soundness of a design.

**Further integration of the tools**

In the beginning of a design cycle of creating software for mechatronic examples, transformation from the 20-sim control structure to the gCSP dataflow view has to be performed manually. Importing 20-sim models, besides automating transfer of the topology, would offer also a possibility to automate incorporation of executable models within the CSP/CT software.

Finally, looking at the ideal of providing an integrated toolchain, outlined in Figure 2-7 on page 61, one can easily envisage an integrated tool consisting of the 20-sim engines for control design, the gCSP layers for software aspects specification enriched with a few UML views and parts of the FDR and ProBE engines. Now, when all the modelling concepts have been worked out and followed by current active adjustments in the implementation libraries, the vision of having a tool-based integrated design trajectory for dependable mechatronic systems is closer to become reality than ever before.

### 7.2.2 Exception handling

The exception handling mechanism elaborated in this thesis proved feasibility of handling intermittent errors in concurrent software in a structured way. **Besides the formalization of the exception handling mechanism, there are many other technical issues that can make the use of the mechanism simpler and more comfortable, for example a higher flexibility in dealing with runtime errors or more support from the gCSP tool. The main points of attention are:**

- More consistency checks built in the tool, preserving the EHM implementation as simple as possible, allowing implementations in various programming languages in a generic way,
- Adding the resumption mechanism in error handling next to the established termination model, as discussed in (Jovanovic *et al.*, 2005),
• Working out specificities for serving an atomic actions framework within CSP/CT on ideas as outlined in Appendix B.

7.2.3 Dependability design patterns

Besides enabling the framework of atomic actions for programming critical portions of concurrent executions, which would complement the proposed dependability techniques, several improvements in the prototyped mechanism can be proposed:

• The developed *watchdogging mechanism* is just conditionally to be named fault tolerant: its current potentials are much more important for safety critical systems, where driving a system to a safe state justifies an abrupt termination of the nominal functionality of the whole network. Asynchronous process termination (as announced for the SIP extensions) and a possibility for inducing exceptions only to the problematic processes would be the most important additional features. Identification of a culprit process can be supported also with allowing the implementation library to address the lowest-level software layers (as programming interrupt routines).

• The current *N-version programming* scheme has just one cross-version checkpoint – the (final) one implemented by the comparator process. Barriers as synchronisation means, although considered as an interesting modelling concept, have been getting low attention in the CSP/CT implementation. This advanced synchronisation feature would provide an elegant support to versions cross-checking on multiple checkpoints.

• Distribution of the *logging and monitoring layer* is certainly an interesting and not a new problem. But also for one-node-applications improvements of these mechanisms can further complement the dependability arsenal. For instance, run-time property assertions or acceptance checks, known from formal verification and atomic actions, find in the L/M mechanism a friendly environment. The monitoring facility could algorithmically implement various acceptance checks of the results travelling along channels and furthermore implement executable assertions applicable in run-time. These assertions may be generalized to formalized versions of likelihood checks used in process control (Laprie et al., 1990).

For all design patterns holds a general recommendation to complete and possibly extend the gCSP code generation support in the course of further experimentation and using the patterns in practice.
7.2.4 Distributiveness for the future

“Our hunger for computing power is largely being satisfied by the continuous development of ever faster serial processors. However there are limits to serial hardware technology that are likely to be approached within the next decade. Explicit parallelism will then become the only means of extending the performance of computers and the field of concurrent programming will finally have come of age” (Martin, 1996, p.134).

“Explicit parallelism” in the previous quoting pertains to distributing concurrent functions of a design from one processing node over many. The insatiability for performance is just one of the reasons to believe that the technologies of the future have to have distributiveness as one of the features. The other strong reason is the apparent imperative for distributed infrastructures of smart surroundings and ubiquitous computing. There are also many more mundane reasons to approach design of real-time systems with the distributive tools.

The CSP/CT approach as a methodology aspires to support predictions of the time behaviour of an embedded design (Orlic, 2002-2006). The price to pay for that ability is high (as complication of algorithms and burdens to the nominal code) and may be too high (limiting use of some design methods and techniques, to an extent which is possibly unacceptable for industry). The distributiveness is the key to making a fruitful trade-off between vital but contradicting design aspects. For example, one may assume that in all reasonable cases the system architect who understands the system by modelling it can make a rough choice of the processing unit to accomplish the functionality under the real-time constraints. He or she should be allowed to speculate on a minimal (cheapest) processing unit in a selection of prospective targets. The only ultimate requirement should be that the chosen unit is programmable with a CSP-based library to make it capable for efficient connecting with the same units, as it was possible with transputers. Since it is our assumption that a CSP/CT software design can be easily distributed, possible problems with the real-time feasibility of the initial design are to be solved simply by hardware parallelisation – by adding as much as necessary simple processors till the desired performance is achieved. For this, it is required that parts of a design (processes) straightforwardly stretch (by flexible channels) over multiple processing nodes.

It has been already discussed how is the distributiveness of the CSP/CT designs enabled by the portable kernel and the linkdrivers concept. This kind of transparent distribution can be easily modelled in the gCSP tool. Since the distributed software components all comply with the high-level CSP/CT concepts implementable in various languages and hardware platforms, all of them are transparently connectable. Yet to be ensured is that the compositional structure of an original one-node design keeps the same semantics. Why and how to preserve sequential execution of processes delegated to two separated processing units or how to make sure that the communication and synchronisation overhead does not diminish wins achieved by the distribution are just some of questions posed to the ongoing research into the CSP/CT distributiveness (Orlic, 2002-2006).
7.3 Closing

A general recommendation and conclusion is that this thesis, together with the recently published (Hilderink, 2005a) and the upcoming results (Orlic, 2002-2006) and the 20-sim package is and should be forming one body of the continuous research carried out at the Control Lab already for several years. Results of this joint work have emerged as a mechatronic design trajectory supported by automation tools in several crucial phases. Many attractive practical case studies have been demonstrated and many lessons have been learned. The components in the tool chain have substantially influenced each other. However, as results of thoughts in different periods and in context of different projects with disparate priorities, there is still a lot to do to harmonise all aspects of the cornerstones of the discussed technology: 20-sim, gCSP and CT (e.g. its successors as SIP and CP). The time has come to lean back and consider carefully the big picture. With luck in reserving sufficient further development resources, the future of this tool chain and the underlying methodology is bright.

For the dependability aspect in particular, it is the author's wish that this text will serve initiate brainstormings on development of safe and reliable embedded systems, employing the CSP way of building concurrent process-oriented software and initiating ideas for further research proposals in these directions.

Also, a true success will be recognized if examples in this thesis or shown proofs of concepts encourage the embedded software industry to start thinking on applying and extending principles around which this work is built.
Appendices

Appendix A  Some implementation details of the CSP/CTC++ exception handling mechanism

This appendix shows a few design decisions that back the required concurrency-specific requirements of the prototyped EHM. A complete implementation documentation can be found in (Van Engelen, 2004).

Exceptions, types of exceptions and exception derivation

Exception handlers must be able to tell exception objects one from another. In the C++ EHM type of an exception (i.e. object) is determined by the RTTI—Run Time Type Interface mechanism—(Stroustrup, 2000). In order to provide the required functionality on the level of libraries for different languages, a generic type checking facility based on linked lists is prototyped.

Every Exception object contains a linked list of ExceptionType objects (Figure A-1). When an exception is derived from another, a new ExceptionType object is added to the list. A string member of the ExceptionType object is initialized with the name of the derived exception. The ExceptionType class contains two methods to determine the type of an exception in an exception set (isOfType and isDerivedFrom). These methods are accessible through the public member type of the Exception class. This way, an exception handler can distinguish different exception occurrences.

The linked list is actually used to enable an exception handler to detect (and perhaps handle) derivatives of an exception the handler is programmed for. As discussed on page 147 (see Exception derivation), this mechanism is used to structure gradual and reusable exception handling. Method isDerivedFrom serves these purposes. About the use of the methods see on Exception handling processes on page 157.
Exception set
Idea of exception sets plays the central role in the mechanism of handling exceptions occurred during executions of alternatively or parallel composed processes, dealt with at the level the constructs (Hilderink, 2005a). In this work this concept is used to enable handling of simultaneous exception too.

ExceptionSet class is a standard collection of objects, an exception set can be manipulated by self-explanatory add, remove, next and current methods (Figure A-2). Besides adding an exception to the set, it is possible to add the contents of an ExceptionSet to another ExceptionSet: all exceptions are removed from the former and added to the latter.

Exception handlers typically use the isHandled method, by which the exception set is notified that the “current exception” is handled. The exception is removed from the set and deleted. The isHandled method returns the next exception in the set. If handing of an exception is not supported by the handling process, the next exception can be retrieved by calling the next method. This returns the next exception in the set without removing/deleting the current one.
Appendices

231

Figure A-2

<table>
<thead>
<tr>
<th>ExceptionSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>first : Exception*</td>
</tr>
<tr>
<td>last : Exception*</td>
</tr>
<tr>
<td>current : Exception*</td>
</tr>
<tr>
<td>+add(e : Exception*) : void</td>
</tr>
<tr>
<td>+add(e : ExceptionSet) : void</td>
</tr>
<tr>
<td>+remove() : Exception*</td>
</tr>
<tr>
<td>+first() : Exception*</td>
</tr>
<tr>
<td>+current() : Exception*</td>
</tr>
<tr>
<td>+next() : Exception*</td>
</tr>
<tr>
<td>+isHandled() : Exception*</td>
</tr>
<tr>
<td>+isEmpty() : bool</td>
</tr>
<tr>
<td>+ExceptionSet()</td>
</tr>
<tr>
<td>+ExceptionSet(e : Exception*)</td>
</tr>
<tr>
<td>+ExceptionSet(e : ExceptionSet)</td>
</tr>
</tbody>
</table>

Exception handling process

A general scheme for exception handlers is shown in the following listing (example for a handler that handles two families of exceptions).

```cpp
void AnExceptionHandlerProcess::run(ExceptionSet *eSet) {
    Exception *exc = eSet->first();
    while (exc) {
        if (exc->type.isDerivedFrom("SomeExceptionName")) {
            //...handling the exception as of type "SomeExceptionName"
            //...or more specifically by using isOfType()
            exc = eSet->isHandled();
        } else if (exc->type.isDerivedFrom("SomeOtherException")) {
            //...handling the exception as of type"SomeOtherException"
            //...or more specifically by using isOfType()
            exc = eSet->isHandled();
        } else {
            //...if the exception at hand cannot be handled by
            //...this exception handler
            exc = eSet->next();
        }
    }

    //Check if all exceptions in the set are handled or not
    if (!eSet->isEmpty()) THROW(eSet);
    else delete eSet;
}
```

After attempting fetching the first exception in the exception set, the collection is iterated as long as the pointer to the next exception is null. Through the conditional branches an appropriate handler is sought. Using the isDerivedFrom method, a handler checks if the exception at hand is
derived from a type that it can handle. To execute handling code written exactly for a specific exception, a selection that further examines the type of the exception using `isDerivedFrom` (for subtypes) and `isOfType` (for a specific type) can be nested. In this way an exception can be handled gradually.

If an exception is handled completely, it is removed from the collection by invoking the `isHandled` method. Only if all exceptions in the collection are completely handled, the collection is destroyed. Otherwise, the set is rethrown, with all unhandled exception or possibly partially handled exception. It can also happen that the exception handling process causes new exceptions. In that case, these are added to the exception set and (re)thrown.
Appendix B  Atomic actions in CSP/CT – an outline

This appendix comments suitability of the prototyped EH Mechanism for implementing the architectural pattern of atomic actions. It also indicates directions for a possible research in developing this dependability pattern.

Atomic actions are the principal error-confinement architectural pattern for dynamic error recovery of set of concurrent processes engaged in a collaborative operation (Lomet, 1977). It is a heavyweight design pattern, since it would require restructuring an error-unaware concurrent design. However, it is an effective means of architecting a concurrent system which requires high dependability.

Requirements for atomic actions

The purpose of an atomic action is to structure an operation that changes state of a system, for which a collaborative activity (encapsulated within processes) is necessary. The objective of such an activity is launching correct outputs from the collaborative process only when all of them pass assigned tests of correctness (acceptance tests). Before all processes pass corresponding acceptance tests, none of the processes is allowed to interact with processes outside participants of the atomic action. This communication atomicity prevents propagation of incorrect information from one or more participating concurrent processes in case that error in one (or consequently more) process(es) occurs, which would otherwise contaminate the system (causing so-called information smuggling). The first error detected in the acceptance tests causes all of the processes to engage in error recovery for that particular error.

Therefore, a framework for programming atomic actions must support the following requirements:

1. clear identification of the participants (Jalote and Campbell, 1984),
2. recovering from a situation when one or more participants fail to enter the collaborative atomic action - the deserter problem (Kim, 1982),
3. atomicity of communication from the system’s perspective,
4. synchronization on acceptance tests,
5. uniform error recovery (possibly collaborative) if one or more processes fail the acceptance test.

The last requirement is the principal focus in research into atomic actions architectures. The error recovery techniques are mostly based on exception handling facilities, which must comply with the concurrency-specific demands as elaborated in Chapter 5. Further motivation of using the CSP/CT EHM for atomic actions is its CSP background, which has been recognized as a superior foundation in an attempt to provide formally sound atomic actions framework (Jalote and Campbell, 1986). Xu et al. (1995)
provide a comprehensive overview of problems and concepts in using atomic action for fault tolerance in concurrent software, such as recovery line (as the state of the system when an atomic action begins and the system rolls back upon detection of an error), test line (barrier of acceptance tests where processes synchronize) and side firewalls (means of confining the communication among participants during the atomic action).

**Atomic actions within the CSP/CT framework**

In Figure B-1 a scheme is depicted that outlines a design pattern for programming atomic actions within presented CSP/CT framework.

![Figure B-1 Three processes participating in an atomic action](image)

In the CSP/CT framework a large data processing (body) of a process can be segmented in sequential subprocesses, which is suitable for separating those parts of processing that should be put in an atomic action with (parts of) other processes. Organized in that way, processes P1, P2 and P3 in Figure B-1 engage in an atomic action by synchronizing executions of their subprocesses (in this example, P13, P22 and P34) by using hypothetic AAchannels (atomic action channels). This way the design environment (the gCSP tool) may keep track of the processes engaged in the action. A collection of the AAchannels in fact declares an atomic action. The network structure
information may be always used for checking if the indivisibility of an atomic action after any system modification is violated.

Initial states of the processes P13, P22 and P34 represent the recovery line for backward error recovery. Test line (acceptance tests) would be performed just before termination of the participating subprocesses. Should be found useful, visualization of the recovery line and the test line would be straightforward. Special provisions for side firewalls for preventing breaching the communication atomicity is not an issue in the CSP/CT paradigm, since the processes may communicate only through channels. Sequential subprocesses within P1, P2 and P3 must pass values to each other through local variables. Apparently, there is a risk of information smuggling if, for instance P13 updates one of the variables later used by P14. But, since the variables are connected to the processes also by the channel mechanism (var-channels, not depicted in the figure), the gCSP tool may easily check if those variables are written to before the acceptance test.

These diagrams capture both backward and forward error recovery schemes. For backward error recovery, exception handling processes associated to the processes P13, P22 and P34 represent recovery blocks. For forward error recovery, handling processes try to remedy the situation as it is at the moment exceptions are raised; the recovery may be performed in several stages (chained handling processes). In both cases (backward or forward recovery), if the recovery is not successful, the last chained handlers throw exception sets outside the processes (in that case, the sequences of subprocesses are terminated) and handlers Handler1, Handler2 and Handler3 may try recovery in ways specific for the P1, P2 and P3 processes. If this is not necessary (or the specific handlers fail), the control comes eventually to the default handler (HandlerAA in this example). In the case of concurrent exception occurrences, handlers guarding P1, P2 and P3 have all necessary information to construct exception hierarchy on basis of thrown exception sets.

To prevent the possibility that one process in the atomic action leaves it (after passing its acceptance test) before the others do so, after the acceptance testing all participating processes should engage in synchronization (for pro’s and con’s see (Romanovsky, 2001)). Using channels for such one multiparty synchronization may be inelegant. The CSP/CT barrier (Hilderink, 2005a) would serve this purpose naturally – a fixed number of processes is required to synchronize their execution at some point before proceeding. For using within the CSP/CT EHM, the barrier also should be a subject to suspension (poisoning).

Fulfilment of the requirements

1. Clear identification of the participants.
   The state of the art in programming atomic actions relies on named participating processes, which limits reusability, when names and number of the participating processes changes. Identification of the processes participating in an atomic action in CSP/CT is graphical: an atomic action is determined by the set of AAchannels. The network builder has all the information to check all necessary consistencies.
Reusability is further reinforced by the identity irrelevance within the CSP/CT networks. Furthermore, visualization of recovery and test lines is straightforward.

2. **Recovering from a situation when one or more participants fail to enter the collaborative atomic actions - the deserter problem.**

   By consulting Figure B-1, it is clear that the deserter problem cannot manifest itself within a CSP/CT network. If it happens that one of the participating processes encounters problems before coming to the recovery line of an atomic action, an exception would be raised notifying the other processes about the problem.

3. **Atomicity of communication from the system’s perspective.**

   As the underlying EHM relies on the gCSP tool to perform all necessary consistency checks, it is the tool that would issue warning on any possibility of breaching the communication firewalls around an atomic action. This property draws from the general advantage of the CSP/CT process orientation, i.e. confinement of all interactions among processes within channels.

4. **Synchronization on acceptance tests.**

   It has been mentioned that barriers, as means of multiway rendezvous synchronization, are proper tool to model (and visualize) the acceptance test lines. Since an atomic action consists of a defined number of participating processes, the tool would generate appropriate barriers for each atomic action within the system.

5. **Uniform error recovery (possibly collaborative) if one or more processes fail on the acceptance test.**

   This, the most challenging research and design issue, is completely taken care of by the properties of the CSP/CT EH Mechanism, as discussed in Chapter 5.

The major deficiency of the outlined framework is already discussed inability of immediate terminating all involved processes upon throwing an exception in one of them (Chapter 5). The other complication is the necessity of explicit synchronization of exception handlers before they engage in cooperative error recovery. It would be incorrect if an exception handler start handling an exception immediately, before the absence of concurrent exceptions is assured. For this reason, additional synchronization primitives (channels or barriers) must be used, which adds to the complexity of the design. The issue of nested atomic actions is not discussed; a satisfactory solution would probably rely on the compositability of the underlying CSP/CT framework.
Appendix C  Some implementation details of the watchdog mechanism

Using the gCSP graphical notation itself as a metalanguage for describing the watchdog construct, the following interpretation can be given:

![Diagram of watchdog construct modelled by the alternative construct with timeout]

Hypothetical rendezvous channels in Figure C-1b (palegrey) represent calling the hit operation on the watchdogs, each of which created from the NominalNetwork, is represented by a reader. All readers and the EmergencyNetwork are composed with an alternative construct running parallel with the nominal network at higher priority. All readers are comm-guarded, except the emergency process which is timeout-guarded.

The way the programmable timer and the real-time clock interrupt serve the watchdog mechanism is illustrated in the following figure. Darkgrey bar represents the value of the programmable timer being decremented each time the real-time clock interrupt routine is activated. This value is in Figure C-1b modelled by TimeOut attached to the EmergencyNetwork. The interrupt routine also decrements variables keeping the remaining time till timeout for each instantiated watchdog (lightgrey bars).
Figure C-2a is a snapshot of the timer register and the watchdogs’ variables just before watchdog 1 is hit. Note that the timer register is equal to the value of the watchdog 1 variable – the shortest in that moment. That is because when a watchdog is hit, timeout of that watchdog resets, and the timer is reprogrammed to the shortest watchdog timeout at that moment (in the \texttt{ReprogramTimeout}). Subsequently the alternative construction restarts.

Figure C-2b shows state of the timer register and watchdogs’ variables just after watchdog 1 is hit. Value of that watchdog variable is reset to its timeout, and the timer starts counting down from the least watchdog variable at the moment (watchdog 2 in this case). In case that a process within the \texttt{NominalNetwork} process does not hit its watchdog before it reaches zero (i.e. the timer decrements to zero), the timeout guard elapses. Depending of the programmed watchdog timeout policy, the interrupt routine detecting this condition deploys one of the intervention levels. That may be interference with the CT scheduler, causing activation of the \texttt{EmergencyNetwork}.

![Figure C-2 Programming the watchdog timer policy](image)
Appendix D  

CTC++ code generation and templates for 20-sim

For each process defined in a gCSP model, a C++ class is created, derived from the CTC++ Process class. The source code organization in directories is flat. The tool generates a directory with the same name as the name of the model. In that directory a .cpp source file for each process is created, named after the process. The .h files of the same names are stored in a directory that contains all header files, called include. An additional file, named gCSPmain.cpp contains the top level network builder and possibly some hardware configurations. The adopted code structure has the advantage of being simple and giving an easy view over the generated source files. However, it means that processes with the same name in different parts of the hierarchy correspond to a single source code. This is favourable as long as the modeller really intends to have completely the same functionality in these processes. As soon as for instance a different message is issued to the screen from these processes, one of them will be overruled by the other.

20-sim CTC++ code generation templates

20-sim code generation templates described in this appendix are a simplified variant of the original implementation (Hilderink, 2005a, p.218). Instead of creating two pairs of .cpp and .h files for each submodel in 20-sim, templates for interfacing 20-sim control code consist of only three source files' templates for each 20-sim submodel: one .cpp file, one header (.h) file and an ASCII .info file, all named after the 20-sim submodel. The .cpp file contains the equations of the model and the .h file necessary interface for integrating in the rest of the compileable resources. The .info file is used to allow matching the input and output variables of the 20-sim submodel with ports of 20-sim code blocks within gCSP.

All three files generated from 20-sim are placed in a separate directory for each submodel. Next to all the directories (named after submodels) has to be a directory containing standard numerical C routines and data structures needed for 20-sim calculations (the name of this directory is common). The common and submodel directories are in turn contained by a 20code directory, to be found next to the gCSP generated source directory.
Samenvatting

In dit proefschrift wordt betrouwbaarheid bepleit als cruciaal onderdeel van softwarekwaliteit. Procesoriëntatie wordt in dit proefschrift gedefinieerd als het zien van een proces als basiscomponent van een dataflowgeoriënteerde softwarearchitectuur. In de voorgestelde variant van procesoriëntatie wordt de betrouwbaarheid verbeterd met een aantal specifieke voordelen van een op de principes van CSP-procesalgebra gebaseerde dataflowgeoriënteerde softwarearchitectuur.

De CSP/CT procesgeoriënteerde modelleer- en programmeeromgeving voor besturingstoepassingen is door dit werk verrijkt met een aantal elkaar aanvullende instrumenten voor de verbetering van de betrouwbaarheid van concurrent software. Naast de verbeterde ontwerpmethodologie is de belangrijkste bijdrage een grafische CASE-tool, genaamd gCSP, die ondersteunt bij het modelleren, visualiseren en beheren van softwaremodels van groeiende complexiteit. Door eerder ontwikkelde modellen te manipuleren, gebruikt gCSP de formele basis van de methodologie om formele verificatie van de ontwerpen mogelijk te maken via een automatisch gegenereerde formele specificatie in de CSPm-taal. De efficiëntie van de productie en de betrouwbaarheid van het eindproduct – de implementatiecode – wordt substantieel verbeterd door de automatische generatie van C++-code die gebruik maakt van de CTC++ bibliotheek voor concurrent programmeren. In dit proefschrift wordt middels voorbeelden en mechatronische illustraties gedemonstreerd dat het procesgeoriënteerde CSP/CT-framework geschikt is om diverse bewezen betrouwbaarheidstechnieken te implementeren, zoals concurrent exception handling, N-version programming, logging, monitoring en verschillende varianten van watchdogs.

Dit proefschrift bepleit tool-based visual programming, het gebruik van de toenemende capaciteiten van computers om de overhead van betrouwbaarheid voor complexe softwaresystemen te compenseren, het onderscheiden van diverse aspecten van de software tijdens de modelleerfase en het ontwikkelen van een ingenieursdiscipline voor softwareontwikkeling gebaseerd op mathematische voorspelbaarheid. Deze voorstellen hebben tot doel de kwaliteit van (embedded) software tijdens de ontwerpfase te verbeteren.
Sažetak


CSP/CT proces-orijentisano okruženje za modelovanje i programiranje sistema automatskog upravljanja je u ovom radu obogaćeno raznovrsnim međusobno dopunjujućim sredstvima za povećanje pouzdanosti konkurentskog softvera. Uz poboljšanje tehnologije konstruisanja, glavni proizvod je grafički CASE alat, nazvan gCSP, koji olakšava modelovanje, vizualizaciju i rukovanje modelima softvera koji su uopšteno odlikovani stalnim porastom složenosti. Pri manipulaciji razvijenim modelima, gCSP alat koristi formalizovanu zasnovanost predložene metodologije u cilju formalne verifikacije softverske konstrukcije preko automatizovanog generisanja formalnih specifikacija u jeziku CSPm. Efikasnaja produkcija i pouzdanost u konacni proizvod procesa konstruisanja – implementacioni programski kod – suštinski su unapredeni automatizovanim generisanjem programskog koda prilagođenog CTC++ implementacionoj biblioteci za konkurentsko programiranje. U ovoj disertaciji je ilustrovana, razrađena i na primerima i mehatroničkim eksperimentalnim sistemima pokazana pogodnost procesno-orijentisanog CSP/CT okruženja za uključenje različitih afirmisanih instrumenata pouzdanosti: konkurentsko razrešavanje izuzetaka (engl. *concurrent exception handling*), programiranje višestrukim verzijama (engl. *N-version programming*), registrovanje i nadzor (engl. *logging and monitoring*) i nekoliko varijanti kontrolnih agenata (engl. *watchdogs*).

Ova se disertacija zalaže za afirmaciju programiranja zasnovanog na vizuelnim alatima, ulaganje permanentno uvećavaćih računarskih resursa u pouzdanost složenih softverskih sistema, razdvajanje raznorodnih aspekata funkcionalnosti softvera u fazi modelovanja i promociju razvoja softvera u istinsku inženjersku disciplinu zasnovanu na predvidljivost kao rezultatu matematički baziranog konstruisanja. Sve ovo zajedno predloženo je u cilju uvećanja kvaliteta (u napredne uređaje ugrađenog – engl. *embedded*) softvera već u vreme ranih faza konstruisanja.
Acknowledgements

Holding this thesis suggests that all those efforts in past five years turned into a successful pursuit. These few pages are dedicated to crediting some people for those things that directly contributed to that pursuit. Building a software design methodology and the supporting tools is not an one-man-job. However, the dissertation is of only one author, which is not quite fair.

In the first 30 years of this life I had (or found?, or chose?) four couples of parents, which I want to thank first.

Some mystics (and scant scientists) argue that a person chooses where and when to be (physically) born. For some reason, my choice was Baljevac in Serbia, where I found my biological parents Vera (Mamica) and Slobodan (Tatica). At the University I chose to learn about engineering control, and had there my engineering parents Srbijanka (Srba) and Miroslav (Silja). Then I chose to refine my being academically further, and by that way I found a chance to develop my being also spiritually in parallel. For the former I had my spiritual parents Ivana (Andeo 1) and Slobodanka (Andeo 2).

The gender misbalance of the last couple was balanced with my academic parents Job (Jobje) and Jan (Jantje), who of course are getting the greatest attention in the sequel. There is to be found a word that has the clear meaning only for me: “Job&Jan”.

At times when I lacked quite a bit of dependability, they provided a lot of safety for me. Later, after recovering my dependability, they acted quite fault-tolerantly. Only at the time of starting writing this thesis and fine-tuning what is described in it, I heard what a PhD degree is about. It’s said that it is a proof of someone’s capability to pursue academic research independently (or “to conduct independent scientific research”, as stated in the letter from the dean that I received only during typing this page).

I wish to thank Jan Broenink further for supporting my PhD research according to the given definition and for managing the research project. Also, it was Jan’s initiative to dub the tool—the most characteristic outcome of this effort—gCSP. Isn’t he then the godfather of our little baby?

I am indebted to Job further for seeing substantial things in sometimes cloudy situations. And for cherishing aesthetics in the things we as engineers produce. And for supporting investments in marrying research and the beauty.

And a few more words about Job&Jan as a collective. In the course of obtaining their approval of the thesis, the quality of the story improved dramatically! (By the way, I strongly believe that the appreciated reader has no doubts that this thesis is written in English; but without having really numerous corrections of these two guys, it could happen that the reader would have some doubts.) That is perhaps obvious from these acknowledgments – this was not corrected by Job&Jan.

The quality of the text got an increment also from the graduation committee members, among which professor Arie van Deursen to my delight was particularly productive.
While being close to the region of management acknowledgments, I want to thank people and institutions that made working on these matters possible. Those are the Technology Foundation STW and its embedded systems research program PROGRESS, established by the Dutch organization for Scientific Research NWO and the Dutch Ministry of Economic Affairs. They funded the project “Design framework for heterogeneous real-time embedded systems”, officially called TES. 5224 (for whose name I realized rather late that it stemmed from “52-weeks-a-year-24-hours-per-day-dedication-expected”).

I wish to thank (the people who created and lead) the University of Twente for its excellent scientific facilities, a beautiful campus, intensive sportive and cultural activities, the superfast Internet connectivity and equally superb phenomenon of Campusnet, which is all undoubtedly important for pursuing a jovial and fruitful research.

Now, credits to a very special person: My Programmer Mister Geert Liet, my (ex-)spouse regarding our gCSP baby! How many times did he hear from me “...well...can you...one more thing...to add?” He also gave a lot of creative input into that development and related texts. How many different things, just to make the application decent, which will be not explicitly reported as a feature, nor noticed by the end user, neither even me?

It is impossible to forget crediting Geert’s successor Matthijs ten Berge, who incredibly quickly picked up and polished Geert’s heritage, providing me with precious support in the last stage of the development.

And then my “embedded” fellows, strongly embedded in my work in Twente: Bojan, Gerald, Peter and Marcel Groothuis (“klein”), who started his PhD studies exactly when I stopped; however, he found time to help me with our set-ups. Another kind of irreplaceable fellows are the hi-tech supporting guys: Marcel Schwirtz (“groot”), Gerben and Gerrit – who would certainly be my third “paranimphe” if the regulation allowed that. Under “the mechatronici” I thank collectively all fellows living together with us “the embedded” and giving us an inexhaustive amount of enthusiasm for explaining what “we embedded” in fact are doing.

I benefited a lot from collaboration with several students I supervised. A lot of work of Thiemo van Engelen and Mark Huijgen is part of this thesis. The others are Peter Visser (the one from the previous passage when he was “only an ordinary” student), Robin Stephan, Hans Hendriks, Ceriel Mocking, Bas Goudriaan, Laurence Smith and Martin Ros. By working with them I learned a lot about steering joint work, its successes and failures.

Why, oh why the secretaries come so often at the end, who are so supportive in all we do? Well, Carla is one exception(al)! So! And then can come other teaching staff of our group, whose enlightening contributions are ingredients of my new title.

Certainly, in becoming a “Doctor of Philosophy” a significant contribution comes from the housemates. In my five PhD-research years, chronologically: Bojan (yes, the one also in the category “fellows”), Dejan (Raša), Nataša (Trinity), Tanja (Mala), Ana (Starters) and Jelena (Lane). I have so much to thank you guys! But look the number of pages already! Don’t be angry that I pick up the flanks of the club as your representatives. Yes, those two „extremists”:
Bojan is the most criticizing, honest, curious, direct discussing-feedback-giving person I have ever met. I can only hope that the life will bring closer more people like him. Herewith I challenge all appreciated readers to try to achieve such a high standard! I encourage trying sincerely, you would be provided with my regular feedback.

Jelena and Bojan have however more in common than it may seem: for instance my insatiable passion towards their notebooks, which were so important in some critical moments along the way. Thanks for understanding, especially to Jelena, who always was so forgiving for the occupation of her email-checking-device while I was finding resorting moments in watching the Star Trek episodes. Thanks also for supporting the last stages of creating this book.

I thank all the 3HA3(X) mailing list members and their families. Live long and prosper!

Outside the PhD students (with a numerous Serbian-speaking community within), on the bow of my integration in the Dutch-speaking community was Jeroen Latour, the captain of our Dutch-lust group: Svetlana, Henning, Tanya, Vojkan, Jelena (Lane). Our last assignment with me inside the club was the Dutch translation of the summary of the thesis. While regretting going outside this charming group, I wish Jeroen to see the obvious impossibility to be outside – not only this company, but also the community of PhD students.

Special thanks to the people in my company Neopost, for their courage to admit an idealistic academic fellow, for their support in finishing this book and all-in-all a warm welcome into my new collective.

And outside the Netherlands, I thank all those refreshing people and places, specially in Serbia, among numerous friends and relatives particularly Dragica (Seka): my sister, my dentist, my lifelong comrade.

Thanks to Microsoft that has begun building interestingly reliable PC tools. Believe or not, this thesis is written in MS Word 2003! So, the mission accomplished, of course with experiencing various kinds of the performance and reliability qualities (and also some safety aspects, when it comes to the delivery deadlines!).

Shoulder to shoulder in the big software world: thanks CLP for thriving messaging the world about the power of the bond-graph modelling and for making it so more attractive with their visualization skills. Earn a lot and prosper!

Lastly, I express my wishes that this text be found useful for more people than myself and I am using this chance to communicate that I am in constant quest for new parents that would guide me through not yet (enough) explored dimensions of the All-That-Matters Existence. You are warmly welcome to subscribe!

Yours DJ
djov@consultant.com
Enschede, 20-02-2006
About the author

Dusko Jovanovic was born on 6\textsuperscript{th} of May 1975 and grown up in a small village Baljevac in the vicinity of Belgrade in Serbia. After finishing the high school of Electrical Engineering “Nikola Tesla” in 1994 (in Computer Engineering) and obtaining engineering degree from the University of Belgrade in 2001 (in Control Engineering), he joined department of Control Engineering at the University of Twente in May 2001 as PhD student to work on a research project into design process of embedded control systems.

Although principally focussed on issues of embedded software, by working on the diverse project of system engineering for smart systems, he was assigned with several mechatronic development activities as well as teaching responsibilities.

In 2003 he found himself enthusiastic in reasoning on dependability issues of mechatronic devices. This interest extrapolated into a somewhat broader target field of computer-supported surroundings to be found dominating the XXI century.

Dusko started his post-pure-academic career in January 2006 as Software Engineer Trainee towards Software Coach with the R&D department for embedded software at Neopost Technologies in Drachten, on the very north of the Netherlands. There he exposed himself to the interaction complexity in making sophisticated machines specialized for automatised papermail flow and document handling.

Photoseries “PhD evolution in Twente 2001-2006”
References


Aceto, L. and A.D. Gordon (Eds.) (2005), *Algebraic Process Calculi: The First Twenty Five Years and Beyond*, BRICS Notes Series, University of Aarhus, Denmark.


References


References


Design Tools project (2001-2005a), PROGRESS/STW TES.5224 User's Committee Meetings.


Eglence, M. (2003), Design and realization of a safe control system for a parallel manipulator, MSc thesis no 010CE2003, Control Engineering dept, University of Twente, NL.

ESSI-SCOPE project (1997), http://www.cse.dcu.ie/essiscope/sm2/9126ref.html, Dublin City University, Ireland.


References


Hendriks, J.P.A. (2001), Realization of Tool Support for CSP Diagrams and Generation of Concurrent Java Software, MSc thesis no 02SR2001, University of Twente, NL.


References


Huijgen, M.C. (2005), Patterns for dependable and distributed embedded control, MSc thesis no 032CE2005, University of Twente, NL.

Huima, A. (2005), Proving software quality - validating test software, Symposium on Embedded Software Quality, 31. August, Institute for mathematics and computer science, CWI Amsterdam, NL.


References


References


OMG (2001), *Model Driven Architecture (MDA)*.


SEI (2005), *Capability Maturity Model® Integration (CMMI®) Overview*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, US.


Smart Surroundings project (2005), University of Twente, http://smart-surroundings.org.

Smith, L. (2002), JIWYNET: A project to implement the control of the mechatronic system JIWY over a computer network, Internship report no 017CE2002, Control Engineering dept, University of Twente, NL.


Van Amerongen, J. (2005), Definitions in control, personal communication.


Van Drunen, J.M. (2000), Realization of link drivers implementing CSP-channels on 20-controller, MSc thesis, University of Twente, NL.

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


Verhulst, E., (2005), Open License Society - Unifying and systematic system development methodologies with trustworthy embedded components, presentation at Communicating Process Architectures 2005, Eindhoven, NL.


Once a year, all engineers should be put on a life support system they helped design.

Bob Perrin

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies.

C. A. R. Hoare

We can write good or bad programs with any tool. Unless we teach people how to design, the languages matter very little.

David Parnas

No architect charged with designing an earthquake proof building would build it and then wait for an earthquake to see if the design was correct.

Guy Broadfoot
The paper used for printing was BIO TOP 3:

BIO TOP 3®

The natural white, elegant offset paper.

100% chlorine-free bleached
No optical brightening agents
Pleasant to touch
Meets all requirements regarding uses for further processing, including stamping and embossing.

BIO TOP 3® is exclusively manufactured from fibres sourced from sustainable forestry. Without the use of chlorine and chlorine compounds in pulp bleaching and optical brighteners in paper production, the production process meets the highest environmental standards. Optimum energy efficiency, low water consumption and effluent treatment in state-of-the-art water treatment plants are standard procedure.

BIO TOP 3® is the first paper that is bleached exclusively with oxygen and oxygen compounds to an attractive whiteness degree of 89%.

(The certificate origin: paper suppliers of Wöhrmann Print Service)