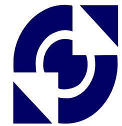
SystemCSP

a graphical language for designing concurrent component-based embedded control systems

Graduation committee:

prof.dr. H. Brinksma, prof. dr. ir. M. Aksit, prof.dr. H. Corporaal, prof. dr. A. van Deursen, prof.dr. ir. J. van Amerongen, dr.ir. J.F. Broenink



Control Eengineering, Centre for Telematics and Information Technology (CTIT), Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente

CTIT PhD thesis series, ISSN: 1381-3617

CTIT number: TR-CTIT-07-55

This research was financially supported by the PROGRESS program of STW, the Dutch Technology Foundation

ISBN: 978-90-365-2573-2

Cover design by Bojan Orlic

Copyright © 2007 by B. Orlic

No part of this work may be reproduced by print, photocopy, or any means without permission of the author.

Printed by PrintPartners Ipskamp, Enschede, The Netherlands

SYSTEMCSP

A GRAPHICAL LANGUAGE FOR DESIGNING CONCURRENT COMPONENT-BASED EMBEDDED CONTROL SYSTEMS

DISSERTATION

to obtain
the doctor's degree at 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, 20th September 2007 at 16.45h

by

Bojan Orlić

born on 11th February, 1976.
in Bor, Serbia

This dissertation is approved by:
prof.dr.ir. Job van Amerongen, the promotor
dr.ir. Jan F. Broenink, the assistant promotor

TABLE OF CONTENTS

ACKNOWLEDGMENT XI

SUMMARY XIII

1 INTRODUCTION 1

1.1 CONTROL SYSTEMS APPLICATION AREA 2
1.2 SOFTWARE DEVELOPMENT PRACTICE 5
1.2.1 Structured approaches *5*
Visual programming 6
Formal verification 7
Concurrency 7
1.3 SYSTEMCSP APPROACH 9
1.3.1 Problem statement *9*
1.3.2 Local context *10*
1.3.3 Brief overview of SystemCSP *11*
1.4 OUTLINE OF THE THESIS 13

2 BACKGROUND 15

2.1 CSP AND ITS IMPLEMENTATIONS 15
2.1.1 Basic elements *15*
2.1.2 Grouping *17*
2.1.3 Event prefix and sequential operators *18*
2.1.4 Parallel operator *18*
2.1.5 Choice operators and constructs *19*
External (deterministic) Choice 19
Conditional (IF) choice 21
Internal choice 21
2.1.6 Finite state machine based designs *22*
2.1.7 Priorities *22*
2.1.8 Exceptions *23*
2.1.9 Discussion *24*
2.2 VISUALIZING CONCURRENT SYSTEMS 25
2.2.1 UML *26*
Critique 30
2.2.2 Finite State Machines *31*
2.2.3 GML *32*
Sequential 34
Parallel 34
Choice 35
Grouping 35
Recursion 37
Representing FSM-like designs 37
Priorities 38
Exceptions 39
Design process 39
Data-flow orientation 39
2.2.4 Discussion *41*
2.3 COMPONENT ENGINEERING PRACTICE 42

2.3.1 Component frameworks ... 43
2.3.2 Interaction management – the notion of contracts and connectors 45
2.3.3 Discussion ... 47
2.4 CONCLUSIONS ... 48

3 SYSTEMCSP ... 49

3.1 BASIC ELEMENTS ... 50
3.1.1 Events ... 50
Event-ends ... 50
Event prefix ... 51
3.1.2 Processes ... 51
Process labels ... 51
Interacting processes ... 52
Process blocks ... 53
Non-interacting process blocks ... 54
3.1.3 Comments ... 55
3.2 CONTROL-FLOW ORIENTED ELEMENTS ... 55
3.2.1 Elements related to CSP operators ... 55
Hiding and Renaming Operators ... 55
Conditional choice ... 56
Preconditions and postconditions ... 57
Guarded alternative ... 57
Start and Exit Control Flow Elements ... 58
Specifying synchronization alphabet ... 60
Exception Handling ... 62
3.2.2 Supervision elements ... 63
3.2.3 FSM-like diagrams in SystemCSP ... 64
3.2.4 Hiding as a filter of possible interactions ... 64
3.2.5 SystemCSP sequence diagrams ... 66
3.3 INTERACTION ORIENTED ELEMENTS ... 68
3.3.1 Binary compositional relationships ... 68
3.3.2 Synchronization events and data flow ... 71
3.3.3 Refinement operator ... 72
3.4 INTERACTING COMPONENTS ... 72
3.4.1 Structural units ... 72
Components ... 72
Ports and interfaces ... 73
Interaction Contracts ... 74
Contexts ... 75
3.4.2 Interaction and control-flow diagrams ... 76
3.4.3 Discussion ... 79
3.5 DISTRIBUTED SYSTEMS – ALLOCATION ... 80
3.6 RELATED WORK ... 82
3.6.1 SystemCSP vs. UML ... 82
3.6.2 SystemCSP vs. GML ... 85
3.7 POSITIONING ... 86
3.8 IMPLEMENTATION ... 87
Metamodel of the notation ... 87
Mapping to software domain ... 87
3.9 CONCLUSIONS ... 88

4 REAL-TIME AND CSP 91

4.1 SPECIFICATION OF TIME PROPERTIES 91

4.1.1 Discrete time event ' tock' 91

4.1.2 Timed CSP 92

4.1.3 Time specification in SystemCSP 94

Time specification in control flow diagram 94

SystemCSP time diagrams 95

Timing subsystem 96

Watchdog Design Pattern 97

Timed interrupt operator 98

Timeout operator 100

4.2 REAL-TIME IN THE IMPLEMENTATION OF CSP-BASED SYSTEMS 102

4.2.1 Identifying problems 102

Origin of time constraints in implementation of control systems 102

Scheduling theories 103

Fundamental mismatch between CSP and classical scheduling – communication induced precedence constraints 104

Influence of assigning priorities on analysis 107

4.2.2 Classic scheduling approach 108

Priority inversion and using a buffer process to solve it 108

Absolute versus relative specification of priorities 109

RM 110

EDF 110

EDF* 112

Design with rendezvous channel communication, convert them to shared data objects when necessary 112

4.2.3 Event based approaches 113

Event-based scheduling 113

Equivalent automaton 114

Mapping time properties to equivalent automata 117

4.3 CONCLUSIONS 120

5 DESIGN PATTERNS IN SYSTEMCSP 121

5.1 COMMUNICATION PATTERNS 122

Design patterns – goals and ideas 122

Design description 123

Remarks 124

5.2 PATTERNS RELATED TO COMPONENTS 125

5.2.1 Inter-component function calls 125

Design patterns – goals and ideas 125

Design description 125

Remarks 127

5.2.2 Switch interaction contract 128

Design pattern – goals and ideas 128

Design description 128

Remarks 129

5.2.3 Diversity interfaces 130

Design pattern – goals and ideas 130

Design description 130

Remarks 131

5.3 PATTERNS RELATED TO CONTROL SYSTEMS 132

5.3.1 Layered structure of control systems *132*
Design pattern – goals and ideas 132
Design description 132
Remarks 134
5.3.2 Supervision and monitoring layer *135*
Design pattern – goals and ideas 135
Design description 135
Remarks 137
5.4 FAULT TOLERANCE PATTERNS 137
5.4.1 Recovery block *138*
Design pattern – goals and ideas 138
Design description 138
Remarks 139
5.4.2 Watchdog design pattern *139*
Design pattern – goals and ideas 139
Design description 140
Remarks 141
5.4.3 Replica Management *143*
Design pattern – goals and ideas 143
Design description 143
Remarks 146
5.4.4 Exception Handling *147*
Design pattern – goals and ideas 147
Design description 147
Remarks 148
5.4.5 Checkpointing *148*
Design pattern – goals and ideas 148
Design description 148
Remarks 150
5.5 CONCLUSIONS 150

6 PRODUCTION CELL SETUP 151

6.1 PRODUCTION CELL SETUP 152
Purpose and properties 153
Detailed description 154
Structural deadlock 155
6.2 OTHER WAYS TO DESIGN SOFTWARE 156
6.2.1 Time-table based approach *157*
6.2.2 GML/CT library design *157*
6.2.3 POOSL *160*
6.3 SYSTEMCSP - DEVICE-ORIENTED DESIGN 161
6.3.1 System structure *161*
Basic structure 161
Adding monitoring layer 162
Assigning priorities 163
6.3.2 Loop controllers *163*
Structural part 163
Behavior definition 1 - Position based loop controller 165
Behavior definition 2 - Velocity based loop controller 166
6.3.3 Sequence control *167*
Basic blocks 167

Dependencies 168
Generic sequence control 170
Time based schedule 172
Sensor based sequence control 173
Event based sequence control 173
6.4 INTERACTION CONTRACT BASED DESIGN 177
6.5 CONCLUSIONS 181

7 CONCLUSIONS AND RECOMMENDATIONS 183

7.1 CONCLUSIONS 183
7.1.1 Summary 183
SystemCSP core elements 183
Real-time support 184
Design patterns in form of reusable interaction contracts 185
Complex control system test case 185
Implementation issues 185
7.1.2 Evaluation 185
Mapping to some existing formal verification method 185
Support for specification and analysis of time properties 186
Support for component-based development 186
Expressiveness and readability 186
Scalability 187
Unambiguous interpretation 187
Applicability to complex control systems 187
Conclusion 187
7.2 RECOMMENDATIONS AND OPEN ISSUES 188
Tooling 188
Mapping to hardware domain 188
Distribution 188
Simulation 189
Software implementation 189
Usage 189

A METAMODELS 191

A.1 METAMODELS AS BASIS FOR CODE GENERATION 191
A.2 METAMODEL OF SYSTEMCSP DESIGN DOMAIN 193
A.2.1 Event ends 194
A.2.2 Control flow elements 197
Elements with direct mapping to CSP 197
Start/exit control flow elements 198
Advanced fork/join control flow elements 200
A.2.3 Binary compositional relationships 202
A.2.4 Components and processes 203
A.2.5 Supervision elements 205

B SOFTWARE FRAMEWORK DESIGN 207

B.1 WHY YET ANOTHER CSP LIBRARY? 207
B.2 EXECUTION ENGINE FRAMEWORK 209
Brief overview of execution engines 209
Discussion 210
Four- layer execution engine architecture 210

Allocation 212
Priority assignment 212
Components, processes and variables 214
Function call-based concurrency inside components 216
B.3 IMPLEMENTING CSP EVENTS AND CHANNELS 220
Event synchronization mechanism 220
Solving the mutual exclusion problem 222
Channels capable for multidirectional communication 225
Distribution/networking 225
B.4 OTHER PARTS OF THE SOFTWARE DESIGN 226
Exception handling 226
Logging 227
Tracing 228
B.5 CONCLUSIONS 228
REFERENCES 231
CURRICULUM VITAE 241

Acknowledgment

The author would first like to thank his supervisors, colleagues and students at the Control Laboratory. Next, the author thanks his family and friends for their support during this work. Finally, the author thanks to life.

Bojan Orlić

Summary

Realization of embedded control systems is a complex task. Increasing part of this complexity is nowadays located in the design and implementation of software that runs them. A major source of difficulties is the limitation of the average software developer to understand and design complex behavioral scenarios. A big part of this complexity comes from the interaction of concurrently existing entities.

Similar to the way energy-flow based modeling is used in bond-graph theory as a common view in multi-domain physical system modeling, in our approach concurrency is viewed as a glue layer that relates different domains and views. Concurrency used in a structured way should lead to systems that are simple to design and understand, easy to distribute, reconfigure and reuse.

The aim of this thesis is to develop a graphical design specification language that can offer a structured way of handling concurrency. A structured way of using concurrency is expected to raise the abstraction level away from concurrency problems. In that way, the introduced design specification language should become a vehicle for reducing complexity in inherently concurrent systems (e.g. complex control systems).

Our approach is not to invent the wheel all over again. Instead, we want to build upon the existing body of knowledge provided by relevant formal methods.

SystemCSP is based on the principles of both component-based design and CSP process algebra. It is applicable for specifying, documenting, visualizing and formal verification of component-based designs. It provides a way to visualize architecture, behavioral patterns of components, intra–component interactions and execution relations among components.

First, the elements of the language are introduced. Than, this core set of language elements is extended with elements dedicated to the specification of time properties. The thesis also offers some insights and some possible approaches for analysis and implementation of real-time systems. The core of the notation is further extended by introducing a set of higher-level primitives in the form of reusable design patterns (see Figure 1-6). These patterns can be used in designs as basic building blocks, extending in that way the vocabulary and raising the level of abstraction of design processes.

The usability of the notation for specifying interactions in embedded control systems is tested by designing software for a complex control setup consisting of several cooperating devices. This test case is a step towards industrial-strength setups.

SystemCSP seems to be convenient way to specify interactions in concurrent component-based embedded control systems.

1 Introduction

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

Bertrand Russell

Making embedded control systems is a multidisciplinary activity. It includes physical system modeling in various domains (e.g. mechanical, electrical, chemical, and thermodynamic), designing control algorithms, and implementing them in software programs. This research is focused on the software implementation part.

Compared to conventional programming, the software development in area of embedded control systems is somewhat specific. The embedded control systems application area does introduce somewhat more stringent constraints. For instance, there is a need for more rigorous verification of functional and timing properties. A need for performance optimization is additionally increased due to mass-production and related needs for minimizing software/hardware costs per unit and minimizing time-to-market.

Existing control-domain related CAD tools do provide solid code generation support for the software implementation of control laws. However, building complex control systems, where event based interaction between concurrently existing components is very important, is not primary concern of such tools. Attempt to apply the methods, the design specification languages, and the tools from general software development, turns out to be problematic with respect to timing predictability, structuring concurrency, minimizing costs, and optimizing performance.

This research introduces SystemCSP, a graphical language for design specification of concurrent, component-based embedded control systems. SystemCSP is designed to be an expressive, readable, and structured way to design interactions of concurrently existing components. It is based on the CSP formal algebra. The relevant application area for this language is in practical implementations of distributed real-time control systems. The proposed design specification language can, as well, be used in other software development domains where concurrency is inherently present and where focus is on interaction of concurrently existing components.

This chapter starts with introducing some of the key concepts of the research context. Throughout this chapter, instead of providing strict definitions, the chosen approach is to provide descriptions that will allow a reader unfamiliar with the used terms to grasp their meaning and create intuitive representation of the key notions. Section 1.1 of this chapter briefly identifies some notions from the control systems application area. Brief description of the current state of the art, concepts and problems in software engineering area is the topic of section 1.2. Section 1.3 attempts to address issues like: somewhat more detailed problem statement, brief

exploration of previous attempts to solve it, setting demands on language features and sketching the basic ideas and principles behind the SystemCSP language. Finally, section 1.4, gives a short outline of the complete thesis.

1.1 Control systems application area

The success of human evolution is mainly due to human's ability to understand, model, control and adapt their environment.

Modeling is a process of creating an abstracted description of reality. The aim of the modeling process is to identify a competent model - the simplest model that captures, with sufficient accuracy, all aspects of the system behavior relevant for the problem at hand. The purpose of the modeling process is gaining insight in the way real entities behave. This insight can later serve as a basis for controlling the modeled system by manipulating some of its variables.

A *control system* (controller) can be seen as a component added to a system (plant) that needs to be controlled. The essence of control is to maintain the desired values of certain system variables by manipulating values of input variables of the controlled system.

A *control system* interacts with its environment via various *sensors* and *actuators*. Sensors convert physical signals to signals understandable by the *control system* (digital quantities in case of computer-based control). Actuators (motors, valves and so on) perform the transformation in the opposite direction (see Figure 1-1). In Figure 1-1 the different types of arrow symbols are used for digital and analogue signals.

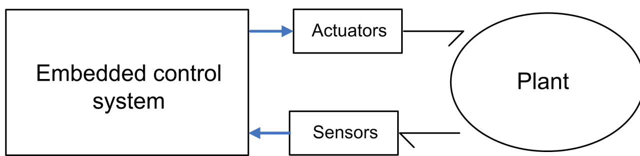


Figure 1-1 Typical control system (Figure based on (Broenink and Hilderink, 2001))

At first, design models of plants and control systems were captured on paper-like media and controllers were realized using mechanics and analog electronics. Application of computers brought more flexibility in the areas of modeling and real world implementation of control systems. With computers, it became easily possible to adapt parameters and algorithms of control systems during system usage. An even more important gain is the opportunity to easily simulate the behavior of a complete system and thus obtain valuable insights into its behavior while remaining purely in the virtual world.

Simulation is the process of exploring the behavior of some system through the execution of its model according to the related operational semantics. Different initial conditions lead to different traces through the state-space of a simulated system. System parameters can be modified in a controlled way and their influence on the overall system behavior can be estimated.

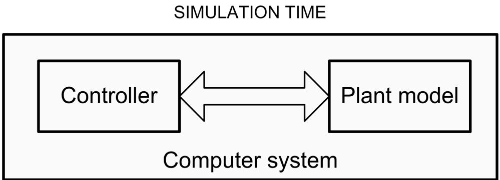


Figure 1-2 Simulation

Figure 1-2 illustrates co-execution of the model representing a control component ('Controller block' in Figure 1-2) and the one representing the controlled system ('Plant model' block in Figure 1-2). In simulations, the time flow is a property of the simulated system and as such independent of real clock time.

Unlike in simulation, in the interaction with a real plant (see Figure 1-3) the control system has to keep pace with the progress of real time. The combination of software and hardware that is implementing the control system, should be designed to guarantee proper temporal properties of the complete system. Simulation can give some insight in the time behavior of the system. Still, only real-time analysis can guarantee satisfaction of temporal constraints. A special branch of computer science ('*real-time* systems' and related 'real-time scheduling theories') is dedicated to the task of real-time analysis.

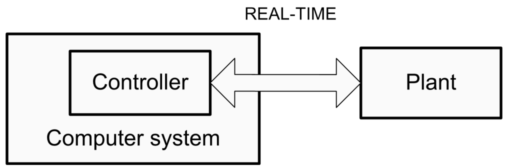


Figure 1-3 Real system

In real-time systems, "correctness of the system depends not only on the logical result of the computation but also on the time at which results are produced" (Buttazzo, 2002). In those systems, the response should take place in a certain time window. Real-time is not about the speed of the system, but rather about its relative speed compared to the required speed of its interaction with the environment. Rather then fast, the response of those systems should be predictable in sense that a guarantee can be given that the response will fall in the predefined time window. A fast system will not be real-time if its environment is requiring a faster response. A controller implemented with slow computer system can work in real-time if it is faster than the response that the controlled plant requires.

Concurrency is one of the most essential properties of the real world. We can perceive that many activities take place simultaneously. Obviously, a (control) system and its relevant environment (plant) exist concurrently. In fact, both plant and control system are often decomposed into smaller parts existing concurrently and cooperating to achieve a desired behavior. The main source of complexity in designed systems stems actually from the simultaneous (concurrent) existence of many objects, events and scenarios. Better control over the concurrency structure,

should therefore automatically reduce the problem of complexity handling. Thus, a structured way to deal with concurrency is needed.

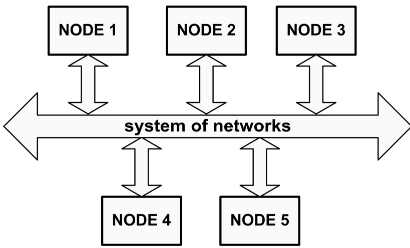


Figure 1-4 Distributed computer platform

This research puts focus on concurrency applied to control systems implemented on top of distributed computer platforms. By the term *distributed computer platform*, we will assume a system built on top of a hardware topology consisting of several computing nodes interconnected via some kind of network, as illustrated in Figure 1-4. In the control systems application area, the most commonly used network interconnections are fieldbuses. The term *fieldbus* is a common name for a set of various serial data communication protocols used in control and instrumentation system architectures.

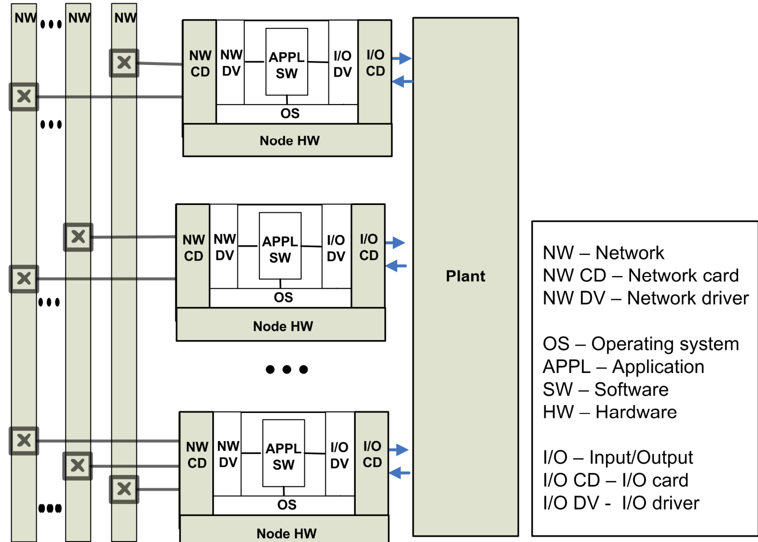


Figure 1-5 Control system based on distributed computer platform

In distributed systems, tasks residing on different nodes execute truly in parallel. For studying concurrency phenomena (deadlock, livelock, race hazard, etc.) of a system, it is not relevant whether its concurrent execution is truly parallel or parallelism is only mimicked via some kind of CPU scheduling. However, for studying the real-time behavior of the system, issues like allocation of tasks to nodes, processing capabilities of the nodes, network delay times and applied scheduling algorithms become relevant.

Control systems as the one depicted in Figure 1-1 are often implemented on top of distributed computer systems (see Figure 1-4). Distributed computer system specialized for the implementation of control systems is typically based on a hardware/software platform that has a topology like the one depicted in Figure 1-5.

In addition to network interconnections between nodes, it contains input/output (I/O) interfaces as points where a control system can use actuators to exert influence on a plant, or sensors to observe changes in some of the plant's signals.

1.2 Software development practice

Since its very beginning, software development is located somewhere in the middle between art, craft and engineering. A major source of difficulties is the limitation of the average software developer to understand and design complex behavioral scenarios.

The current state of the art in the software development area is still marked by the *software crisis* phenomena (Dijjkstra, 1972). *Software crisis* is a term used to denote the incapability of existing methodologies, programming languages and tools to offer adequate support for development of complex software programs.

Ever increasing demands on additional features and performance resulted in increased complexity of developed software systems. There is a constant need to decrease production costs and development time. Teamwork does not scale well, because partitioning into subsystems and their later integration is not always trivial. As a result, software projects are often late and adding more man power makes them often even more late. Verification and debugging of complex systems is difficult. (Keding, 2004)

In the embedded system area, due to cost-efficiency needed for mass production, resources used per product are minimized. This results in an increased effect of the software crisis. Additionally, in embedded systems, non-functional requirements like: performance, reliability, maintainability, safety, fault tolerance, security and power consumption are often of equal or higher relevance then just producing correct results. In addition, standards and algorithms applied are constantly changing, necessitating implementation of flexible and upgradeable architectures.

All these problems illustrate a need for introduction of a novel design methodology. The design methodology should offer a way to structure complexity of the system in a way that efficiently utilizes human comprehension capabilities. A good idea could be abstracting away from the hardware/software development domain, allowing in that way that most of the design is performed at system level and postponing hardware/software allocation to later phases.

1.2.1 Structured approaches

All major advances in software development are actually in some way extending human capabilities to deal with complexity. Most often, this is achieved by raising

the abstraction level and introducing some notion of hierarchy. In the beginning, programs were written in low-level machine and assembly code. Subsequently the view of a programmer was lifted to higher abstraction levels in several phases: the concept of *operating systems,* the introduction of high-level programming languages, and finally, the modern *object-oriented* approach. *Object-oriented programming* (OOP) has added, to the world of software development, many concepts comparable to the way the human mind operates.

Complex software implemented by an immediate 'coding and fix' ad-hoc approaches is bound to fail. Often it is compared to making a building without any prior plans and calculations (Booch, 1993). Like a properly designed building has a certain architecture corresponding to the requirements of the end users, the same can be said for a properly designed software system. Instead of the 'nothing works until everything works' approach, iterative incremental design is often suggested (Booch, 1993).

One of the structured approaches to software design is based on reusing *design patterns* as common solutions to recurring problems. Design patterns (Gamma et al., 1994) systematically name, explain and evaluate important and recurring designs in order to capture them in a form that can be reused effectively. Besides this design description, a pattern contains a description of the problem and the context where it is applicable, as well as the results and trade-offs emerging from its application.

A *software architecture* is a set of high-level design decisions regarding the employed components and the way they are composed and connected. Design decisions left to downstream architects and implementers are not considered part of the architecture on the current abstraction level (McConnell, 2003).

Software architectures are often based on components. A *component* is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition (Szyperski, 1998).

Thus, a new design methodology, that aims to respond to the problems in current software development practice, should be capable to reduce complexity by imposing more structure in system design. Design patterns and component-based approach seems to offer a structured way to efficiently approach designing complex software architectures.

Visual programming

It is well known that visual information is easier to understand. Obviously, any design methodology will benefit from being based on a visual notation that is intuitively clear and self-explaining. Engineers often need to visualize their designs in order to obtain better understanding and also to be able to share that understanding with other members of the team. As a natural consequence, a class of visual modeling languages has emerged. More and more tools are developed with an aim of providing a graphical modeling language for entering designs. Most of the tools are also able to generate source code and/or executables from such

models. Some of the popular commercially available tools belonging to this category are Labview, Rhapsody, Rational Rose, Matlab/Simulink, dSpace, 20-sim, etc. A distinction can be made between domain specific (e.g. control systems) CAD (*Computer Aided Design*) tools (Matlab/Simulink, Labview, dSpace, 20-sim, etc.) and the more general CASE (*Computer-Aided Software Engineering*) tools intended for modeling software designs (Rhapsody, Rational Rose, etc.). The latter category most often relies on the UML graphical modeling language, which is the de facto industry standard for designing software systems.

"The *Unified Modeling Language* (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system" (Booch at al., 1999). It is tightly coupled to the *Object-oriented programming* (OOP) design philosophy. UML is a general modeling language. Its capability to model abstractions and structural relationships between them, regardless of the domain, makes it suitable for defining a domain vocabulary and basic structural laws of the domain (i.e. it is suitable for defining a metamodel of the domain). UML offers several kinds of diagrams that allow insight in the system structure and behavior from different viewpoints. The main disadvantage of UML is the lack of a glue layer able to connect all views into a consistent model of the system. In addition, a rigorous software engineering approach would require that such a glue layer is amenable to formal verification.

Formal verification

In all non-trivial projects, a significant part of product development time is spent on verification. Most of redesign is still caused by functional errors that could have been fixed with proper verification (Keding, 2004). Informal verification relies on performing a large quantity of custom, application-specific tests. *Formal verification* performs property checking based on mathematical techniques and precise requirements. In practice, the general weakness of formal verification tool implementations lies in the computational complexity (also referred to as "state-space explosion") needed to execute verification algorithms. Therefore, in practice *simulation* is often a main vehicle to verify complex systems. Computational complexity of verification analysis would be significantly reduced if a system could be hierarchically decomposed into separately verifiable building blocks. *Hierarchical verification* takes pre-verified blocks and focuses on block interconnections (which is usually a major source of problems).

It is desirable that designs are amenable to formal verification based on some mathematical formalism. Thus, a novel design methodology able to respond to the challenges of the software crisis should be related to some formal method capable to allow compositional hierarchical verification of the system.

Concurrency

Hardware vendors answer to increasing demands for processing power with parallel computer architectures— hyper-threading and multi-core multiprocessors. Software must be able to exploit these parallel hardware architectures in a both safe and efficient way (Sutter, 2005).

The computer science area, up to now, did not provide daily software development *practice* with an adequate response to the need for a simple and structured way of handling concurrency. Synchronization primitives offered by existing operating systems and programming languages are too low-level (Lee, 2006) and do not scale well with complexity (Hilderink, 2005a). Since the higher-level synchronization has to be designed for every problem in an ad-hoc manner, it is easy to unintentionally misuse those synchronization primitives. This usually results in problems that can stay hidden during most of the software lifetime and emerge in the worst possible moments.

The way concurrency is applied is getting more and more mature and structured. It is expected that a structured way of using concurrency will soon become mainstream and the most important breakthrough in software development after the invention of object-oriented programming. Many libraries, e.g. CT libraries (Hilderink et al., 1997; Orlic and Broenink, 2004), YAPI (Kock at al., 2000) , Kent libraries (JCSP, 2007), design and development tools like Ptolemy (Eker et al., 2003), UPAAL (UPPAAL, 2007), gCSP (Jovanovic et al., 2004), SHESim/POOSL (2007) are based on various theories concerned with discrete event systems (Cassandras and Lafortune, 1999; Roscoe, 1997; Schneider, 2000; Milner, 1989; Kahn and MacQueen, 1977) in order to structure concurrency.

Obviously, as concurrency becomes a more and more important design issue, it should be easily expressible in terms of the ideal design methodology.

Concurrency is an unavoidable issue in managing complex software projects. Similar to the way energy-flow based modeling is used in bond-graph theory as a common view in multi-domain physical system modeling (van Amerongen and Breedveld, 2002), in our approach concurrency is viewed as a glue layer that relates different domains and views. Concurrency used in a structured way should lead to systems that are simple to design and understand, and easy to distribute, reconfigure and reuse. Furthermore, designs are expected to be revised step-by-step with formally verifiable conformance to the initial specification. Formal checking can ensure freedom from concurrency related hazards (like deadlocks, livelocks, and race conditions), that are otherwise hard to analyze and prevent in complex problems.

Many *formal methods* (Formal methods, 2007) and process algebras were introduced in order to bring mathematical rigor into the way concurrent systems are described and analyzed. For instance, CSP (Hoare, 1978; Roscoe, 1997; Schneider, 2000) is one of the first and most influential formal methods. It offers a relevant parallel programming model, and seems to be a promising choice for handling concurrency. A commercial powerful tool, FDR (Formal Systems, 2005), exists, that can provide support for formal checking CSP expressions.

CSP is also one of the rare formal methods, whose subset was used as a basis of a programming language, and thus applied in software development practice. Occam (INMOS, 1988) is a simple and easy to use parallel programming language designed on CSP principles. It implements a certain subset of CSP constructs and allows construction of programs that can be formally checked. Occam and its hardware companion, the transputer (Welch et al., 1993) were very powerful and

scalable tools for building complex distributed systems. The application area of control systems has seen many successful projects based on the combination of the two (Fleming, 1988; Sunter, 1994). But with transputers being outperformed by other microprocessors, occam also felt into oblivion. Actually, this is not completely true. Occam is used in several research environments (Hilderink, 2005a; JCSP, 2007; Welch and Wood 1996) as a role model in shaping the way of for structuring concurrency.

1.3 SystemCSP approach

1.3.1 Problem statement

The aim of this thesis is to develop a graphical design specification language that can offer a structured way of handling concurrency. In accordance with previous text, a structured way of using concurrency is expected to raise the abstraction level away from concurrency problems. In that way, the introduced design specification language should become a vehicle for reducing complexity in inherently concurrent systems (e.g. complex control systems).

Key demands for such a design specification language are:

- Mapping to some existing formal verification method (this is necessary in order to provide support for formal verification without investing effort in creating new verification methods and tools)
- Support for specification of time properties and ways to analyze them (this is an essential demand stemming from the control systems application domain where timing properties are crucial)
- Support for modern notions of component-based development
- Expressiveness (it should be possible to easily express most important design decisions)
- Readability (the notation should be intuitive and easy to understand. It should be possible for a person familiar with the notation to grasp fast the key design decisions depicted in diagrams)
- Unambiguous interpretation of specified designs (the language is, in perspective, expected to result in a design tool with code generation facilities. Ambiguous interpretation would lead to uncertainty in the actual behavior of the implementation, making it dependent on tool vendors and the way in which code generation facilities are implemented.)
- Scalability (in order to be helpful in managing complexity, designs should scale well with increasing the number of used elements. In the language definition, this demand maps to the need for careful design of graphical elements and a well-thought balance between mandatory and

optionally visualized elements of the notation. One way to address the scalability issue is dividing systems into subsystems via a tree-like containment hierarchy. A somewhat more advanced approach is to provide ways to focus, in different diagrams, on different aspects of the interaction of a single component.)

- Applicability of the introduced notation for the design of interactions in complex control systems

Our approach is not to invent the wheel all over again. Instead, we want to build upon the existing body of knowledge provided by some existing formal method. We opt for CSP theory because of its wide acceptance, significant influence in the scientific world, solid body of knowledge and existence of the tools. Additional reason in favor of CSP is a lot of experience present in our lab in applying its occam implementation in the design of complex control systems.

The basic idea is to create the graphical representation of CSP and in that way bring the benefits obtained by relying on CSP closer to industrial practice. A graphical way of entering designs will allow engineers, ignorant to mathematics behind the CSP theory, to draw programs that are liable to formal verification based on CSP and the FDR tool (Formal Systems, 2005).

1.3.2 Local context

This research was carried out within a context defined by several preceding research projects.

During Hilderink's project (Hilderink, 2005a), basic concepts, and libraries were designed in order to bring occam-like concurrency in a modern software development practice. A first step was the implementation of libraries that provide occam-like compositional and synchronization primitives in popular programming languages (C++, Java, C) (Hilderink et al., 1997). Those libraries are known as CT libraries. A second step was the creation of a graphical modeling language (GML) (Hilderink, 2003) tailored for the design of occam-like CSP-based applications.

The next project (Jovanovic, 2006) in the line was oriented towards practical exploitation of the concepts through the implementation of a design tool (gCSP). gCSP tool (Jovanovic et al., 2004) is based on GML language and it is capable of generating source code for CT library, occam and machine readable CSP scripts. In addition, that project dealt with dependability issues in GML/CT based systems. The gCSP tool is capable to generate an FDR compatible CSP representation of the GML designs.

However, some important topics, like simulation of functional and extra-functional properties, real-time analysis, distributed communication, component-based development and many other topics were not researched in the scope of the aforementioned projects.

The initial aim of this project was to use the GML/CT approach in distributed real-time control systems with focus on the distribution aspects (e.g. designing the

support for distribution in CT library and estimating the influence imposed by using fieldbus network interconnections inside control systems). During the work on various practical assignments, some advantages and disadvantages of GML/CT approach were identified. As a result, the decision was made to move the research topic towards improving the ways to specify concurrent systems in a graphical way. Chapter 2 provides background information that motivates this decision.

Briefly stated, GML is based on relating process blocks via binary relationships. This approach does offer significant design freedom, which is useful in early stages of the design process. However, using the approach based on binary relationships to specify control flow, makes the control flow difficult to grasp fast in complex GML based designs.

The scalability of the GML design is problematic. In GML, the only way to handle scalability problems is using containment hierarchy, where internals of some process blocks are hidden on the current abstraction level and visualized in a separate view.

Thus, despite the offered design freedom, the expressiveness of the GML notation is, in more complex designs, often hampered due to the cluttered readability of control flow and the resulting reduced scalability.

In addition, GML is closer to occam then to CSP. Compared to CSP, it lacks some useful concepts like: control flow orientation, possibility to implement finite-state-machine like designs in intuitive way, mutual recursions. Compared to the practice of modern component based development, GML also lags behind.

SystemCSP inherits and extends some useful ideas from GML, while introducing new concepts that were lacking there.

1.3.3 Brief overview of SystemCSP

SystemCSP is based on the principles of both component-based design and CSP process algebra. Such a combination offers more expressiveness than offered by the occam-like approach targeted in Hilderink's and Jovanovic's work.

According to Roscoe (1997), "CSP was designed to be a notation and theory for describing and analyzing systems whose primary interest arises from the ways in which different components interact". CSP is a language offering a relevant model for parallel programming and SystemCSP aims to foster its utilization in the practice of component-based design.

SystemCSP is aimed to serve as a basis for the specification of formally verifiable interactions among concurrent entities (e.g. components). It aims to cover various aspects needed for the design and development of distributed real-time control systems. The structured approach is backed up by defining the metamodel of the SystemCSP design domain. In that way, the foundation is set for practical tool implementation.

SystemCSP is applicable for specifying, documenting, visualizing and formal verification of component-based designs. It provides a way to visualize architecture, behavioral patterns of components, intra–component interactions and execution relations among components.

The introduced set of graphical elements is related to the basic elements of the CSP process algebra. In addition, a set of basic elements related to component-based design is added to notation. Those elements can also be mapped to appropriate CSP expressions. In this way, designs have immediate mapping to CSP expressions. The library is designed that can provide support for the proposed concepts.

SystemCSP offers a control flow oriented and an interaction oriented visualizations of a design.

The control flow oriented visualization has its focus point on specifying control flow as a set of cooperating control flows composed via CSP operators. This kind of visualization defines compositional (execution) structure of the application.

Interaction oriented visualization allows the designer to design an interaction of few components in isolation from the rest of the system. In this approach, the structure, the set of data flows and the relative compositional relationships among participants can be emphasized. The interaction-oriented part of SystemCSP is inspired by the GML approach.

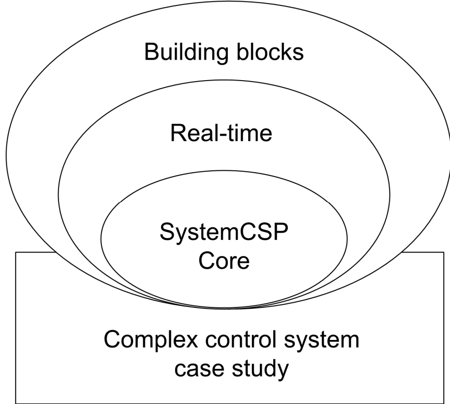


Figure 1-6 Overview of the results of this thesis

CSP has no widely accepted way to specify and analyze time properties. In this thesis, set of language elements is introduced that is dedicated to the specification of time properties. The thesis also offers some insights and some possible approaches for analysis and implementation of real-time systems. The language elements for specifying time properties, together with the proposed ways to do real-time implementation, extend the core of the SystemCSP graphical language with real-time part (see Figure 1-6).

The core of the notation is further extended by introducing a set of higher-level primitives in the form of reusable design patterns (see Figure 1-6). Defined patterns can be used in designs as basic building blocks, raising in that way the

level of abstraction. The introduced set of design patterns makes the initial body of reusable SystemCSP designs. Introduced patterns can also be viewed as a set of examples illustrating a way to design interactions of concurrent entities in SystemCSP. As such, patterns were used to evaluate whether and to what extent the proposed graphical language fulfills the criteria set in 1.3.1. Feedback obtained was used to further improve the notation.

Usability of the notation for specifying interactions in embedded control systems is tested on a complex control setup consisting of several cooperating devices. In that way, foundation is set (see Figure 1-6) for practical usage of the notation in complex control systems. This test case is a step towards industrial-strength test case. It should give us insight whether SystemCSP is a convenient language to design interactions in complex control systems.

1.4 Outline of the thesis

Chapter 2 introduces the context via focusing on three basic ingredients for the creation of SystemCSP: software implementation of CSP, visual representations of concurrent systems and component based development.

Chapter 3 introduces the core of the SystemCSP. In this chapter the elements of control flow oriented and interaction-oriented parts of the notation are introduced. The language elements presented in this chapter are in fact improved version of the ones introduced in (Orlic and Broenink, 2006a; Orlic and Broenink, 2006b).

Chapter 4 extends the core of the SystemCSP language with real-time aspects. The first part of the chapter introduces a way to specify time properties in SystemCSP. The second part contemplates about applicability of CSP and SystemCSP based systems in the area of real-time systems. The contents of this chapter were introduced in (Orlic and Broenink, 2007a).

Chapter 5 introduces set of SystemCSP design patterns useable in component-based safety critical real-time control systems. Patterns are grouped into sets related to communication, component-based design, structuring control systems and the ones related to fault tolerance. Many of the patterns presented in this chapter are based upon the patterns described in (Orlic and Broenink, 2006b) and (Orlic and Broenink, 2007a).

In chapter 6, SystemCSP is used to design software for the complex control setup consisting of several interacting simple robotic devices. For the purpose of this project, the production cell setup was made.

Finally, in chapter 7, conclusions and recommendations are summarized.

The appendices deal with creating preconditions for practical usage of SystemCSP.

Appendix A defines the metamodel of the notation, creating in that way a foundation for structured design of a prospective tool.

Appendix B details the design of a framework that provides support for proper functioning of code generated from SystemCSP models. The design principles for SystemCSP software framework were introduced in (Orlic and Broenink, 2007b).

2 Background

Eventually, you learn to read groups of words. Where a student will see three motions, the experienced man will see one, because he sees the overall energy path.

Bruce Lee

The primary aim of this chapter is to introduce the context vocabulary. It provides background information for three different aspects that will be in the next chapters unified into a new graphical design specification language based on CSP. This chapter also illustrates a need for such a holistic approach.

The first aspect, presented in section 2.1, is related to comparing CSP and its practical software implementations – occam and the CT library. Comparison is performed for every significant part of the vocabulary.

The second aspect, presented in section 2.2, is related to ways of visualizing concurrent systems. Most attention is given to the graphical modeling language GML which is developed by (Hilderink, 2003) in our lab. GML brings in an original way to visualize CSP-based occam-like concurrency.

The third aspect, presented in section 2.3, is related to the current state of the art in component-based development. This part introduces basic elements that will be used to extend our approach with typical notions from the area of component-based development.

2.1 CSP and its implementations

Although there are other equally or more relevant representatives of occam-like libraries (JCSP, 2007), we choose to focus on the CT library (Hilderink et al., 1997; Orlic and Broenink, 2004) because it was developed in our lab and is thus a starting point of this research.

The basic elements of occam, CT library and GML are based on CSP operators. However, the exact meaning of the concepts in CSP, occam, CT and GML differs in some respects. Further text will attempt to carefully observe and emphasize the most important differences and similarities.

2.1.1 Basic elements

A *CSP process* describes some behavior in the form of a set of event synchronization points related via control flow operators. Process progresses through its control flow until it reach an event synchronization point. At this point a process attempts to engage in the event associated with the point, and waits until its environment (all other processes participating in that event) accepts the offered event. An event can take place only when all processes participating in the event

are ready to perform it. A set of events in which a process can participate makes its *alphabet*.

Control flow is specified using several predefined operators. Most important operators of CSP are *event prefix*, *sequential*, *parallel* and *choice* (*internal*, *external*). These control flow elements are in a CSP process used to relate event synchronization points, and to construct in such a way an interaction pattern. Such interaction pattern defines a set of event orderings (traces) possible for the process.

The *sequential* operator defines a strict order (or sequence) of execution for the associated group of processes. The *parallel* operator defines that the associated group of processes is executed concurrently, synchronizing at least on start and termination. When a process offers to the environment a choice between several alternatives, those alternatives are grouped with the *external choice* operator. *Internal choice* operator specifies that the behavior of a process is known to be according to one of several possible alternatives. In addition, there is the *conditional choice* (IF) operator, that will, depending on whether some logical condition is satisfied, transfer control to either the true or the false branch.

Any composition of processes is again a process that can be further nested in CSP expressions. It offers to its environment events of the contained subprocesses. It is in fact possible to hide or rename some events offered to the environment of a process. Special operators (*hiding* and *renaming*) are dedicated to fulfilling that task in CSP.

Occam puts a restricted form of CSP into practical use. CSP operators are in occam represented via PAR (*parallel*), SEQ (*sequential*), ALT (*alternative*) and IF constructs. ALT is like the external choice with the difference that it assumes joining of control flow. In occam, the *internal choice* operator is not implemented. In addition, occam introduces a prioritized version of the *parallel* construct (PRIPAR). In transputers the PRIPAR construct was supported via a hardware scheduler that could deal with two levels of priority. In occam, events of CSP are represented in the form of channels.

The CT library (Hilderink et al., 1997; Orlic and Broenink, 2004) implements an API, that maps to the syntax elements of occam. The CT library follows the occam model as far as possible. Basic occam primitives are implemented in an object-oriented way. Every process is actually implemented as an object and is thus a tangible entity that has both behavior and structure. Special classes are implemented to provide the functionality of the constructs.

In addition to implementing all occam constructs, the `PriAlternative`, a prioritized version of the ALT construct is introduced. In CT, constructs are created by instantiating classes defining the behavior of that construct (classes `(Pri)Parallel`, `Sequential`, `(Pri)Alternative`). Leaf processes in a tree-hierarchy are user-defined classes that must be derived from the common class `Process` in order to allow `Construct` classes to deal with them.

Two types of communication relationship are possible: *rendezvous channels* (synchronization points comparable to events of CSP) and *variable channels* (not

synchronized, comparable to variables of CSP) named sometimes *var-channels* in context of the GML/CT approach.

2.1.2 Grouping

In CSP, grouping of event expressions is done via parenthesis symbols. In addition, it is possible to name some parts of event expressions and refer to them by name from any other part of a CSP description. The process expression P;(T;(Q || R)) means that there is the *sequential* composition (;) of process P and the process (T;(Q || R) that is a *sequential* composition of process T and of the process (Q || R) that is *parallel* (||) composition of processes Q and R.

The way grouping is specified in occam, is based on the hierarchical structure of occam programs. Every construct starts with its symbol. The scope of the construct is actually determined via indentation that mimicks the hierarchical structure of program. The CSP expression P;(T;(Q || R)) would in occam implementation look like:

```
SEQ
  P
  SEQ
    T
    PAR
      Q
      R
```

Listing 2-1 Code snippet illustrating grouping in occam

In occam, a process has access to the variables defined anywhere in the hierarchy of its parent processes.

In CT, constructs are created by instantiating classes defining the behavior equivalent to the appropriate construct (classes `(Pri)Parallel`, `Sequential`, `(Pri)Alternative`). Grouping is thus made by creating an instance of the construct and assigning chosen user-defined processes to be its subprocesses. The CSP expression P;(T;(Q || R)) is in CT library specified as:

```
Parallel* par = new Parallel (Q, R);
Sequential seq1= new Sequential (T, par);
Sequential seq2 = new Sequential (P, seq1);
```

Listing 2-2 Code snippet illustrating grouping in the CT library

Since in CT both user-defined processes and their parent processes are implemented as objects, a user-defined process does not automatically gain access to the variables defined in the parent scope (as it was the case in occam). Instead, copying data back and forth or passing references to those variables must be performed. The CT libraries use the concept of *variable/var channels* (see Table 2-1) for this purpose. *Variable channels* are channels without synchronization, used to pass values of shared variables across process boundaries.

2.1.3 Event prefix and sequential operators

CSP makes a distinction between the *event prefix* operator (→) and the *sequential* (;) operator. Both specify the order of execution. The prefix operator is used to connect an event end on the left side of the operator and the rest of the process on the right side. The *sequential* operator is used to group two processes where the second process is executed immediately after the first one has terminated. E.g. the process expression: ev1 → (P1;P2) starts with event 'ev1' and after its occurrence, *sequential* composition of processes P1 and P2 is executed.

In occam and CT libraries, the code of a single process is normally written in a sequential manner, statement by statement. Consequently, the *prefix* operator is not needed: an event from the left side of a *prefix* operator is placed in one statement and the rest of the process in the next statements. The same holds for the *sequential* operator. However, since the *sequential* operator also has a role of grouping parts of code into reusable and maintainable parts, both occam and CT libraries provide a `Sequential` construct.

2.1.4 Parallel operator

The CSP *parallel* operator allows grouping of two or more processes that are obliged to synchronize on start and termination (events). In addition, the *parallel* operator provides ways to specify whether the combined processes should synchronize on some additional, user-defined, events. The connection between user-defined event ends belonging to composed processes is made by specifying them as a part of the *synchronization alphabet* associated with the operator. If the *synchronization alphabet* is not specified, then the assumption is that it contains all those events that do appear in both composed processes. *Interleaving parallel* is the *parallel* operator with no events in its synchronization alphabet.

Event synchronization is named *channel* communication in case when only two event-ends participate and there is an associated set of data communications in one or both directions.

In occam, a PAR construct contains two or more processes and it defines that all of its subprocesses are executed in parallel, that is by separated flows of control. It is not relevant for the construct whether a pair of processes is intended to synchronize on some user-defined event or not. Instead of explicitly specifying events on which processes synchronize per every parallel operator, the concept of channels is used.

A *channel* is an infrastructure element that encapsulates the point of event synchronization. Occam channels assume both rendezvous synchronization and unidirectional communication, where exactly two event-ends participate in synchronization and data transfer is directed from the writer side to the reader side. The processes accessing the channel are not aware of each other; they communicate by accessing the same channel for a read or write operation. Upon accessing the channel, their further execution is blocked until the data transfer is performed.

The a*lphabet* on which two parallel processes synchronize is in this way implicitly specified in their channel interconnections. For the operational semantics of a *parallel* construct in occam, explicit enumeration of *alphabets* on which processes synchronize is thus not needed and not foreseen in the language syntax. Similarly, the *renaming* and *hiding* operators are superfluous in a channel-based interconnection of processes.

Considering this issue, the CT library follows the occam style. A `Parallel` construct in the CT library spawns separate user-level threads for every subprocess. Synchronization points are defined by channel interconnections.

2.1.5 Choice operators and constructs

Like the parallel operator, the *choice operator* essentially branches the related control flow on several independent control flow paths. Unlike the parallel operator where all branches are executed in parallel, the choice operator means that only one branch is chosen to be executed. An *external choice* is letting the environment of the process (all other processes) to choose a branch (one of the offered alternative control flow paths) to be executed.

External (deterministic) Choice

In case of a *guarded alternative* operator (|), the choice is offered between events. After the environment has chosen one of the offered events, the control flow will continue following the branch associated with the chosen event.

In case of the CSP *External choice* operator (□), the choice is offered between processes. The process that first accepts some event from the environment will be chosen. This usually boils down to the *guarded alternative* operator because offered processes usually start with an *event*. In the CSP description below, process `choice1` based on the *guarded alternative* operator and process `choice2` based on the *external choice* operator offer actually the same options to the environment.

```
Choice 1 = ev1 → Process 1  |  ev2 → Proces2
P1 = ev1 → Process1
P2 = ev2 → Process2
Choice2 = P1 □ P2
```

Neither the guarded alternative operator, nor the external choice operator, assumes that there is a common join point of the alternative branches.

The ALT construct of occam is a somewhat modified guarded alternative operator of CSP. The main difference is that every fork of the control flow on two or more alternative branches is paired with a join where all those branches meet. Introducing this limitation makes the structure of occam programs expressible as a strict tree-hierarchy consisting of constructs as branches and user-defined processes as leaves.

In occam, the offered events are taking the form of channels guarded for communication attempts. The ALT process waits until one of the channels (for which the associated logical condition is set to true) becomes ready, and then it executes the associated process. If the environment in the same time accepts two or more events, this is in theory resolved by choosing one of the ready channels randomly. In practical implementations, often the first ready channel from the list is chosen (that is why the occam/transputer implementation of ALT is sometimes referred to as prioritized ALT or PRIALT).

A typical ALT construct looks as in:

```
[logical condition 1] & channel1 ? data
   Process1
[logical condition 2] & channel2 ? data
   Process 2
[logical condition 3] & SKIP
   Process 3
```

Listing 2-3 Typical ALT construct in occam

In occam, only input channels can be guarded, there are no output guards.

The CT library implements the `Alternative` construct as a class whose behavior is based on the ideas of the occam ALT construct. The implementation of the `Alternative` construct (Orlic and Broenink, 2004) allows several different working modes (*preference alting*, *prioritized* (guards), fair, FIFO), introduced to allow several criterias for making a deterministic choice in case more than one of the alternatives is ready for execution at the same time. The *preference alting* mechanism (Hilderink, 2005a) enables that, in case several guards are ready, a choice is made by comparing the priorities of the processes from the environment that are attempting to access the guarded channels. The *prioritized* version of the `Alternative` construct (`PriAlternative`) gives preference according to the order in which guards are specified in the `PriAlternative` construct.

The alting in the CT library assumes that a channel can be guarded by some `Alternative` construct only from one of the exactly two event-end sides (there can be either an input or an output guard associated with a channel). A guarded channel is just a channel with an associated guard. A guard is an object inside an alternative construct associated with a channel and a process. When a guarded channel is accessed by the peer process, then the guard becomes ready and is added to the alting queue. The way in which guards are ordered in this queue, determines the working mode (*preference alting*, *prioritized*, fair, FIFO) of the alternative construct. An `Alternative` construct is thus a single point where the decision of a choice is made.

A guard that behaves the same as the combination of a logical condition and a *skip* process as in branch 3 of the occam sample code in Listing 2-3 is in the CT library named a *skip* guard. A *skip* guard has no channel associated. Readiness of this guard depends only on the state of the associated logical condition.

Conditional (IF) choice

CSP defines a conditional choice operator. If the associated condition equates to true, then the control flow will perform the process specified as left operand. In case that condition equates to false, control flow will perform the process specified as right side operand. The following example illustrates this:

```
P = (ev2?x→P) ⊀condition⊁ (ev1?x→SKIP)
```

In this example, if the condition equates to true, a process expression is executed that starts with event ‘ev2’. After performing the associated data communication (from channel ‘ev2’ to variable x), the process will behave again as process P. In case that the `condition` equates to false, process P will perform event ‘ev1’ and than successfully terminate by performing the SKIP process (process that does nothing, but successfully finishes).

Occam, CT and GML have different levels of support for specifying a conditional (IF) choice. In occam, the IF conditional statement is considered as a decision making construct. It is somewhat similar to the ALT construct with the distinction that, instead of guarded by a combination of logical condition and channel readiness, the alternative options are guarded only by logical conditions. Consequently, the branch to be executed is in the IF construct resolved immediately when the IF construct is entered: there is no waiting involved. In occam, the IF construct is priority structured. The process associated with the first condition that equates to TRUE is executed. If no condition equates to TRUE, the resulting action is equivalent to STOP and thus causes a deadlock. If it is possible that no condition equates to true, usually an additional branch is added at the end, with a condition that is always true and has as resulting action the SKIP process. The code below illustrates the typical use of the IF construct in occam:

```
IF
   [logical condition1]
      Process1
   [logical condition2]
      Process2
   [TRUE]
      SKIP
```

Listing 2-4 Typical IF choice construct in occam

In the CT library, it is assumed that the behavior of the IF construct can be implemented using *if/then* and *switch* control blocks of the native language (e.g. C++).

Internal choice

The *Internal choice* operator ($\sqcap$) specifies that the process will, in some way, internally choose one of the several possible branches. Using the *internal choice* operator is a way to specify that a process will behave in one of the specified ways, but without specifying exactly how. The power of the *internal choice* is thus that it provides abstraction capabilities to describe the important aspect of process behavior without going into details of actual implementation. In that way it is

allowing incomplete but formally checkable specifications. Through the procedure of refinement, *internal choice* operators are usually replaced with other more detailed specifications.

The *internal choice* is thus an important abstraction mechanism, extremely useful in early stages in the process of top-down, stepwise refinement design. It is not very useful for operational execution. Therefore, the occam and the CT library do not have support for the *internal choice*.

2.1.6 Finite state machine based designs

Simple CSP processes, made out of only event synchronization points connected via instances of the *event prefix* and of the *guarded alternative* operator, are often visualized using a *Finite State Machine* (FSM). Every node in such a FSM represents a state of the process and every edge/transition is associated with some event. A single transition leading from some state defines an *event prefix* operator and multiple transitions leading from a single state define a *guarded alternative* operator. With the *guarded alternative* of CSP, no join of branches is assumed, and the branches can lead to any other state, or one can reference other processes in alternative branches. This is compatible with the FSM-based visualization.

The Occam/CT/GML choice (ALT construct) is, as explained before, somewhat different than the CSP *external choice* operator. The ALT kind of choice requires that all alternatives are eventually joined. Consequently, FSM-like designs is not straightforward to implement in occam, and one needs to develop special design pattern for this purpose.

For instance, a solution that seems to be most close to common sense is to have a variable that keeps track of the current state, and a single `Alternative` construct. The `Alternative` construct has a dedicated subprocess for every transition (the occurrence of an event that leads the FSM from one state to another) in an FSM. At any time, only the branches representing the transitions leading from the current state would have associated logical conditions enabled. Associated subprocess will be invoked by the event triggering FSM transition and the task of the invoked process is only to update the variable representing the state.

The code obtained using this pattern is not intuitively readable.

Another issue relevant for implementing FSM-like designs is that, in CSP, recursion is possible simply by referencing (in CSP expressions) names of the processes defined elsewhere. Occam and CT lack ways to express recursion in ways other then repeating the same part of behavior in a loop.

2.1.7 Priorities

CSP is ignorant of the way concurrency is implemented. Concurrency phenomena involving parallel processes interacting via rendezvous synchronizations, are the same regardless whether the processes are executed on separate processors or using

some time-sharing algorithm on the same processor. However, the temporal characteristics are different in the two cases. In case processes are executed by the same microprocessor, the most commonly applied scheduling schemes are based on associating priorities with processes. In real-time systems, achieving proper temporal behavior is of utmost interest. Therefore, in real-time systems priorities are attached to schedulable units according to some scheduling algorithm that can guarantee meeting time requirements.

In addition to the PAR (*parallel*) construct, a prioritized version of the PAR construct, the PRIPAR construct, was introduced in occam. It specifies parallel execution with priorities assigned according to the order of adding subprocesses to the construct. However, on transputer platforms only two priority levels were supported and a PRIPAR was therefore used only on top level. Additional priority levels were sometimes implemented in software (Sunter, 1994).

As in occam, the `PriParallel` construct of the CT library defines relative priority ordering of composed processes, based on the order of adding processes to the construct. This allows for a user-friendly priority assignment based on the notion of the, more or less intuitive, relative importance of processes compared to other processes. The possibility to use a hierarchy of `PriParallel` and `Parallel` constructs creates a possibility to have an unbounded number of different priority levels in a program. Note however, that priority ordering, of all processes in a system, is not necessarily a strict ordering, but rather a set of partial orderings.

Scheduling in the CT library is distributed in `PriParallel` constructs. When a construct gets the processor, it will continue executing its subprocesses untill they are finished or until some higher-priority process has become ready to be executed.

No underlying theory exists that can be directly applied to guarantee real-time execution in cases where processes communicate via rendezvous based message-passing. Chapter 4 deals with those issues in more details.

2.1.8 Exceptions

The concept of exceptions is an essential part of modern software development practice, but it was not common at the time CSP theory appeared. In CSP, the concept of the *interrupt operator* (Δ_i) is most suitable for describing the way exception handling works. Process $P \Delta_i Q$ is a process that behaves as process P until either P terminates successfully or until an event ‘i’ occurs and activates process Q. In the latter case, further execution of process P is aborted and process Q is executed instead. Despite its name, the semantics of the Δ operator is much closer to the termination model of exception handling than to *interrupt handling*, since it implies termination of the process used as the left hand-side operand.

Occam does not have support for exceptions.

In the CT library, upon an exception occurring in a process, the process terminates and an exception is thrown and propagates upwards the process/construct

hierarchy; the exception is eventually being caught by the `exception construct` that forwards it to the associated `exception handler`. If the process where the exception occured is inside an `Alternative` or `Sequential` construct, the exception that cannot be handled by the subprocess will interrupt the associated parent construct. The interrupted construct forwards an exception upwards the tree hierarchy. With a `Parallel` construct, the situation is however different because other subprocesses do continue to execute until they terminate. All exceptions thrown by the subprocesses of a `Parallel` construct are collected in an `ExceptionSet` object and handled only after the `Parallel` construct terminates (i.e. when all subprocesses terminate, successfully or exceptionally). One can however force other subprocesses to terminate via the *channel poisoning mechanism* (via infecting a channel with an exception). Every attempt of a process to access a poisoned channel, results in raising an exception.

2.1.9 Discussion

An application area of interest in this research is control systems area. CSP offers a structured way of handling concurrency. However, since CSP is an asynchronous (i.e. not synchronized with time) approach, it might not be the perfect match for implementation of precisely periodic control loop calculation schemes. Chapter 4 will attempt to shed more light on this issue.

Instead of synchronization with time, rendezvous based synchronization of event-ends participating in the same event is performed. This makes CSP convenient to structure concurrency in event-driven supervision and sequence control layers and in additional components necessary in complex control systems.

The CSP description does not really have a strict hierarchy. Any named process can be invoked from any part of the description. As a consequence, after forking the control flow, as is done in the case of *parallel* or *choice* operators, the created branches do not have to join in the same place. In this way, a CSP description can capture any concurrent behavior (e.g. FSM-like descriptions that do not have a tree-like hierarchy with processes as firm structural units).

Occam, as a language inspired by CSP, adopts process architecture with message-passing channel communication. This kind of architecture turned up to be convenient for building large-scale scalable distributed programs. The CSP notion of events is in occam restricted to channels. Channels are used to perform rendezvous synchronization and unidirectional communication between exactly two processes.

The notion of a process is in occam also somewhat different than in CSP. A CSP process is essentially a description of a behavior that can be just a named point in a control flow. An occam process is a structural unit alike to a component in every-day software development practice. Occam processes are grouped into constructs in such a way that every process has one immediate parent construct (except the top-level construct of course). An occam program is thus a tree-like hierarchy with user-defined processes as leaves and constructs as branches. With this approach,

some possibilities offered by CSP (e.g. mutual recursion and natural expression of FSM-like designs) cannot be expressed in occam elegantly.

Obviously, although the basic elements of occam are adopted from a subset of CSP, the related philosophy and the way of thinking in occam is rather different from the one in CSP.

At the time when CSP and occam appeared, the dominant programming style was imperative. *Object-oriented programming* (OOP), which is the preferred style nowadays, was not yet conceived. Although the *object-oriented programming* worldview introduced a more structured way to handle complexity, it failed to provide an adequate concurrency model. Blending the process orientation of occam with the *object-oriented programming* orientation seems a promising concept.

CT libraries adopt the occam-like programming model implemented in an *object-oriented programming* way. However, from the users point of view, *object-oriented programming* is in occam-like libraries restricted to lower level objects encapsulated inside user-defined processes. In this way, user-defined process became alike to component whose internal content and behavior is specified in sequential *object-oriented* way. The components obtained in that way can be further composed using constructs and do synchronize with other components exclusively via access points named channels.

Though a CSP process can be implemented as an object, essentially a CSP process is *not* an object. While an object is an entity, a process is focused on defining some behavior, possibly a behavior attributed to an object, part of an object or a set of objects.

In the CT library, processes and constructs are implemented as objects, imposing more strict boundaries for the variables scope then in occam. The scope of variables belonging to some process is, due to encapsulation, limited by the borders of the object implementing that process.

Following the semantics of CSP events, the channels of occam and the CT library are synchronous (rendezvous based). However, existing real-time scheduling theories are based on process models with asynchronous communication.

Occam and CT libraries do not use the full expressiveness of CSP. Things that are easy to express in CSP, like recursion other then looping and *finite state machine* kind of behavior, is not straightforward to implement in occam and CT libraries. The *internal choice*, which is an important abstraction mechanism, is also missing. The `Alternative` construct differs from the *external choice* of CSP in a sense that it implies not only forking but also a common join point of alternatives.

2.2 Visualizing concurrent systems

Humans can much better comprehend and communicate visualized behavioral scenarios then the same scenarios given via some mathematical description, e.g. CSP formulas. Still mathematical descriptions are necessary for precise analysis of

models. A workaround is to introduce a set of intuitive visual elements that can be automatically mapped onto proper mathematical descriptions. There are many tools and technologies that define own visual notations. In this section, however, we focus our intention to those that were most influential in the scope of this project.

UML is the de-facto standard in software development practice. After briefly discussing the UML with focus on its support for specifying concurrent systems, the overview goes towards visualizing CSP based designs. A *finite state machine* based way of visualizing CSP processes is often used in CSP literature and is thus described in a separate subsection. GML, as the CSP based, occam-compatible graphical language (Hilderink, 2003), is relevant to explain in detail because it is predecessor and starting point of SystemCSP in the local research context. GML has in fact served as an inspiration for the interaction oriented part of SystemCSP.

2.2.1 UML

UML is designed to offer general support that can be used in various development processes (Booch at al., 1999). UML contains several kinds of diagrams (see Figure 2-1) based on different basic elements. There is no mapping or firm relation between those separate views in which the same entities can participate.

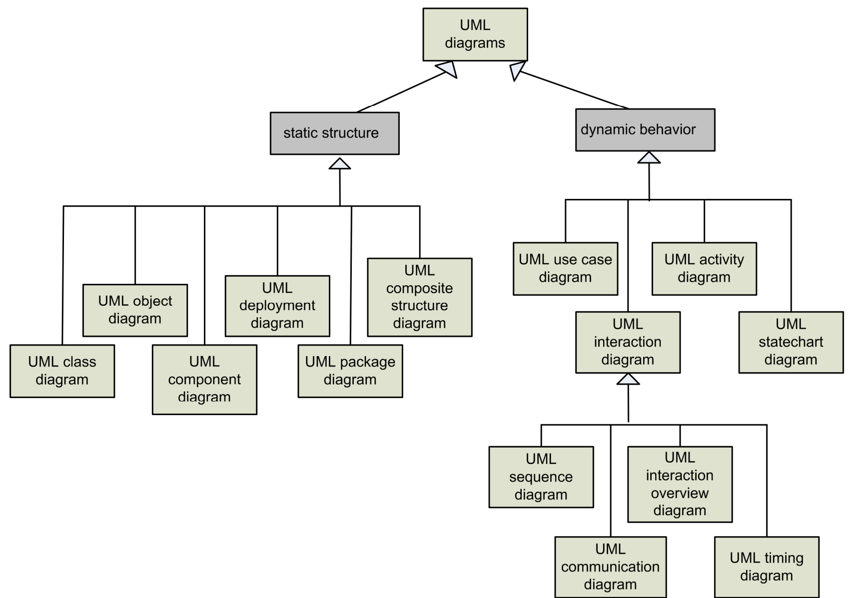


Figure 2-1 UML is a set of various types of diagrams

UML use-case diagrams specify the functionality and behavior of a system as viewed by its user (*actor* in UML terminology). It defines what the system does, but not how. An example of a UML use-case diagram is given in Figure 2-2.

A *communication relationship* relates an *actor* with a *use-case* (oval block representing the functionality offered by system). The behavior of *use-case* blocks

is expressed via unstructured text, pseudocode or *UML interaction diagrams*. An *include relationship* specifies that the *use-case* on the source side of the relationship includes behavior defined by the use-case on the target side of the relationship. E.g. in Figure 2-2, *use-case* A includes behaviors as defined in *use-cases* A11 and A12. An *Extend relationship* means that the extension *use-case* implements some behavior that is an optional part of the basic behavior. In Figure 2-2, *use-case* 'A12extension' defines a functionality that can optionally be implemented in A12. A *generalization relationship* is used to relate a *use-case* with the *use-case* that is special case of it. In Figure 2-2, 'B impl1' and 'B impl2' are two different special *use-cases* implementing the behavior defined in 'B generic'.

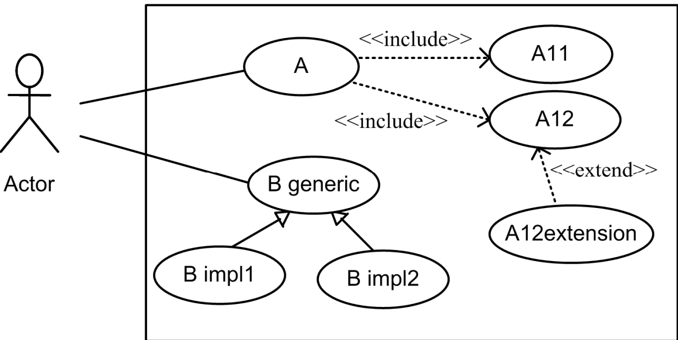


Figure 2-2 UML use-case diagram

UML class diagrams (see Figure 2-3) allow defining abstractions (classes) in terms of attributes they encapsulate, operations they can perform and responsibilities delegated to them.

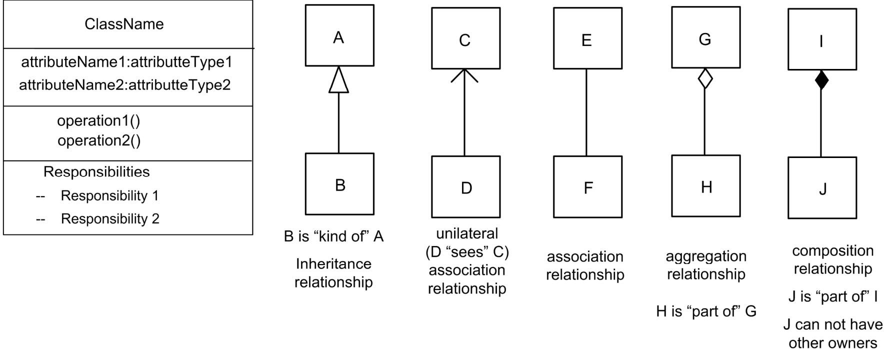


Figure 2-3 Elements of UML class diagram

Classes can be related via *inheritance* ("a kind of") and *association* relationships. A special kind of *association* relationship is a "part of" relationship. It comes in less strict ownership (*aggregation* relationships) where more than one owner can exist and more strict ownership (*composition* relationship) where only a single owner can exist and is responsible for creation and destruction of the contained part. The introduction of abstractions and the possibility to define relationships

among them, in a way close to the way human mind classifies, thinks and establishes relationships, makes *UML class diagrams* very convenient for defining the vocabulary of a domain (i.e. defining a metamodel of the domain).

UML sequence diagrams (see Figure 2-4) focus on time aspects of interaction. Time is flowing downwards and is reflected in the *lifeline* of every object participating in the interaction. It is possible to visualize several kinds of interactions among objects, like function calls, returning results of a function call, asynchronous signals, creating and destroying of objects. Mentioned types of interactions are visualized as directed lines that do connect the lifeline of a source object with the lifeline of the target object. *Control focus* is a rectangle around lifeline used to emphasize control flow. A sequence diagram can contain more than one independent control flow, depicting in such way concurrency present in the interaction.

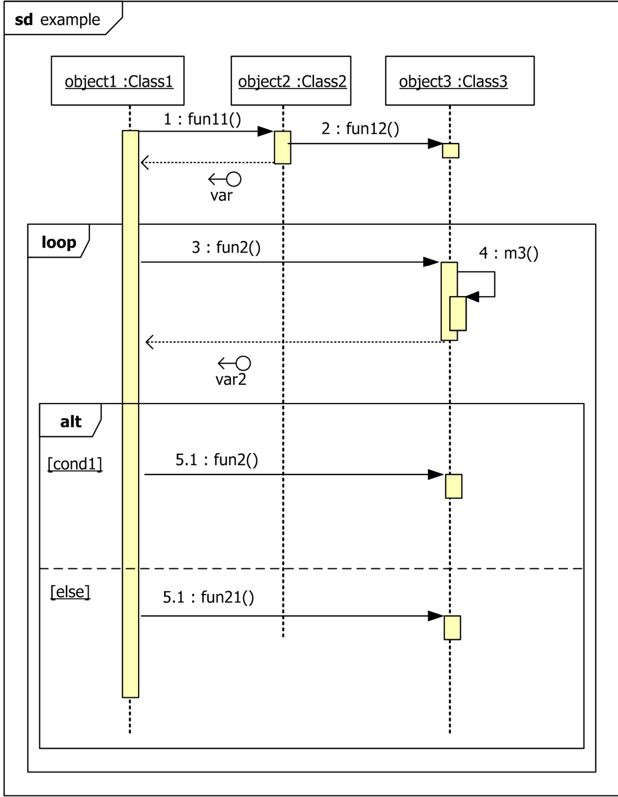


Figure 2-4 Example of a UML sequence diagram

UML2 defines interaction operators like *loop*, *optional* and *alt* (see Figure 2-4) used respectively to illustrate the scope of a loop control structure, optional part of an interaction whose execution depends on some condition, and conditional branching with all possible alternative options of control flow execution given in rectangle boxes separated via dashed line. However, those operators, although

bringing in the possibility to depict several scenarios in a single diagram, tend to clutter diagrams, especially for more complex interactions.

Concurrency is in UML diagrams best specified in *activity diagrams* and *statechart diagrams*. However although those diagrams can specify concurrency structure, that is not their primary focus.

An *activity diagram* (see Figure 2-5) depicts control flow passing through structural instances (e.g. objects) and can display action blocks executed along the way and conditional branching of control flow according to associated logical guards. Concurrency is in an *activity diagram* represented via the *fork* control flow element that forks control flow into several parallel flows of control, and the *join* control flow elements that joins flows of control. In UML2, *activity diagrams* incorporate some Petri Net concepts (tokens).

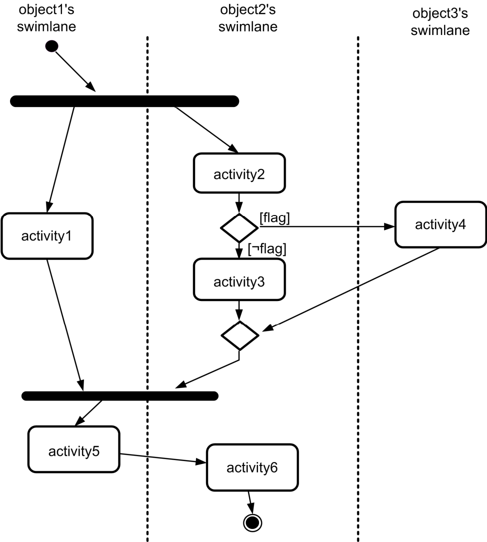


Figure 2-5 Example of a UML activity diagram

A *Finite state machine* (FSM) defines sequences of states through which an object goes as a respond to events. In state machine diagram, nodes represent states and vertices represent transitions from one state to another. An approach based on FSM is used in CSP books to visualize CSP systems and will be the focus of the next section.

UML state diagrams (see Figure 2-6) origin from the *statechart* approach (Harel and Politi, 1998). Compared to the classic finite state machine (FSM) approach, they build upon the state machine idea by adding many concepts useful in practice (e.g. allowing nesting states and remembering deep or shallow history while moving vertically through the hierarchy of states). Substates in UML statechart diagrams can be sequential or concurrent. An object implementing a state machine with nested sequential states can be at any time only in one of the nested states. An object implementing state machine consisting of nested concurrent substates is in

all substates in the same time. Entering composite state, that contains concurrent substates (e.g. substates s21 and s22 of state 2 in the example from Figure 2-6), assumes existence of fork of control flow to concurrent substates. Exit from such state assumes the join of control flow.

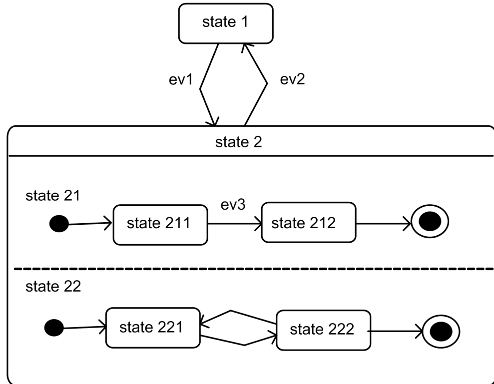


Figure 2-6 UML statechart diagram

Critique

As stated in (Marwedel, 2003), UML does not have precise semantics, and as such, it is most useful in early stages of the design, and for informal communication. The lack of precise semantics makes complete consistency checks between different diagrams impossible. This informal way of using, based on local conventions, is often creating problems in communicating designs between stakeholders.

Another significant problem with UML diagrams is that they do not put main focus on visualizing the concurrency structure of a program.

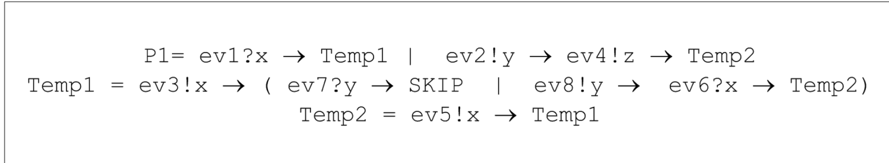
According to the survey (Lange et al., 2006) on UML usage in practice: adherence to standards is loose, there are no objective criteria to verify that a model is complete or satisfies some tangible notion of quality, miscommunication is reported in more than half of the projects, a “wrong” product delivered is mentioned and a high amount of testing effort is needed. According to this survey, some of the main problems with UML are: design choices scattered in unrelated views, informal use, limited possibility for checking consistency between different views, disproportion between specified architectural details and the needs of implementers.

In UML, semantics necessary for precise specification of a system can be obtained only if UML is combined with some formal language (e.g. SDL). Approaches based on a combination of UML and CSP also exist. (Crichton et al., 2002) proposed a design pattern for specifying concurrency patterns using a subset of UML, in a way that is formally verifiable via CSP. Their approach is based more or less on the combination of statecharts and activity diagrams. But, such an approach is fitting existing diagrams into something they were not designed for. It also suffers from not allowing full expressiveness of CSP. The aim of that research

was to create a formally verifiable concurrency design pattern using existing UML diagrams. For our purpose, insisting on the usage of UML diagrams is not an issue.

2.2.2 Finite State Machines

In CSP related literature (Roscoe, 1997; Schneider, 2000) *finite state machine* (FSM) kind of diagrams are often used to illustrate CSP interaction patterns. Every node in such a FSM represents a state of the process and every edge/transition is associated with some event. Figure 2-7 presents one CSP description and its associated visualization based on a FSM. In fact, the FSM in Figure 2-7 is a typical UML-like visual representation of a state machine. State machine diagrams in CSP literature differ from UML statecharts in depicting states as small circles with state names written outside the state circle. In addition, the start and exit states bare no special visual difference to other states. The difference is of course that there are no transitions leading to the start state and no transitions leading from the exit state.



```
P1= ev1?x → Temp1 |  ev2!y → ev4!z → Temp2
Temp1 = ev3!x → ( ev7?y → SKIP  |  ev8!y →  ev6?x → Temp2)
Temp2 = ev5!x → Temp1
```

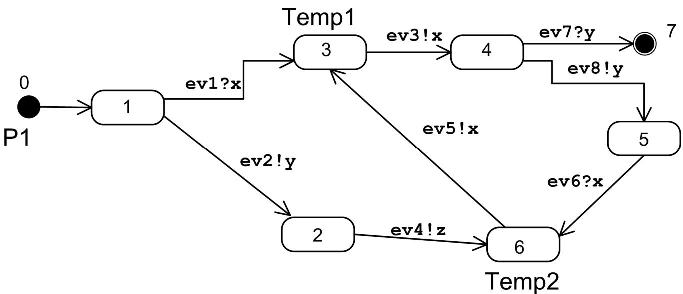


Figure 2-7 Classical FSM diagram

One can thus visualize some CSP descriptions using FSM diagrams. Looking in another direction, CSP expressions can be seen as an attempt to capture visual FSM specifications in the form of a textual language sequential stream of characters. In order to transform an FSM into CSP expressions, some states are given names and considered to be CSP processes. Note that the same FSM can be mapped onto different CSP descriptions depending on the choice of the states to which names are assigned. However, a CSP representation of an FSM, based on the minimal number of process names is unique (of course, provided that one abstracts away from differences in chosen names). Such a representation is obtained if only the start state and states reachable from more than one other state (states where a join of several control flows is performed) are named. In Figure 2-7, states 3 and 6 are named respectively Temp1 and Temp2, defining in such a way auxiliary processes needed as recursion entry points in the CSP description.

An obvious question here is why invent something new, if one can use FSM diagrams as they are for specifying CSP processes. If one aims to capture only simple CSP processes that contain only guarded alternatives, prefix operators and event occurrences, then an FSM is a good enough abstraction. Those diagrams are however not used to depict examples containing, for example, parallel, internal choice, hiding and renaming operators. Other issues are that different diagrams can be drawn for the same process (depending on whether recursion is expanded and how many times), and that parameterized processes with infinite number of states are impossible to draw.

Processes composed in parallel are sometimes (Schneider, 2000) represented as rectangle boxes adorned with ports representing events in the alphabet of the process. Such process boxes are then related via lines that connect ports representing event-ends offered to the environment for synchronization.

One of the approaches to a structured way of specifying concurrency derived from CCS (Milner, 1989) and CSP process algebras and using state transition diagrams is named FSP (Magee and Kramer, 1999). FSP provides a Java library implementation and a tool for model animation and checking. Compared to CSP literature, this approach goes one step further in visualizing process expressions. The initial (*start*) state is shaded. The letter E inside a state circle denotes the *end* state. A *sequential* composition is created by concatenating two finite state machines: making a transition from the end of a subprocess to the start of the next one in line and hiding *start* and *end* states resolved in this way. A *parallel* composition is also visualized via creating an equivalent state machine or alternatively again as connecting appropriate ports of boxes representing subprocesses. In addition, the notation allows systematic adding of prefixes to the names of event transitions by altering the process labels associated with states. *Hiding* is performed by replacing the event name with the keyword `tau` and reducing the state machine by merging states related by the `tau` labeled transitions. *Renaming* is performed by making a new state machine with names changed according to the replacing function. Time is introduced using `tick` events.

Another tool based on the CSP and FSM way of visualizing is i-MathicStudio (Hilderink, 2006). Software design in i-Mathic consist of hierarchies of *black-box* (abstract) and *white-box* (concrete) specifications, which are subject to refinement verification. The model checker FDR is used for refinement verification and detecting deadlocks and livelocks in the software specification. In i-MathicStudio one of the ways to edit FSM-like CSP specifications is in the form of a state transition table, where each row contains one transition from some state to some other state. Rows are grouped according to the source state. Columns specify conditions, events, actions, the next state and a reference field respectively.

2.2.3 GML

In previous research at our Lab, GML (Hilderink, 2003, 2005a) was developed. GML is a design methodology and visual notation related to occam. At the end of

the design process, a complete GML model always defines an occam-like tree hierarchy with constructs and wrapper processes as branches, and user-defined processes as leaves. The main difference compared to a simple visualization of occam-like hierarchies is that in GML processes are related via binary relationships.

There are two types of binary relationships: compositional and communication. Any two processes are related by one of the compositional relationships: *sequential*, *(pri)parallel*, *(pri)alternative*. The meaning of those relationships is derived from the analogue CSP operators and occam constructs.

Communication relationships can be either unsynchronized *var-channels* or *rendezvous-based channels*. *Var-channels* are equivalent to using shared variables and cannot be used for communication between processes related via a parallel relationship (concurrently existing processes). *Rendezvous channels* are equivalent to occam and CSP channels and can be used only between processes that do exist concurrently.

A *process* is in GML depicted as a rectangle. A line representing compositional relationship is adorned with the symbol of appropriate CSP-like compositional relationship. Symbols for two types of communication relationships are given in Table 2-1.

Table 2-1 Communication relationships

CSP	occam	CT library	GML Symbols
channel (kind of event)	channel	Rendezvous (synchronous) channels	channelName : dataType (line with filled arrowhead)
variable	variable	VAR (asynchronous) channels	channelName : dataType (line with open arrowhead)

It is possible to display only communication relationships (*communication view*) or only compositional relationships (*compositional view*). The *Communication view* and the *composition view* define two separate layers of edges on top of the same process nodes layout. *Communication* and *compositional view* are therefore considered orthogonal. Sometimes it is convenient to view both layers in the same time (*hybrid view*).

Specifying a binary compositional relationship between two processes is not identical to relating them via a CSP operator or an occam construct. For instance, a single parallel binary relationship relating two processes, does not imply synchronization on start and termination events as the *parallel* operator of CSP and the `parallel` constructs of occam and the CT library do. Only a complete set of

binary relationships or explicitly grouped relationships of the same type can imply that some set of relationships is actually participating in making some occam-like construct.

GML models can express designs that are illegal or underspecified or ambiguous. Additional consistency checks are needed before designs can be translated to CSPm scripts, occam or CT library code. Compared to constructing an occam-like application as a hierarchical-tree, GML models seem to offer more flexibility as a design entering view, especially in early stages of the design, when relationships between some processes are known, but the exact borders of the constructs are not yet quite clear.

Sequential

The *binary sequential relationship* of GML is a weaker concept then the *event prefix* or the *sequential* operator of CSP. The GML *binary sequential relationship* specifies precedence rather then exact order. The `Sequential` construct created from the set of *binary sequential relationships* implies grouping based on established strict order among the involved processes.

Since event-end synchronization points are in GML considered basic processes (*writer* & *reader*), GML uses the *binary sequential relationship* to specify both the prefix and the *sequential* operator. Naturally, *sequential relationship* whose source is an event-end is assumed to be *event prefix* operator. In Figure 2-8 the binary sequential relationship between the *writer* basic process and the process P1 is mapped to an CSP *event prefix* operator and the one between P1 and P2 into *sequential* operator.

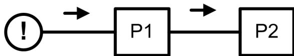


Figure 2-8 Binary sequential compositional relationships

Parallel

In GML, a *parallel* binary relationship is again weaker than specifying a `Parallel` construct in occam/CT approach. Synchronization on the ‘start’ and ‘termination’ event is implied for processes grouped in a `Parallel` construct, but not for processes related via *binary parallel relationship* in isolation. The *parallel compositional relationship* does imply that somewhere upwards in the hierarchy of constructs there will be a common parent `Parallel` construct containing both processes related via this *binary parallel compositional relationship*. A `Parallel` construct is constructed by grouping a set of processes related via parallel binary relationships.

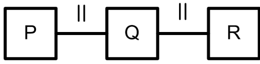


Figure 2-9 Binary parallel compositional relationships

A set of parallel relationships as depicted in Figure 2-9 is ambiguous. Intuitively, one would conclude that it is a `Parallel` construct consisting of P, Q and R.

However, the undefined relation between P and R also leave space for combinations: (P; R)) || Q and (R;P)||Q

Choice

GML does not define special symbols for the *conditional choice* and the *internal choice* operators. As in occam, *internal choice* is assumed not useful in practice.

The *conditional choice* can only be specified inside *code blocks* using the IF control block of native programming language (C++, Java…). In fact, introducing an IF *choice binary relationship* would make GML more expressive. The symbol associated with such a relationship could for instance be the same as in CSP. Generalization of the IF construct, with more then 2 outgoing branches, is in most programming languages supported in a form of SWITCH control structure. Semantically, the *switch* control structure can be implemented using nested *if* statements. Again, introducing the *switch* symbol as a part of basic vocabulary would make GML more expressive.

GML does define the *alternative binary relationship.* A grouped set of *alternative* binary compositional relationships is equivalent to specifying the ALT construct in occam. As in occam and the CT library, a common join point is assumed for all alternative processes. Figure 2-10 gives the GML visualization of CSP expression:

$$(cond_p \ \& \ (action_p \ ; (event_p \rightarrow P'))) \ \square \ (cond_q \ \& \ (action_q \ ; (event_q \rightarrow Q')))$$

$event_p$ [$cond_p$] $action_p$ $event_q$ [$cond_q$] $action_q$

□

P Q

Figure 2-10 GML symbol for alternative relationship

The graphical notation maps to the occam notion of ALT construct as a group of alternative processes where each one offers an initial event to the environment. Comparing the notation to CSP, the way of visualization implies that a choice is made between processes as in *external choice*, but also that events are part of the choice operator as in *guarded alternative*.

Grouping

If binary relationships being specified between any two processes, and providing as such a complete specification, then it implicitly defines grouping of processes into a tree-hierarchy of constructs. For instance, the model in Figure 2-11 a) is ambiguous and can have two possible solutions that resolves it to a tree-like hierarchy of constructs: P;(Q||R) and (P;Q)|| R, respectively visualized in Figure 2-11 b) and c).

Sometimes it is safe to leave a relationship unspecified because there is only one possibility that is not illegal and it can be deduced from other relationships. Sometimes it is not. Relying on specifying a complete set of binary relationships, in order to specify constructs, obviously clutters even simple diagrams. However,

using the grouping notation enables one to reduce significantly the number of specified binary relationships. From these reasons, the gCSP tool requires explicit grouping.

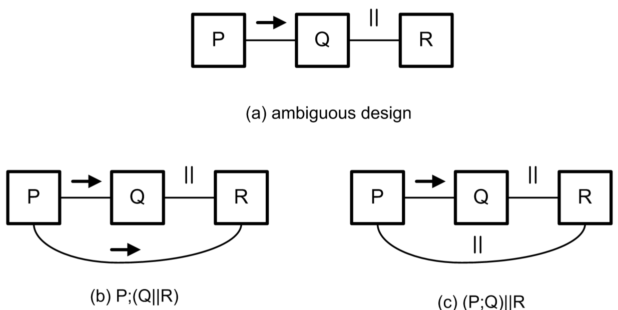


Figure 2-11 Resolving ambiguity (adapted from (Hilderink, 2005a))

Normally, grouping is explicitly defined by using explicit grouping symbols. CSP construct borders are most naturally visualized using a rectangle around the group of processes (so-called *box notation*) as depicted in Figure 2-12 b) for the CSP expression P;(Q || R) from Figure 2-11 b).

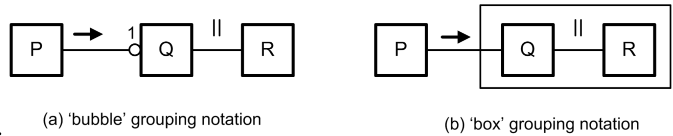


Figure 2-12 Grouping in GML models (parenthesis and box notations)

The preferred way to specify grouping in GML is using the *parenthesis notation*, as in Figure 2-12 a). This notation relies on specifying indexed bubbles on the ends of binary relationships. The *parenthesis notation* is in fact a more compact representation of the *box notation*. Wherever a box crosses a binary relationship, instead of the box, a bubble is drawn (or if it already exists its index is incremented) on the side of the relationship that belongs to the internal area of the box. The index of a bubble is the number of such boxes crossing the relationship on that place. In fact, this way to visualize grouping is similar to the way parentheses are used in CSP expressions. A *bubble* is always placed on the side of the compositional relationship next to the process that is considered to be inside the parenthesis.

When processes are grouped into a construct, this construct can actually be considered as a new process that inherits compositional relationships of all subprocesses with other processes from its environment. Inheriting two different types of relationships to the same outer process indicates that the attempted design is illegal.

A disadvantage of the preferred *parenthesis notation* is that for the untrained eye it is quite difficult to spot borders of the constructs. Regardless of the chosen grouping notation, using binary relationships makes reconstruction of the exact control flow difficult, especially in complex examples.

The gCSP tool introduces an additional view that gives more insight in the way grouping is resolving the set of processes related via binary relationships into the tree-like hierarchy (Jovanovic et al., 2004).

A GML design is in fact a single diagram that can be split into several hierarchical views by collapsing/hiding parts of the hierarchy in separate views. It is however not envisioned that the same component participates in several views focused on different aspects of the system, as is the case for instance in UML models.

A GML modeling is a free-choice combination of deep and flat hierarchy grouping. Flat hierarchy modeling represents several compositional levels in the same view and signifies grouping via bubbles on relationship ends. Deep hierarchy modeling means that borders of the groups are made clear by either a rectangle around it (*box notation*) or hiding submodels in separate views (*containment hierarchy*).

Recursion

Recursion in CSP has a meaning of jump to the place in control flow marked with the process name used in recursion. Occam, CT and GML lack ways to express recursion in ways other then repeating the same part of behavior in a loop (WHILE construct in occam and loop control blocks of native programming languages in CT library). Special GML symbols for loops are displayed in Figure 2-13.

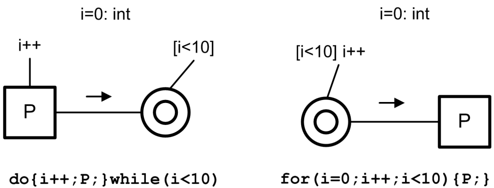


Figure 2-13 Control loop symbols in GML

CSP recursion is for instance used to split FSM-like designs on several process descriptions. The lack of similar concept in GML is one of the reasons that GML is not suitable for making FSM-like diagrams.

Representing FSM-like designs

Simple CSP processes, made out only of event synchronization points connected via the prefix and the guarded alternative operator, are often visualized using a *Finite State Machine* (FSM). Every node in such FSM represents a state of the process and every edge/transition is associated with some event.

Alternative binary relationships are grouped in ALT constructs, and as in occam and CT library, such a construct requires a common join point and is thus not suitable for FSM-like designs. A CSP expression is visualized naturally via FSM approach, as depicted in Figure 2-7. In GML it can be represented for instance relying on the design pattern for coding FSM-like designs in occam-like approaches described in section 2.1.5.

If in Figure 2-14 the variable `state` is set to the value equal to one, the transitions associated with events 'ev1' and 'ev2' are allowed (note in FSM given in Figure 2-7 that the transitions associated with events 'ev1' and 'ev2' lead from the state marked with 1). The occurrence of event 'ev1' updates the value of the variable `state` to the value 3 and the body of the process 'from state S1 to S3' is executed. In the next iteration, since the value of the variable state is equal to three, only the event 'ev3' is allowed. Figure 2-14 illustrates that this approach results in a rather unreadable diagram. The size of diagram in fact scales with number of transitions instead of with number of states.

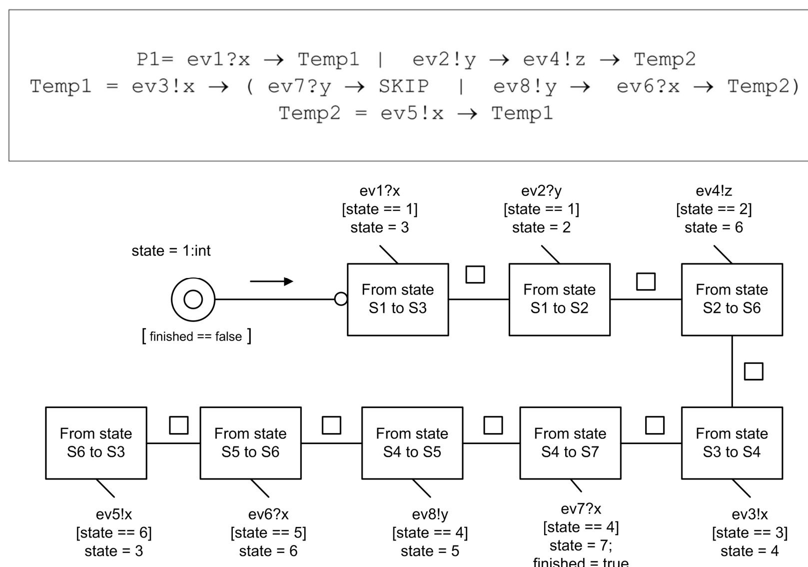


Figure 2-14 The FSM visualized in GML

GML was designed for adding the concurrency structure to existing data flow diagrams and not for FSM kind of designs. The conclusion here is that GML is not suitable for this purpose. The recommendation is to envision other ways to capture FSM-like designs in CSP manner.

Priorities

GML provides symbols for the *prioritized* versions of binary parallel and alternative compositional relationships. In this way, a relative priority ordering can be specified between any two processes. As a result, the related processes will have common prioritized construct of appropriate type located somewhere upwards in the hierarchy of parent processes. A tool can check whether the priority specification is conflict-free.

Symbols for priority ordering are created by adorning existing symbols of parallel and alternative relationship with an arrow directed towards the process of higher priority.

Exceptions

GML defines the *exception relationship* based on the concept similar to the *interrupt operator* of CSP. In Figure 2-15, the process P is guarded for exception and process Q handles the exception.

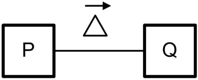


Figure 2-15 Exception compositional relationship in GML

In the CT library implementation, the *binary exceptional relationship* is replaced with an `exceptional construct` containing both processes P and Q. The process Q is an `exception handler`. After an exception is thrown by process P, it will be caught by the associated `exception construct` and forwarded to the `exception handler` Q. It is possible to specify a chain of exception handlers.

Design process

In GML, the recommended design process starts with process blocks existing in isolation. The first design step is making the communication structure as a standard data flow model. Next, the concurrency structure is added to this model by specifying binary compositional relationships between involved processes. Some relationships are known in advance and some are subject to various trade-offs. For instance, instead of specifying immediately that there is a parallel composition of processes A, B and C, one might first conclude that processes A and B should be executed in parallel. Only after the same type of relationship is made between for instance B and C, it becomes possible to group A, B and C into one parallel construct.

Data-flow orientation

GML is a convenient method for resolving the concurrency structure of the data flow diagram without changing the existing layout of the original data-flow model.

Let us consider an example of a control task that can be further decomposed into a hierarchy of models related in a *precedence constraints diagram* as given in Figure 2-16. The origin of the specified precedence relations is in the data flow between submodels.

In GML, given data flow block scheme can be refined by adding compositional relationships – e.g. *sequential relationship*s with non-synchronized data passing in between (*var-channels*) or *parallel relationships* with *rendezvous synchronized channels* in between. E.g. the data flow model given most left on the Figure 2-16 can for instance be implemented as (P1;P3;P5) || (P2;P4;P6) (see the middle diagram in Figure 2-16).

As a consequence of chosen compositional relationships some data channels are of the rendezvous type (pairs (P2,P3), (P3, P4) and (P5,P6) in Figure 2-16) and some are var-channels as illustrated on final version of *communication diagram* (depicted most right on Figure 2-16).

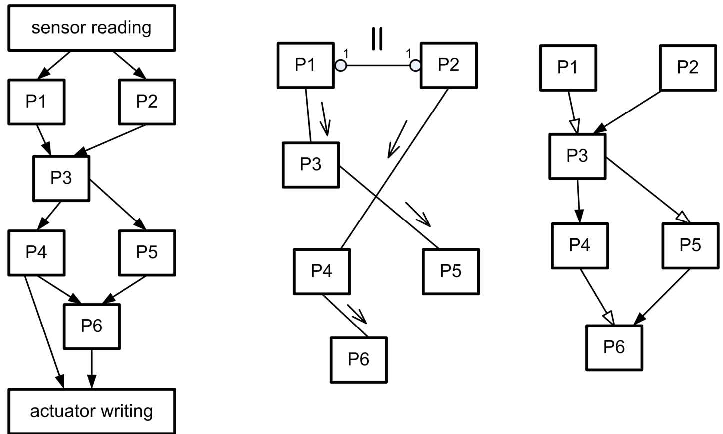


Figure 2-16 Refining data flow model to compositional and communication view

Comparing *communication* and *compositional view* on Figure 2-16, one can come to the conclusion that they are not completely independent because the same information is encoded in both views. Var-channels indicate *sequential relationships* and rendezvous channels indicate *parallel relationships*. However, note that the *communication view* is in the general case not enough to reconstruct the compositional view, since *parallel* and *sequential relationships* might exist where channels are not present and there might be alternative relationships that do not allow any kind of channels between related processes. In addition, the existence of some compositional relationships, without the complete specification or explicit grouping symbols, does not say enough about their grouping into a hierarchy of constructs.

GML visualization of a model easily becomes overcrowded with data and thus hardly readable. This is especially the case when both communication and compositional models, as carrying together the complete information of the system, are depicted simultaneously. To illustrate this, on Figure 2-17 the *hybrid view*, containing both *communication* and *compositional views* from Figure 2-16, is depicted.

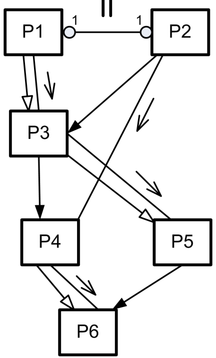


Figure 2-17 Hybrid view

2.2.4 Discussion

While the CT library is adopting an occam-like viewpoint, GML is positioned somewhere in between CSP and occam viewpoints. GML is a description language that aims to allow significant design freedom needed in early stages of the design trajectory and to allow easy transition to a final stage where the model is shaped as an occam-like tree hierarchy. Design freedom is based on relying on specification of binary compositional relationships between process blocks. This is done in order to: specify relative compositional relationship between two processes in the isolation from the rest of the system, allow compositional ambiguity during the design process and in that way postpone design decisions about exact concurrency structure. Final models, shaped as occam-like hierarchies, allow a choice of code generation based on primitives from the CT library or occam.

Allowing one to, instead of immediately specifying constructs, make explicit binary relationships (sequential, parallel, alternative) between any two processes offers a high flexibility during the design process. GML models can express designs that are illegal or underspecified or ambiguous. Compared to a making an occam-like tree-hierarchy directly, GML models seem to be much better suited as a design entering view. Still, design freedom comes with a price that the specification should not be illegal or ambiguous in order for code generation to be possible.

As illustrated in Figure 2-16, refining the data flow model with compositional relationships expressing the concurrency structure can be done without changing the 2D layout of the original data-flow model. This feature makes a prospective tool based on GML suitable for application in a chain of tools, with the preceding tool in the chain producing a data flow model e.g. based on some application-specific domain. In our research group focus is on development of control systems and a possible predecessor to a GML-based tool is 20SIM, a tool for modeling and simulation of control systems based on bond graphs and control theory.

In fact, one can state that the data flow orientation of GML makes it not suitable for FSM-like designs that are essentially specifying control flow. In control systems application area, control loop internals is convenient to represent in a data flow model and event based supervisory and sequence control processes are better visualized in the control flow based FSM-like approach. Thus, both control flow and data flow oriented views are needed to visualize concurrency in control systems.

Compared to CSP as its source, GML suffers from the similar expressiveness deficiencies as CT libraries and occam. The internal choice is missing, recursion is possible only in a form of loop constructs, and the external choice is possible only in the shape of Alternative construct of CT and occam. The control flow element for conditional branching (IF) of control flow is also missing. As in occam/CT, a finite-state-machine-like behavior is not straightforward to express. In fact, a good idea would be to extend GML with a different view that can capture finite state machines. A simple way to do that would be a table layout approach as the one in i-MathicStudio.

GML considers that the *communication view* (data flow expressed via rendezvous and var channels) and the *compositional view* (set of binary compositional relationships among process blocks) are orthogonal. However, as illustrated in the example of Figure 2-16, this is not completely true since the presence of *rendezvous channel* communication (equivalent to a CSP event) implies a parallel *compositional relationship* and *var-channels* imply either *sequential* or parent-child relationship.

Perhaps the biggest disadvantage of GML is that relying on binary relationships does not scale well with complexity. Using bubbles as a way of grouping reduces the number of binary relationships that needs to be specified and is more compact then the box notation. However, it leads to somewhat reduced readability, at least for untrained eyes. This is especially the case in hybrid diagrams, depicting in the same time both compositional and communication view as in Figure 2-17. Consequently, GML makes the representation of complex designs visually cluttered, instead of making it simple and intuitive. It is hard to envision GML-like diagrams without external media like paper or a computer screen.

A GML design is intended to be one diagram/graph, whose compositional / containment hierarchy structure can be either fully exploded to show everything in a single view, or internals of some processes/constructs can be hidden on the current abstraction level and be depicted in a separate view. In other words, in GML and gCSP it is not envisioned that same element can exist in different views. Different views are thus related exclusively via the *containment hierarchy*. It is not envisioned to focus on one part or aspect of behavior of some entity in one diagram/view and on another aspect of the same element in another diagram/view, as is possible in UML.

GML relies on CSP formal checkers for checking consistency of designs whose compositional structure is not illegal or ambiguous. Before the code generation to CSPm scripts is possible, ambigouities in designs need to be resolved. The gCSP tool does this by enforcing explicit grouping and using knowledge of the grouping action to keep track of growing sub-trees in tree-hierarchy of constructs and user-defined processes. When all binary relationships are resolved in constructs placed somewhere in the tree-hierarchy and all processes except the top-level process have a parent construct, the design is ready for the next stages: formal checking and source code generation.

2.3 Component engineering practice

CSP allows one to structure concurrency in a compositional way, with focus on the interaction between components. However, a component in the CSP sense of the word is something different from a component in the modern sense of the word. Normally, a component is reusable structural unit that has an identity. CSP basic components are processes, which are more behavioral than structural units. A CSP process can though specify the way in which some structural unit, e.g. a component will behave in its interaction with other components in its environment. This interaction is viewed only via the event synchronization pattern describing the

behavior of the component on its interface. The same structural unit can in fact exhibit different behavior on different interfaces.

CSP has a potential for describing the interactions between components. Occam, CT and GML however miss to use the capabilities of CSP for component-based software engineering. They do use processes as components with channels playing the role of ports. However, they do not keep up with modern notions of component-based design.

An *Architecture description language* (ADL) expresses system (software/hardware) architectures using terms as ports, components, connectors. Some ADL (e.g. MetaH (Honeywell, 2007)) also include specification of additional properties like execution time and failure modes.

In (Crnkovic and Larsson, 2002), basic notions of component-based development are defined as follows. "A *component* is a reusable unit of deployment and composition that is accessed through an interface. An *interface* specifies the access points to a component. The component specification can be achieved through *contracts*, which make sure certain conditions hold true during the execution of a component within its environment. A *framework* describes a large unit of design with defined relationships between participants of the framework. The last term discussed is *patterns*, which define recurring solutions to recurring problems on a higher level of abstraction. Patterns enable the reuse of the logical solutions and have proven to be very useful. "

It is further stated that, due to the requirement of integration of a component into an application, the most important feature of a component is the separation of its interfaces from its implementation. Component integration and deployment is completely independent of the component development life cycle and there should be no need to recompile or relink the application when updating with a new component.

2.3.1 Component frameworks

Many component frameworks do exist. Here we chose to focus attention on the few most interesting regarding the needs of this project for adding support for basic component based development on top of the CSP-based design specification notation.

COM / DCOM / MTS / COM+ (Eddon and Eddon, 1998) is a stream of subsequent Microsoft component frameworks, with each one keeping backwards compatibility with previous ones. DCOM extends COM with distribution, and MTS extends DCOM with persistency and transaction services. A COM object comes in binary code, which is a way to avoid recompiling and relinking and to allow plug-and-play integration of components. All interfaces inherit from the IUnknown interface, which contains functions to dynamically discover the type of interface of the addressed COM object and to maintain the existence of a COM object via obtaining/releasing references to it.

The Koala (van Ommering, 2004) component model is used by Philips to create product line architectures for consumer electronic products. As in COM, a Koala interface is a set of functions. Connections between components are possible in the form of direct binding of *provided* and *required* interfaces, in the form of the *glue module* and the *switch connector*. The *glue module* is used, for instance, when two interfaces do not exactly fit. In the *glue module* it is possible to add glue code or fit parts of several different interfaces as a replacement for a single one supporting all those features. Koala defines a custom expression language for specifying *glue code*.

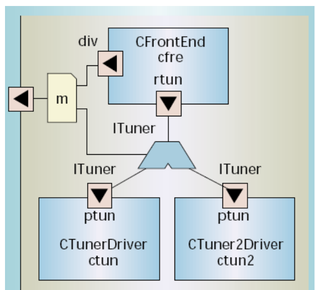


Figure 2-18 An example (van Ommering, 2004) of system specified in Koala

Switch connector is an example of a *glue module*, but it is used so often that it has a separate symbol. *Switch connector* is used to switch binding between components. In addition, Koala components can be parameterized through *diversity interfaces* – those are just required interfaces marked with the ‘div’ label, that needs to be binded to some other component that can provide the values for those parameters.

In UML2, components are adorned with a special symbol (see upper right corner of component1 in Figure 2-19).

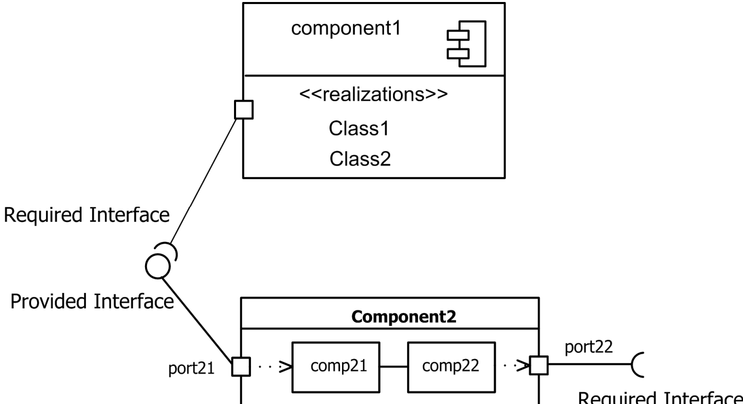


Figure 2-19 Ports, provided and required interfaces in UML2

Optionally, components can have a compartment with contained classes (Class1 and Class2 in component1 in Figure 2-19). Port is an access point to a component

or a complex class. It is visualized as a small square on the top of the border of a component/class (e.g. see port21 and port22 in Figure 2-19). Port is associated with one or more interfaces. Two types of interface exist: *provided interface* and *required interface* (see Figure 2-19). In addition, port may be connected to inner part that provides/requires implementation of the interface (e.g. see relation between comp21 and port21 in Figure 2-19).

The ACME Architecture Description Language (ADL) (Garlan et al., 2000) defines components, ports, connectors and roles. The structure of a system is specified by a set of *components*, a set of *connectors*, and a set of *attachments*. An *attachment* links a *component port* to a *connector role*. Introduction of *connectors* allows one to reason about them in isolation. A component can internally contain other components and a set of *bindings* (links from an internal port to an external port).

WRIGHT (Allen, 1997) is an Architecture Description Language (ADL) that relies on CSP to describe the architecture of software systems. Basic abstractions of WRIGHT are: *components*, *connectors* and *configurations*. A *component* consists of two parts: a computation and an interface. The *interface* consists of ports. Each *port* is an interaction in which the component can participate. The use of ports is to allow consistency checking and to guide programmers in the use of the associated component. WRIGHT is a textual way to describe architectures. No visual notation exists. However, ACME studio (Schmerl and Garlan, 2004), an editing environment and visualization tool for software architectural designs based on the Acme, does provide space for specifying protocols using text-based syntax of Wright. The accompanying Wright toolset can translate Wright-annotated Acme description into Wright, parse the software architectures defined using the Wright ADL and translate them into machine readable CSP (for use with a formal model checker – e.g. FDR) and Acme ADL.

2.3.2 Interaction management – the notion of contracts and connectors

Component-based software engineering is in practice most often based on the client-server architecture model, where one component (server) provides a certain service and the other component (client) uses that service. In some application areas, typical generic patterns are captured in the shape of standardized and precisely defined client side and server side interfaces (e.g. OPC (2007)). This allows system integration based on components supplied by different vendors. Integration efforts are minimized as long as the used components adhere to these prescribed interfaces.

In a client-server system, the contract specifying interaction scenarios and adjustable parameters of service delivery is implicit and partially reflected in the interface definitions of the provided and required services. Sometimes those interfaces also offer services for negotiating contract parameters.

Making an explicit entity that implements such a contract is considered to be unwanted overhead. The client/server approach is justified for data processing systems with clearly directed data flows. Data flows are in such systems starting with a client's request to the service provider, which can, in order to provide its service, further delegate part of its task to some other service provider(s) and in that way act as a client of the next component(s) in the client/server chain/tree. The obtained results travel in the opposite direction.

Complex client/server systems may however require existence of components that provide the management of interaction between several involved components. Components managing interaction of other components are in fact specifying and implementing explicit contracts governing interaction. For instance, a typical case would be providing, for fault tolerance reasons, redundancy in the form of replicated server components and an additional component/contract governing the interaction of the involved components.

Sometimes, e.g. in complex control applications, interaction between components is not natural to structure as a chain or tree of clients and servers. For instance, devices in an industrial production cell system need to cooperate as peers in order to provide a result. Every participating device has a precisely defined role, but it is not always clear what is the service, and if some component is in that interaction playing the role of a server or of a client. Instead, interaction between components is an interaction of peers that work together to achieve some higher-level behavior. In those situations, a structured approach is to introduce entities that will manage and supervise interactions between components. Such an entity is in fact defining an explicit contract between the involved components.

In Wright (Allen, 1997), the connector specifies the interaction between a set of components. It does that by providing the description of Roles representing expected behavior of participants and the *Glue* representing the specification on how the participating roles cooperate in the scope of the interaction managed by the connector. A Configuration is a set of component instances combined via connectors.

In (Zorzo et al., 1999), *Coordinating Atomic Actions* are introduced as a way to structure safety-critical systems involving complex concurrent activities. A *coordinated atomic action* (CA action) is an entity in which two or more threads of control implementing the roles of the participating components meet and synchronize their activities performing atomically a set of operations on a set of objects belonging to the CA action entity. In this way, a CA action behaves as a transaction and represents a general framework for dealing with faults and providing ways of recovery. Obviously, a CA action managing interaction contains more information needed for handling composite exceptional occurrences than any of the participating components in isolation. This makes CA actions a structured design pattern convenient for usage in safety-critical systems. The CA action design pattern is in (Zorzo et al., 1999) illustrated on a model of the Production Cell case study.

In (Boosten, 2003), a *formal contract* is introduced as a design pattern that manages interaction among components in a side-effect-free way. A formal

contract is implemented as a state machine that codes interaction between components relying on a system of asynchronous modification requests from components to contract and state change notifications from contract to components. The contract is promoted as an interaction entity that should substitute occam/CSP channels, since its ability to capture a complete N-directional specification of interactions between involved components makes it superior to channels usage. It is suggested that it is possible to transform a formal contract, being a state machine, into a CSP specification allowing in that way formal checking of the interaction patterns managed by contracts. The usage of such a formal contract is foreseen as a support useful during the full development cycle. The paper reports that usage of formal contracts in a real-life software problem resulted in a significant reduction of complexity and elimination of some typical problems related to the unstructured use of concurrency.

In (Beugnard et al., 1999), making components contract aware is argued in order to be able to trust components employed in mission-critical applications. This paper deals with client-server architectures and identifies four levels of increasingly negotiable properties in the contract specification.

On the *basic (syntactic) level*¸ there is the *interface description language* (IDL)–like description of contract properties. This includes services/operations a component can provide/perform, associated input and output parameters and possible exceptions that may be raised during operation. Component frameworks that support (only) first-level contracts are, for instance: CORBA, Component Object Model (COM), JavaBeans.

Level 2 contracts are *behavioral contracts*. These contracts offer the possibility to specify pre-conditions, post-conditions and invariants for the performed operations. Typical examples of level 2 contracts are “design by contract” in the Eiffel language (Meyer, 1992; Nienaltowski and Meyer, 2006) and Object Constraint Language (OCL) of UML.

Level 3 are *synchronization contracts*. Contracts on this level specify behavior in terms of synchronizations and concurrency, e.g. whether a dependency between provided services is parallelism, sequence, shuffle, etc. The Service Object Synchronization (SOS) mechanism is an example for contracts on this level.

Finally, level 4 contracts allow dynamic adaptation of the contract based on *Quality of Service* (QoS) requirements. TAO (the adaptive communication environment object request broker) is an example for a level 4 contract.

Although the “four level” classification of contracts was introduced for implicit contracts of the client-server architecture, the classification is still a useful way to define more precisely the position of explicit notion of a contract.

2.3.3 Discussion

Component-based frameworks in general lack the structured way of handling concurrency. Wright ADL does handle that issue, but there is no visualization of

the concurrent behavior. The design methodology aiming to reduce complexity of the design process does need a support for component-based development. This include ways to specify components as structural units of deployment and composition, ease of dynamic reconfiguration, interface management system, and interaction management via some kind of a formal interaction contract. Visual notation should, as a minimum, provide ways to specify components, ports and provided and required interfaces. Some notion of interaction contract would help to specify interactions in more structured way.

2.4 Conclusions

This chapter illustrated that in the current state of the art, there is a considerable gap that can be filled with combination of several complementing aspects. CSP is a theory that allows creating relevant models focused on interaction in concurrent systems and allows formal checking. The programming language implementations of CSP that exist in practice are not only subsets of it, but also introduce a somewhat different way of thinking.

CSP formulas are hard to follow and proper visualization could make the CSP designs more readable and more applicable in practice. GML attempts to achieve this, but is geared towards an occam-like approach and is based on the idea of extending existing data flow diagrams with binary compositional relationships. However, this results in reduced readability of the control flow and is often a less suitable way to visualize designs than the introduction of simple control flow elements.

Finally, component-based frameworks in general lack structured ways to capture interaction on their interfaces. Even when this is taken care of, visualization of specified interaction is missing, which makes this way of design less attractive to a prospective software designer.

3 SystemCSP

Absorb what is useful, discard what is useless, and add what is essentially your own.

Bruce Lee

This chapter introduces the SystemCSP – a new graphical design specification language aimed to serve as a basis for the specification of formally verifiable component-based designs of distributed real-time systems.

SystemCSP is based on the principles of both component-based design and CSP process algebra. Such a combination promises to offer a more structured approach and more expressiveness than the one offered by the occam-like approach targeted in GML.

GML is geared towards producing occam-like programs. It provides a lot of design freedom in early stages of design by relying on idea to relate processes via binary compositional relationships instead of starting immediately with occam-like constructs. However, experiences with GML lead to the conclusion that although binary relationships are useful in early stages of design, they tend to clutter readability of even relatively simple diagrams.

With SystemCSP, we attempt to make a paradigm shift from occam towards CSP. CSP offers more expressiveness for specifying concurrent systems then its occam related subset. Instead of relying solely on binary relationships as in GML, SystemCSP tries to make a trade-off between the need for design freedom in early stages of the design process and the need for providing readability of control flow in late stages of the design process.

Graphical elements introduced in SystemCSP are related to basic elements of the CSP process algebra. In this way, designs have immediate mapping to CSP expressions. A CSP description can also be mapped to an appropriate graphical representation in SystemCSP.

Section 3.1 deals with symbols used for basic elements. Section 3.2 puts focus on the specification of behavior in CSP-based, control flow oriented, manner. Section 3.3 deals with ways to specify interaction between components. Section 3.4 presents ways to specify basic structural units – components. Interactions are specified in separate views (interaction views) centered around interaction contracts. Participating components are related via binary compositional relationships. Section 3.5 illustrates a way to visualize the distribution of components (belonging to the same interaction) to different nodes. Section 3.6 deals with a comparison to some other approaches for visualizing software models. Section 3.7 specifies the position of SystemCSP diagrams in a more broad development process. Section 3.8 deals with implementation issues. At the end in section 3.9 some conclusions are given.

3.1 Basic elements

In SystemCSP, behavior is visualized using diagrams whose basic elements are: event-ends, event prefix operator, process labels, process blocks.

3.1.1 Events

Event-ends

Basic event-ends are: START, EXIT, STOP, EventSync, Writer, Reader, and EventAccept (Figure 3-1).

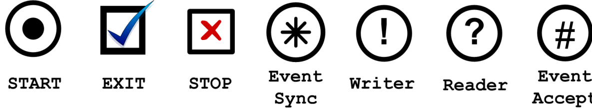


Figure 3-1 Basic elements

The existence of a *start* event is implicitly assumed in CSP, but is not written down in CSP expressions. In the visual notation, however, it is very useful to mark the entry point of a process or a component. The START element is marking the entry point of a component.

The EXIT element is a point of successful termination It is equivalent to the SKIP process in CSP. The name EXIT is chosen because compared to the name SKIP, it reflects better the intended purpose of the element.

STOP is, as in CSP, the process that does not engage in any event.

EventSync is an elementary process playing the role of event-end. It participates in event synchronization with one or more peer *EventSync* elementary processes executing in parallel. An event takes place when all participating *EventSync* processes are ready (*rendezvous synchronization*).

An *EventSync* can in general initiate or accept events. The difference is not important from the CSP point of view, but sometimes in designs, it is handy to know which side initiates the interaction. For a side that can only accept interaction, the *EventAccept* symbol is used.

The two other special kinds of *EventSync* symbols are *Writer* and *Reader* symbols that emphasize the direction of unidirectional communication associated with an event occurrence. The Writer and Reader basic processes are alike to channels in the occam approach.

EventSync processes usually have associated an *event label* that specifies the event name and the details of the related data communication, if any.

Event prefix

As the *event prefix* operator of CSP, the *event prefix* (abbreviated *prefix* in further text) is in SystemCSP shaped as an arrow (see left-hand side of Figure 3-2). As in CSP, it is used to specify the sequential order between an event and the rest of the process executed afterwards. A *prefix* can lead to some other diagram node element – e.g. to an event end.

Figure 3-2 Event prefix

At right-hand side of Figure 3-2, the first *event prefix* element relates event ‘a’ to the process expression starting with event ‘b’, and the second *event prefix* element relates event ‘b’ with the EXIT element. Thus, the specified process will participate in event ‘a’, than participate in event ‘b’ and successfully terminate.

3.1.2 Processes

Process labels

A CSP process is essentially a named entry point in some control flow. In SystemCSP, *process labels* (see Figure 3-3) are used to visualize process entry and recursion points. Thus, a distinction is made between a *process entry label* and a *process recursion label*. A *process entry label* represents the entry point of a process, and is attached via a prefix operator to an element of a SystemCSP diagram (with the exception of a prefix operator). A prefix leading to a *process recursion label* means that after the prefix operator, the process will continue behaving in a way as defined by the process entry label carrying the same name. This combination of *process entry labels* and *process recursion labels* allows natural visualization of recursions. In Figure 3-3, process P1 will first initiate event ‘a’, than wait for event ‘b’ and subsequently behave as process P2.

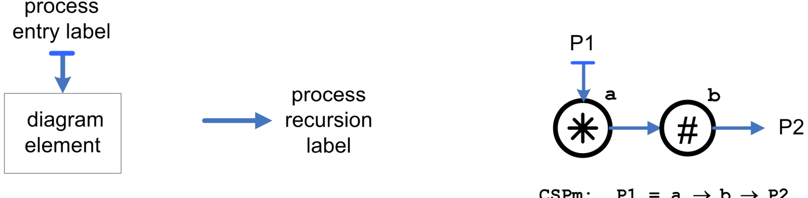


Figure 3-3 Process labels

Instead of using *process recursion labels* one can directly draw prefix arrows to entry points of appropriate processes (provided they are in the same view). However, using the *process recursion labels* makes diagrams more readable.

Interacting processes

Figure 3-4 illustrates how the previously introduced elements can be combined for describing three processes that interact by synchronizing on certain events. Process entry points are marked with *process entry labels* carrying names `P1`, `P2` and `P3`.

```
CSPm:
P1 = ev1?x!y → activate_P3 → SKIP
P2 = ev1!w?z → ev2?q → SKIP
P3= activate_P3 → ev2!r → SKIP
```

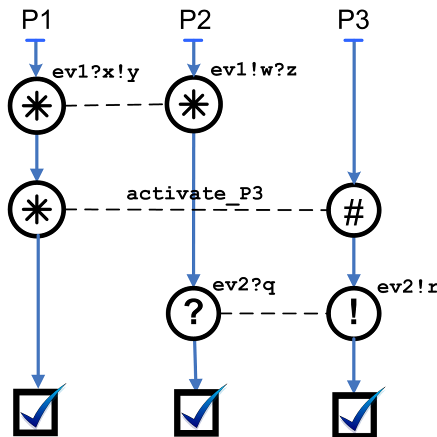


Figure 3-4 Combining basic processes with prefix control flow elements

In case data communication is present, the event label contains, in addition to the event name (‘ev1’, ‘activate_P3’ and ‘ev2’ in Figure 3-4), the description of the data communication. Signs “?” (read) and “!” (write) are representing the direction of the communication. The names following the “?” or the “!” signs represent the local variables used as destination (e.g. variable `q` in `ev2?q` event label in Figure 3-4), or source variables (e.g. variable `r` in `ev2!r` event label in Figure 3-4). As the source of data, it is possible to use an expression involving multiple variables, or a function that evaluates to a value of an appropriate data type. One event occurrence can have multiple data communications associated (e.g. `ev1?x!y` in Figure 3-4).

Note that in case of event ‘ev1’, either process `P1` or process `P2` can initiate the interaction, and that when both processes are ready to perform the event (*rendezvous synchronization*), the associated data communication will take place in both directions. The value of variable `w` of process `P2` will be written into the variable `x` of the process `P1` and the value of variable `y` from the process `P1` will be written into the variable `z` of the process `P2`.

Event ‘activate_P3’ is initiated by process P1, and accepted by process P3. This event has no associated data communication.

In the third interaction (event 'ev2'), focus is on emphasizing the direction of unidirectional data communication. Therefore, the basic processes *Writer* and *Reader* are used. Rendezvous synchronization is implied and it is not considered important which side initiates the interaction.

Interaction between two processes is represented via a *dashed line*. This line connects a pair of peer event-end synchronization points (*EventSync* elements) located inside those processes (e.g. 'ev1' event-end in process P1 with 'ev1' event-end in process P2 in Figure 3-4). Dashed lines are an intuitively good choice because a dashed line indicates discontinuity and the *EventSync* processes are points of discontinuity in the control flow of a process. Prefix arrows are, for instance, *solid, directed lines* because they do indicate that, between the points they connect, the control flow is not interrupted by interaction with the environment.

Process blocks

A *Process block* is a process separated from its environment via a rectangle box. Since a process is in CSP a named entry point in the behavior specification, not every process can be depicted as a *process block*.

Processes or process parts can be visualized with their internals exposed (*transparent-box* approach as used for processes Q2 and Q3 in Figure 3-5) or hidden (*opaque-box* approach as used for the instance q1 of process Q1 in Figure 3-5).

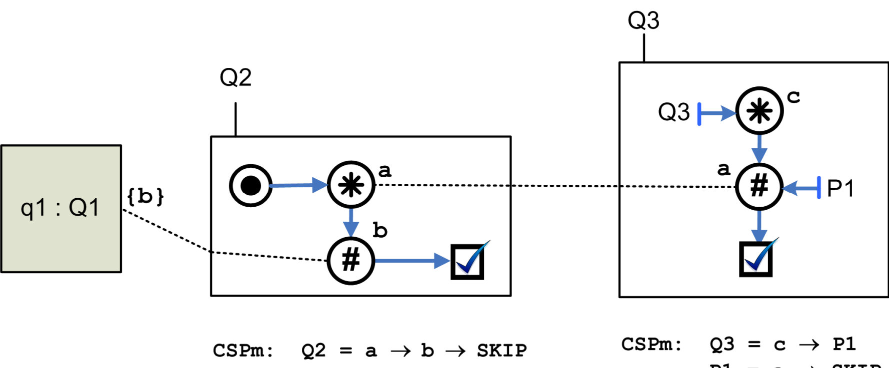


Figure 3-5 Process specification

The label carrying the name of a process is in case of the *opaque-box* approach depicted inside the *process block*. In case of the *transparent-box* approach, it is associated via a line to the *process block*.

A *process block* needs to have a single entry point. This entry point is marked either with the START event (as for process Q2 in Figure 3-5) or via a *process entry label* (as for process Q3 in Figure 3-5).

A process block can represent either a type definition or an instance of some type. Process types are abstractions that can be reused and instantiated in any context. Instances of processes can be named or not. When an instance is named, the

naming notation takes the form `instanceName:TypeName` (in Figure 3-5, q1 is an instance of a process of type Q1). If left unspecified, a process name is assumed to represent an abstract type or a nameless instance of a named type (e.g. nameless instance of process Q2 in Figure 3-5), depending on the context of its usage.

The rectangle box representing a process can have visualized a set of events exported to the environment in the shape of *event port labels* (e.g. event 'b' offered to the environment by instance q1 of process Q1 in Figure 3-5). An *Event port label* contains inside curly braces a name of an event or names of a set of events participating in the specified interaction. This label is associated with a point on the outer side of the *process block* rectangle. Explicit specification of *event port labels* is often omitted e.g. when the interaction line crossing the border of the *process block* makes its participants obvious (e.g. event 'a' in Figure 3-5) or when it is not considered crucial to visualize event names.

Non-interacting process blocks

Non-interacting process blocks are process blocks that do not interact with their environment. Internally, however they can contain any number of interacting subprocesses and EventSync processes. A special kind of *non-interacting process block* is a *code block*, which contains only pure computation code.

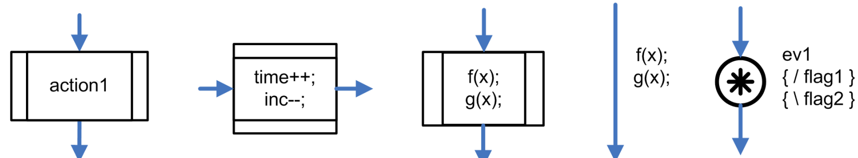


Figure 3-6 Non-interacting processes

A *non-interacting process* is not relevant for the CSP model at the level of abstraction where it is invoked and can thus be omitted in the resulting CSP description. *Non-interacting processes* are in SystemCSP specified in one of the ways depicted in Figure 3-6. The first three symbols visualize *non-interacting process block* descriptions like a rectangle box, isolated from the interaction with the environment via walls on the sides lateral to the control flow. The types of description illustrated in Figure 3-6 are respectively: a textual description or the name of the action or the scenario it represents, a detailed description of internals, or a sequence of functions invoked. Using a brief textual description is especially useful in early stages of the design process, e.g. while specifying in abstract way the behavior of use-case elements in use-case scenarios (see Figure 3-35 for example). The fourth example illustrates that in case of a *code block*, it is allowed to skip the box element, and associate the description directly with the prefix operator. This is, for instance, convenient in order to reduce number of displayed blocks when the *code block* is not so relevant for understanding the diagram.

Another issue is that often one needs only to set/reset some flag variable(s) or set the value of a variable that maintains the current state in the control flow. Specifying a code block using a rectangle is in such cases a bit of overkill. That is why the concept of *action block* is introduced. It is a small code block restricted to

setting/ resetting the value of flag variables and/or updating the state variables. An *action block* specified inside curly braces and a special notation is introduced to abbreviate setting (the symbol "/" that reminds visually on raising the value of signal) and resetting (the symbol "\" that visually resembles on falling value of signal) variables. An *action block* is always associated with some prefix arrow. It can, however (as in Figure 3-6), be visualized as associated with an event-end. In that case, in fact it is associated with prefix arrow that origins in that event-end. In Figure 3-6, immediately after the occurrence of event 'ev1', flag variable flag1 is set and flag 2 is reset. The need for the *action block* elements was realized during the work on designing the software for the study cased described in chapter 6.

3.1.3 Comments

Figure 3-7 Comment block

The shape of the comment block (see Figure 3-7) is deliberately cloud-like, because in that way it is completely different from other elements of diagram and it is easy to visually separate it from the rest of the design. Besides, a cloud is a symbol intuitively related to comments, thoughts and ideas. A comment block is intended primarily for short comments and keywords indicating the contents of an actual comment. A prospective tool is expected to offer a place for detailed description in an element property viewer e.g. located below the design editor area. When a comment block is selected in the design editor, its property field would display the complete comment.

3.2 Control-flow oriented elements

3.2.1 Elements related to CSP operators

Hiding and Renaming Operators

The hiding and renaming operators are applied to a process and the result is again a process. For this reason, in SystemCSP, those two operators are visualized using a rectangle element that relates the *process entry point* of the resulting process with the entry point of the (named or nameless) process used as the operand.

The environment of some process does not know its CSP description; it sees only the set of offered (ready) events. In CSP, the *hiding* operator is applied in the form: newProcessName = oldProcessName \ {set of hidden events}. The result is hiding the chosen set of events from the environment. In SystemCSP, the part of this specification containing the *hiding* operator and the set of hidden events is specified inside a rectangle box that relates entry points of the new and the old

process. In Figure 3-8, the hiding operator is applied on process SWITCH in order to hide event 'off' from the environment and to offer the resulting process under the name TURN_ON. The TURN_ON process can offer to its environment only the event 'on'.

```
CSPm:
SWITCH= on → off → SWITCH
TURN_ON = SWITCH \ {off}
ELECTRIC_LIGHT_SWITCH = light_on → light_off
                                → ELECTRIC_LIGHT_SWITCH
```

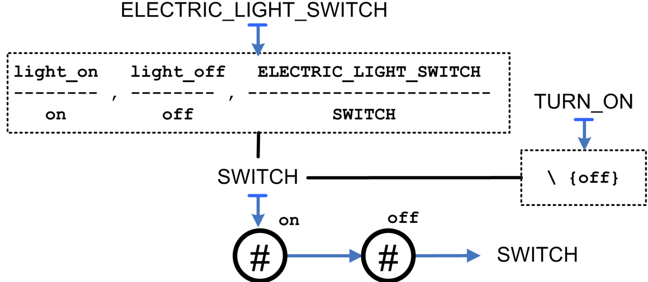


Figure 3-8 Example illustrating the usage of the renaming operator

The *renaming* operator of CSP replaces all occurrences of event/process names or event/process expressions with some other event/process names or event/process expressions. The symbol used in SystemCSP for the *renaming* operator relies on a notation that resembles to multiplying with the ratio of the new and old name (expression). This is an intuitively clear way to create the illusion of canceling the old expression and replacing it with the new one. *Renaming* is especially useful in situations when a process is reused in a context with a need for different names of events. In Figure 3-8 the *renaming* element specifies that events 'on' and 'off' are renamed into light_on' and 'light_off' respectively. The process created in this way is named ELECTRIC_LIGHT_SWITCH.

Conditional choice

An IF element (see left-hand side of Figure 3-9) specifies, inside square brackets, a boolean expression that represents a condition. It has two *prefix* arrows leading from it: the TRUE and the FALSE paths.

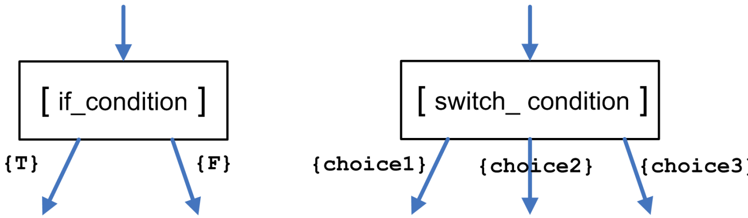


Figure 3-9 Conditional choice elements

Generalization of the IF control flow element is the SWITCH control flow element (see right-hand side of Figure 3-9). Unlike IF element, it can have more then two

outgoing *prefix* arrows, each one with a different constant value associated. In CSP, the SWITCH element can always be represented by several nested IF choice operators.

Preconditions and postconditions

Figure 3-10 displays on the left hand side, the operational semantics related to checking preconditions and postconditions, and on the right hand side, the symbol used to abbreviate that.

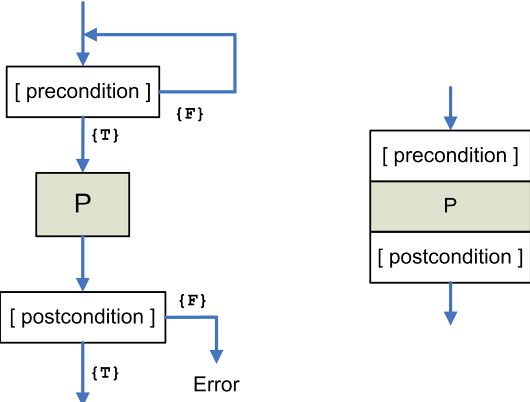


Figure 3-10 Specifying preconditions and postconditions

Guarded alternative

The *guarded alternative* is a process that offers to its relevant environment, a choice between several events. The branch starting with the chosen event will be followed. In SystemCSP this element (see left-hand side of Figure 3-11) is depicted as a rectangle box in which *SyncEvent* processes ('ev1' and 'ev2 in Figure 3-11) are half-emerged.

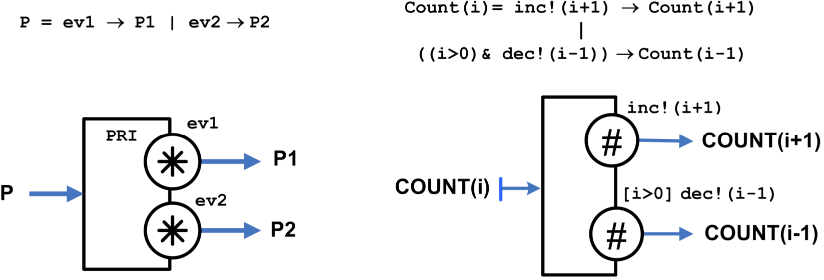


Figure 3-11 Guarded alternative

From the outer side of the *SyncEvent* circles, *prefix* control flow elements lead to communication patterns representing the alternative control flow branches. This is an intuitive representation of the fact that the guarded events are in same time offered to the environment and considered to be part of the operator.

The *Guarded alternative* can be in one of several working modes (prioritized, FIFO, FAIR, *preference alting*). The chosen mode is represented with an abbreviation (PRI, FAIR, FIFO, PREF) inside the rectangular box defining the operator. When the specification of the mode is omitted, the default mode (FIFO) is assumed.

At right-hand side of Figure 3-11, a counter is specified using an array of processes of type `COUNT` indexed with parameter `i`. The parameter `i` is used as the counter value. After accepting the 'inc' event, the `COUNT(i)` process will behave as `COUNT(i+1)`. After accepting the 'dec' event, it behaves as the `COUNT(i-1)` process. The event 'dec' `is` guarded with the logical condition set to allow the counter value to be decremented only if the value of `i` is greater than zero.

Start and Exit Control Flow Elements

In CSP, operators like *sequential*, *paralle*l, *external choice* and *internal choice* are used to combine two or more processes into a new process. In SystemCSP, control-flow elements are introduced to represent those operators.

The operators and theirs operands are in CSP grouped via parentheses. Every process composed via one of the CSP operators has an implicit START event and either a termination (EXIT) event or a *process recursion label* leading to the entry point of some other process. In a s*equential* combination of processes, the EXIT of one process is triggering the START of the next one in line. The START event of a *sequential* composition corresponds to the START event of the first element in *sequence* and the EXIT event of the composition corresponds to the EXIT event of the last subprocess in *sequence*. In case of *parallel* and *choice* operators, a START event is a point of forking control flow to branches and an EXIT event is a point of joining control flows of branches. In a *parallel* combination, all involved processes synchronize on both START and EXIT events. In case of a *choice*, only one branch is executed. Instead of using explicit grouping symbols (like parenthesis bubbles or boxes in GML), SystemCSP chooses to merge the START and EXIT events of the composition with CSP operators, creating in that way an extended set of control-flow elements as illustrated in Figure 3-12.

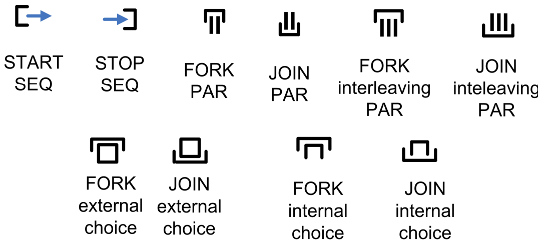


Figure 3-12 START and EXIT grouping symbols

Thus, the pair of open and close parentheses, bounding the scope of a CSP operator, can actually be mapped to the pair of control flow elements representing the synchronization on START and EXIT events (see Figure 3-12).

A special kind of a START grouping symbol is the FORK symbol that branches control flow on two or more branches. A special kind of EXIT grouping symbol is the JOIN symbol, where control flow branches are joined. Often, but not always a FORK element is paired with an appropriate JOIN element. In Figure 3-13, the words "FORK symbol" and "JOIN symbol" are used to cover all possible FORK and JOIN symbols given in Figure 3-12.

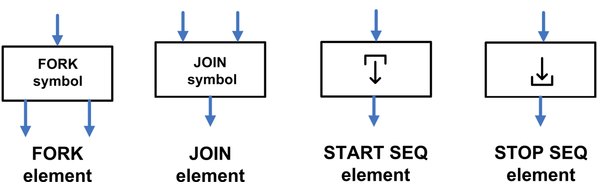


Figure 3-13 START and EXIT control flow elements

FORK and JOIN symbols can alternatively be specified in the style depicted in Figure 3-14. The look based on rectangles is more convenient when additional details need to be specified (e.g. synchronizing alphabets related to a *Parallel* operator). The look based on lines is more space efficient and resembles more FORK/JOIN elements of a *UML activity-diagram*.

START SEQ and STOP SEQ elements are in Figure 3-13 represented via rectangle boxes with the appropriate symbol inside. However, to abbreviate and compress the size of diagrams, it is allowed to omit the rectangle and to associate one or more START SEQ and/or STOP SEQ symbols directly with a *prefix* control flow element as in Figure 3-14.

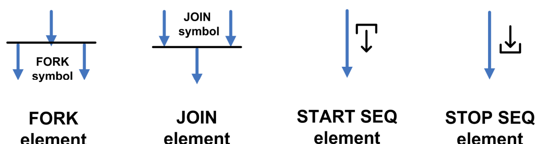


Figure 3-14 STARTand EXIT control flow elements - abbreviated forms

Comparing the CSP expression and its SystemCSP representation in Figure 3-15, one can see that all brackets used to group CSP operators with operands are present in symbols associated with control flow elements

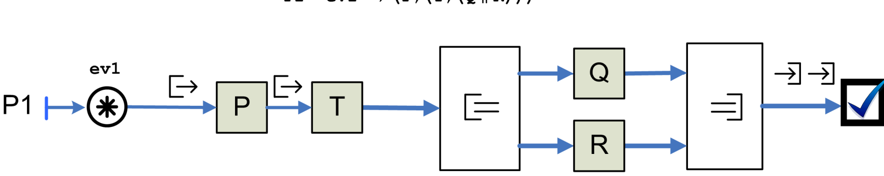


Figure 3-15 Grouping in SystemCSP

Process P1 will perform event 'ev1' and then behave as a sequence of process P and a process constructed as a sequence of process T and the parallel composition of processes Q and R.

The same example of a non-negative counter from Figure 3-11 is implemented in Figure 3-16 using a FORK choice element instead of a *guarded alternative* element. The only difference compared to the *guarded alternative* based design, is that events offered to the environment ('inc' and 'dec' events) are not considered a part of the control flow element but instead are considered part of branches to which the FORK choice control flow elements lead. The *guarded alternative* of CSP is thus in fact a special kind of the *external choice* operator and can always be replaced with a FORK *external choice*. The opposite is not the case. However, a *guarded alternative* is especially convenient for specifying *finite-state-machine* like designs.

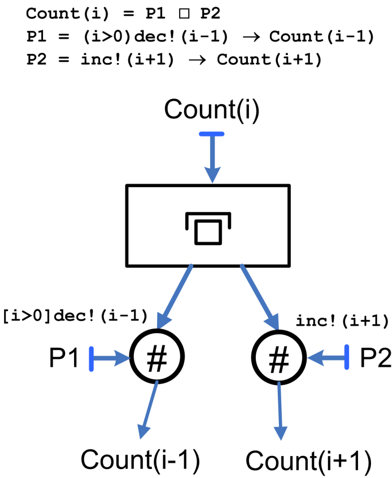


Figure 3-16 Example of a counter based on FORK choice

The *internal choice* (symbol given in Figure 3-12) is different from the external choice in the sense that it will internally make a choice and offer only the chosen branch to the environment. The internal choice is not so useful for implementation, but it represents a powerful abstraction mechanism; It is used when one needs to specify that some process will in some, unknown way choose one of several branches.

Specifying synchronization alphabet

The *synchronization alphabet* is a property of every parallel operator in CSP. It defines the set of events on which its subprocesses synchronize. In SystemCSP, the *synchronization alphabet* is a property of the FORK PAR operator. The *Synchronization alphabet* can be visualized inside the rectangle box representing the FORK PAR element. It is depicted as a list of events specified inside curly brackets.

It makes sense to visualize the synchronization alphabet in cases where it is different than the set of all events common for participating processes and in that way significantly affects the expected execution semantics, as in the example given in Figure 3-17.

```
Race   = Runners ‖ RaceCtrl
Runners= Runner1 {raceStart} ‖ {raceStart} Runner2
Runner1 = raceStart → 100m?time11 → 200m?time12 → finish!time13 → SKIP
Runner2 = raceStart → 100m?time21 → 200m?time22 → finish!time23 → SKIP
RaceCtrl = raceStart → Milestones
Milestones = (100m!time → SKIP)
                  □ (200m!time → SKIP)
                      □ (finish!(time, count++) → SKIP)
                        →(SKIP ≮ count == participantsNr ≯ Milestones)
```

Figure 3-17 Specifying synchronization alphabets

Figure 3-17 represents a complex process named `Race`. `Race` is a *parallel* combination of two processes: `RaceCtrl` managing the race and `Runners` being a *parallel* composition of two runners participating in the race. Both runners are

described via the same process description specifying that they will engage in the 'raceStart' event, and in the events '100m', '200m' and 'finish'. The FORK PAR marked with *process entry point* named `Runners` specifies that its subprocesses - two runners - actually synchronize among themselves only on the 'raceStart' event.

The 'raceStart' event is initiated by the race control mechanism (`RaceCtrl` process). The race control mechanism will on request of the runners deliver them the current time on the milestone events '100m', '200m' and 'finish'. After both runners have separately engaged in the event 'finish' with the `RaceCtrl` process, the `count` variable becomes equal to the number of runners and the `Race` process is finished.

Exception Handling

Handling exceptional situations that occur during the execution of a process is an important issue for well-designed programs. In the software development world, the implicit agreement exists that in some way, the part of code or a visual design, which specifies/implements handling exceptional situations, should be visually isolated as far as possible from the design/code specifying normal execution. In general, exceptional situations in some process can be handled by attempting recovery within that process (*recovery model*) or by aborting the process and executing some other process instead (*termination model*).

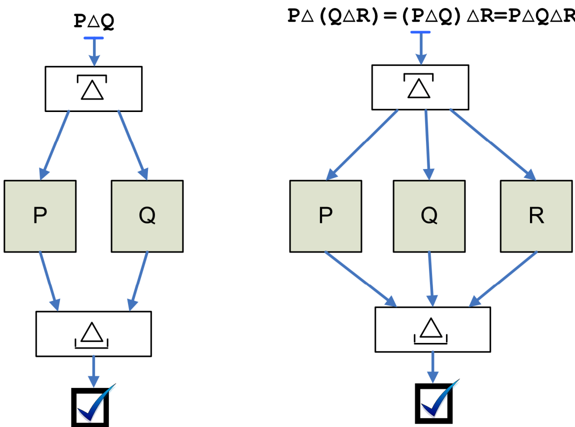


Figure 3-18 Take-over operator

CSP defines the *interrupt operator* that covers the termination model. Process P Δ_i Q is a process that behaves as process P until either P terminates successfully or until an event occurs that activates process Q. In the latter case, further execution of process P is aborted and process Q is executed instead. Despite its name, the semantics of the Δ_i operator is much closer to the termination model of exception handling than to *interrupt handling*, since it implies aborting the left hand-side operand. To avoid terminology confusion, the equivalent of *interrupt* operator is in SystemCSP named *take-over* operator. The *take-over* operator is

specified with a pair of FORK and JOIN symbols (see Figure 3-18) branching control flow on normal (most left) and on one or more exceptional modes.

In Figure 3-18, in the first case the process Q can take over process P. The second example in Figure 3-18 illustrates the fact that the take-over interrupt is associative by definition. Process Q can take over process P. Process R can take over both P and Q.

3.2.2 Supervision elements

Logging is the activity of collecting data about the changes in values of a certain chosen set of variables during some time interval. In tracing, the information communicated to the human operator is the current position in execution flow of the application.

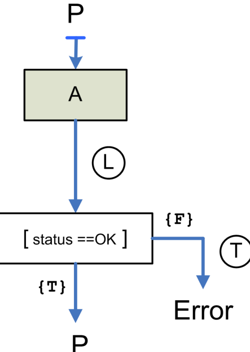


Figure 3-19 Supervision layer

In SystemCSP, logging and tracing points are predetermined in the design. In a control-flow diagram of SystemCSP, the symbol used for a logging/tracing point (a circle with big L for logging, or T for tracing, inside) is associated with a prefix arrow as its property. The reason for this is a choice to consider a set of logging points to be an optionally visible layer added on top of the design. In the implementation, however, prefix arrows do not exist, while logging points are inserted into the appropriate location in the execution flow, as defined by the position of the related prefix arrow in the design.

The reason to opt for this kind of logging is predictability. The logging activity is considered to be part of the design and all the needed resources (e.g. CPU time, memory, network bandwidth and storage capacity) can be preallocated. Placing logging points on predefined places in control flow is, compared to logging every change of some variable, considered to be more structured and more predictable approach, especially in the sense of processing time requirements. Places to insert logging points can be chosen in a way that allows reconstruction of the relevant changes of value of a variable.

3.2.3 FSM-like diagrams in SystemCSP

If only events, prefix and guarded alternative elements are used, any CSP process can be visualized both via a SystemCSP diagram and an FSM and easily converted from one visualization into the other. In the example below, a one-to-one mapping between the SystemCSP diagram in Figure 3-20 and the FSM diagram in Figure 2-7 is obvious. The states of the FSM map to *waiting done on events* and *guarded alternative* elements in the Figure 3-20. This is illustrated by enumerating states in Figure 2-7 and the corresponding EventSync and guarded alternative elements in Figure 3-20 with numbers 0 to 7.

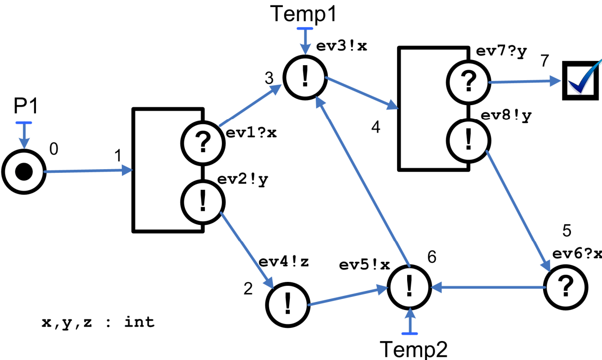


Figure 3-20 SystemCSP diagram

Note that the topology of SystemCSP diagram in Figure 3-20 visually resembles the one of the FSM diagram of Figure 2-7. The main paradigm shift is emphasizing events instead of states. Events are not shown as transitions (as in the FSM), but as *EventSync* processes. This allows for direct line connections to peers in the environment or to ports of the component. Mapping from SystemCSP to CSP is more direct then it is the case for an FSM, where such a mapping requires distinguishing between states with *exactly one* outgoing transition and states where more then one outgoing transition is present.

SystemCSP makes a difference between *internal* and *external choice*, while FSMs imply always the *external choice*.

3.2.4 Hiding as a filter of possible interactions

The example in Figure 3-21 illustrates how the *hiding* operator is used to hide 'install'/'uninstall' events from the user that has restricted access rights.

When the process `Program` is at its entry point, it is ready to be installed (event 'install'). However, the users who can access the program only via the `Restricted_program_use` *process entry point* cannot engage in that interaction with the program.

The program is installed (event ‘install’ accepted) by some process from the environment that can see the process under name `Program` and thus can initiate ‘install’ event. After installation, program is in `Start_Menu` entry point in its control flow, where any user can open the program (‘openProg’ event). A user that has no access restrictions can at this point also decide to uninstall the program (‘uninstall’ event). If the program is opened (after ‘openProg’ event), then it can be used (`UseProg` entry point in control flow of the program). Using the program initially offers two options: closing the program (‘closeProg’ event) or opening some document (‘openDoc’ event). Upon opening a document, one can work with it (`Work` entry point in control flow of the program). Working with the document includes making choices between several actions: updating the document (‘updateDoc’ event) saving the document (‘saveDoc’ event), closing the document (‘closeDoc’ event) or closing the entire program (‘closeProg’ event). Note that in Figure 3-21, the assumption is that the process selected to be displayed in the figure is `Restricted_program_use` and that because of that, the hidden events ‘install’ and ‘uninstall’ are shaded.

```
Restricted_program_use = Program \ {install, uninstall}
Program = install → StartMenu
StartMenu = openProg  → UseProg | uninstall  → Program
UseProg = closeProg  → StartMenu | openDoc → Work
Work = updateDoc → Work | saveDoc  → Work
          | closeDoc → UseProg | closeProg → StartMenu
```

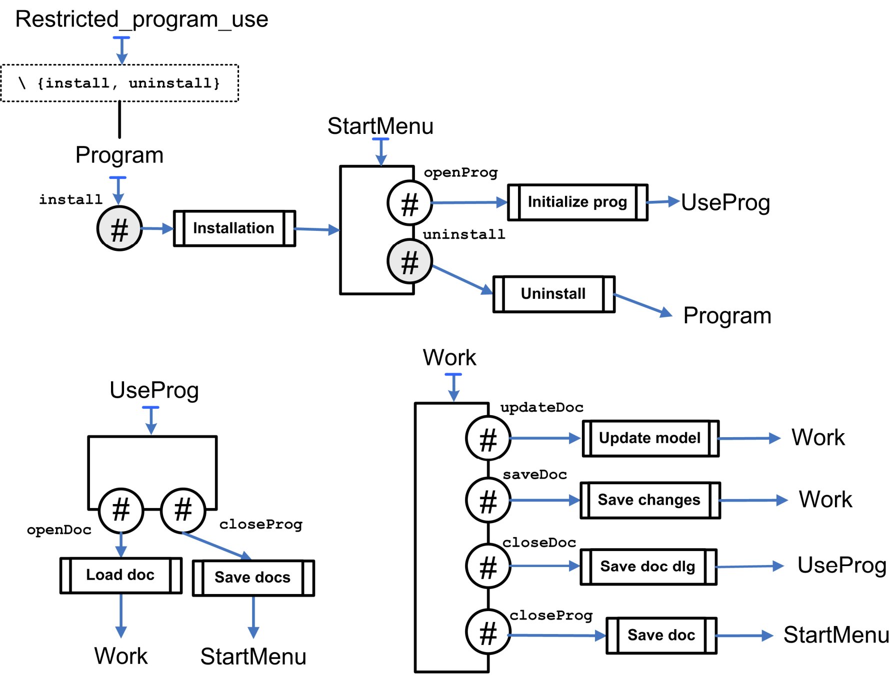


Figure 3-21 Hiding as a way to filter possible interactions

In this example, one can also notice the difference between a process as a structural unit and a process as a behavioral unit. `Restricted_program_use` and `Program` are two different CSP processes in behavioral sense, but both behaviors are provided by the same structural instance. Thus, the processes `Restricted_program_use` and `Program` need to synchronize on their common set of events and each one of them needs to synchronize with own set of users.

3.2.5 SystemCSP sequence diagrams

Code blocks are allowed to specify arbitrary complex sequential OOP designs. In principle, they can be designed using *UML sequence diagrams* or some other type of diagrams. The aim is, however, to be able to create in SystemCSP a complete specification sufficient for an efficient coding process, code generation and reverse engineering. For specifying sequential code, special diagrams (see Figure 3-22) inspired by *UML sequence diagrams* are proposed.

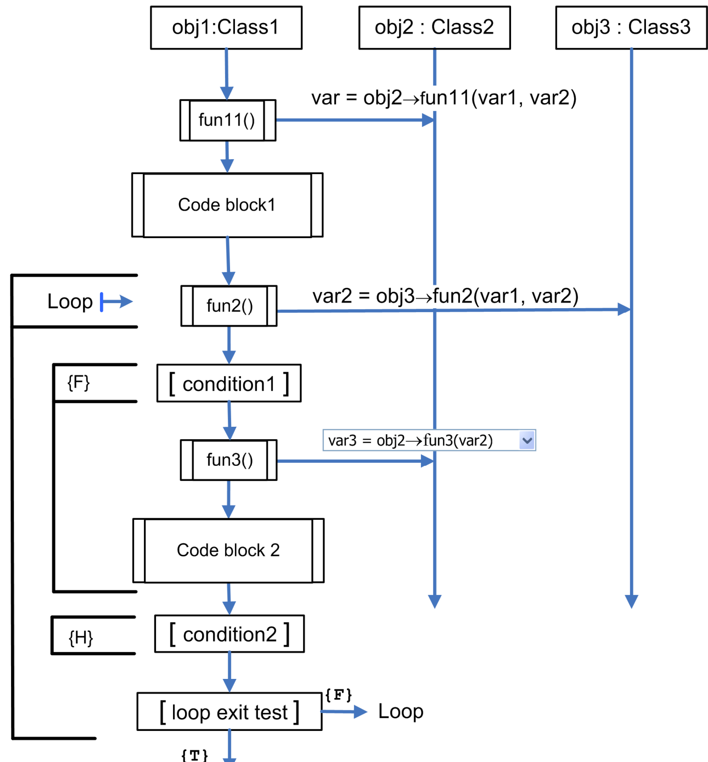


Figure 3-22 Design of sequential code in SystemCSP

The basic structure resembles the one of *UML sequence diagrams*: every object participating in the interaction has a *lifeline* that reflects the flow of time, function calls are visualized using directed lines connecting the *lifeline* of the object making

the function call with the *lifeline* of the one providing function (see Figure 3-22). SystemCSP elements include *code block* elements, *conditional choice*, *entry labels* and *recursion labels* (see Figure 3-22). In addition to elements from SystemCSP and *UML sequence diagrams*, the sequence diagrams in SystemCSP introduce notation elements for specifying the scope of the control blocks (i.e. for visualizing conditional branching and loops). The borders of the control-flow blocks are visualized by means of large square brackets located on the most left side of the diagram (see Figure 3-22).

A problem with *UML sequence diagrams* was that due to the single time axis, they can depict only one scenario. UML2 attempts to solve this by introducing *interaction operators* as described in Chapter 2. Those operators allow one to specify multiple scenarios in same diagram by e.g. displaying options of conditional choice in separate compartments of a box introduced to represent conditional choice. However, this does in fact reduce the readability of the diagram. The idea in SystemCSP is that at any moment, for every condition either the `true` branch or the `false` branch is depicted. A prospective tool, providing support for SystemCSP sequential diagrams, should allow displaying control flow in a way that will allow hiding irrelevant parts and displaying only the scenario for selected values of switch conditions. The best solution is that toggling true/false values for the condition fields, results in the diagram updated to display the chosen branch. The model of the diagram captures the complete control flow of the function and the actual diagram visible on the screen depends on the current user selections for offered conditional switches. To allow hiding irrelevant parts, in addition to true and false branches, it should also be possible to hide both. (in Figure 3-22 for condition 1, false or {F} switch is selected, and for condition 2 the hidden or {H} switch.

Every function that is specified via a SystemCSP sequence diagram is expected to have defined properties like: description of service that it provides, input/output parameters, preconditions, postconditions, and exceptions it may raise.

Another potential problem with *UML sequence diagrams* is that they chose to display nested function calls, but not code blocks. In that way, a sequence diagram does not provide complete specification sufficient for code generation. Displaying nested function calls is not relevant for the current abstraction level, especially because the used function calls are in good object-oriented programming assumed to provide certain services regardless of their internals. Displaying nested function calls breaks this kind of encapsulation. In SystemCSP, every sequential design diagram focuses on modeling the internals of exactly *one* function (either a global function or a member function of some object) at the time, and thus only the function calls made directly from the internals of the visualized function are specified. It is assumed that used objects provide well-defined services, and peeking into their internal implementation from the internals of currently designed function is considered to be bad design practice. Finding out the implementation of those functions would be (with a support of prospective tool) anyway just a mouse click away, in the *SystemCSP sequence diagram* defining the used functions.

A side benefit of the decision not to display nested function calls is that the source code model needed for construction of the diagram can easily be reconstructed from code (*reverse engineering*), since the contents of only *one* function needs to be parsed. When editing of the function is finished, code is generated and no additional data about the visualized function needs to be preserved.

Introduction of a special kind of diagrams for representing sequential designs allows visualization of the coding process, which results in a completely visual design process. Entering the design by switching continuously between mouse and keyboard slows down the design process, especially when one applies it at the lowest design level where most of source code is. For efficient coding, it is expected that the prospective tool allows users to enter a complete diagram using the keyboard only (e.g. arrows instead of mouse movements as a way to select existing diagram elements or to browse for the type of element to insert). The presented form of the diagram (e.g. as in Figure 3-22), with predefined placement areas for visual elements, provides a form that enables design entering via keyboard only.

3.3 Interaction oriented elements

3.3.1 Binary compositional relationships

The interaction-oriented part of the language is inspired by the binary compositional relationships of GML.

As explained in Chapter 2, GML binary compositional relationships have a weaker meaning than the related occam constructs or CSP operators. For instance, specifying a *parallel* relationship between two processes does not mean that they are synchronizing on the *start* and *termination event*, as it would be the case if they were composed via a PAR construct of occam. Such a *weak* relationship gets a stronger meaning only after the processes it connects are grouped together (e.g. via the *bubble notation*). A *weak* binary *parallel* relationship in isolation (without specified grouping) does imply only that the two related processes will have a common *parallel* construct somewhere in the hierarchy of their parent constructs.

In the section 3.2.1, grouping symbols of type START and EXIT (including FORK and JOIN types) were introduced (see Figure 3-12). The START SEQ control flow element defines the starting point of a *sequential* construct. In the interaction-oriented diagrams, we introduce binary SEQ relationships that relate two processes executed in a sequence as part of the same *sequential* construct. In an interaction-oriented diagram, START SEQ implies that, in addition to the semantics of the SEQ binary relationship, one of the two process blocks, as indicated by the direction of the START SEQ symbol, is the first process in the enclosing *sequential* construct. In analogue way, the STOP SEQ binary relationship defines that the related pair of process blocks are part of the same sequential construct and that one of them as indicated by the direction of the associated STOP SEQ symbol, is the last process in the enclosing *sequential* construct. Thus, since START SEQ

and STOP SEQ binary relationships add to the semantic of the binary SEQ relationships, we choose to call the binary SEQ relationship WEAK SEQ.

In the section 3.2.1, the FORK PAR control flow element is introduced to mark forking of the control flow on two or more processes that will execute concurrently. In the interaction-oriented diagrams, we introduce the binary PAR relationships that relate any two of the process blocks forked by the same parallel construct. FORK PAR as a binary relationship indicates that, in addition to being executed concurrently (weak version of parallel relationship as used in GML), the related subprocesses of parallel construct do synchronize on ‘start’ event. The JOIN PAR binary relationship indicates that the two related process blocks, in addition to being executed concurrently (weak version of parallel relationship), do synchronize on the termination event. Specifying both FORK PAR and JOIN PAR binary relationships indicates that, besides being executed concurrently (weak version of parallel relationship) the related process blocks do synchronize on both start and termination events, which is equivalent to grouping them via a CSP parallel operator or occam PAR construct.

When two process blocks are related both with a FORK and a JOIN of the same kind of a binary relationship, we introduce one symbol instead of two and call such binary relationship a STRONG relationship. The symbol for a STRONG relationship (see Figure 3-23), in addition to the symbol of the operator, contains both FORK and JOIN symbols.

Besides FORK, JOIN and STRONG, we also explicitly introduce WEAK binary compositional relationships. A WEAK PAR exists between those process blocks in parallel branches that do not synchronize on START or EXIT events. Process blocks related via a WEAK PAR binary relationship can however synchronize on any number of user-defined events. The WEAK interleaving PAR specifies that the related process blocks are executed concurrently and that there is no synchronization *at all* between them.

The sequential binary relationship in addition to its STRONG and WEAK form also has a PRECEDENCE form. It specifies that the involved process blocks are executed one after another, but not necessarily immediately after each other. Thus, the PRECEDENCE relationship is weaker then WEAK SEQ. In fact, the PRECEDENCE binary relationship matches the SEQ binary relationship of GML.

Figure 3-23 depicts symbols for WEAK and STRONG binary relationships.

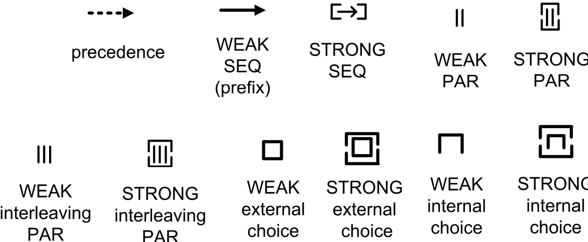


Figure 3-23 STRONG and WEAK relationships

Figure 3-24 illustrates an example of an interaction diagram and related CSP expression. In interaction-oriented diagrams, a process block defines the preference indexes for its binary relationships (see numbers at the ends of the relationships). Lower the number, the related construct is closer in hierarchy of constructs. Number 1 indicates an immediate parent construct (thus, in Figure 3-24, the immediate parent construct of P, Q and R is sequential, of M it is parallel and of N it is external choice). Note that in this ordering, the same number implies the same construct. Thus, a process block can use the same preference index only for the binary relationships of matching types (e.g. in Figure 3-24 for process block Q, two SEQ relationships that have the same preference index can be resolved via the same sequential construct).

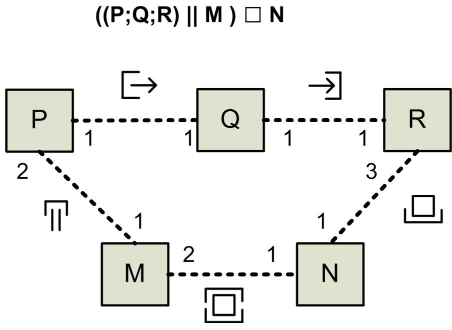


Figure 3-24 An interaction-oriented diagram

A binary relationship is uniquely defined for any pair of process blocks belonging to the same control-flow diagram. From a complete CSP expression or control flow diagram, it is possible to deduce a binary relationships and associated preference indexes for any pair of contained process blocks. Simply, the type of the relationship is the one of the nearest parent CSP operator that contains both of processes located somewhere inside its operands. The STRONG relationship relates two process blocks if they are both located directly after the same START control flow element and directly before the same EXIT control flow element. WEAK relationship indicates that the related process blocks have a common parent construct somewhere in the hierarchy.

Compared to GML, introduction of START and EXIT, WEAK and STRONG variants allows incomplete specifications for the borders of constructs during the design process. Instead of explicit grouping (via box or bubble notation), and in addition to GML-like binary relationships, reasoning about synchronization on starting and termination can be specified. The preference index is in essence the same as the index of the bubble in GML increased for 1. However, in SystemCSP idea is to allow the designer to just order and reorder the list of binary compositional relationships of a process block. Actually, the index number is in fact preference of the process block about its distance (in the hierarchy of parent constructs) from the construct that resolves the binary relationship. This insight does allow one to forget GML idea of explicit grouping via bubbles as parenthesis and to partition interaction-oriented diagrams in any number of smaller interaction-

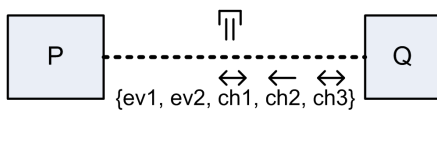
oriented diagrams with the same blocks allowed to participate in many interaction diagrams.

Interaction-oriented diagrams allow one to focus attention on several processes in the isolation from the rest of the system. Binary relationships specified in interaction diagrams define restrictions that control flow diagram is expected to fulfill. Such a restriction can be satisfied with using the same or higher strength relationship in the control flow diagram. If higher strength relationship is used, that should be reflected back by automatic update of the relationship between process blocks in all related interaction diagrams. The relationships between roles inside interaction contract stay the same regardless of the context of using the contract.

Visualizing binary relationships and preference indexes is optional in interaction-oriented diagrams. Interaction oriented diagrams are convenient for putting focus on interactions, structure and data flows. Thus to make them useful, there should be a way to visualize events on which related processes synchronize, and to display the related communication data flows.

3.3.2 Synchronization events and data flow

In Figure 3-25, processes P and Q are related via FORK PAR binary relationship, meaning that they are executed concurrently and that their execution starts at the same time. In addition, in Figure 3-25, below the interaction line, a set of user-defined events on which they do synchronize is specified inside curly braces. Note that some events are channels. The data flow direction of channel is indicated via directed lines immediately above the name of the channel (e.g. in Figure 3-25 channel ‘ch1’ transfers data in both directions, while ‘ch2’ transfers data from Q to P). In addition, it is allowed to associate numbers with events, in order to specify ordering of events and in that way indicate the scenario of interest.



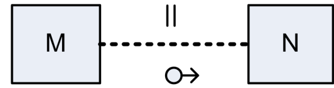


Figure 3-25 Binary compositional relationships and data flow

Sometimes, however, it is useful to visualize only data flow without listing all synchronization events and channels. In lower part of the Figure 3-25, the used symbol indicates the existence of a data flow directed from process block M to process block N.

3.3.3 Refinement operator

The refinement operator defines that one of the related processes is a specification that the other one needs to refine. The symbol used to adorn binary relationship is the same as in CSP (see Figure 3-26). Refinement verification is a very important vehicle in formal checking. For instance, in a stepwise development process, the implementations of systems are built through set of incremental iterations with checking conformance to specification after every cycle.

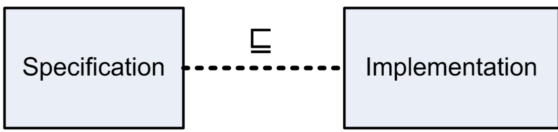


Figure 3-26 Refinement relationship

3.4 Interacting components

SystemCSP is a graphical modeling language based on both CSP and concepts of component-based software development. SystemCSP provides a way to visualize architecture, behavioral patterns of components, intra–component interactions and execution relations among components.

The notation makes a distinction between components, interaction contracts and processes. CSP is focused on the interaction between processes. A process is viewed as a behavior described via some named pattern of event synchronizations. In SystemCSP, the term *component* is used as in modern notions of software development: “A *component* is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition” (Szyperski, 1998). From a CSP point of view, the behavior of a component is captured as a complex process, described with the help of one or more auxiliary processes. An *interaction contract* is a process or a component that has the responsibility to manage the interaction of some other components.

3.4.1 Structural units

Components

Component is alike to *process block* element defined in section 3.1.2. The main difference is that a component is larger structural unit of composition that can be deployed independently. A *component* can contain smaller structural units, like variables, code blocks, process blocks, processes, subcomponents. It is a reconfigurable and reusable unit, which can be used in different contexts and different interaction scenarios. In CSP sense, the behavior of a component is described as a process. As it was the case for *process blocks*, a distinction is made

between component types and component instances. Everything said in section 3.3 for process blocks applies for components as well.

In our notation, components are enclosed in a rounded rectangle (see Figure 3-27) specifying the boundaries of the component. As with process blocks, *opaque-box* and *transparent-box* notations are possible. The entry point of a component is always marked with a `start` element. Variables defined in the component scope can optionally be represented via labels floating somewhere inside the borders of the encompassing component. A *variable label* contains the variable declaration consisting of its name and its type separated by a colon.

Ports and interfaces

An *event port* is an access point that exports an event end to the environment of the component. In Figure 3-27, *event ports* are depicted as little rounded rectangles attached to the outer side of the component border. The interaction line connecting an event-end with its associated port is visualized optionally (e.g. in Figure 3-27 it is visualized for 'ev1', but not for event 'ev2').

A *port label* carrying the externally visible name of the event associated with the port, can be inside the port symbol or next to it. The name used in a port label can be different from the one of the related event-end (e.g. in Figure 3-27 event–end 'event3' from component C1 is exported to environment via port named 'ev3). In mapping to CSP, this would be implemented with the renaming operator.

A *channel port* (see 'ch1' port in the Figure 3-27) is an *event port* with *port label* containing one or more signs of type "?" or "!" to indicate the direction(s) of associated data flow(s). *Channel ports* and *event ports* are on both participating sides depicted on the outer side of the involved components.

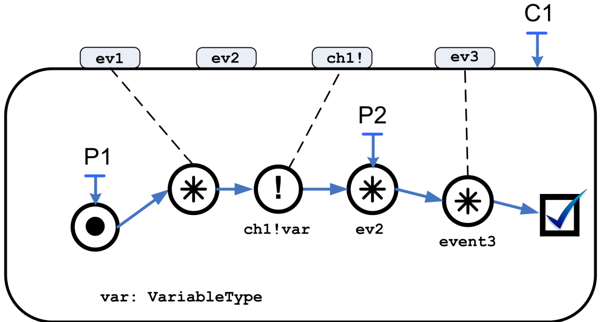


Figure 3-27 Component, ports and interfaces

In Figure 3-27, the behavior of the component C1 is represented via process P1. In case of event 'ev1', the relation between the EventSync processes and the associated port is specified via a dashed line. In case of event 'ev2' the same is achieved via using the same name. Both representations are allowed.

An *interface port* (e.g. see 'user interface' and 'printing' ports in Figure 3-28) is a higher-level access point to a component, that internally contains *event port(s)*

and/or *channel port(s)*. In addition, it can specify interaction pattern of the associated role implementation or specification.

Interface ports appear in pairs consisting of the *required* and the *provided* interface. The *provided interface* is related to the implemention (provided by the component) of a role in an interaction. The *required interface* specifies the role in the interaction required by the component. In SystemCSP, both implementation and specification of a role in some interaction are in fact some CSP processes, and refinement must hold between specification and implementation.

The printer component on Figure 3-28 is a server providing the implementation of the service needed by its users, but in the same time it is a client of some lower level printer device controller that takes care of the actual implementation of those services.

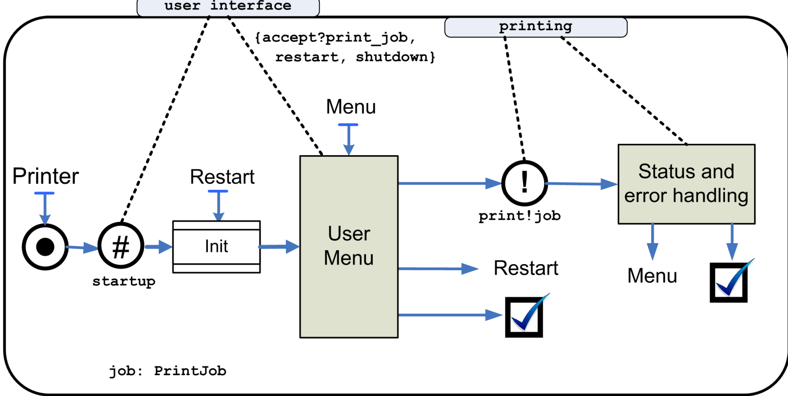


Figure 3-28 Interface ports

Interface ports are depicted in same shape as event ports but larger in size. Ports representing a *provided interface* are located at the outside of the components (see port ‘user interface’ in Figure 3-28). The ports representing a *required interface* are depicted as plug-in sockets (see port ‘printing’ in Figure 3-28).

Interaction Contracts

Specification and implementation of interaction among participating components is formalized via the notion of an *interaction contract*.

An *Interaction contract* is a special purpose component, dedicated to managing an interaction of some components, in a structured and formally verifiable way. The existence of *interaction contracts* allows reasoning about the interaction in isolation, without a need for actual components.

An *interaction contract* specifies the *roles* of the participating components. Component participating in interaction contracts must provide an *implementation* that is a *refinement* (in the CSP sense) of the role *specification* given in the interaction contract.

Contracts usually contain an internal component dedicated to managing the contract. For instance, let us imagine that the process `Race` from Figure 3-17 is specifying an interaction contract. In that case, the processes `Runner1` and `Runner2` could be considered to be roles implemented by some external components and all the rest would actually be an internal component managing the `Race` interaction contract.

Some of the simple and widely used interaction contracts are: shared memory, buffered channel, Any2One, One2Any, Any2Any channels. Interaction contracts can be made for any kind of application-specific scenario and can be reused in the same way as components.

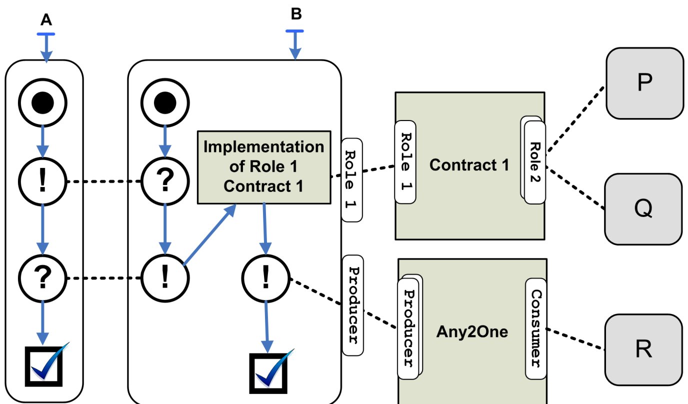


Figure 3-29 Interaction - direct and via contract

Figure 3-29 displays interaction of several components cooperating directly or via simple interaction contracts. Component A and component B interact directly and for simplicity, even the port elements are omitted in the diagram. Component B participates in interaction with components P and Q in a way specified in the interaction contract named Contract 1. It implements Role 1 of that contract. Component B also participates in an Any2One contract as one of several possible Producers. In the same contract, component R plays the role of the Consumer. The ports displayed in Figure 3-29 are associated with provided and required roles.

Both Contract1 and the Any2One contract in Figure 3-29 are visualized using the *opaque-box* approach. The internals of an interaction contract can also be visualized in *transparent-box* approach. In the *transparent-box* approach, roles are depicted in separate areas and associated with appropriate ports.

Contexts

A component can contain subcomponents and contexts (see Figure 3-30). A *Context* is a structure used by the component management system to organize logically related information about available *interaction contracts* and *provided* and *required interfaces* of components. As such, it provides a mechanism that is particularly useful for dynamical reconfiguration, allowing a component to discover its environment through querying the contexts of the host component.

Contexts are a kind of meeting point for components and *interaction contracts*. Upon entering a *context*, a component needs to register with the *context*. During the registration it is expected to provide the list of the interfaces that it is ready to publicly share and make available to other components and interaction contracts present in the same context. A component can query an opened *context* and find out about other *interaction contracts*, components, *provided* and *required interfaces* present in the *context*. A *context* can also contain links/passages to other related *contexts* in the same or in some other component. For instance, a *context* that does not provide some *interaction contract* or interface might have a link to a related *context* that does.

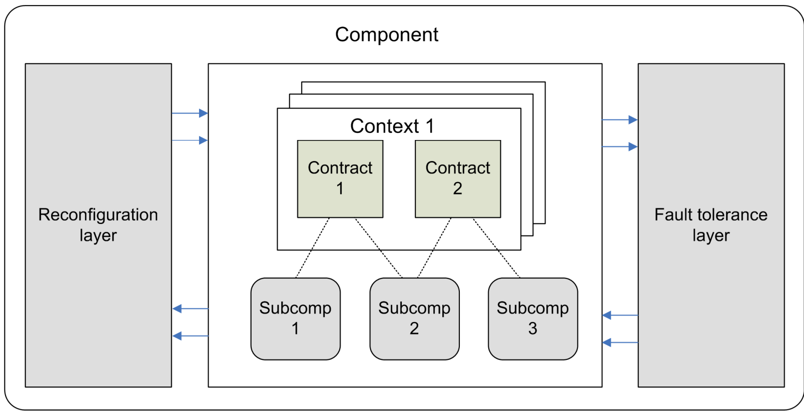


Figure 3-30 Contexts and contracts

Interaction contracts are abstract definitions, whose instances can be located inside some *contexts* (see Figure 3-30). *Contexts* provide concrete environments in which *interaction contracts* can appear. Parameters of the *interaction contract* might need to be initialized with data from the *context* or mapped to objects existing in the *context*. For instance, a football game can be considered an *interaction contract* with certain rules. The abstractions used in the description of rules (e.g. football terrain elements) must be mapped onto existing objects in some real world *context* (i.e. real world objects that are used to play the role of the football terrain elements).

In SystemCSP, besides a process describing the normal execution mode, components optionally contain a process managing possible reconfiguration scenarios and a process specifying the recovery activities upon occurrence of exceptional situations (see processes named ‘Reconfiguration Layer’ and ‘Fault tolerance layer’ in Figure 3-30).

3.4.2 Interaction and control-flow diagrams

The experience with using GML, showed that specifying binary relationships among components is very useful in early stages of the design, but is somewhat cluttering readability in later phases when focus is on control flow.

This project goes a step further with the philosophy behind using binary compositional relationships. Insight is obtained that a representation based on binary relationships is in fact a good basis for fragmentation of the design into a set of diagrams. The same entity can appear in any number of such partial diagrams. This does introduce early design freedom in a sense that a design can be specified iteratively with focusing on particular interactions among several components, specified in isolation from the rest of the system.

The description of every component contains one *control-flow diagram* and one or more *interaction diagrams*. A *control-flow diagram* focuses on control flow elements that determine possible execution orderings of components. An *interaction diagram* is a diagram that specifies the way in which components interact. It usually contains a set of components (in *opaque-box* or in *transparent-box* notation) centered around an *interaction contract*.

SystemCSP allows the same component to participate in any number of different *interaction diagrams*. This is in a way similar to UML diagrams where one can focus on certain aspects of some entity in one diagram and on other aspects in other diagrams. Unlike in UML notation, where there is no relation between different diagrams, in SystemCSP all *interaction diagrams* inside one component provide a single, consistent, formally verifiable, model of the component. This model is reflected in the *control-flow diagram* of the component.

A component specifies its binary compositional relationship with other components in different *interaction diagrams* it participates in. The set of all its binary compositional relationships from all *interaction diagrams* it participates in, determines its position in the *control-flow diagram* of the parent component.

At the left-hand side of Figure 3-31 two *interaction diagrams* are given, and on the right-hand side the associated control flow diagram is shown.

In Figure 3-31, components B and C appear in both interaction diagrams because they engage in both *interaction contracts*. Component B is in the upper *interaction diagram* depicted via the *opaque-box* approach, and in the lower one via the *transparent-box* approach. Contract 1 from the upper *interaction diagram* is depicted via the *transparent-box* approach and Contract 2 from the lower interaction diagram is depicted via the *opaque-box* approach.

Components A and B are, in upper *interaction diagram* from Figure 3-31, related via FORK PAR binary relationship. For component B, the preference of this relationship is set to 1 and for component A it is 2 (preference 1 has its STRONG SEQ relationship with C). That means that in the control flow diagram components A and B are in the same *parallel* construct, with component B as direct subprocess and component A as a part of its subprocess organized as a *sequential* construct in which component A is the first subprocess and component C is the second one. Component D is composed in STRONG CHOICE relationship with component B. For component B this is the relationship of preference index 2 (after the parallel relationship with components A and C) and for component D of preference index 1. This indicates that component D is composed in an *external choice* construct with parallel composition consisting of component B and a sequential composition

of components A and C. The other specified binary relationships are redundant and can serve as a check whether the specification is illegal or not.

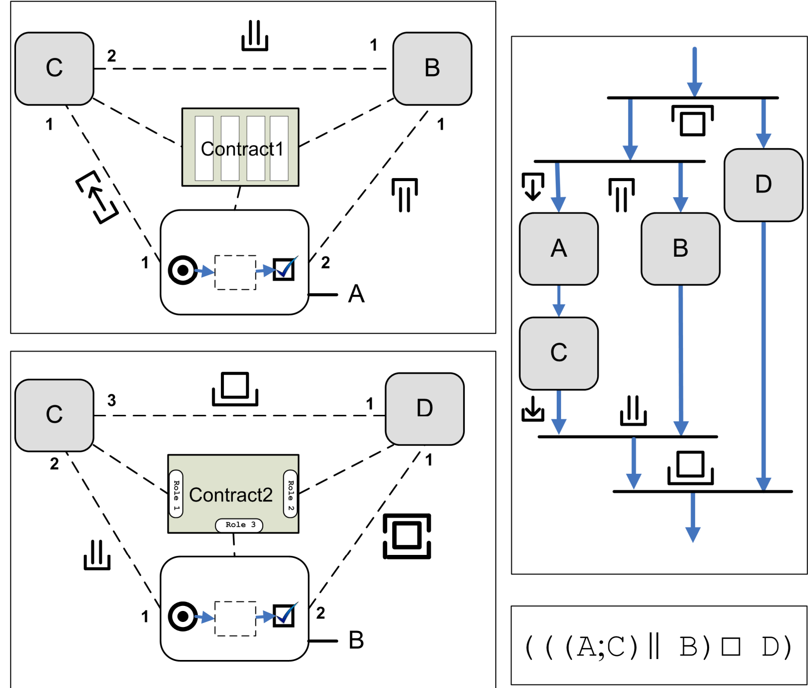


Figure 3-31 Specifying execution relationships in interaction views

By aligning elements of the control flow diagram in such a way that the control flow goes downwards with FORKs and JOINs as horizontal lines connected via prefix arrows to the involved components below and above, a form that resembles the *UML activity diagram* is created (see the control flow diagram in Figure 3-31).

In principle, the design starts by specifying interactions among few components in isolation. An interaction diagram captures only the binary relationships relevant for the given interaction. Often, at first the WEAK relationships (PRECEDENCE, WEAK SEQ, WEAK PAR, WEAK CHOICE) are specified, which are then either gradually refined to stronger variants (START, STOP and STRONG kinds of binary relationships) that specify implicit grouping or they stay weak if that is an intention.

Components participating in an interaction diagram do not exist in isolation; they are nested in some parent component that specifies a set of their possible execution orderings in the associated control-flow diagram (e.g. the right-hand side diagram of Figure 3-31). Thus, there is always some compositional relationship between participating components. Specifying binary compositional relationships is optional in interaction diagrams. Only the binary relationships relevant for the interaction diagram are visualized.

Benefit from introducing *interaction diagrams* is that they provide significant

design freedom, as necessary in early stages of the design, by allowing incomplete specifications scattered in many diagrams focused on particular aspects of the system.

The used approach enables incremental design of *control-flow oriented diagrams* by adding restrictions in different *interaction-oriented diagrams* throughout the process of system design. In this way, all different diagrams are combined into a single formally verifiable system.

3.4.3 Discussion

In the following text, an attempt is made to position the notion of *interaction contract* as used in SystemCSP, compared to the other approaches described in chapter 2.

The interaction contract of SystemCSP is matching the concept of the *connector* concept in WRIGHT. The name *interaction contract* was chosen because it is more general and better suits its purpose then the name connector. Indeed, most simple *interaction contracts* (event, Any2One channels, buffered channels, etc) can be classified as *connectors*. But the entity specifying an interaction among devices in an industrial production cell is more then just a *connector*. Unlike WRIGHT, SystemCSP can visualize *interaction contracts*, as well as other entities useful in the development of concurrent, component-based systems.

Compared to the formal contracts of Boosten (Boosten, 2003), *interaction contracts* are directly implemented as CSP processes and there is no need for transformation in order to achieve formal checking. In addition, *interaction contracts* are not considered a substitute to channels, but a higher-level primitive described via event (channel)-based interactions with participating components.

The *Coordinated atomic actions* (CA actions) safety pattern (Zorzo et al., 1999) illustrates the importance of a centralized entity maintaining the interaction between participating components and thus serves as a motivation for introducing interaction contracts as separate, explicitly existing entities. Compared to CA actions, interaction contracts are considered to be a more structured approach because they achieve the same purpose, but rely on a safer and more structured way to use concurrency. As in CA actions, one of the main powers of interaction contracts is the opportunity to nest handling of exceptional situations in contract facilities, where more knowledge is available about the current state of interaction than in participating components in isolation.

An *interaction contract* is an abstract entity whose main purpose is specifying and managing interactions between components. By defining interaction as an abstract entity (that can be instantiated in the same way as components can), a possibility for reuse of the design patterns captured in a form of interaction contracts is introduced. An interaction contract prescribes the roles of the participants and offers additional interaction management support. It can introduce additional constraints in the way component instances interact, provide buffering support and exception handling facilities. A contract normally consist of three phases: checking

preconditions, performing the action and checking postconditions. An action can contain an interaction pattern specified via events or via subcontracts.

In the light of the four level of contracts classified in (Beugnard et al., 1999), the interaction contract of SystemCSP provides a natural support for the first three levels and the possibility to build an application specific Quality of Service layer on top of the first three layers. On the basic contract level, an interaction contract is described via event/channel interconnections, operations/actions they represent with associated input/output parameters and a defined set of possible exceptions that can propagate via the event/channel infrastructure. The second level is achieved by dividing every contract into three parts: optional checking of preconditions, the mandatory action and optional checking of postconditions. The third level is naturally supported by the CSP structure of the contract including the participating roles and the optional contract manager. In addition to the used channel/event ports, the CSP description of the contract encapsulates all possible scenarios for contract execution. Level 4 contracts can be built as an additional layer in the application specific contracts. General design patterns can be made to construct reusable QoS contract layers.

An interaction contract specifies roles for which a component willing to participate must provide an implementation. The implementation of a role must be a refinement in the CSP sense (traces, failures, failures/divergencies levels of refinement) of the role description required in the contract. In CSP, the implementation is considered to be the trace refinement of the specification when a set of event traces that can be produced via execution of the implementation process is a subset of the set of traces that can be produced by the execution of the specification process. In other words, a behavior of an implementation must stay within the behavior defined in the specification. This approach allows stepwise refinement during the design process and a formal verification of even early stages of the design. The same role/component can be represented via several process descriptions on different abstraction levels ranging from a high-level specification to a low-level implementation. The refinement property between process descriptions on different levels can be formally verified.

Chapter 5 illustrates the concept of an interaction contract by introducing a set of design patterns in the form of reusable SystemCSP interaction contracts.

3.5 Distributed Systems – Allocation

This research is focused on systems that execute on distributed computer platforms and need to interact with processes from the physical world (plant in the further text). Interaction between computing nodes takes place over network interconnections and interaction between application and the plant takes place via I/O interfaces. In our approach, plants and computing nodes are considered components, and networks and I/O interconnections are considered system-level interaction contracts. The system level interaction diagrams specify the topology of distributed system consisting of nodes and networks, and the allocation of components to nodes. Some components can be replicated on several nodes.

Alternative network routes are visible in such a diagram. It is already stated that specifying binary compositional relationships is optional. In the case of system-level interaction diagrams, that focus on the topology and allocation, the preferred choice is to completely omit binary compositional relationships, as is the case in Figure 3-32.

System-level interaction diagrams are usually used to illustrate the allocation of components and contracts participating in same interaction.

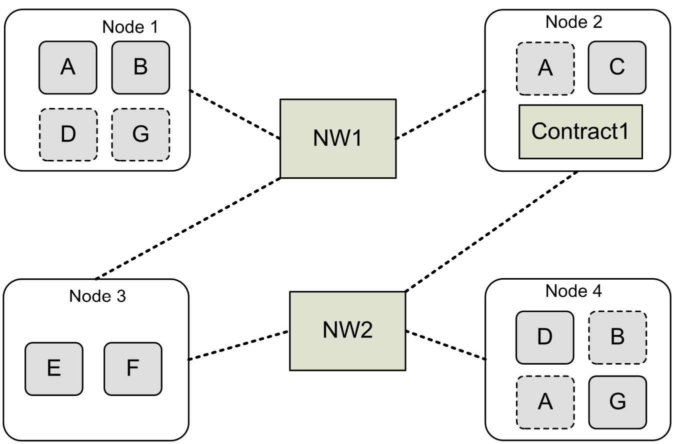


Figure 3-32 System-level interaction diagram

Figure 3-32 illustrates a system-level interaction diagram concerned with the interaction of components A, B, C, D, E, F and G via interaction contract Contract1. In this configuration, Node 4 and Node 1 actually contain the same components. The connectivity of Node 1 to Nodes 2 and 3 is via network NW1 and Node 4 is connected to Nodes 2 and 3 via network NW2. This network topology suggests that this might be a design of a fault tolerant system that can survive a failure of either network NW1 or NW2. It can also survive the failure of either Node 1 or Node 4. For the proper synchronization of replicas, an additional interaction contract is needed. Chapter 5 provides designs for the *hot-standby*, *cold-standby* and *majority voting* replica managing interaction contracts.

Replicas are optional. In Figure 3-32 a special symbol is introduced to mark *optional* components (components D and G on node 1, A on node 2, and A and B on node 4). The used symbol is the symbol for a component, but with a border depicted using a dashed line instead of a solid line. The *optional process block* is, in analogue way, represented as a process block whose borders are visualized via a dashed line.

Software components and contracts must always belong to some node. One can express this visually by assigning a special port (see Figure 3-33) to every software component. This port needs to be attached to some node kind of component.

The difference of such *node ports* as opposed to interaction related ports, is comparable to the difference between power supply ports in electric circuits compared to data signal ports. Thus, a *node port* is depicted in a somewhat

different way than ordinary ports. It is located inside the component/contract giving a visual impression of a plug-in socket for putting the software component on top of the node.

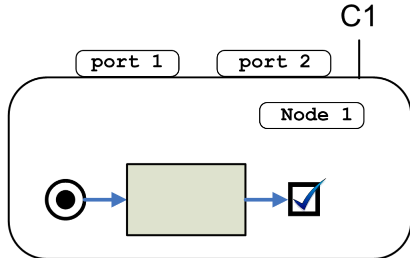


Figure 3-33 Node ports

3.6 Related Work

This section attempts to compare SystemCSP with other approaches for visual specification of interactions in concurrent software systems. First, a comparison is made to UML, which is the de-facto industry standard for software development. Then a comparison is made to GML, as a predecessor of SystemCSP based on an occam-like approach and the concept of using binary compositional relationships.

3.6.1 SystemCSP vs. UML

UML defines several different kinds of unrelated diagrams. Section 2.2.1 introduces briefly the most relevant types of UML diagrams. In UML, every diagram puts focus on certain aspect of the system and there is no clear relation between different types of diagrams.

SystemCSP gives precise relation between entities in a way that spans across various diagrams. It does this by relying on a formal method.

Structuring concurrency is in UML not primary focus. UML can describe control flows/activities going through passive objects. But it does not offer any model for structuring concurrency. UML aims to be general modeling language.

In SystemCSP, focus is on offering structured way to deal with concurrency. Consequently, a way to structure concurrency is prescribed. This is achieved through the introduction of basic elements that map to CSP operators, giving in that way the possibility for formal verification. The motivation behind the choice to prescribe the way of structuring concurrency, instead of just providing generic elements to describe any way to do that, is that insight is obtained that the main source of complexity in most systems, is in fact a concurrent existence of interacting entities. Thus, well-structured concurrency can significantly decrease the complexity of the system.

A UML design starts with capturing intended functionality in the form of the *UML use case diagrams* (see section 2.2.1 in chapter 1). In Figure 3-34, an UML use-case scenario describes ways to use a program.

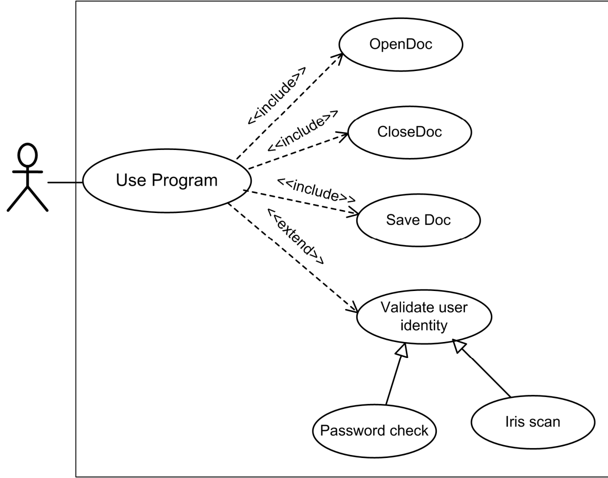


Figure 3-34 A use-case diagram in UML

Figure 3-35 depicts the same use-case specification but using SystemCSP elements defined in this chapter. In SystemCSP, it is possible to specify whether a user/system/use-case is implemented as a process or as a component.

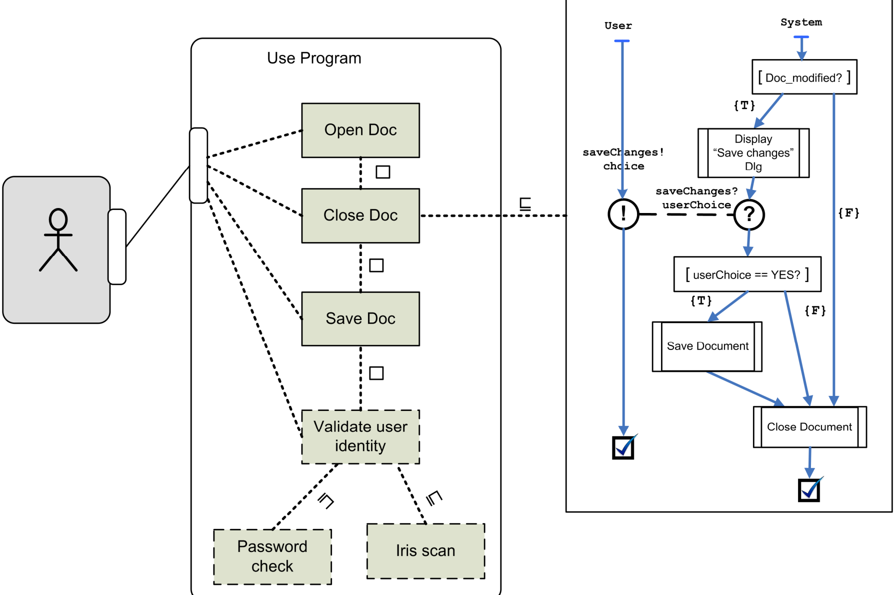


Figure 3-35 A use case scenario

A communication relationship relating the user and the use-case from the *UML use case diagram,* maps in SystemCSP to the *interaction* between appropriate processes/components.

An include relationship of a use case diagram is in SystemCSP just a containment of the process/component representing the included use-case (e.g. processes ‘Open Doc’, ‘Save Doc’, ‘CloseDoc’ are specified nested inside ‘Use Program’ component in Figure 3-35).

An extend relationship of UML use-case diagrams is mapped to the optional component/process (e.g. ‘Validate user identity’ is in SystemCSP specified as an optional process block in Figure 3-35).

Generalization/specialization is indicated via a *refinement* relation (in Figure 3-35 “Password check” and “Iris scan” are possible specializations of the more general “Validate user identity” scenario).

In addition, in SystemCSP it is possible to specify binary compositional relationships between processes/components that indicate for instance expected order of using, whether use-cases can be used concurrently, or a choice is made and so on.

In SystemCSP, specializations are expected to be refinements of more general specifications. The same is true for implementations of use-cases. In that way, the development process can be based on a step-wise refinement paradigm, starting with more abstract specifications and going iteratively towards more specific implementations that still behave in the ways defined on previous more abstract levels.

On right hand-side of a Figure 3-35, the implementation of the `Close Doc` use-case is given using a standard SystemCSP diagram. Note that activities like displaying a “`Save changes`” dialog, and performing `save document` and `close document` activities are actually depicted as non-interacting processes. The reason is that on the depicted level of abstraction, those activities do not communicate to the environment. However, inside they can contain events and processes.

UML state diagrams are based on the StateChart approach that is a more expressive and a more semantically rich version of the classic finite-state-machine (FSM) automata. A comparison between FSM and SystemCSP was given in section 3.2.3. As it was concluded, the main paradigm shift is in putting focus on events instead of on states. Otherwise, FSM diagrams can be considered directly translatable to SystemCSP. SystemCSP contains elements that do not have counterparts in FSM diagrams.

UML activity diagrams of are most similar to control-flow oriented SystemCSP diagrams. An *activity* element of an activity diagram is essentially a process or a component in SystemCSP. Fork and join of activity diagrams are PAR FORK and PAR JOIN in SystemCSP. A control-flow arrow in an activity diagram is a prefix element in a SystemCSP diagram. A conditional branching based on the value of a logical guard in an activity diagram, is an IF choice in a SystemCSP diagram. The

data flow in an activity diagram has in SystemCSP the more precise semantics of *EventSync* peer synchronization and data exchange.

UML communication diagrams (previously known as object diagrams) emphasize structural aspects and the associations between objects via which messages are transferred. Ordering is specified by attaching numbers to operations. In a way, this is similar to the interaction-oriented diagrams of SystemCSP where focus is also on structural aspects and it is allowed to assign numbers to events in order to give better insight in the underlying interaction scenario.

UML sequence diagrams emphasize the time ordering of messages among objects participating in an interaction scenario. This is done by associating a vertical dimension of the diagram with time. Code blocks of SystemCSP are on a lower level where there is no possibility for concurrency and they can be designed or documented e.g. using *UML sequence diagrams*. *SystemCSP sequence diagrams* attempt to take best properties of *UML sequence diagrams* and SystemCSP diagrams and combine them in a more efficient way of entering code for pure computation code blocks. The comparison is done in more details in section 3.2.5 where the *SystemCSP sequence diagrams* are introduced.

3.6.2 SystemCSP vs. GML

SystemCSP started as an attempt to enhance the expressiveness of GML, especially to include state machine behavior. However, SystemCSP diverged to a completely different notation based on CSP.

Compared to CSP, the occam subset of CSP offers only limited expressiveness for specifying and designing concurrent systems. As a consequence, GML as an occam-oriented approach suffers from the same limited expressiveness. SystemCSP is CSP-based, and not occam-like.

GML has, as occam, only the ALT kind of choice. ALT is a choice between two processes with a control flow continuing at the same place after a chosen alternative regardless of what was the choice. SystemCSP strictly follows CSP semantics and introduces a guarded alternative control flow element that essentially forks alternatives without forcing the existence of a common join place. This allows building diagrams alike to FSM diagrams.

The introduction of *process entry labels* and *recursion process labels* in SystemCSP is one of the elements that enable full expressiveness of CSP. While GML does not allow recursions other then loops, SystemCSP allows mutual recursions to be specified. Process labels also help to make diagrams more readable by omitting the prefix lines connecting recursion invocation to a recursion entry point.

In GML, as in occam, a process is a structural unit like a component in modern software development. In SystemCSP, a process is, as in CSP, more a behavioral than a structural unit. A process is a named point in a control flow and sometimes it can be separated from other processes using process blocks creating in that way a

basis for defining structural units. Process blocks are abstract types that can have named or unnamed instances. Process blocks interact with environment via event/channel ports. Components are larger structural units.

GML has a somewhat cluttered readability, caused by using only binary relationships and no control-flow elements. Control-flow oriented part of SystemCSP gives better readability concerning observing control-flow patterns.

Binary compositional relationships of GML have weak semantics. E.g. specifying a *parallel binary relationship* means that the related processes do have a common parent construct of type `Parallel` somewhere upwards in the hierarchy of processes. Exact location of such a common parent construct is imposed via explicit grouping.

SystemCSP introduces WEAK, START, EXIT and STRONG types of relationships. This enhances the expressiveness by allowing the designer to specify expected synchronizations on start and termination events in addition to specifying only weak versions of relationships.

In addition, the meaning of indexes associated with relationship ends is interpreted as the preference ordering of the compositional relationships, and not as a consequence of grouping. This way of interpretation created basis for fragmenting design in any number of interaction diagrams. The same component can appear in different interaction diagrams concerned with different aspects of the designed system.

Interaction diagrams are allowed not to be fully specified. By grouping, in separate diagrams, the components participating in certain interactions, and by displaying only the relevant binary compositional relationships, cluttering of readability is avoided in SystemCSP interaction diagrams. In the same time, early design freedom is enhanced.

GML lacks explicit support for modern notion of component-based design and development. SystemCSP does introduce the basic notions of component-based development.

3.7 Positioning

Figure 3-36 represents an attempt to position SystemCSP in the overall software development process. SystemCSP diagrams aim to cover the areas of component-based design, structuring the concurrency and specifying deployment. In addition, the sequential, pure computation code, can be specified via SystemCSP diagrams. However, UML class and object diagrams are still useful in combination with SystemCSP. A prospective SystemCSP tool should provide design editor for all those kinds of diagrams. On lowest level, there is domain specific modeling, with domain specific diagrams and tools. Idea is that code generated by these tools can be imported in the prospective SystemCSP tool.

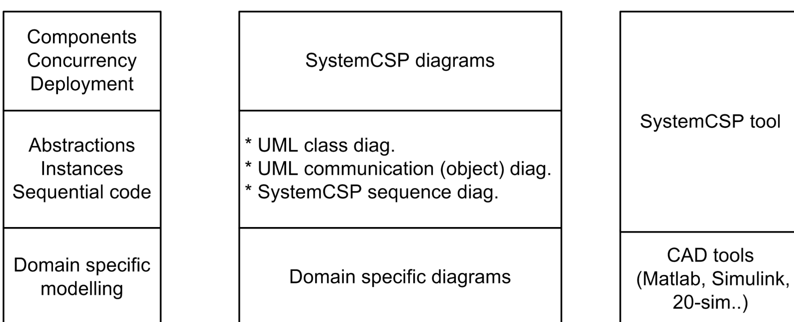


Figure 3-36 Development areas, and diagrams and tools intended to cover them

3.8 Implementation

At the moment of writing, there is no tool that can support SystemCSP design methodology. Diagrams given in this thesis are drawn in Microsoft Office Visio by creating a template containing dedicated set of shapes corresponding to the elements of the notation. Thus, no code generation is possible, and no automated formal checking is available. This thesis provides theoretical support for the prospective tool in the form of the basic notation elements, illustration of their mapping to CSP, metamodel of the notation and the design of software framework library that can support execution of code generated out of SystemCSP models.

Metamodel of the notation

A model in any domain is made out of instances of abstractions, interconnections and laws of the domain. A metamodel provides definitions of the abstractions used in a modeling domain and captures their possible structural relationships. Any model in a domain is, seen in that light, some combination of instances of the abstractions and relationships allowed by the underlying metamodel. These abstractions, as well as a set of possible relationships between them, can, for instance, be expressed via a set of UML class diagrams.

A purpose of creating a metamodel is making a definition of the modeling domain. Such a definition can, for instance, be used as a basis of a structured way to capture models in tool implementations. Another advantage of using a metamodel is its potential to introduce a standard, tool independent, way of data interchange between different domains and tools.

Appendix A uses UML class diagrams to define the metamodel of the SystemCSP design domain.

Mapping to software domain

In previous sections, a graphical notation was introduced. However, to be really useful, a design entered in a visual notation does need to have an efficient implementation in software, hardware or some other domain. Figure 3-37 depicts domains relevant for SystemCSP models. Inside a prospective tool, creation of

models should be possible via using graphical editors and views to update and inspect the current structure of the model. From a single non-ambiguous model, it should be possible to generate CSPm scripts, software and hardware implementations in different software and hardware programming languages.

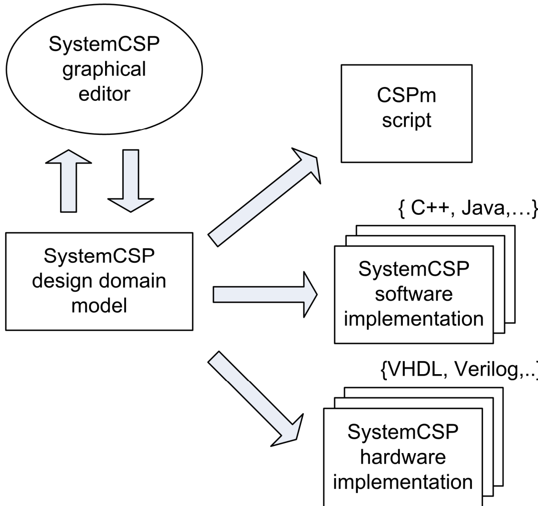


Figure 3-37 SystemCSP source and target domains

Code generation is a procedure of transforming a model from its source domain representation (the SystemCSP design domain in Figure 3-37), to a target domain representation (CSPm script, SystemCSP software implementation and SystemCSP hardware implementation in Figure 3-37).

Appendix B focuses on the infrastructure needed in the 'SystemCSP software implementation' target domain to support the implementation of a model specified in SystemCSP. The architecture of this framework for the software implementation of SystemCSP designs is also visualized using UML class diagrams. At the moment of writing, specified software framework is partly implemented.

3.9 Conclusions

A novel graphical description language for concurrent, component-based systems is introduced. The SystemCSP notation is intuitive, readable and based on CSP formal algebra. The notation has two essential viewpoints: control flow oriented part and interaction oriented part.

SystemCSP also includes concepts from modern component-based software engineering practice. The notion of *interaction contract* allows specifying, studying, analyzing (e.g. formal checking) interaction patterns in abstract way, in the isolation from the actual context of usage.

Incremental design of *control-flow diagrams* is possible by adding restrictions in different *interaction diagrams* throughout the process of system design. This is

particularly useful in early stages of the design, when the focus is on interactions of sets of components in isolation from the rest of the system. At the end of the design process, all different diagrams converge into a single, formally verifiable, system expressible via control-flow diagram.

The notation has been compared to relevant related graphical design specification languages, namely UML and GML. SystemCSP does incorporate some ideas from both. From its predecessor GML, concept of binary compositional relationships is inhereted and reused in interactionoriented diagrams. UML has also strongly influenced SystemCSP. The comparison with UML illustrated that SystemCSP is capable to offer alternative to the most types of UML diagrams. The exception to this is the *UML class diagram* that is convenient for usage in combination with SystemCSP diagrams.

4 Real-time and CSP

Everywhere is within walking distance if you have the time.

Steven Wright

Various approaches attempt to introduce ways to specify time properties in CSP theory (Roscoe, 1997; Schneider, 2000). SystemCSP as a design methodology based on CSP and intended to be suitable for the real-time systems application area, offers a practical application of those theories. The way in which time properties are introduced in SystemCSP also makes a connection between the two referenced approaches of theoretical CSP. Section 4.1 presents ways to specify time properties in SystemCSP.

Specifying time properties is one part of the problem. It allows capturing time requirements and execution times. In practical implementations, the resulting time behaviour of processes is also the consequence of time-sharing of a processor or network bandwidth. This time-sharing implies switching the context of execution from one involved process to another, where the order of execution is based on some kind of scheduling.

Classical scheduling theory offers recipes to give real-time guarantees for systems where several tasks share the same processing or network resource using some priority based scheme. However, as it will be illustrated in section 4.2, there is an essential mismatch between the programming paradigm assumed by classical scheduling techniques and the one offered by the CSP way of design. This mismatch raises the fundamental question: are CSP-based systems suitable for usage in real-time systems or should one rely for this application area on some other method? This chapter will attempt to show possible directions in solving the problem of achieving real-time in CSP-based systems. The first direction in addressing this problem is constructing CSP-based design patterns that can match the form required by the classic scheduling techniques. The second direction is oriented towards the creation of scheduling or real-time analysis theories specific for CSP-based systems.

4.1 Specification of time properties

4.1.1 Discrete time event ‘tock’

Roscoe (1997) specifies time properties by introducing an explicit time event named *tock*. This implicitly introduces the existence of a discrete clock that advances the time of the system one step with each occurrence of the ‘tock’ event. Time instants can thus be represented by a stream of natural numbers, where every occurrence of the ‘tock’ event can be considered to increase the current time for one basic time unit. All processes with time constraints synchronize with the progress of time by participating directly in the `tock` event, or via interaction with

processes that do. Advantages of this approach are that it is simple, easy to understand and flexible. It does *not* introduce any theoretical extensions to CSP theory and thus formal checking is possible using the same tools (FDR) as in untimed CSP.

4.1.2 Timed CSP

Timed CSP (Schneider, 2000) extends CSP theory by introducing ways to specify time properties in CSP descriptions. There is, however, (yet) no tool that can verify designs based upon Timed CSP.

Time instant associated with an event occurrence is, in timed CSP, a non-negative real number, thus assuming a dense continuous model of time. This assumption makes the verification process complicated and *not* practical. The difference between this approach and introducing the explicit time event ('tock') is comparable to the difference between continuous-time systems and their simulation on a computer using discrete time.

The method is also *not* related to real-time scheduling. It defines the operational semantics for introducing time properties in CSP-based systems. Several essential extensions to CSP are the basis for making a system of proofs according to the ones that exist in basic CSP theory. Newly introduced operators include observing time, timeout operator, timed interrupt operator, time delay and evolution transition.

Time can be observed at any event occurrence. The observed time can then be used in a following part of the process description as a free variable.

$$P = ev1@t1 \rightarrow ev2@t2 \rightarrow display(t2-t1) \rightarrow SKIP \quad (1)$$

The expression (1) specifies that the time of occurrence of event 'ev1' is stored in variable t1 and the time of occurrence of 'ev2' is stored in variable t2. Afterwards a function is called that displays the time interval between the occurrences of event 'ev1' and event 'ev2'.

The *timeout* operator is a binary operator representing the time-sensitive version of the *external choice* operator of CSP. It is offering the choice between the process specified as its first operand and the process specified as second operand. In case when a timeout event takes place before the process guarded via the timeout operator engages in some external event, the control is given to the process specified as second operand.

$$P = (ev1 \rightarrow P1) \overset{d}{\triangleright} Q \quad (2)$$

The expression (2) specifies that if the event 'ev1' is accepted within d time units from the moment it is offered, then the process will subsequently behave as process P1. Otherwise, it will behave as process Q.

The *timed interrupt* is a binary operator representing the time-sensitive version of the *interrupt operator* of CSP. The main difference is that the event that triggers the interrupt is actually a timeout event. The process specified as the second operand will be executed after the timeout event signifies that the guarded process did not succeed to successfully finish its execution in the given time interval. As opposed to the timeout operator that uses timeouts to guard only a single event, the timed interrupt operator is guarding the completion of a process. If that process does not finish its execution in the predefined time interval, its further execution is abandoned.

$$(\mathtt{ev1} \rightarrow \mathtt{P1}) \triangle_{d} \mathtt{Q} \qquad (3)$$

The expression (3) specifies that the process ev1->P1 will be granted a time interval of d time units to be performed. When the given time interval expires, further execution of the process ev1->P1 is aborted (interrupted) and the process Q is executed instead.

Introducing *time delay* (*delay event prefix* in Timed CSP) is a step from the world of ideal computing devices capable of infinitely fast parallel execution (as assumed by CSP) to the world of real target implementations. Time delay is used to extend process descriptions with the specification of execution times. In software implementations, the execution times get certain values depending on the processing node which executes a process. During this delay time, a process cannot engage in any event, that is, it acts as a STOP process. In fact, specifying the *delay event prefix* is equivalent to applying the timed interrupt operator on a STOP process as the first operand and the rest of original process as the second operand. A *delay event prefix* is specified by augmenting an event prefix arrow with a time delay value. Instead of a single number denoting a fixed execution time, it is possible to specify an interval for the expected time delay. In that case, a pair of values is grouped via square brackets.

$$\mathtt{P} = \mathtt{ev1} \overset{10}{\rightarrow} \mathtt{ev2} \overset{[10,20]}{\rightarrow} \mathtt{SKIP} \qquad (4)$$

The expression (4) specifies that after the occurrence of event ‘ev1’, process P is for 10 time units unable to participate in any event. After the interval of 10 time units expires, process P will offer event ‘ev2’ to the environment. Then, after the event ‘ev2’ is accepted by the environment, it will take between 10 and 20 time units before process P can successfully finish its execution.

Evolution transition is a way to display an observed delay between events in some particular execution of the process description. The expression (5) represents an execution in which the event ‘ev1’ has taken place 10 time units after it was initially offered to the environment, and where the event ‘ev2’ has taken place 20 time units after it was initially offered to the environment.

```
P = ev1 → ev2 → SKIP
     ⦚ 10
    ev1 → ev2 → SKIP
     ↓ ev1
    ev2 → SKIP                    (5)
     ⦚ 20
    ev2 → SKIP
     ↓ ev2
    SKIP
```

4.1.3 Time specification in SystemCSP

SystemCSP recognizes that operators introduced in TimedCSP are practical for describing time properties of systems. However, there is no real need to introduce a dense continuous model of time for modelling software and hardware implementation of processes. Therefore, in SystemCSP we start with the discrete notion of time as in (Roscoe, 1997) and introduce the basic event `tock` produced by the timing subsystem in regular intervals. Upon the `tock` event, we construct a process that implements a timing subsystem. This subsystem provides services used in the implementation of the higher-level design primitives that provide functionality analogue to the one defined by the timeout and the timed interrupt operators defined in TimedCSP (Schneider, 2000). In this way, it is possible to create designs using Timed CSP-alike operators, to describe them in basic CSP theory, making these designs amenable to formal verification equally as untimed CSP designs.

Time specification in control flow diagram

SystemCSP specifies time properties inside square brackets positioned in a separate node element or next to the element that they are associated with (see Figure 4-1). In Figure 4-1, the keyword `time` is used to denote the current time in the system. The occurrence of event 'ev1' is a point when time is stored into the variable `t1`. The time when the event 'ev2' occurs is stored into the variable `t2`.

Execution times can also be visualized in SystemCSP diagrams. In SystemCSP, as in Timed CSP, the *time delay* is specified inside square brackets. Instead of specifying a single time delay value, e.g. representing a fixed or average execution time, it is possible to display a pair of values that defines a range (compared to timed CSP formula (4) in section 4.1.2). The range of possible execution times is bounded by the *minimum execution time* (minET) and the *worst case execution time* (WCET). In addition, often it is useful to keep track of the *average execution time* (avET). In that case, a triple is specified.

The position of the time delay specification is related to the associated diagram element (e.g. next to the associated computation process block, or the prefix arrow replacing it, or next to the event that allows progress of the following computation

block). The specified delay can be just a number, in which case the default time unit is implied. Otherwise, the specification of time delay should also include a time unit. Time delay can also be specified as an expression or a variable that will, at run-time, evaluate to some time value.

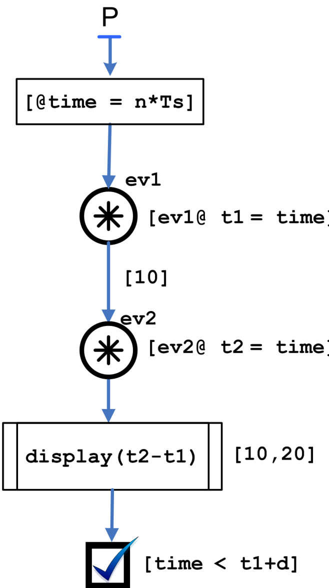


Figure 4-1 Specifying time requirements

In addition to operators defined in Timed CSP, SystemCSP also introduces a notation for visual specification of time constraints. Those constraints are not directly translated to the CSP model of the system. The time constraints specify that certain events should take place before some deadline or precisely at some time. A deadline can be set relative to some absolute time or as a maximally allowed distance in time between the occurrences of two events. Deadline constraints are independent of the platform on which they are executed. In Figure 4-1 process P is scheduled to be triggered in precisely periodic moments in time with period nTs. The time constraint associated with the `SKIP` event in Figure 4-1 specifies that its occurrence should take place strictly less then `d` time units after `t1` moment in time, or, in other words, process P must finish successfully at most `d` time units after an occurrence of the event 'ev1'.

SystemCSP time diagrams

The graphical representation of process P was in Figure 4-1 extended with specified time properties. In Figure 4-2, the time scenario of one execution of process P is depicted using a SystemCSP time diagram.

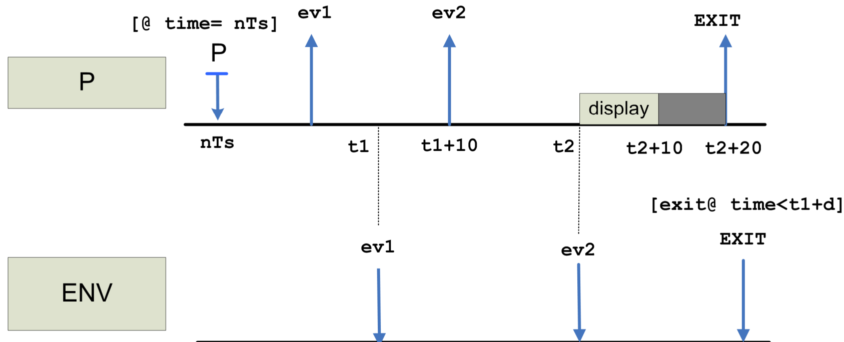


Figure 4-2 SystemCSP time diagram

Process names ‘P’ and ‘Env’ on most left side of the Figure 4-2, stand for process P from Figure 4-1 and for its environment. Arrows directed upwards indicate offered events (event ‘ev1’ being offered before time t1, and event ‘ev2’ being offered 10 time units after the occurrence of event ‘ev1’). Arrows directed downwards indicate accepted events (e.g. event ‘ev1’ is accepted at time t1 and event ‘ev2’ at time t2). Time constraints are specified inside square brackets above the related event or process entry label. E.g. temporal constraint above process entry label of process P indicate that it is executed periodically with period Ts. The temporal constraint related to EXIT event indicates that it should take place before time moment equal to t1+d.

Timing subsystem

Figure 4-3 introduces one possible design of a timing subsystem. The purpose of this example is *not* to provide a ready-to-use design, but rather to illustrate the possibility for constructing a timing subsystem starting with the ‘tock’ event.

The timing subsystem in Figure 4-3 contains several processes executed concurrently.

HW_TIMER is a process implemented in hardware that forks instances of the hardware interrupt process, `HW_INT`, in regular intervals. The `HW_INT` process synchronizes with the CPU on the event `tock`, invoking in that way the timer interrupt service routine (TIMER_ISR process). TIMER_ISR increments the value of the variable `time`. TIMER_ISR than queries a sorted list of processes waiting for timeout events. It will use the ‘wakeup’ event to awake the processes in this list for which the specified timeout time is less than or equal to the current time. The awoken processes will be removed from the top of the list.

The process CPU acts as a gate that can disable (event ‘int_d’) or enable (event ‘int_e’) the timer and other interrupts. When interrupts are enabled, event ‘tock’ can take place and as a consequence the interrupt service routine TIMER_ISR is enabled. The case of occurrence of the ‘int_d’ event, as a next event only the ‘int_e’ event is allowed and until then, the event ‘tock’ cannot be accepted, or in other words interrupts are *not* allowed to happen.

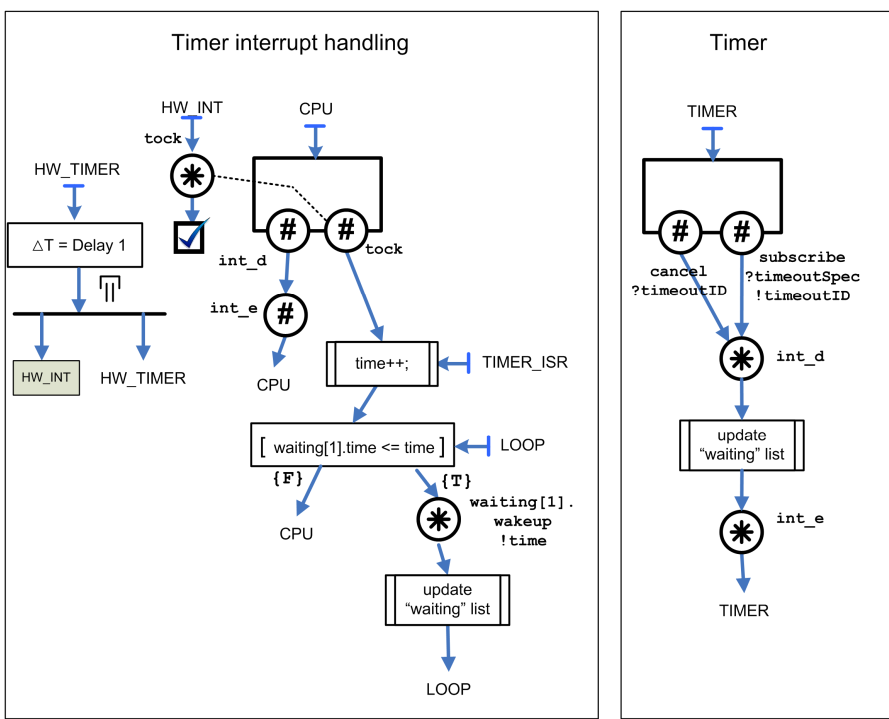


Figure 4-3 Timing susbsystem

Processes using services of the timing subsystem can, via the TIMER process, either subscribe (via event 'subscribe') to the timeout service or generate a cancel event to cancel a previously requested timeout service. Since these activities are actually updating the waiting list, this list must be protected from being updated in the same time by TIMER and TIMER_ISR processes. That is achieved in this case via disabling/enabling interrupts ('int_d' / 'int_e' events).

Watchdog Design Pattern

The interaction view specified in Figure 4-4 illustrates the interaction between a user-defined component and the timing subsystem component via the watchdog interaction contract. The watchdog pattern is used to detect timing faults and to initiate recovery mechanisms.

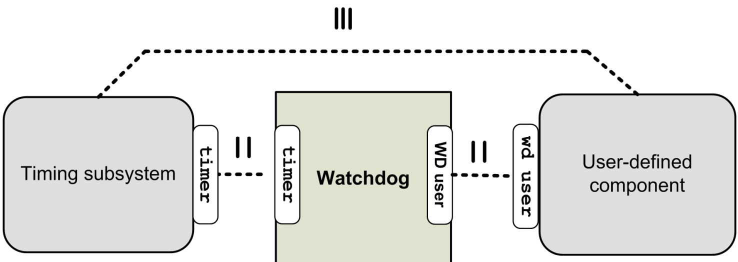


Figure 4-4 Interaction diagram: using a watchdog interaction contract

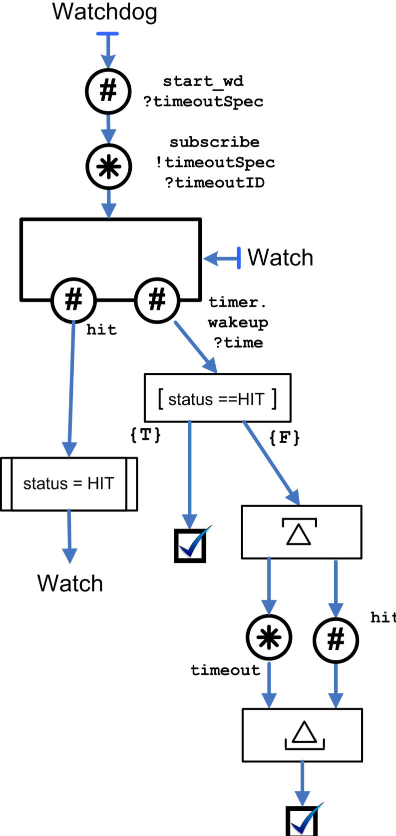


Figure 4-5 Watchdog design pattern

The design pattern for the watchdog process (see Figure 4-5) relies on services provided by the timing subsystem. A user initializes the watchdog using the 'start_wd' event, which results in a watchdog request to be notified by the timing subsystem when the specified timeout expires. In case when the watchdog user chooses to initiate the 'hit' event, the watchdog is disarmed. Otherwise, upon being notified by the timer subsystem that the specified timeout has expired (event 'wakeup'), the watchdog will initiate the 'timeout' event, notifying the user about the occurrence of timing fault.

Timed interrupt operator

The *timed interrupt* operator is simply a time-sensitive version of the *interrupt* operator of CSP. Its implementation, as depicted in Figure 4-6, contains the *interrupt* operator, and an additional events ('start_wd', 'hit' and 'timeout') for synchronization with a *watchdog* process.

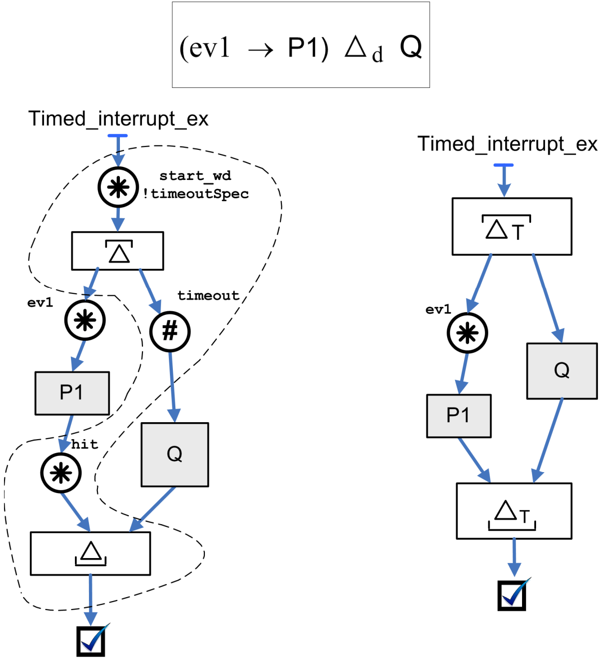


Figure 4-6 Timed interrupt – implementation and symbol

The closed dotted curve at the left-hand side of Figure 4-6 encircles elements that are providing the implementation of the behaviour specified by the *timed interrupt* operator. The right-hand side of Figure 4-6 abstracts away from those implementation details by providing a way to specify the *timed interrupt* operator as a basic element of the SystemCSP vocabulary. In fact, a pair of blocks with a *timed interrupt* symbol is used to determine the scope of the operator, similarly as brackets are used in CSP expressions.

The interaction with the associated watchdog is hidden in the implementation of the *timed interrupt* operator. This interaction (interaction of watchdog from Figure 4-5 and its user from Figure 4-6) starts with initialization of the watchdog via the 'start_wd' event. The timeout value as specified in the *timed interrupt* operator is passed to the watchdog as a part of `timeoutSpec` specification.

When the guarded process (ev1->P in the example of Figure 4-6) finishes and the 'hit' event takes place, the associated watchdog process will be disarmed. This scenario is depicted in Figure 4-7.

The occurrence of the 'hit' event will disarm the watchdog (cancel 'timeout' event). The canceled 'timeout' event is indicated via dashed line in Figure 4-7, and a big vertical arrow is used to indicate that canceling this event is a consequence of the occurrence of the 'hit' event.

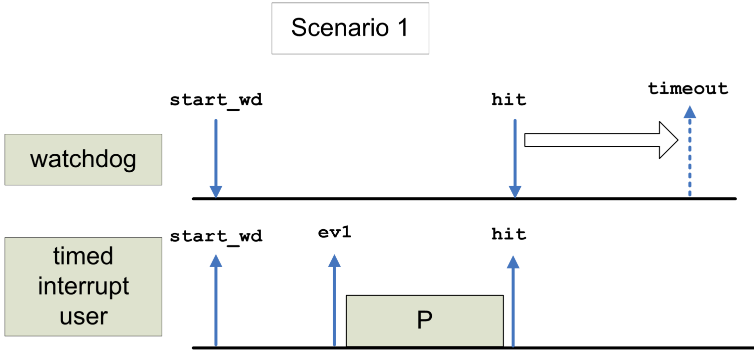


Figure 4-7 Timed interrupt -scenario 1

If, however (see Figure 4-8), the 'timeout' event takes place, it will cause the guarded process to be aborted, and the process specified as the second operand (process Q in the example of Figure 4-6) is executed. In this scenario, the occurrence of 'hit' event is prevented by the 'timeout' event as indicated again using dashed lines for the event occurrence that will not happen, and vertical big arrow relating the cause and consequence.

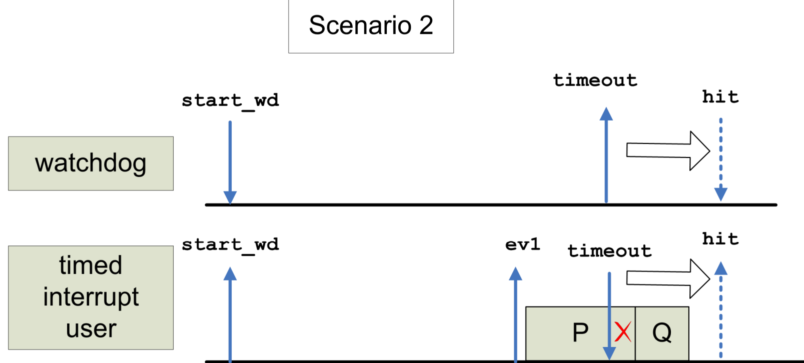


Figure 4-8 Timed interrupt - scenario 2

Timeout operator

The *timeout* operator is simply a time-sensitive *external choice* where one of the branches is a guarded process and the other one starts with a time event that will be initiated by the associated watchdog after the requested timeout expires. Following the 'timeout' event (see Figure 4-9), the process specified as second operator is executed.

Figure 4-9 depicts the implementation and a symbol of the *timeout* operator on a simple example and its associated visualization using the symbol for the timeout operator. Instead of the letter d inside the *timeout* operator symbol, it is possible to use any number or variable representing time. The left hand-side of Figure 4-9 depicts the implementation details encircled via the dotted curve, while the right-

hand side introduces notation elements used to represent the *timeout* operator as one of the basic building blocks in SystemCSP.

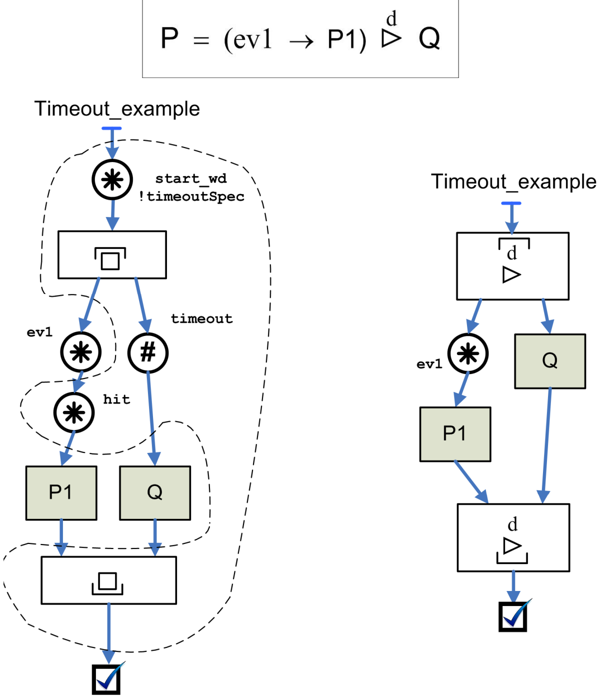


Figure 4-9 Timeout operator – implementation and symbol

Again, two scenarios are possible. In first scenario, depicted in Figure 4-10, the ‘hit’ event takes place before ‘timeout’ event, which results in the watchdog being disarmed. Difference compared to the analogue scenario for the *timed interrupt* operator (see Figure 4-7) is that in the case with *timeout* operator, only the event ‘ev1’ needs to take place before ‘hit’ event. Process P can take place any time later, since the used timeout operator guards only the occurrence of the event ‘ev1’.

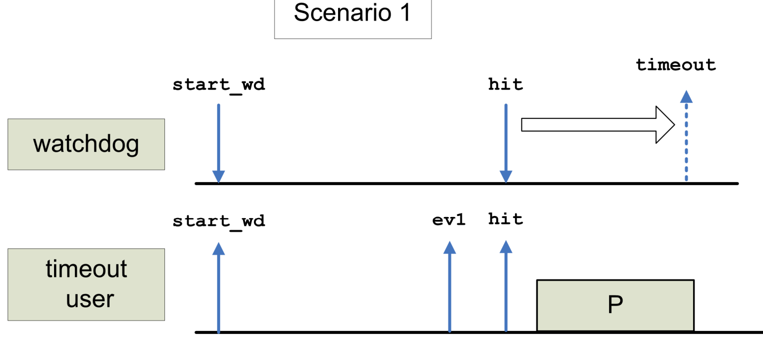


Figure 4-10 Timeout operator - scenario 1

In the second scenario depicted in Figure 4-11, the 'timeout' event will occur before event 'ev1'. Consequently, 'ev1' and 'hit' events will not occur, and instead of process P, process Q will be executed.

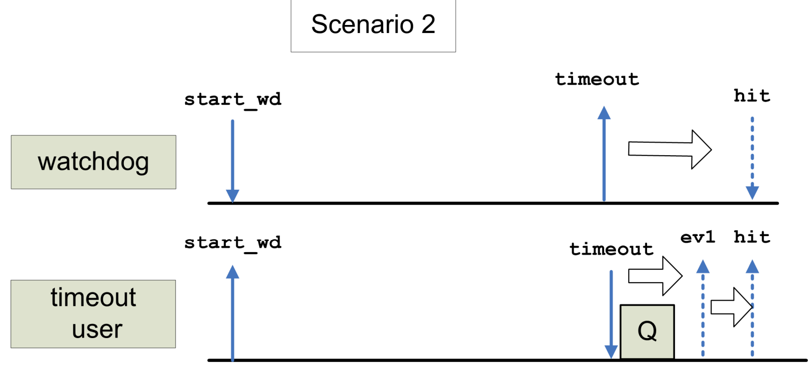


Figure 4-11 Timeout operator - scenario 2

4.2 Real-time in the implementation of CSP-based systems

4.2.1 Identifying problems

Origin of time constraints in implementation of control systems

An *embedded control system* interacts with its environment via various sensors and actuators. Sensors convert analogue physical signals to signals understandable by the embedded control system (digital quantities in case of computer-based control). Actuators (motors, valves, and so on) perform a transformation in the opposite direction. The time pattern of the interaction between the control system and its environment is based on the time constraints imposed by the underlying control theory. The computer system implementing the embedded control system must be able to guarantee that the required timing properties will be met in real time.

A control loop starts with sensor data measurements and finishes with delivering command data to the actuators. The time between two subsequent measurement (sampling) points is named *sampling period* and the time between a sampling point and the related actuation action is named *control delay* (Wittenmark and Torngren, 1995). Digital control theory assumes equidistant sampling and a fixed control delay time. On an ideal computer system, the control loop computation is performed infinitely fast. In reality, it takes a certain time that should be bounded. This gap between ideal and real computing devices reflects itself in a design choice between two possible patterns used in practice for ordering sampling and actuation tasks.

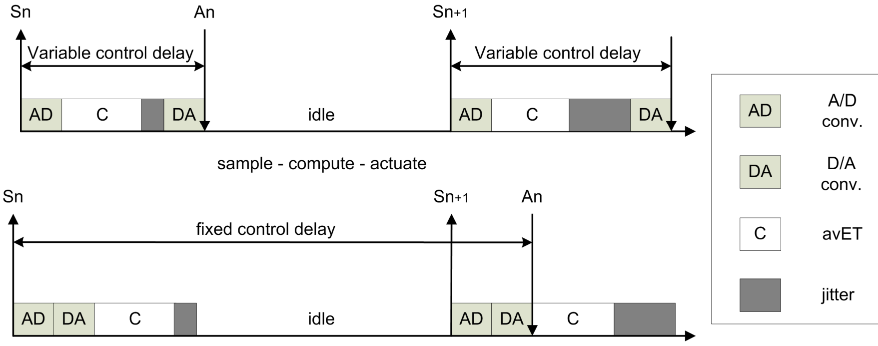


Figure 4-12 Sampling period and control delay (adapted from (ESI, 2006))

In the Sample-Compute-Actuate approach, depicted in upper part of Figure 4-12, the computation time is usually assumed to be negligible, implying that the computing device is close to the ideal one. A rule of thumb is that the behaviour of a control system will still be acceptable when this computation time is kept smaller then around 20% of the sampling period. Obviously this approach does not really guarantee that the system will always work as expected by control engineers. Especially in complex control systems that contain more than one control loop, or control loops closed over a network, the influence of a variable control delay becomes an important factor in the resulting behavior of the control system.

The second approach, Sample-Actuate-Compute, takes into account the non-ideal nature of the computation devices. In this approach, depicted in the lower part of Figure 4-12, the control delay is fixed and usually set to be equal to one sampling period. By fixing the point of actuation to be immediately after the sampling point for the next iteration, two goals are achieved: first, actuators are prevented from disturbing the next cycle of the input sampling and second, the control delay is fixed, which allows compensating for it in the control algorithm using standard digital control theory.

From these temporal requirements imposed by control theory, real-time constraints are imposed on the implementation of control systems. In both described approaches, a constant sampling frequency is achieved by performing the sampling tasks in *precisely periodic* points of time. In the first approach, the computation and actuation tasks need to get processor time as soon as possible, resulting in assigning them high priority value. The relative deadline of this task can be set using the aforementioned rule of thumb, to be 20% of the sampling period. In the second approach, as a consequence of fixing the actuation point in time, a hard real-time deadline is introduced for the computation task.

Scheduling theories

Real-time scheduling is nowadays a well-developed branch of computer science. It relies on the programming model where tasks communicate via shared data objects

protected from the simultaneous access via a locking mechanism. Good overview of the most commonly used scheduling methods is given in (Buttazzo, 2002).

Time constraints are in real-time systems met by assigning different priority levels to the involved tasks according to some scheduling policy. E.g. in Earliest Deadline First (EDF) scheduling, the task with a more stringent time requirement will get a higher priority. In Rate Monotonic (RM) scheduling, a process with higher sampling frequency will have higher priority. A good comparison of all advantages and disadvantages of EDF and RM is given in (Buttazzo, 2005).

An emerging way to schedule tasks in control systems is presented in (Cervin and Ekerz, 2006). There, it is demonstrated that, compared to EDF and RM, better performance of a control system can be achieved when priorities are dynamically assigned according to the values of some performance parameters inside the control system.

Designs based on shared data objects as assumed in classical scheduling theories differ from designs based on rendezvous-based communication, as assumed in CSP. In communication via shared data objects, *no* precedence constraints (set of "before"/"after" relationships between processes specifying set of relative orderings of the involved tasks) are introduced by the communication primitives.

Fundamental mismatch between CSP and classical scheduling – communication induced precedence constraints

A rendezvous synchronization point introduces a pair of precedence constraint dependencies. In Figure 4-13, the control flow specifies that process A must be executed before process C and process B before process D. In addition, due to rendezvous synchronization on event 'ev1', subprocess A must be executed before subprocess D and subprocess B must be executed before subprocess C. In the right-hand side part of the figure, this is illustrated by dashed directed lines specifying precedence constraints from subprocess A to subprocess D (let us abbreviate this with A->D) and from subprocess B to subprocess C (B->C). Note that A, B, C and D are processes that can contain events and synchronize with the environment.

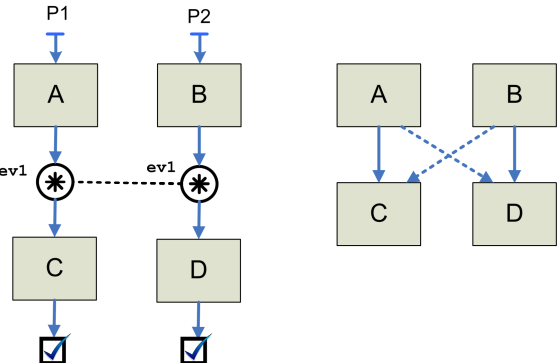


Figure 4-13 Rendezvous communication introduces new precedence constraints

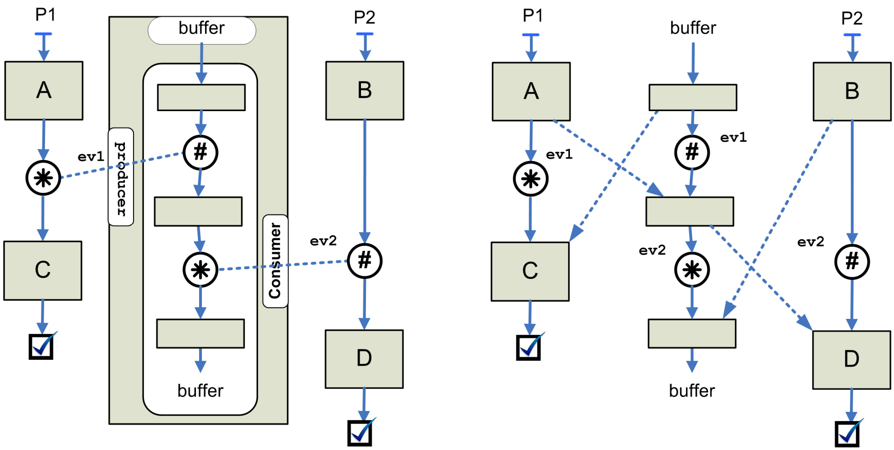


Figure 4-14 One-place buffer

In Figure 4-14, the communication from process P1 to P2 is buffered via an intermediate buffer process. Precedence relations are depicted using the oriented dashed lines. In fact, as depicted in Figure 4-15, the precedence dependency from B to C (B->C) is gone and only the precedence dependency from A to D (A->D) still exists. Its cause is that the data must arrive to the buffer before it can be consumed. If the data flow direction for the buffered asynchronous communication was from P2 to P1, then only the B->C precedence constraint would exist.

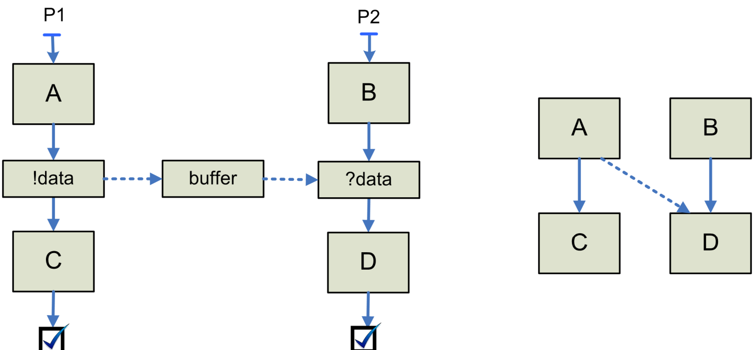


Figure 4-15 Precedence constraints for buffered communication

If, however, communication is via shared data objects (see Figure 4-16 and Figure 4-17), *no* precedence constraints are involved. The reason is that the shared data object has the semantics of an overwrite buffer, where a consumer always consumes the last fresh value available. In fact, in this case, it is more appropriate to use the term reader instead of the term consumer.

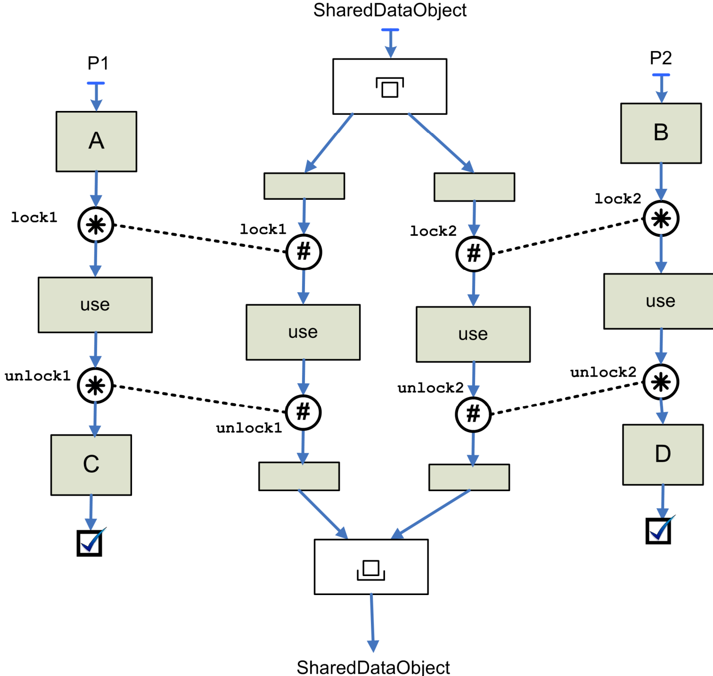


Figure 4-16 Shared data objects

Note that in case of shared data object communication, a process can still be blocked on waiting to access the shared data object. Scheduling theories do take into account this delay by calculating the worst-case blocking time.

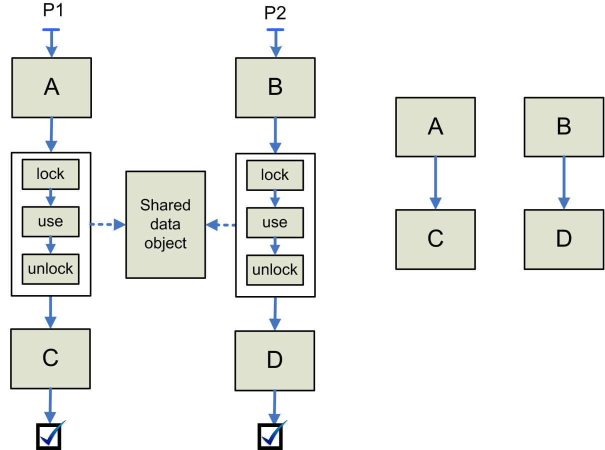


Figure 4-17 Precedence constraints in shared data object communication

In Figure 4-13, usage of a rendezvous channel yields the possible orderings of subprocesses: (A || B) ->(C || D). The symbol A || B is used to abbreviate that A and B can be executed in any order, which is equivalent to composing them in parallel.

When an asynchronous channel is used, it is equivalent to erasing one of the two communication induced precedence constraints (depending on the direction of data flow as explained above) and the resulting set of possible orderings is larger, allowing e.g. ordering A->C->B->D (after A process P1 writes to the buffer and continues), which is not covered originally. When shared data objects are used, the set of possible orderings is even larger because another precedence constraint is removed. Thus, relaxation of precedence constraints, introduced by changing the type of the applied communication primitive, leads to extending the set of possible behaviours.

Influence of assigning priorities on analysis

In systems with rendezvous-based communication, the use of priorities reduces the set of possible traces only in pathological cases (Fidge, 1993). For instance, consider the system given in Figure 4-18, with the assumption that the highest priority level is assigned to process P1, the middle one is assigned to P2 and the lowest priority level is assigned to process P3. This priority ordering can be implemented with a relative or absolute priority settings. In any case, the relative priority ordering is from P1 to P3.

Priorities defined in this way will tend to give preference to P1i blocks compared to P2j blocks and also will give preference to P2j blocks compared to P3k blocks, but in fact the real order of execution can be any depending on the order of events accepted by the environment. Thus, the set of possible orderings of processes P1i, P2j, P3k and the set of related event traces is in the general case not reduced by a priority assignment.

A PRIALT construct is in fact giving relative priorities to event ends participating in the same PRIALT construct. Those priorities are used only in the case when more then one event is ready. Again, the environment determines what events will be ready in run-time and thus as for PRIPAR, all traces are still possible.

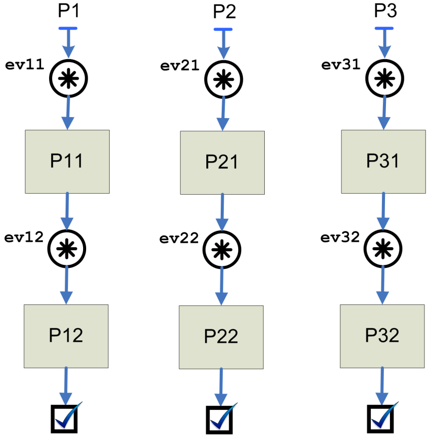


Figure 4-18 Some processes executed in parallel

From the discussion above, it is clear that assigning priorities to processes does not reduce the set of possible traces. Thus, to guarantee real-time, it should be verified that constraints are satisfied along any possible trace in the system. One reasonable approach for checking real-time guarantees is systematically replacing every composition with an equivalent automaton and associating execution times and deadlines with points in the control flow of that equivalent automaton. This approach is subject of discussion in Section 4.2.3.

The conclusion is that structuring a program in the CSP way does influence schedulability analysis, because rendezvous-based communication makes processes more tightly coupled due to the additional precedence constraints stemming from the rendezvous synchronization. In rendezvous-based systems, priority of a process does not influence dominantly the order of execution. The actual execution ordering in the overall system is dominantly determined by the communication pattern encapsulated in the event-based interaction of processes. This interaction pattern inherent to the structure of the overall system, will due to the tight coupling on event synchronization points, always overrule the priorities of the involved processes. Assigning a higher priority to one process, engaged in a complex interaction scheme with processes of different priorities, does not necessarily mean it will always be executed before lower-priority processes. This situation can also be seen as analogue to the priority inversion phenomenon in classical scheduling theory.

4.2.2 Classic scheduling approach

The most straightforward approach to making CSP designs with real-time guarantees is using CSP-based design patterns that match the programming paradigm of classical scheduling.

Priority inversion and using a buffer process to solve it

In classic scheduling, priority inversion is a situation where a higher-priority task is blocked on a resource held by a lower-priority task, which has as a consequence that tasks of intermediate priority are in the position to preempt the execution of the lower priority task and in that way prolong blocking time of the higher priority task.

Let us try to view the rendezvous channels in CSP-based systems as analogue to the resources shared between tasks in classic scheduling. In that light, waiting on a peer process to access the channel can be seen as analogue to the blocking time spent waiting on a peer task to free the shared resource. Viewed in this way, the system consisting of processes P1, P2 and P3 composed via a PRIPAR construct depicted in Figure 4-19 illustrates the priority inversion problem.

Process P1 has to wait for process P3 to enable occurrence of event ev3 and in the meantime process P2 can preempt P3 and execute although its priority is lower than the one of P1.

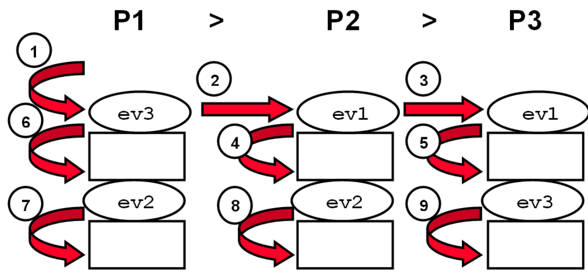


Figure 4-19 Process execution is not dominantly influenced by priority

Buffered channels are proposed in (Hilderink, 2005a) to alleviate this problem. There, a proof of concept is given by implementing the buffered channel as a CSP process. For this scheme to work, the priority of the buffer process is assumed to be equal to the priority of the higher-level process. However, a buffer process does not help when the process of higher priority (P1 in the example) is playing the role of a consumer and the process of lower priority (P3) is playing the role of a producer. In that case, the direction of the data flow introduces the precedence constraint from the event end in process P3 to the event end in process P1. Priority inversion in rendezvous-based systems is caused by precedence constraints leading from a process of lower priority to a process of higher priority. Using high-priority shared-data-object communication primitives between processes of different priorities eliminates the priority inversion problem in some cases.

Absolute versus relative specification of priorities

Classic operating systems offer usually a fixed range of absolute priority values that can be assigned to any of the tasks/processes. In occam and the CT library, the concept of PRIPAR construct is introduced that allows one to specify relative priorities instead of absolute ones. The index of a process inside a PRIPAR construct determines its relative priority compared to the other subprocesses of the same construct. A program shaped as a hierarchy of nested PRIPAR and PAR constructs, results in an infinite number of possible priority levels. This approach also offers more flexibility, since new components can be added on proper places in the priority structure without the need to change priorities of the already existing components.

However, while absolute priority ordering guarantees that any two processes are comparable, this is not the case in PAR / PRIPAR hierarchies. Let’s consider the following example:

```
PAR
  PRIPAR
    A
    B
  PRIPAR
    C
    D
```

The two PRIPARs define A as of having higher priority than B and C having higher priority than D. However, no preference is given to any when for instance B is compared to C, or A to D. Relation between them is PAR, and not PRIPAR. Giving no preference is the same as assuming they are of equal priority. The attempt to deduce strict ordering fails in the described case:

```
priority(C) = priority(A) > priority(B) = priority(D)
priority(D) = priority(A) > priority(B) = priority(C)
```

If only PRIPARs were used, there is *no* confusion. In fact, the hierarchy of PRIPARs is like collapsing a big sorted queue into smaller subqueues that can further be decomposed in subsubqueues. But with a PAR being a parent of PRIPAR constructs, the priority ordering problems appear.

The question is whether the relative priority ordering schemes, as the ones of occam-like PRIPAR constructs, can be efficiently used in combination with the classical scheduling methods, for instance RM and EDF.

To be able to apply any priority-based scheduling method in a way that will avoid introducing priority inversion problems, as concluded in the previous section, processes of different priorities should be decoupled via consistent usage of shared data objects.

RM

The problem with RM scheduling is that it is not compositional. This means that if one composes two components with inner RM based schedulers, the resulting component does not preserve real-time guarantees. Thus, it would not be possible to define a PAR of components on top level and to have PRIPAR based RM schedulers inside each of them. If, however, a hierarchy consisting only of PRIPAR constructs is used to implement the RM priority assignment on global level, this hierarchy can be seen as dividing one big queue into a hierarchical system of subqueues, where strict ordering is preserved. Theoretically, for large queues, a hierarchical organization can significantly increase the speed of searching. In practice, however, systems usually do no need more then 8 or 16 or 32 different priority levels, which allow efficient implementation based on a single status register of size 8, 16 or 32 bits and dedicated FIFO queues for every priority level. In fact the real advantage of such hierarchical tree of priority levels is that it is very flexible for extension allowing new priority levels to be inserted without disturbing the existing ones.

The conclusion is that in principle, by using global priorities or strict priority ordering defined by relying only on PRIPAR constructs, it is possible to apply the RM scheme in CSP based systems. Key precondition for this is that communication between components with different priorities is decoupled via shared data object communication primitives.

EDF

Implementing an EDF scheduler is tricky with fixed global priorities because the actual importance of a task is proportional to the nearness of a deadline and this

nearness is a factor that keeps changing in time. Trying to assign a global priority to a process whose importance in fact changes with time is unnatural. Actually scheduler framework built on top of a framework providing the occam-like relative priorities could result in a very natural solution for implementing EDF scheduler. For instance, one can divide a certain next part of the time axis into several time windows (see the right-hand side of Figure 4-20).

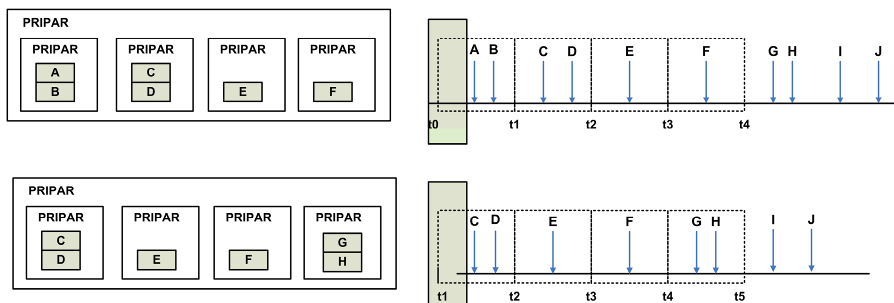


Figure 4-20 EDF scheduler

Everyone of those time windows is associated with a single PRIPAR construct (compare the left-hand side and the right-hand side of Figure 4-20). The top level PRIPAR construct is used to sort the nested PRIPAR constructs, associated with time windows from near future, according to their ordering in time. Tasks with time constraints are then inserted in the PRIPAR construct related to the time window where their deadlines fall in. E.g. tasks C and D have deadlines falling into the interval (t1, t2) and are thus in upper part of Figure 4-20 mapped onto the second PRIPAR construct.

The processes far away in the future and out of scope of any time window will be kept in a separate queue. After all tasks associated with the first time window are processed, this PRIPAR construct is removed from the top-level PRIPAR construct. The removed PRIPAR construct is then reused. It is associated with the first previously not mapped time interval. The non-allocated processes from the far-future queue that fall into that time window (G and H in Figure 4-20) are now mapped onto it and the associated PRIPAR is now added to the top-level PRIPAR as the least urgent time window (lower part in Figure 4-20).

For this scheme to work, again, there should be no precedence constraints among tasks and thus rendezvous or buffered communication is not allowed between scheduled tasks (can exist inside each of them though) or it should be taken into account by deriving intermediate deadlines as in EDF*.

The described approach for implementing an EDF scheduler is based on relative priority orderings. However, relative priority ordering is in described framework used for a generic implementation of the scheduler and not as a way to specify priorities of user- defined processes in the application, as it was the case in occam. In order to apply this scheme in practice, the application itself should specify deadlines and not PRIPAR constructs or priorities.

A problem with using relative priorities based on PRIPAR / PAR constructs, is that it hard-codes priorities in the design, while a design of an application should be independent of priority specification. A priority level is related both to the time requirements, as specified in an application, and the time properties of the underlying execution engine framework. A choice of preference is to design applications without introducing priorities and to postpone the process of assigning absolute priorities or deadlines (for EDF based scheduling) to the stage of allocation, where it is suited more naturally.

Thus, the recommendation is *not* to use occam-like relative priority orderings based on PRIPAR constructs. PRIALT on the other hand makes sense independently of the scheduling method used.

EDF*

In classical scheduling techniques, precedence constraints can be specified between tasks and special extensions exist for some scheduling theories (e.g. *modified EDF* (EDF*) for EDF) to enable them to deal with precedence constraints. EDF* takes precedence constraints into account by deriving the deadline of a task from the WCETs and deadlines of the following tasks.

When rendezvous channels are seen in the light of the introduced precedence constraints, the EDF* scheduling algorithm is applicable to rendezvous based systems. In applying EDF* to rendezvous based systems, calculation blocks can be considered to be schedulable units. A deadline of a calculation block is updated to the minimum value calculated upwards of any trace starting with some fundamental deadline and leading upwards via the chain of precedence constraints to the current calculation block. The value is calculated starting with the time of the fundamental deadline and substracting the WCET of every code block passed while going upwards the trace towards the block whose deadline is being derived in this way.

Design with rendezvous channel communication, convert them to shared data objects when necessary

Regarding solving the priority inversion problem of rendezvous-based system by relaxing the used type of communication primitive, Hilderink (2005a) states that a deadlock-free program with rendezvous channels will still be deadlock-free when rendezvous channels are substituted with buffered channels. This is in fact intuitively explainable if we realize that deadlock is in fact a circle of precedence constraints (some being due to event prefix and sequential composition operators and some due to rendezvous communication). Since substituting rendezvous channels with buffered or shared data objects removes some of precedence constraints, this can remove some deadlock problems, but cannot introduce new ones.

Thus, a convenient design method could start with a design based on rendezvous channels. Such an initial design is amenable to deadlock checking. After allocation and priority assignment, all rendezvous channels between processes of different priorities can be replaced with shared data objects allowing the usage of classic

scheduling techniques, while preserving the results of deadlock checking. However, this approach may not always be feasible. Note that relaxing the precedence constraints associated with rendezvous channels results in an extended set of behaviours by including the possible behaviours that were not formally checked. As a consequence, a new implementation that was produced in this way might not anymore be a refinement of the initial specification. For conformance of such an implementation to its specification both its traces and failures must be subsets of the traces and failures defined in the specification. Chapter 5 presents a design pattern for structuring different layers in complex control systems.

4.2.3 Event based approaches

If however, the intention is to use rendezvous-based channels/events as basic primitives then a distinct scheduling theory must be developed. The topic of achieving real-time guarantees in systems with rendezvous synchronized communication is a research field that still waits for a good underlying theory. The text here proposes two possible directions of searching for the solution.

Event-based scheduling

Figure 4-19 illustrates that attaching priorities to processes that communicate via rendezvous channels influences the behavior of rendezvous-based systems much less then expected.

Instead of trying to apply classic scheduling methods, it is possible to admit the crucial role of events in CSP based systems and assign priorities to events instead of to processes. Deadlines can be seen as time requirements imposed on events or on distances between some events. Furthermore, while priority of processes is local to a node, the priority of an event is still valid throughout the whole distributed system. Such *event-based scheduling* seems to promise a way to get better insight and more control over the way the synchronization pattern influences the execution and overrules the preferred priorities.

Section 4.2.2 has introduced an analogy between the priority inversion problem in classical scheduling methods and the analogue problem in rendezvous based systems. In classical scheduling, the standard solution to the priority inheritance problem is that a lower-priority task holding the resource needed by the higher-priority task gets a temporary priority boost until it frees the resource. If we apply this analogy to a rendezvous channel as a shared resource, then the peer process is holding the resource as long as it is not ready to engage in rendezvous. Thus to avoid priority inversions, the complete control flow of the lower-priority task, that is taking place *before* the event access point, should get a priority boost. The 'before' relation is formally expressed via precedence constraint arrows. Thus, starting from some event end, the priority of all events ends upwards the precedence constraint arrows should be updated to be of equal or higher value. Keep in mind that as explained in section 4.2.1, extra precedence constraints are introduced on every rendezvous synchronization point. Thus, the process of updating priorities propagates through rendezvous communication points to other

processes. Eventually, a stable set of priorities is reached. This set of priorities is in the general case different from the initially specified set of priorities.

The user can set an initial set of preferred priorities to some subset, or to all event ends in the program. For instance, one can initially assign priorities to event ends by assigning priorities to processes, which can for instance result in automatically associating the specified process-level priority with every event end in the process. Those preferred values are used as initial values in the aforementioned procedure of systematically updating priorities of event ends. The set of priorities obtained by applying this algorithm reveals a realistic or an achievable set of values after the synchronization pattern is taken into account. In this process, priority inversions are inherently eliminated.

In the example of Figure 4-19, in event-based scheduling, event ends initially get priorities according to the priority specified for their parent processes. Due to precedence constraints all event ends participating in the same event need to get the same value, which is equal to the highest priority present in any of the event ends. Thus, events 'ev3' and 'ev2' would get the priority of process P1, and event 'ev1' the priority of process P2. However, since a precedence constraint ev1->ev2 exists, the priority of the event ev1 needs to be readjusted in order to avoid priority inversion as described above in the procedure for realigning priorities of events. Thus, although the preferred priorities of process P1, P2 and P3 are different, their execution pattern results in all events 'ev1', 'ev2' and 'ev3' having the same priority. The event-based scheduling approach uncovers realistic, priority-inversion free, values of priority levels, achievable with the given design of synchronization pattern between processes.

The procedure is not so convenient for application in systems with a lot of recursions. There are two types of recursive processes: time-triggered recursion and ordinary recursion. In ordinary recursion there is a cycle and as a result all the events in the process have the same priority. The time-triggered recursions are considered new instances of tasks with new deadline values and there is no need to perform a circular update of priorities.

Equivalent automaton

From the discussion in Section 4.2.1, it is clear that assigning priorities to processes does not reduce the set of possible traces. Thus, one reasonable approach for checking real-time guarantees is treating CSP processes as automata and systematically replacing every composition of CSP processes with an equivalent automaton and associating execution times and deadlines with points in the control flow of the equivalent automaton. In (Ouaknine and Worrell, 2003), it is in fact stated that timed CSP descriptions are closed-timed epsilon automata.

In order to simplify the reasoning, in this discussion we restrict the analyzed models to be free from the non-deterministic and too complex primitives: systems are considered to be free from the usage of the *internal choice* operator and of those cases of the *external choice* operator that cannot be reduced to the *guarded alternative* operator. Internal choice is normally used as an abstraction vehicle and as such, it does not exist in final designs. External choice that cannot be replaced

with a guarded alternative operator is another situation that is rarely used in practice and difficult to implement and also not straightforward to describe in automata representation. The decision here is to restrict final designs to be free of those two special cases. If the set of CSP operators is restricted in this way, the processes constructed using events and this restricted set of operators can be reasoned about using classic automata theory.

Automata theory (Cassandras.and Lafortune, 1999) defines how to make a parallel composition of two automata. The example for this procedure is depicted in Figure 4-21. The start state of the equivalent automaton representing the composition is the combination of the initial states of the composed processes. In Figure 4-21, process P1 can initially engage in event 'a' and process P2 in event 'b'. Since event 'b' must be accepted by both P1 and P2, initially only event 'a' is possible. Event 'a' will take the first automaton to state 2, while the second automaton will stay in state 1. Thus, starting from the initial state (1, 1) and following the occurrence of the event 'a', the composite state (2, 1) is discovered (see Figure 4-21).

For every reachable composite state, all possible transitions are checked (taking into account when synchronization is required and when not). The resulting composite states are mapped onto the equivalent automaton. After a while all transitions either lead to already discovered composite states or to the end state (when both participating processes are in their end states) if any.

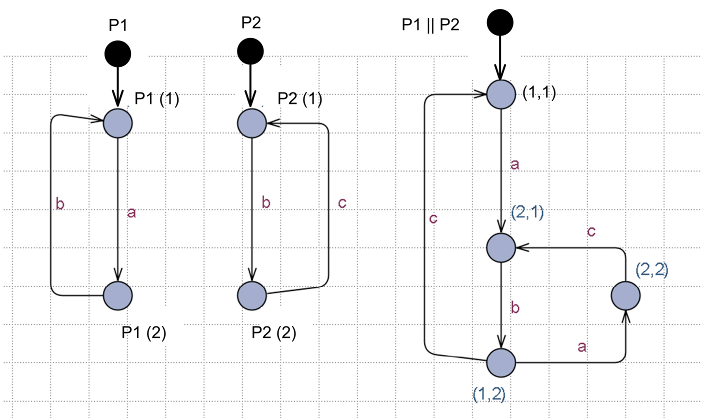


Figure 4-21 Construction of equivalent automaton

Some composite states are not reachable and thus not part of the equivalent automaton representing the parallel composition. In principle, composing, in parallel, a process containing 3 states with a process containing 4 states, yields a process with all 3*4 combinations possible. This is in fact the case whenever processes are composed in *interleaving parallel*. In non-interleaving *parallel* constructs, due to the involved synchronizations, the number of states is less.

A *parallel* composition can be seen as a way to efficiently write down complex processes that contain a number of states. Seen in that light, the introduction of the *Parallel* operator allows decomposing complex processes on entities that are smaller, focused on one aspect of the system and simpler to understand.

The definition of the *parallel* composition is in CSP identical to the one in automata theory. The *external choice* of CSP viewed as automata is equivalent to making a composite initial state that is offering the set of initial transitions leading to the start states of the involved subprocesses and subsequently behaving as those subprocesses. The sequential composition is trivially concatenating the involved automata. If every CSP process is viewed as an automaton, creating an equivalent automaton representing a complete CSP-based application is a straightforward thing to do. The equivalent automaton defines all traces (sequences of events) possible in the system. Thus, it can be used as a model against which one can check different properties of the system – e.g. checking for deadlock/livelock freedom, checking the compliance of implementation to the related specification (refinement checking). For instance, an application has a potential for a deadlock situation if there is a state (not the end state) from which no transition is leaving. The refinement checking is about testing if a set of traces and failures defined by an automaton representing some implementation is a subset of the set of traces and failures defined by an automaton representing the related specification.

In Figure 4-22 SystemCSP based visualization of the equivalent automata from Figure 4-21 is given. The main difference of a SystemCSP representation compared to the automata way of visualizing CSP processes is that instead of on states, the focus is on events. The procedure of constructing an equivalent automaton is exactly the same.

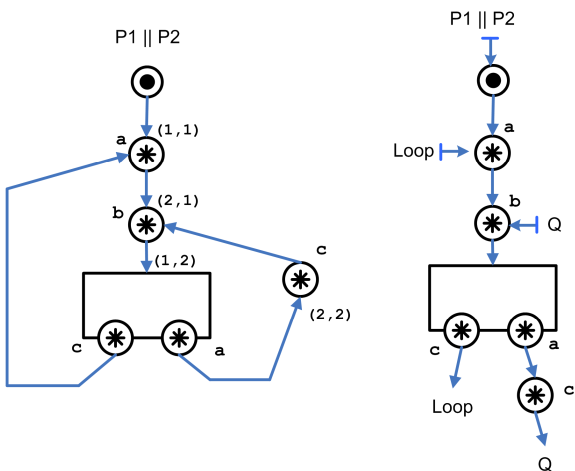


Figure 4-22 SystemCSP with recursion labels makes traces more obvious

Focusing on events ensures that traces are more easily observable, especially if in SystemCSP, instead of the lines going back to the revisited states, as is always the case in automata, usage of recursion labels is enforced. Systematic usage of recursion labels will naturally separate subtraces that are repeated and thus create immediately observable trees of possible event traces. Inspection of Figure 4-22 shows that possible traces in the single 'Loop' iteration of the equivalent automaton are <'a' , 'b' , 'c' > and <'a' , <'b' , 'a' , 'c'> ^{n}, 'b' , 'c'>. The actual trace taken is dependent on the readiness of the environment. Recursion labels

define sequences that are repeated and the IF choice and guarded alternatives divide traces into several subtraces.

Mapping time properties to equivalent automata

The next step is to extend the description of an equivalent automaton with time properties in a way that it will allow us to perform efficient analysis. The idea is to extend CSP descriptions with time properties in such a way that the mapping to the equivalent automaton preserves their meaning.

After specifying execution times and time constraints in the two subprocesses composed in PAR, time properties are mapped onto the equivalent automaton. If that can be done, then the analysis of time behaviour can be performed on the constructed equivalent automaton. If we are able to map the time properties from a pair of parallel composed processes onto their composition, then the same can be done hierarchically in bottom-up manner yielding at the end an executable timed model of complete application.

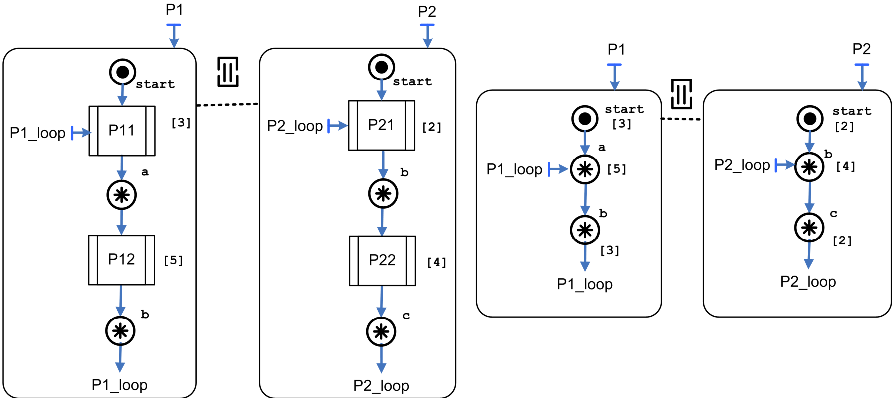


Figure 4-23 Specifying Execution times of code blocks

Analysis of time properties should be performed without the need to perform code block calculations. In such analysis, code blocks are substituted with their execution times and sums of execution times along all possible system traces are inspected with respect to the specified time constraints.

Note that execution times are in Figure 4-23 associated with event-ends. The execution times of the calculation blocks can be seen as related to the event ends immediately preceding them, that is, to event ends associated with events whose occurrence will allow every participating process to progress for the amount of execution time spent on the next calculation block.

The execution time of a process at a certain point of its execution is the sum of execution times along the path that brought the process to the current point. In other words, it is the sum of progress (expressed in time units) allowed by all event ends along the trace that the process is following.

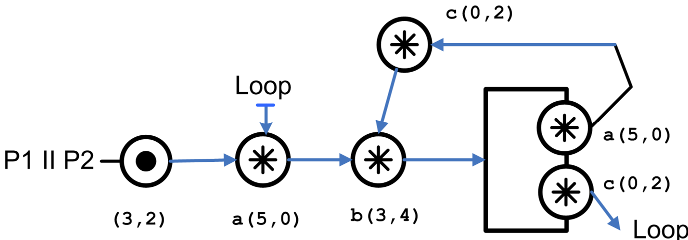


Figure 4-24 Equivalent Event Machine

The basic idea is that an event occurrence allows further progress of processes involved in that event occurrence. The initial event in process P1 (see Figure 4-23) allows process P1 to progress further in execution for 3 time units and offers event ‘a’ to the environment. The initial event in P2 allows process P2 to progress in execution for 2 time units and then offer event ‘b’ to environment. Thus, the composite initial event allows the involved subprocesses to progress for (3, 2) time units, where the first number maps to the event end in the first subprocess and the second number to the event end in the second one (see Figure 4-24). Event ‘a’, once it is accepted by the environment will allow progress of P1 for 5 time units and P2 for 0 time units since P2 is blocked waiting on its environment (including P1) to accept event ‘b’. Thus in the composite automaton, event ‘a’ taking place following the initial event, will allow (5, 0) progress of the involved subprocesses. The subsequent occurrence of event ‘b’ will allow progress of both P1 and P2, for 3 and 4 time units respectively, which is in Figure 4-24 expressed by associating the ordered pair (3, 4) with the event ‘b’.

Note that in general, when a hierarchy of processes is resolved, it is not a good idea to capture progress of subprocesses as n-tuples. In a prospective analyzer implementation, since execution times are related to event ends, bookkeeping of allowed progress would be kept in the participating event ends and not in the n-tuples, containing progress for all composed subprocesses. Different occurrences of the same event in a composite automaton can in fact have associated different progress values. Essentially, execution times are expressed as the amount of pure computation progress that occurrence of an event at immediatelly preceding event-end point allows.

Note that the assumption in this example is that the environment is always ready to accept events. In fact, when the equivalent automaton is constructed hierarchically in a bottom-up approach, it will eventually include the complete system with all events resolved internally.

Unpredictable (in the sense of time) occurrences of events from the environment can of course only be analyzed for a certain chosen set of scenarios. For every scenario, the environment can also be modelled as a process with defined time properties and composed in parallel with the application to form the complete system.

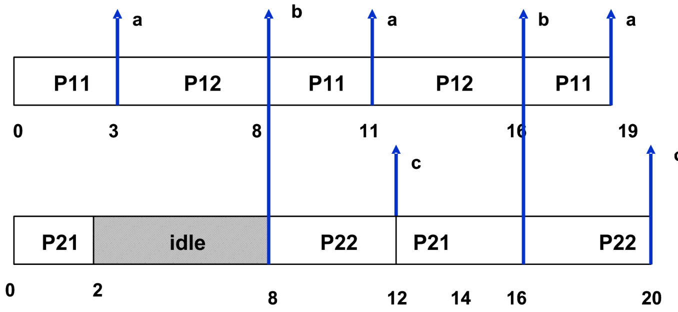
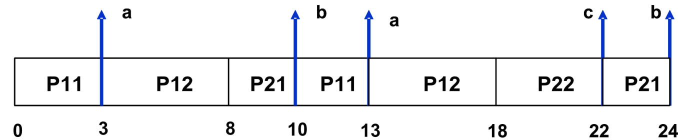


Figure 4-25 True parallelism

The actual time of event occurrences depends on the allocation. For the equivalent automaton of Figure 4-24, in Figure 4-25 the true parallelism case is depicted, and in Figure 4-26 a shared CPU with P1 having higher priority and a shared CPU with P2 having higher priority. The same equivalent automaton keeps information necessary to unwrap actual timings of the involved events in all 3 cases.

1) Process 1 has higher priority



2) Process 2 has higher priority

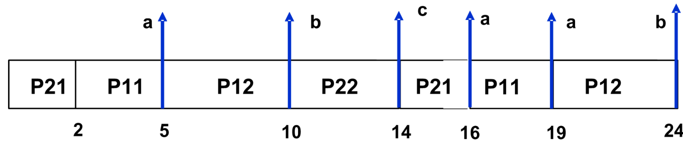


Figure 4-26 Shared CPU

In the case of true parallelism, components P1 and P2 are initially allowed to progress 3 and 2 time units respectively. Then event ‘a’ allows component P1 to progress another 5 units. Both processes synchronize on event b, meaning that their times must be same at the rendezvous point. Thus the time of this rendezvous point is max(3+5, 2+0)=8. Event b will allow components P1 and P2 to progress 3 and 4 units of time respectively. Under the assumption that the environment is always ready to accept events, event a will be accepted at time 8+3=11 and event c at time 8+4=12. This scheduling pattern is depicted in Figure 4-25.

Three independent timed models can be made by using in the analysis either the minimum, average or worst-case execution times. Using only the average execution time is a good first approximation of the system’s behavior. Execution times are dependent on allocation of components to processing nodes and can in

fact be measured or simulated for different targets and stored in some database. A prospective tool should be able to keep track of allocation scenarios and to simulate/analyze/compare effects of the different execution times in different allocation scenarios.

4.3 Conclusions

In this chapter, ways to introduce time properties were defined in the scope of the SystemCSP design methodology. The specification of time properties was deduced by merging the ideas from previous work in the CSP community (Roscoe, 1997; Schneider, 2000). Implementation of CSP-based systems with real-time properties was then investigated. Two major directions were observed for achieving real-time behavior: (1) introducing design patterns that can fit CSP-based systems into requirements of existing scheduling theories and (2) relying on constructing distinct scheduling theories for CSP-based systems. Comparing the two indicates is that the first proposed direction enables immediate implementation, while taking the second direction requires additional research. Thus, a recommendation for a prospective tool used for editing SystemCSP designs is to use the combination of proposed design patterns and classical scheduling theories to provide real-time guarantees. In long term, it is suggested to investigate the second approach. Reason is that transforming rendezvous channels to shared data objects removes some precedence constraints, which results in extended set of possible behaviour, and in practice means that refinement property might not hold after such transformation.

5 Design patterns in SystemCSP

Enter a mold without being caged in it. Obey the principles without being bound by them. Don't get set into one form, adapt it and build your own, and let it grow, be like water. Now you put water in a cup, it becomes the cup; You put it into a teapot it becomes the teapot.

Bruce Lee

In this chapter, a set of SystemCSP design patterns is introduced with the following intentions:

- to make the initial body of reusable building blocks useable in SystemCSP based designs. The choice of designed reusable building blocks is geared towards the application area of component-based, safety-critical control systems. Introduction of the reusable design patterns extends the vocabulary of the notation by introducing set of more complex primitives built upon the basic ones. Using such a higher-level language primitives raises the abstraction level of the design process and is expected to result in reduced software design effort, reduced costs, and in simplified expression of more complex designs. In fact, this introduction of patterns is a way to enhance the scalability of the notation. This is especially the case when a graphical symbol is introduced to represent a pattern. In some cases, intuitive symbols are introduced, and in others the introduction of symbol is only suggested.

- to illustrate the usage of interaction contracts as reusable units in the practice of software development. The advantage of using interaction contracts, as a way to specify interactions among components, is that they are as reusable as components, and can be analyzed in isolation from the actual system and the set of actual components participating in the specified interaction. Most of the design patterns introduced in this chapter are presented in the form of interaction contracts.

- to obtain insight that can be used as a feedback to further improve the notation.

- to test the capabilities of the SystemCSP language when used as a vehicle in visualizing and structuring concurrency in interaction based systems. The evaluation of the notation is done according to the criteria listed as a part of the problem statement in section 1.3.1. Expressivness and readability are the key features expected of the notation that can be evaluated using this chapter.

The methodology used here is to describe every pattern in structured way. First, a brief explanation of the key purpose and ideas of the pattern is presented. In that way, the design problem is introduced. Second, the design is given in the form of a SystemCSP diagram and the associated textual description. Third, a brief

discussion is given containing remarks with issues specific to the described pattern. Those remarks can be, for instance, comments on the usage, or comments on the need for introduction of some additional notation elements, or introduction of a symbol for given pattern, or a comparison with similar patterns in other component-based frameworks. The remarks contain, when convenient, comments about scalability, expressiveness and readability properties.

Expressiveness is in fact a criterion that should evaluate how easy is it to design interaction for given descriptions of the key objective and behavior of a pattern.

Readability can be evaluated by the speed and the level in which SystemCSP diagram can be correctly interpreted knowing the problem statement, but without reading the associated textual description of the diagram.

Design patterns are grouped in several sections. Section 5.1 introduces basic communication patterns. Section 5.2 defines some patterns useful in component-based software development. Section 5.3 introduces a possible way to structure layers in a control system. Section 5.4 presents a set of fault tolerance design patterns. Section 5.5 briefly states conclusions.

5.1 Communication patterns

The purpose of this section is to design a set of SystemCSP interaction contracts representing commonly used communication primitives.

Design patterns – goals and ideas

In the *shared memory* pattern, the shared part of memory needs to be protected from simultaneous access by multiple users. Shared memory can be structured in any way, e.g. it can contain complex data structures. The ownership of shared memory is given to the process that *locks* it.. Other processes wanting to access the memory need to wait until the process holding it *unlocks* it.

The *shared variable* pattern provides the mutual exclusion in accessing a single variable by multiple readers and multiple writers.

In the *one-place buffer* pattern, some data is placed in a buffer by a producer, and stays there until it is consumed by a consumer. This is different from the shared variable pattern where the data value is present in variable until it is overwritten by a new value.

FIFO buffer is a buffer capable to store multiple data chunks, which are consumed in order of storing. In that way, it provides more flexible synchronization in case when frequencies of `producers` and `consumers` accessing it are varying in time. FIFO buffer discussed here is assumed to have a fixed storage size. Thus, it can happen that such a buffer is filled (no space available) and not able to receive a next chunk of data. It can also happen that buffer is empty (no data available) and thus not able to deliver data to a consumer.

Design description

The shared_memory interaction contract (see Figure 5-1) is designed to initially allow a user to *lock* the shared memory using 'lock' event. After the memory is locked, the next event that the contract offers to its environment is to *unlock* memory ('unlock' event). It is assumed that the same user that has locked the memory will unlock it. Unlocking the memory means that the current user has finished the current session of data access and update and that a next user is allowed to take ownership over the shared memory. Assumption is that user processes will always invoke events 'lock' and 'unlock' in this sequence. This is described with the role user in the contract. Additional care should be taken that user processes do not synchronize among themselves on 'lock' and 'unlock' events.

In the shared_variable interaction contract (see Figure 5-1), the guarded alternative element allows either the 'read' or the 'write' event to be accepted, allowing in that way, at any point of time, access to either one reader or one writer.

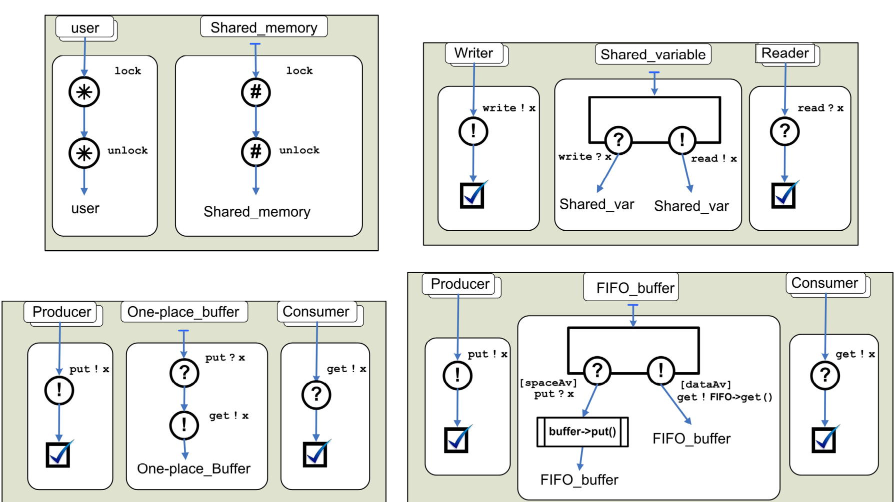


Figure 5-1 Communication patterns

The One-place_buffer interaction contract (see Figure 5-1) is initially ready to receive data from producer (offering a 'put' event). When 'put' event takes place, data is transferred from the producer into the buffer. The contract manager is then ready to deliver the data to the consumer (offering a 'get' event). After the data is consumed (occurrence of the 'get' event), the one-place buffer is again ready to accept new data from the producer (offering 'put' event).

In guarded alternative of the FIFO_buffer interaction contract, events 'put' and 'get' are guarded with logical guards 'spaceAv' and 'dataAv' representing respectively the facts that data can be put to the FIFO only when there is space, and that data can be consumed out of FIFO only if it is available.

Remarks

Multiplicity of specified roles is indicated with multiple interface ports (most often visualized on top of each other) associated with role in the interaction contracts. In a shared memory contract, assumption that users do not synchronize among themselves on 'lock' and 'unlock' events is not made explicit on the diagrams. A way to visually specify that assumption is to draw interleaving parallel binary relationship of the role `user` with self (see Figure 5-2). That is analogue to associations relating a UML class with itself, and carrying the meaning that the specified relationship relates instances of the class.

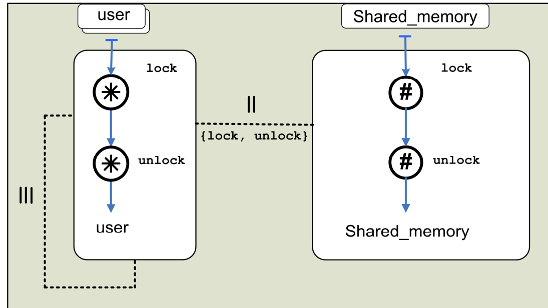


Figure 5-2 Shared memory contract with specified compositional relationships

Note that specifying interleaving parallel binary relationship between roles does not mean that it will still hold among components implementing the roles. Implementations of the roles need to be interleaving with respect to events relevant for those roles ('lock' and 'unlock' in this case), but can synchronize (e.g. by participating together in some other interaction contracts) on some other events not relevant for the roles (any event in their alphabets except 'lock' and 'unlock').

Often, visualizing compositional relationship additionally clutters the intended message of the diagram. Therefore, visualization of compositional relationships among participants in contract definition is optionally visualized, e.g. when it carries additional information important for proper understanding of the interaction contract. When the relationship is not visualized it is still expected to be specified in properties of the contract. The trade-off made here is between expressiveness and readability of the diagram. Since the property left out from the design diagram is expected to be specified in a property page of the contract in a prospective tool, unambiguous interpretation is not sacrificed by leaving compositional relationships out of the diagram.

`The Shared_variable` contract allows participation of multiple writers and multiple readers. At any moment of time, only a single writer or reader can engage in interaction with this shared variable interaction contract. Again, the assumption is that writers do not synchronize among themselves on 'write' events, and that readers do not synchronize among themselves on 'read' events. The same holds for multiple producers and consumers not synchronizing among themselves on 'put' and 'get' events in the `One-place_buffer` and and `FIFO_buffer` interaction contracts.

In all interactions in all presented communication patterns, initiative is taken by the side using the contract, meaning that the event-ends inside interaction contract managers can be depicted using type *EventAccept* and event-ends in the roles using the event-ends of *EventSync* type. However, a choice taken in three of four patterns is to use basic *Reader* (event-end circle with question mark inside) and basic *Writer* (event-end circle with exclamation mark inside) channel-end symbols in order to focus on the direction of data communication. Sometimes, however, it is equally important to emphasize both which side can initiate and which side can only accept events, and to emphasize the direction of the communication. In such cases, it is suggested to use *EventSync* and *EventAccept* event-ends, since the direction of data communication is also specified in the event labels. In fact, *Reader* and *Writer* type of event-ends are redundant and exist in order to increase readability of the information about the direction of data communication for unidirectional channels.

Recommendation is to invent intuitive symbols representing the presented and other relevant communication design patterns in order to raise the level of abstraction in design process.

5.2 Patterns related to components

5.2.1 Inter-component function calls

Design patterns – goals and ideas

Inter-component function calls are usually classified as *synchronous* or *asynchronous*. A *Synchronous function call* assumes that caller is blocked after making the call waiting for the function to be performed and results returned to it. In an asynchronous function call, a caller will send a function call request and continue working on something else. The results will be delivered to the caller asynchronously, e.g. by using a *callback* function.

In function calls, arguments can be passed *by value* or *by reference*. Passing argument *by value* means that the object passed as argument is used to initialize the local object inside function definition. Changes in such a local copy will not change the original variable. Passing arguments *by reference* means that the function will actually work on the original object without creating a local copy. Thus, the function has the right to update the value of variables passed by reference.

Design description

In Figure 5-3, SystemCSP design patterns for implementing *synchronous* inter-component function call is depicted.

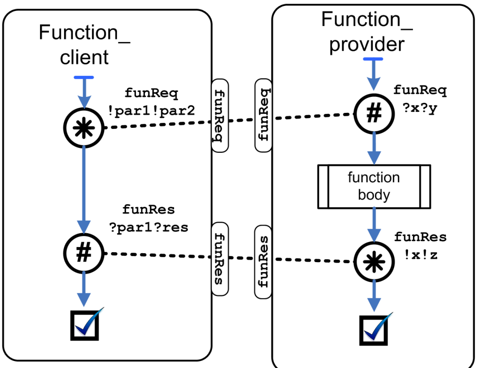


Figure 5-3 Synchronous inter-component function call

Design pattern given in Figure 5-3, is analogue to synchronous function call because control flow of the caller is blocked on the call until function is performed and results are returned. Client uses channel 'funReq' to send parameters `par1` and `par2` to the function provider. The function provider performs calculation and returns back the result (event 'funRes'). In this case, also the first parameter of the function call is returned. Thus, the function provider is allowed to update the value of the first parameter. This is in fact equivalent to passing the parameter par1 *by reference* and par2 *by value*.

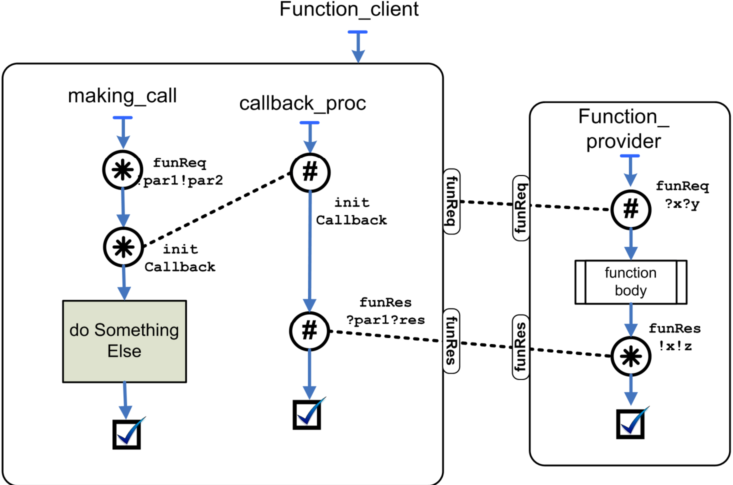


Figure 5-4 Asynchronous inter-component function call

In Figure 5-4, a way to perform an equivalent of *asynchronous* function call is depicted. The caller can do something else instead of waiting for the results. This is achieved by activating (via event 'initCallback') a dedicated process (`callback_proc`) that does execute concurrently with control flow that is making the function call. This dedicated callback process will wait for the results on the behalf of the caller. It can be composed in parallel with the normal control flow of the component.

Remarks

Obviously, two channels are used for implementing an inter-component function call. However, logically this pair of channels makes one interface unit. Also, while channels are symetrical, function call ports have provided and required side. From that reason, a symbol for *function port* is introduced (see Figure 5-5 and Figure 5-6).

A visual distinction of *function port* compared to event/interface ports is the presence of the function name followed by a pair consisting of opened and closed brackets resembling in that way function declarations in modern programming languages. In fact, function name can be left out, but the pair of brackets marks the interface port as *function port*.

In addition, the synchronization connection that relates two function ports, is adorned with 'sync' or 'async' label, specifying in that way whether the control flow of the caller is blocked until the function is performed and the result is returned.

The distinction between provided and required side is as in classic client/server architectures. This can be somewhat different from the previously defined *interface ports* that are related to role specification/implementation, and where required interface is associated with specification and provided interface with the implementation. Thus, the clear distinction in used symbols for function ports and role related interface ports is important for proper understanding of diagrams.

On the client side it is possible to introduce symbols to abbreviate the pattern (see the internals of `Function_client` component in Figure 5-5 and Figure 5-6). In that way, higher-level primitives are created. Comparing Figure 5-5 with Figure 5-3 and Figure 5-6 with Figure 5-4 one can observe parts of the pattern design that are assumed by symbolic representation.

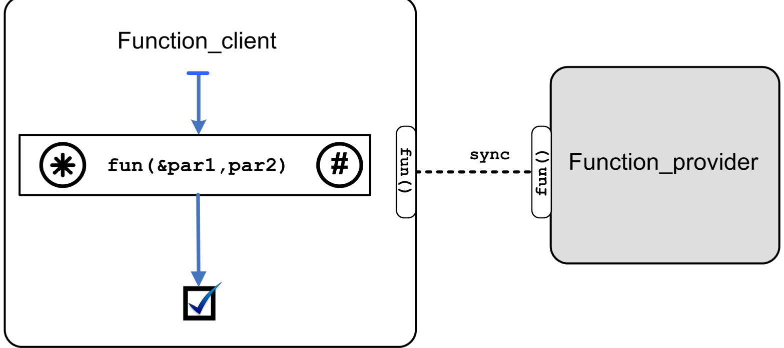


Figure 5-5 Introducing symbols for synchronous inter-component function calls

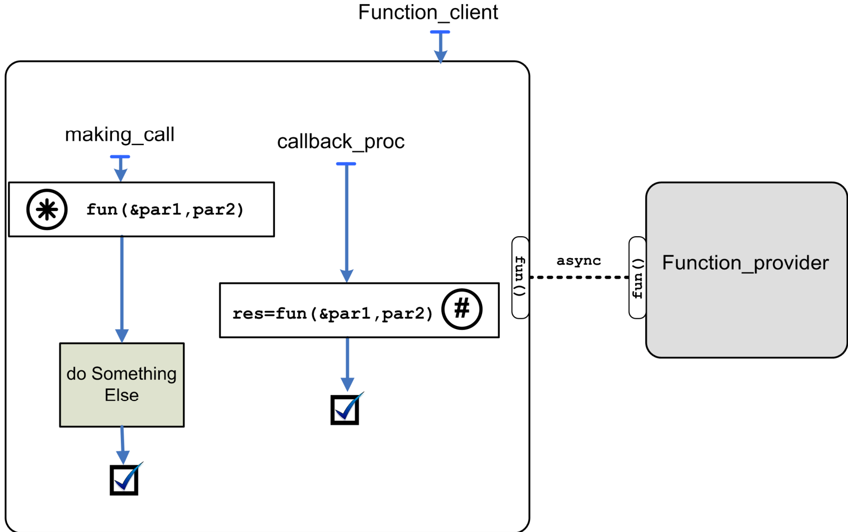


Figure 5-6 Introducing symbols for asynchronous inter-component function calls

5.2.2 Switch interaction contract

One of the more commonly used interaction contracts is the *switch* interaction contract

Design pattern – goals and ideas

The *switch* contract can dynamically switch interface connections between components implementing compatible interfaces. The switching is done depending on the value of some parameter. Here we limit our attention to the case when the parameter can have one of two predefined values. Generalization to the case when parameter can have more then two predefined values is straightforward.

Design description

In this design, the switching mechanism works by using conditional branching (SWITCH element of SystemCSP notation) and renaming operators. In that way, dynamical reconnection of ports, dependant on parameter value, is achieved.

A practical example of using the switch interaction contract is making a design that enables switching dynamically client's requests between two servers that both provide the same required service.

In the design given in Figure 5-7, the parameter, that determines the component to be used, is named `server`. When it has value equal to one, `server1` is used and when its value is equal to two, `server2` is used. `Server_proxy` is a process that accepts requests from a client, forwards it to a server, receives reply from the server and sends the result back to the client. Channel ends of `Server_proxy`

process are, depending on the value of the `server` parameter, renamed to match the names of the appropriate channels of either the `server1` process or `server2` process.

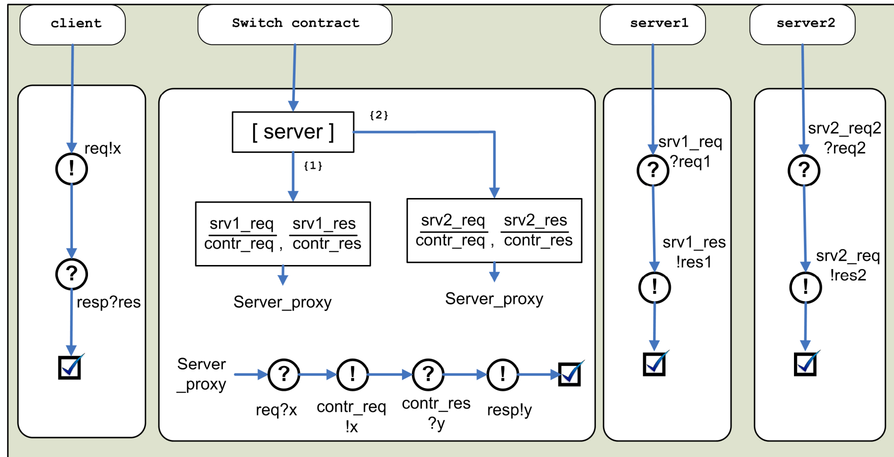


Figure 5-7 Switch interaction contract capable of data processing

Remarks

Note that the CSP feature of renaming event ends is used for dynamical switching of interconnections.

The solution in Figure 5-7 is in fact an elegant way to write down two branches (one for each parameter value and appropriate server) with sequence of events as defined in `Server_proxy` process. Implementation can be the same in both cases.

The solution from Figure 5-7 is also convenient when additional data processing should be inserted inside the switch interaction contract. A place to actually specify that additional data processing is inside the `Server_proxy` process. Function providers for this additional processing can be internally specified as fixed part of the contract definition or can be pluggable external components related to the contract via inter-component function calls.

The switch design pattern can be used for instance to implement service with varying QoS level depending on various optimization criteria (e.g. available processing power), or to switch service providers depending on working mode. In fact, it is so useful in practice of component-based development that in the Koala framework (van Ommering, 2004) switch is one of the basic language elements. Here however no special symbol is introduced for the switch interaction contract.

Recommendation is to create a symbol for the switch element (e.g. alike to one used in the Koala framework) and to provide tool support in specifying port connections and criteria that triggers switching.

5.2.3 Diversity interfaces

Design pattern – goals and ideas

Like in the Koala framework (van Ommering, 2004), that served as an inspiration for introducing this and the previous pattern, a *diversity interface* is just an interface used for a specific purpose of initializing component's parameters with values from its environment (e.g. from related interaction contracts). The required side is the component that needs to be initialized with parameters from the environment and the provided side is a component (often an interaction contract) providing values for those parameters.

Design description

In the example depicted in Figure 5-8, a parameterized calculation is performed. The example illustrates the usage of *div* interfaces to initialize the parameters of components participating in an interaction contract.

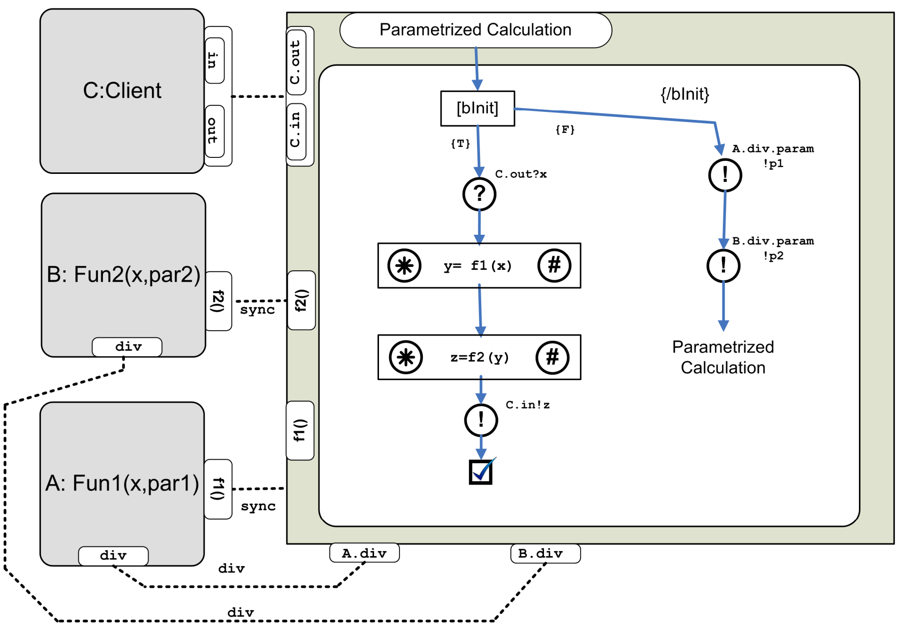


Figure 5-8 Parameterized calculation - an example of using diversity interfaces

Component A is initialized via its `div` interface with vector of parameters `par1`. When it is invoked with vector of input values `x`, the component will calculate the function `Fun1(x,par1)` and provide results. Component B does the same but with a different function `Fun2(x,par2)`. The interaction contract first initializes components A and B with vectors p1 and p2 for vector parameters `par1` and `par2` respectively, and then it can perform calculations by delegating requests of the client first to component A and then forwarding the obtained result to the component B. The result obtained from component B is sent back to the client.

Remarks

The symbol used for the diversity interface is, as in the Koala framework, the keyword `div` associated in this case with a synchronization connection relating provided and required sides of the diversity interface. Using keyword `div` as port label is specific choice made in this case and not a rule. More often ports of diversity interfaces will carry names associated with the meaning of the parameters obtained from environment. The keyword `div` on a synchronization connection clearly indicates the purpose of the interface and in that case it does increase readability.

Note that in this example, the pattern for implementing a synchronous inter-component function call is used. This has resulted in a reduced number of ports (instead of separate ports for passing parameters to function providers Fun1 and Fun2, and obtaining result from them, in both cases single function port is used for both purposes). As a result, both expressiveness and readability of the diagram were enhanced.

Note that the `bInit` flag is used to test whether the interaction contract is initialized. During the initialization phase, an *action block* is used to set `bInit` flag to value `true`. It could have been done in a dedicated *code block* element. Action blocks do increase readability of the notation by avoiding to visualize code block boxes for simple actions (i.e. setting /resetting boolean flags and setting state variables).

In fact, the assumption is that the `bInit` flag is initially set to value `false`. That could have been specified in the variable declaration floating somewhere inside the interaction contract. However, specifying lot of variables, their types and initial values does tend to clutter readability of diagrams. From that reason, declaration of variables and their initial values are assumed to be specified in property pages of the parent component/contract. Visualization of variables on diagrams is optional and is avoided in the general case.

Note that the example of Figure 5-8 would be further simplified by hiding initialization in separate view. In fact, a separate pattern can be created to achieve this – for instance replacing a conditional switch and initialization branch with an initialization process block and assuming in design pattern existence of boolean flag (as `bInit` in this example) indicating whether component is initialized or not.

5.3 Patterns related to control systems

5.3.1 Layered structure of control systems

Design pattern – goals and ideas

Control systems typically function at a number of levels (see Figure 5-9).

At the lowest level, the highest priority layer is situated. *Safety control* makes sure that functioning of the system will not endanger itself or its environment. It is especially important when embedded control systems are employed in *safety-critical* systems. *Loop control* is a hard real-time part that periodically reads inputs from sensors, calculates control signals according to the chosen control algorithms and uses the obtained values to steer the plant via actuators. *Sequence control* defines the synchronization between the involved subsystems or devices. *Supervisory control* ensures that the overall aim is achieved by using functions like monitoring and/or algorithms for parameter optimization. *User interface* is an optional layer that supports the interaction of the system with an operator (user), in the form of displaying important part of the system's state to the operator and receiving the commands from the operator.

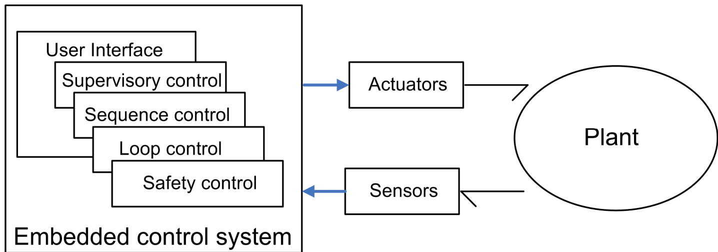


Figure 5-9 Typical control system

In fact, a complex control system (e.g. a production cell) typically contains several devices that need to cooperate. Every device can participate in any or all mentioned layers. The supervisory and sequence control layers are often event based and the control loop is time-triggered and periodic, with the period in general different from one device to another. Software components in charge of devices are either situated on the same node or distributed over several nodes.

Design description

The diagram of Figure 5-10 captures data/event dependencies between layers situated in the same device as well as between layers distributed over several or all participating devices. Because all layers are composed in parallel, two ways of clustering into components are possible: horizontal – where a centralized supervisory layer, sequence layer, control loop layer and safety control layer exist, and vertical - where parts belonging to the same device are considered to be a single component.

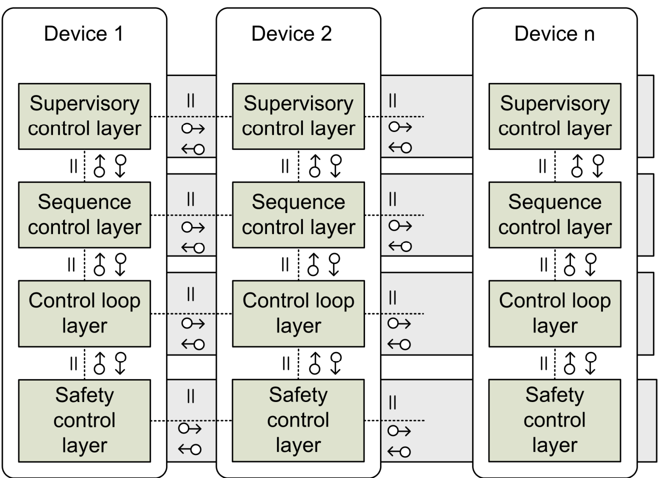


Figure 5-10 Typical layered structure of complex control system

Figure 5-11 illustrates the case where devices are treated as components and layers as interaction contracts. Every device is in this approach a component that provides ports, which can be plugged into one of the four interaction contracts: supervision, sequence control, safety, loop control and data logging (see Figure 5-11). Every contract contains logic for handling several devices and managing synchronization between them.

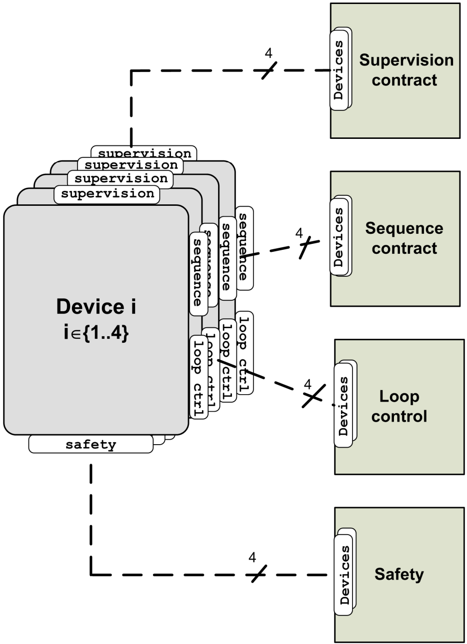


Figure 5-11 One SystemCSP design pattern for complex control systems

Internally, a device component might be organized as in Figure 5-12, with a subcomponent dedicated to the implementation of every role that maps to one of the layers supported by the device, and a subprocess dedicated to maintaining the state data of a device. In Figure 5-12, in order to put emphasis on structure and data-flow and not on control flow, the interaction oriented SystemCSP diagram is used, where communication data flows and binary compositional relationships are emphasized.

Since several processes composed in parallel need to share same data, a centralized process (`State data` in Figure 5-12) is introduced to manage access to the data that captures the *state* variables of a component. This process is in fact the *shared memory* pattern or a set of instances of the *shared variable* communication pattern. Introducing such additional process(es) allows decoupled communication between processes implementing roles of various layers. In practice, for efficiency reasons, the state data process can also be implemented as a passive object that provides the necessary synchronization.

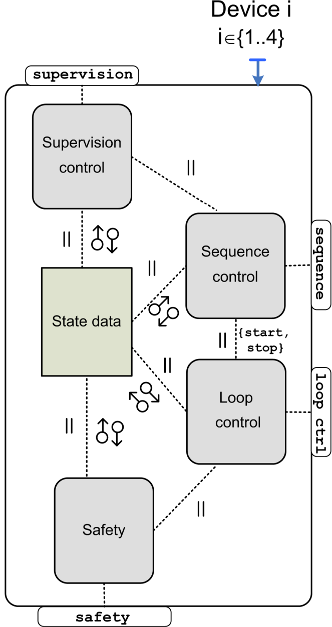


Figure 5-12 Organization of device internals

Remarks

Often, it is useful to merge management of several activities belonging to different layers into a single interaction contract (e.g. some safety measures can be in the sequence control contract or sequence and supervision layer can be merged).

A loop control contract is often implicit since there is no other dependency between control loops except for common usage of the timing, scheduling, and I/O subsystems to ensure precise in-time execution of time-triggered periodic

sampling/actuation actions. Upon performing the time triggered sampling/actuation (I/O subsystem) actions, loop-control processes of related devices are released to perform computation of the control algorithms. A loop control interaction contract can for instance perform scheduling (RM, EDF) of the involved loop control processes, check whether deadlines are missed and raise alarms to the safety or supervision interaction contract when that happens. E.g. in overload conditions, a loop control interaction contract can have a centralized policy to decrease the needed total computation time in a way that reduces performance but does not jeopardize the stability of the control system.

In the typical layered structure of a control system, layers get different priorities according to their importance and time requirements. The safety layer is of highest priority and is activated only when it is necessary to handle alarm situations. The next range of priorities is associated with loop control subcomponents. A range of priority levels might be necessary for the implementation of the scheduling method that will guarantee their execution in real-time. Sequence control is an event based layer and thus of less importance than the time-constrained loop control layer. The supervision layer is performing monitoring and optimization and is usually of least importance.

5.3.2 Supervision and monitoring layer

Design pattern – goals and ideas

The monitoring design pattern enables safe monitoring of components and systems. Every component that needs logging facilities contains a Logger subprocess that relates it to a global monitoring component. The Logger subcomponent performs data logging into a local buffer. Upon the request, data is transferred further to the Monitor component. The Monitor component collects data from several monitored processes and can reason about different safety and optimization issues and update system parameters or invoke some safety measures when needed.

Design description

Every component that has support for logging does internally contain a Logger process working in parallel with the rest of the component. A set of chosen variables can be logged at predetermined logging points in the monitored process. Logging points can be activated/deactivated due to commands from Monitor process. Logging points can be configured to log specific set of component's variables. Configuration of logging points is performed by Logger process block.

In design given in Figure 5-13, monitored components internally contain a `Logger` process block that synchronizes with other processes inside the component via a 'log' event. Processes inside the `Normal mode` process block do not synchronize among themselves on the 'log' event.

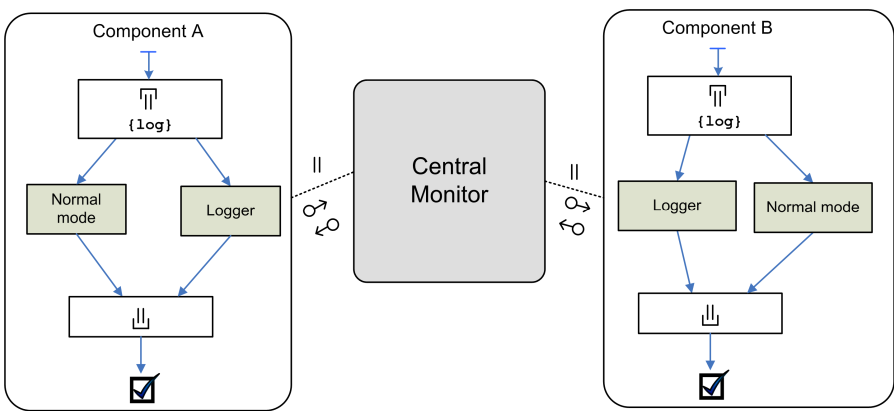


Figure 5-13 Logging and monitoring pattern

Worth noting is that in Figure 5-13 data flow is depicted in both directions: from components to monitor and from monitor to components. The `Logger` process of every component does communicate logged data to `Central monitor` component. In the other direction, the monitor can, when needed, send some commands back to the component. Commands can be related e.g. to specifying the set of component's variables to be logged at certain logging points, activating/deactivating logging points, or activating/deactivating Loggers.

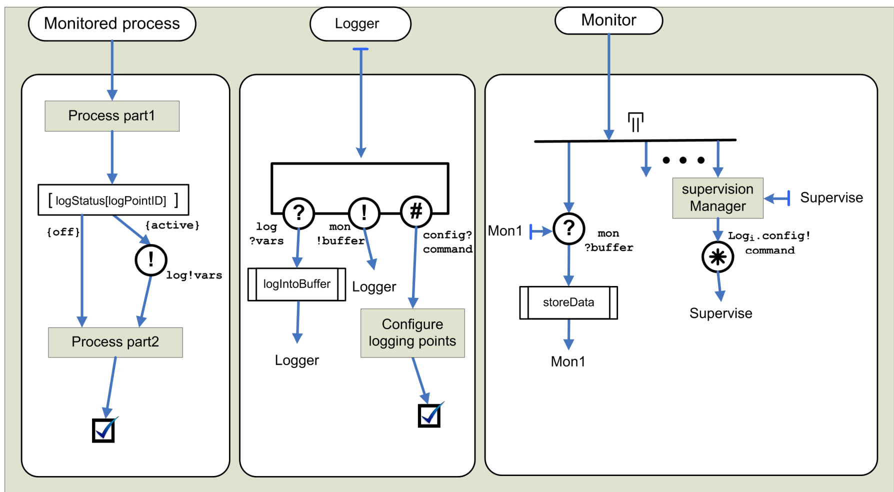


Figure 5-14 Details of monitoring interaction pattern

In Figure 5-14, the monitoring interaction contract is displayed. From the active logging point inside the monitoring process, data is sent to logger via 'log' channel. `Logger` is kind of buffer that collects data from all logging points of its component and occasionally sends bigger chunks of data to central monitor component. `Monitor` component stores received data. It also contains `supervisionManager` that can issue commands to set of loggers it has under its control.

Remarks

Such a monitoring component can be designed to plug-in higher level structures that provides facilities of SCADA (*Supevisory Control and Data Acquisition*) systems e.g. like in the OPC architecture (OPC, 2007). In such an architecture, separate components are dedicated to obtaining data access, archiving, displaying and handling alarms and similar events.

Note that the depicted interaction contract relates monitored processes with central monitor via Logger component. In fact, the interaction contract is just a set of roles that engage in some interaction. Sometimes those roles needs a contract manager to handle interaction, sometimes not. For instance, in this case one can interpret the Logger role as such contract management role or as one of the peer participants in interaction without contract manager. The choice of whether some component is promoted into an interaction contract is in fact a subjective decision depending on the context of the problem.

5.4 Fault Tolerance patterns

In fault tolerant systems, effort is made to design system that can continue providing required or degraded service despite the presence of faults in the system. A *fault* in a system can cause an *erroneous state* of some component. This error can further propagate and cause a *failure* of the expected service delivery. Faults can be *transient* and *permanent*. Fault tolerance can be seen as a process consisting of *error detection*, *error containment* (isolating error from spreading further), *error diagnosis* and *error recovery* (Avizienis, 2004).

In a SystemCSP-based system, functional error detection is naturally located in precondition and postcondition tests related to the execution of interaction contracts and subcontracts. Detecting errors in the timing relies on the *watchdog* design pattern. Upon detection of an error in an interaction contract, this contract can halt further progress of the interaction and in that way isolate the error from spreading further. An interaction contract is also a natural place to perform error diagnosis, since an interaction contract can possess more information about the current state of the interaction than the participating components in isolation can. The purpose of error recovery is to substitute an erroneous state with an error-free state. This state can be some previously saved state or degraded part of that state or it can be a new error-free state.

Forward error recovery attempts to handle errors by finding a new state from which the system can continue further operation. Usually it is based on *replication* redundancy. Replication can be done in software or in hardware (replicated specialized hardware or complete nodes or network interconnections). Forward error recovery is predictable in terms of time and memory overhead and thus often used in real-time systems (Pullum, 2001).

Backward error recovery handles erroneous states by restoring some previous error-free state. Backward error recovery is especially suited for handling errors

caused by transient faults. It has also the capability to handle unpredictable errors. The most widely used backward error recovery mechanism is *checkpointing* (Pullum, 2001).

Another useful fault tolerance design pattern is *exception handling*. It can have *termination* or *resumption* semantics. The *take-over* operator of SystemCSP covers the termination semantics. The resumption semantics upon occurrence of an exceptional situation is in our case just delegating exception handling to another part of the same process. Exceptions that cannot be handled internally (inside components) are propagated across component boundaries.

5.4.1 Recovery block

Design pattern – goals and ideas

The essence of the recovery block mechanism is providing several implementations of the same functionality, possibly with different QoS (Quality of Service) levels. If the results obtained by executing first block fails to pass the acceptance test, then the next block in line is performed, and so on. If all recovery blocks fail the associated acceptance tests, recovery block fails.

Design description

Figure 5-15 illustrates the usage of a *recovery block* fault tolerance mechanism in implementation of a service provider. After obtaining data via an 'input' channel end, data is processed by block F(x). In case when results pass the acceptance test the result is returned to the client. If however the results did not qualify to pass the acceptance test, the input vector is forwarded to block G(x). If new result fails the acceptance test, the 'error' event is used to notify the client that the recovery block has failed to provide the required service.

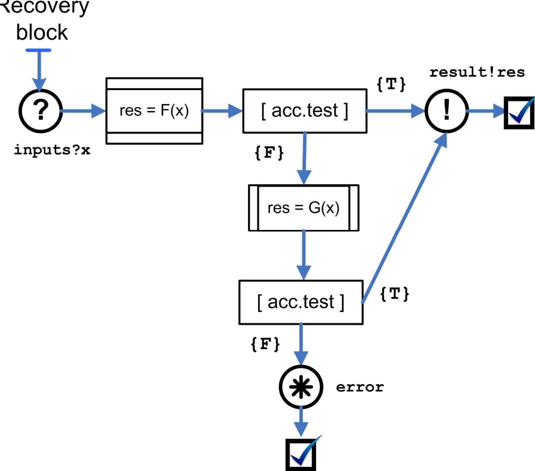


Figure 5-15 An example of using recovery block

In this example, the recovery block mechanism is implemented, but it is not generalized into form of interaction contract. It is possible to externalize the implementation F(x) and G(x) out of the component where the interaction takes place and implement them as service providers. In that way, a generic recovery block interaction contract can be created as illustrated in Figure 5-16. Participants in the interaction specified are function providers F(x) and G(x) and user component.

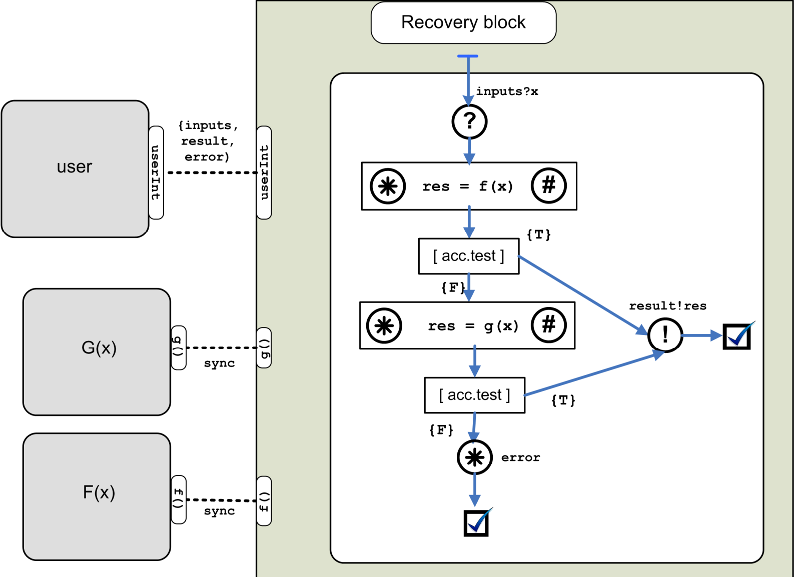


Figure 5-16 Recovery block interaction contract

Remarks

In (Yeung et al., 1998), a distributed recovery block mechanism is described in CSP. The example given in Figure 5-15 is in fact a visualization of a non-distributed version from that paper.

Again, usage of the inter-component function call pattern did simplify the diagram by replacing in two occasions a pair of channel-ends needed for inter-component function call with the appropriate symbols. Such symbols contain function call declarations, which further increase readability compared to specifying just a pair of channel communications.

5.4.2 Watchdog design pattern

Design pattern – goals and ideas

In section 4.1.3, a design for watchdog interaction contract is introduced in order to provide watchdog services used in the implementation of the timeout operator and the timed interrupt operator. Here (see Figure 5-17) a more elaborate version of the

watchdog interaction contract is presented. In addition, a generic role of the watchdog user is specified and two different implementations of the role are provided: one that uses watchdog in one-shot mode and the other one that use it in periodic mode. In periodic mode, watchdog is in charge of initiating both the *precisely periodic* timeout event ('period' event in Figure 5-17), and the *watchdog deadline* timeout event ('timeout' event in Figure 5-17).

Design description

The watchdog interaction contract, as designed in Figure 5-17, defines a contract manager named `Watchdog` and roles for `Timer` and `WD_user` processes. The `Timer` process allows its users to subscribe to a timeout service ('subscribe' event) and to cancel it ('timer.cancel' event). When the timeout interval expires for its user, it will activate it using the 'timer.wakeup' event. In this case, direct user of `Timer` process is `watchdog` process, and indirect user is `watchdog user`. In one-shot mode, watchdog and its user interact in a way described in section 4.1.3.

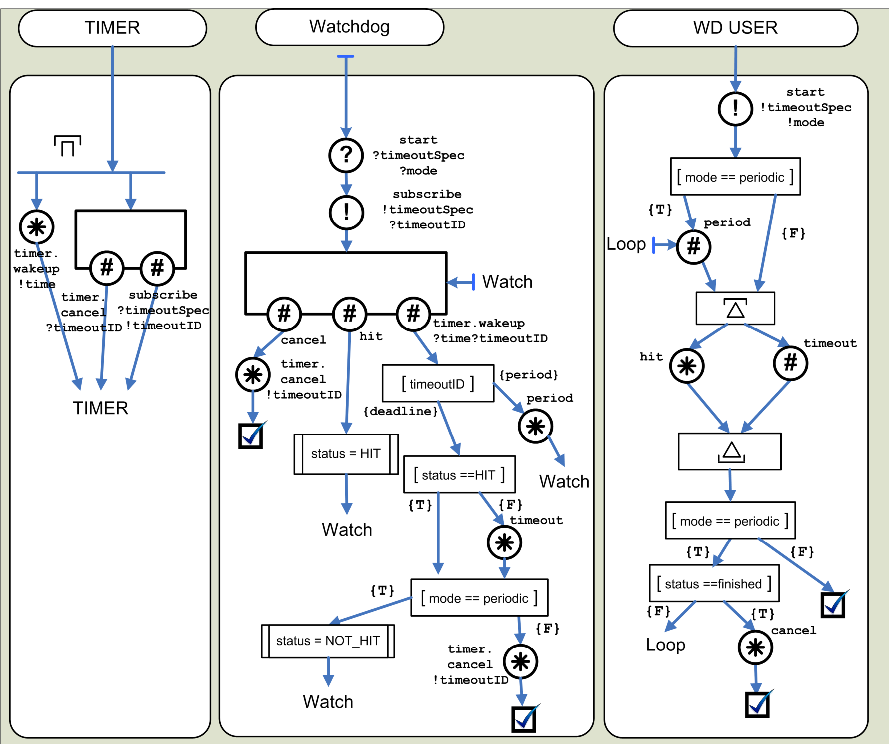


Figure 5-17 Watchdog interaction contract with specified roles of participating components

In periodic mode, the user initiates the watchdog by specifying the periodic mode and time values for period and relative deadline. In every period, the user first waits for precisely periodic timeout ('period' event), then it will attempt to perform its normal mode activity before the deadline expires ('timeout' event). If it succeeds, it will disarm the watchdog via the 'hit' event. If not, its further

execution will be interrupted with a ‘timeout’ event. Compared to the watchdog presented in Section 4.1.3, this one when it works in periodic mode needs to distinguish between two kinds of ‘timer.wakeup’ events – the one due to period and the one due to deadline. TimeoutID is used for that purpose. In case timeoutID evaluates to meaning of period, the ‘period’ event is initiated. In case the meaning is ‘deadline’, it will check whether ‘hit’ event did take place already in that period (that is whether the user has finished its execution for that period. If ‘hit’ event did take place, everything is ok, and the status flag that indicates whether ‘hit’ event did took place can be reset to be ready for a next period. If however the ‘hit’ event did not take place, ‘timeout’ event is initiated that will cause the normal mode execution of the watchdog user process to be aborted.

Figure 5-18 introduces two different implementations of watchdog users, both refinements in CSP sense of the role of the watchdog user specified in the interaction contract. The left hand side is the implementation that uses the watchdog interaction contract in a single-shot manner and on the right-hand side is the implementation that uses the watchdog interaction contract in periodic manner.

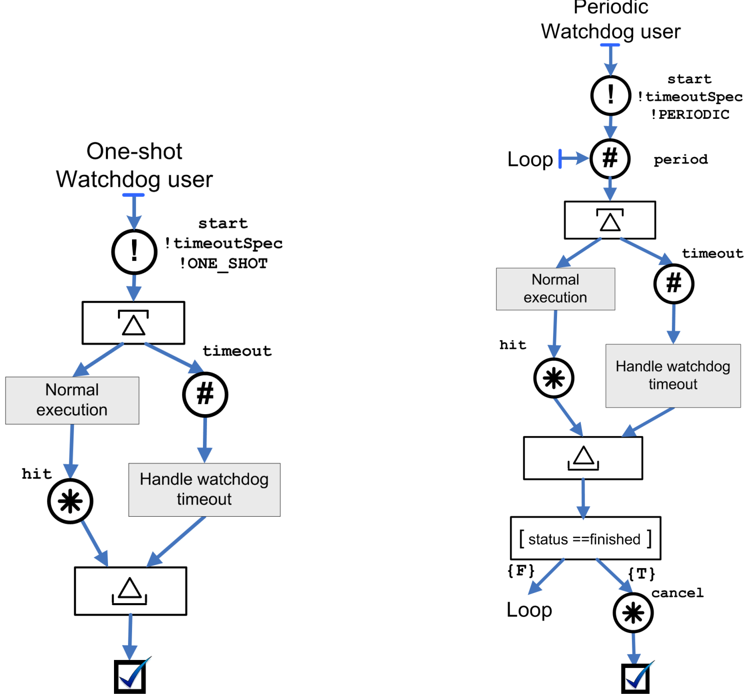


Figure 5-18 One-shot and periodic versions of the watchdog

Remarks

Figure 5-19 presents symbols used for the abbreviated representation of using the watchdog. The one-shot watchdog users do guard a process (`Normal execution`) for a watchdog timeout and are in that sense equivalent to the usage of the timed

interrupt operator (see section 4.1.3). Hence, the used symbol is the same in fact.

Figure 5-19 Symbols used for watchdog usage

The left most diagram is, as in section 4.1.3, based on a fork and join symbol of *timed interrupt* operator. In the middle diagram, the symbol is further abbreviated by showing only the process `Normal execution`. This makes a lot of sense since most of the time designers/users are interested in the normal control flow and not in handling watchdog timeouts. Hiding the process that handles watchdog timeout enhances the readability and scalability of the design. The third figure is used to introduce symbol for using periodic watchdog. T1 and T2 are time parameters that stand for: the time interval specified for period (T1), and the time interval specified for the watchdog deadline (T2).

Note that the internal choice operator is used to specify that TIMER process will internally in some way decide whether to accept the timeout service subscription/canceling or to initiate the 'timer.wakeup' event. For this role, the complete description of how the choice is made is not relevant. Thus, this is an example of how an internal choice operator can be used to abstract away from details of the inner workings that are irrelevant in the context of usage.

The difference between specification of a role and the implementation is that in the specification of the role only the event interactions important for the interaction contract are present. If one compares the role of watchdog users as given in Figure 5-17, and the two implementations of watchdog users as given in Figure 5-18, both implementations, although quite different, are trace refinements of the same role. However, these implementations do contain process blocks (`Normal execution` and `Handle watchdog timeout`) that might internally contain some event interactions irrelevant for this contract. Those process blocks are simply left out in the role specification because they do not contribute to the set of possible traces of the contract specification. The set of traces capturing the interaction specified in contract is based only on the events relevant for the contract.

5.4.3 Replica Management

Design pattern – goals and ideas

In Figure 5-20, an interaction diagram is displayed that relates a client component with a replicated server components via a replicaMgr contract.

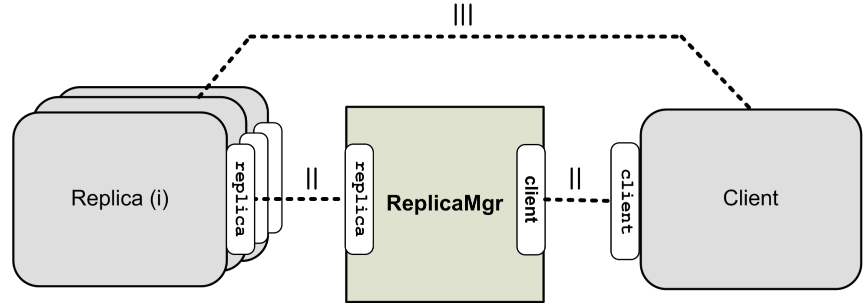


Figure 5-20 Replica management - interaction diagram

All code related to managing the replication is situated in the ReplicaMgr contract, which enables reusing the same components in different replication configurations. Replicas can be on same node or on different nodes, can be identical or based on different designs (N-version programming). The ReplicaMgr can be on the same node as some of the replicas or on a separate node. In this section, SystemCSP based designs are provided that specify *hot-standby*, *cold-standby* and *majority voting* types of ReplicaMgr.

In the 'hot-standby' design pattern, upon receiving a request from a client, all replicas are activated, but only the first available result is actually used. After one of the replicas comes up with results, further execution of other replicas is aborted.

In the *cold-standby* design pattern, replicas are activated one by one. If the first one fails to deliver the result, the replica next in line is activated.

In the 'majority voting' design pattern, all replicas are active in parallel. Results delivered by replicas are then compared, and the decision about the result is made using voting.

Design description

In this section, designs will be introduced for the three mentioned types of replication management. In all designs, replicas are servers named r1, r2 and r3, and each replica has 'request' and 'res' channels.

In the 'hot-standby' design pattern (see Figure 5-21), the `ReplicaMgr` first receives a request from a client ('request' event) and then it will distribute the request (events 'r_i.request') in parallel to all involved replicas. In order to protect the interaction contract from being deadlocked by waiting on synchronization with failed replicas, sending the request to every replica is guarded using the watchdog design pattern.

The replicas work in parallel, and the interaction manager waits for a limited time (again the watchdog pattern is used) for one of the replicas to produce its results ('r_i.reply' events). This kind of selective waiting is realized using an *external choice* element. In case one of the replicas comes up with the result, the process starting with the associated event is chosen. After the occurrence of 'r_i.reply event, further execution of the other two replicas is aborted using 'r_i.abort' events. Again, watchdog protection from failed replicas is necessary because 'abort' event is event like any other and can not be performed if other side does not exist.

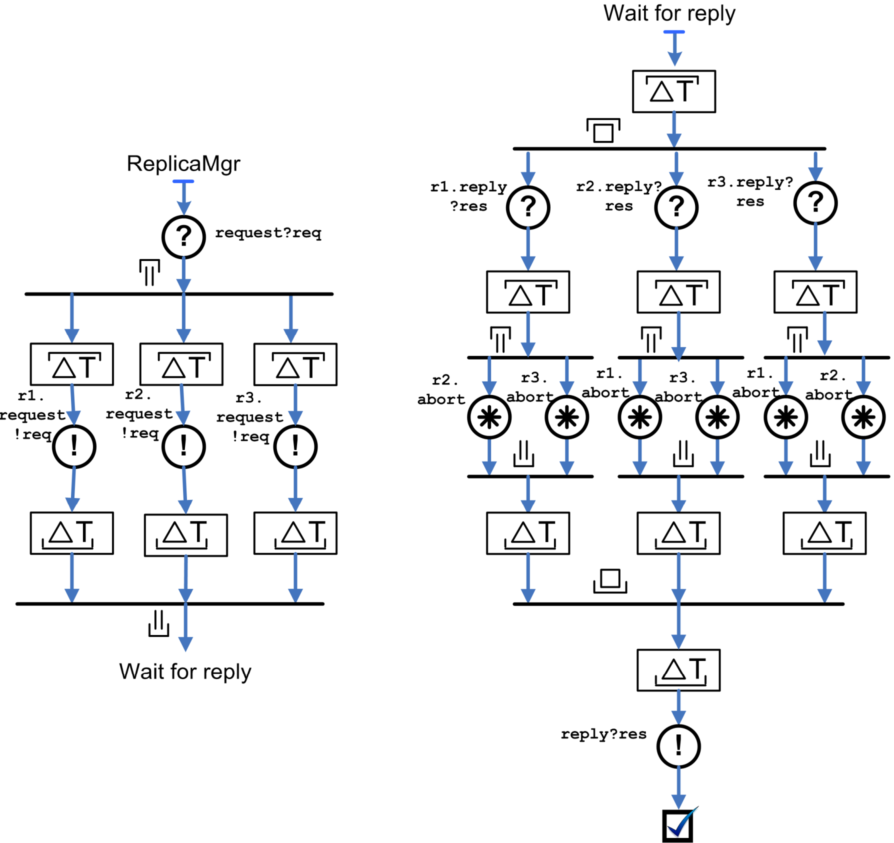


Figure 5-21 Hot-standby

In the *cold-standby* design pattern, given in Figure 5-22, a request is first sent (via 'r1.request' channel) to the first replica (`r1`). If the first replica does not accept the request within the predefined time (notice the use of the watchdog pattern), the request is forwarded to the second replica (process label `Try2ndRep` is followed), and if this one also fails to accept it, the request is forwarded further to the next replica in the chain.

After the request is accepted by one of the replicas (e.g. `r1` accepted event 'r1.request' within a given time interval), the ReplicaMgr waits for a reply ('r1.reply' in case replica `r1` is active) for a predefined time interval. If the reply

does not arrive (note the usage of the watchdog design pattern) within predefined time interval, a request is sent to the next replica in the chain (e.g. if replica `r1` failed, then process label `Try2ndRep` is followed). If the replica replies, then the result is forwarded ('result' event) to the client. If no replica in the chain is able to provide the result, then the 'error' event is initiated.

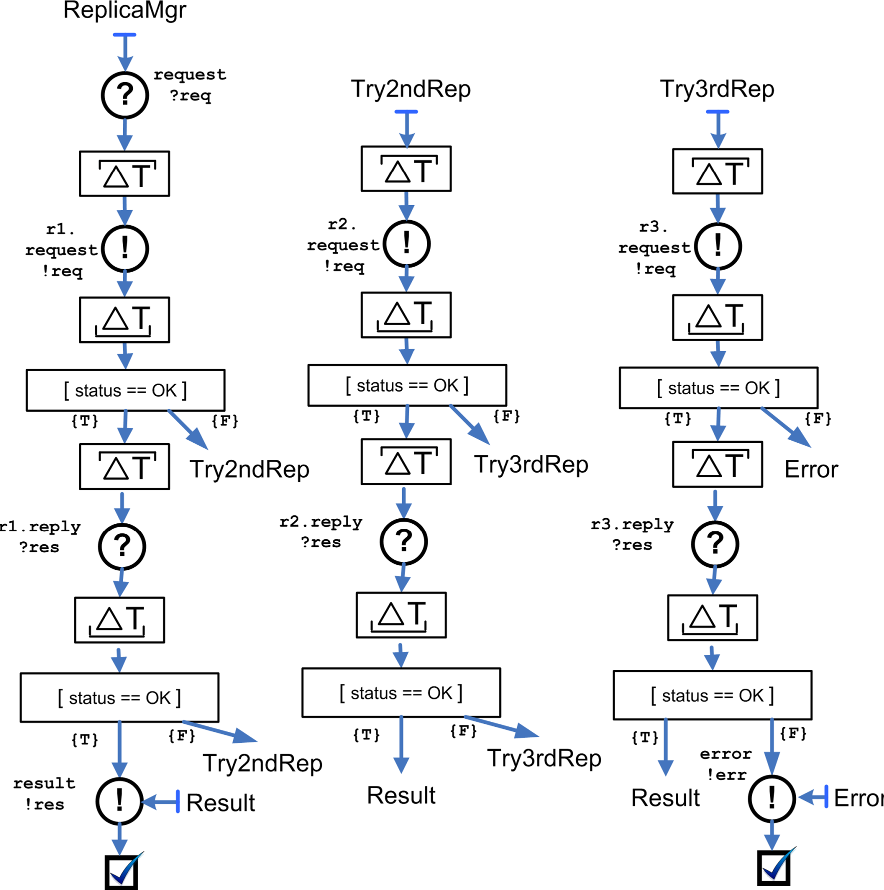


Figure 5-22 Cold-standby

In the 'majority voting' design pattern, the request is sent in parallel to all replicas. The sequence of sending the request to a replica and obtaining the reply from it, is guarded by the watchdog pattern. In that way, a failing replica cannot block the ReplicaMgr process. Obtained results and status flags are used in the *majority voting* process block to make an agreement about the correct result. In case when it is impossible to deduce a result, the `error` event is initiated.

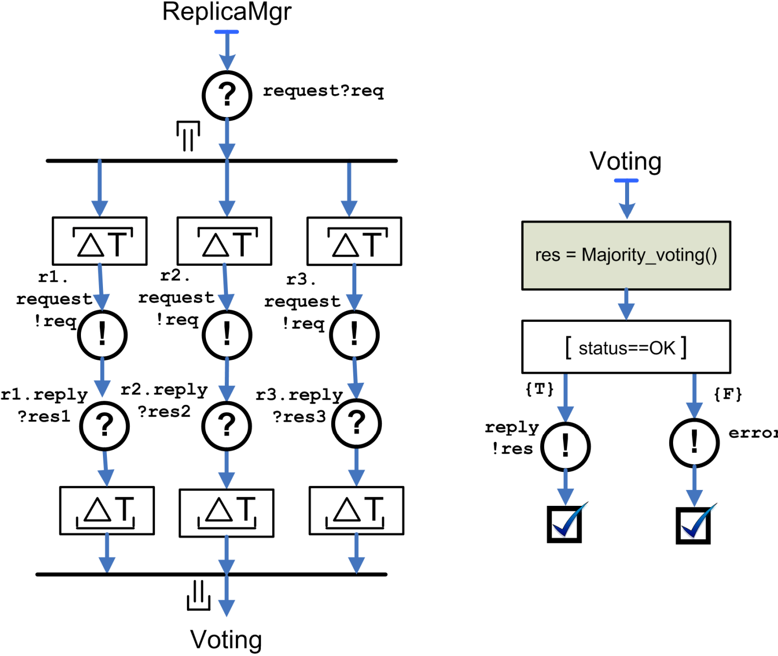


Figure 5-23 Majority voting

Remarks

'Hot-standby' replica management pattern is convenient to use when there is enough processing power available, and the time constraints are such that there is no time to invoke replicas in sequence only in the case when previous one failed.

In the *cold-standby* design pattern, given in Figure 5-22 only one replica is used at any time. This pattern is recommended in the case when CPU resources are scarce and the associated time constraints are relaxed enough to allow for handling replica failures in sequence.

In the 'majority voting' design pattern, all replicas perform (unless they fail) the complete computation. Thus, the average CPU usage is larger compared to other two patterns. This pattern is recommended when replicas are most likely to fail by producing incorrect results.

In case when the replicas are invoked periodically and contain state (e.g. if replicas are controller implementations), the state of the replica that produced the result should be communicated together with result to the ReplicaMgr. In the next iteration, the state from the previous iteration should be communicated to all replicas. This approach will prevent internal states of replicas to drift away.

Note that usage of watchdog design pattern makes specification of the `replicaMgr` interaction contracts much simpler. Creating symbols for design patterns in order to abbreviate their description does in fact extend the vocabulary of the language. Using symbolic abbreviations of design patterns increases, in the

same time, readability, scalability and expressiveness.

Normally, process labels are used to mark logically separate parts of control flow. However, in Figure 5-21, process label `WaitForReply` is inserted to visually split diagram into two parts. This illustrates the way in which the introduction of process labels does enhance scalability. In the general case, usage of process labels enhances expressiveness by allowing jumps to named points in control flow. Usage of process labels also increases readability, because it substitutes lines connecting recursion points to process entry points.

5.4.4 Exception Handling

Design pattern – goals and ideas

The termination model of exception handling (see the take-over operator in section 3.2.1) is not always convenient because it destructs all the work performed in the aborted process. Often, it is more desirable to first attempt recovery and if possible to avoid the need to abort the process. It is possible to construct a design pattern that will perform exception handling with the *resumption* semantics. Idea is to separate the part that attempts to handle exception from the rest of the design using a special process block. In that way, a visual separation of normal mode and exception handling mode (EHM) is obtained. If an invalid state is observed in normal mode, further control flow may be designed to lead to the EHM via a recursion label. When the exception is handled a process recursion label can be used to bring control flow back to some resumption point in normal mode.

Design description

Figure 5-24 depicts a case where EHMs of the contract and involved roles/components can interact and agree on ways to handle the exceptional situation. `Role1` of `Component1` offers both resumption and termination methods for exception handling. Resumption method relies on a jump from `normal mode` to the `EHM` part of the process that attempts to handle the exception. The `normal mode` and `EHM` blocks are two visually separated parts of the same process block. One can also understand this as analogue to state-diagram composed of two state-diagrams, where transitions between diagrams can lead in both directions.

The termination method of exception handling is performed by the take-over operator and the `Abort procedure` block. The contract, named `Contract1` in this case, is a single process that is also visually divided into two process blocks: normal mode and EHM.

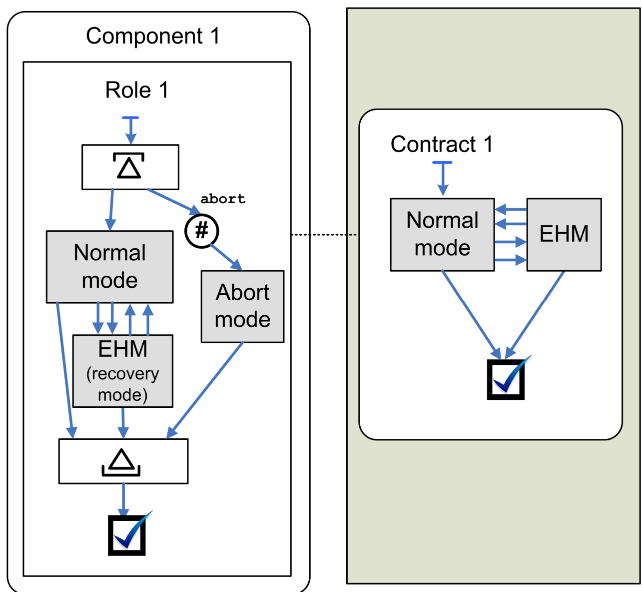


Figure 5-24 Cooperation of contract and component EHM layers

Remarks

The power of the configuration given in Figure 5-24, comes from the fact that the interaction contract has additional knowledge about the current state of the interaction and can also obtain/maintain info on the status of the participating components. In that way, the interaction contract can pinpoint more precisely on possible causes of the exceptional situation and propose, to the participating components, appropriate ways to handle it.

If EHM cannot handle the exception, it can trigger an 'abort' event that will trigger the termination model of exception handling.

5.4.5 Checkpointing

Design pattern – goals and ideas

Checkpointing (see Figure 5-25) is a backward recovery mechanism that relies on correcting an erroneous state by rolling back to some previous correct state.

Design description

In example given in Figure 5-25, two recovery lines are defined (marked with labels `recovery line 1` and `recovery line 2` in the design), splitting the participating components and interaction contract into two phases. After an exceptional situation is detected inside Phase 1 or Phase 2 of the contract, control is given to the EHM part of the interaction contract. EHM will use the events belonging to the alphabet of every participating component to transmit exception

information to them. As a consequence, components will quit executing the current phase and go into the EHM part of their process definition. From there, it will use a dedicated channel ('$role_i$') to communicate its status to the EHM part of the contract. The Contract will wait for a predefined period of time to obtain the status information of all involved components. After that, it will perform analysis and establish whether the interaction should be reverted to some recovery line or aborted. Its decision will be communicated (either event 'r_i.abort' or event 'r_i.rcvLn') to the participating components. Component will, depending on the communicated event, either be aborted or jump to the recovery line. The used mechanism makes sure that all involved components are rolled-back to the same phase.

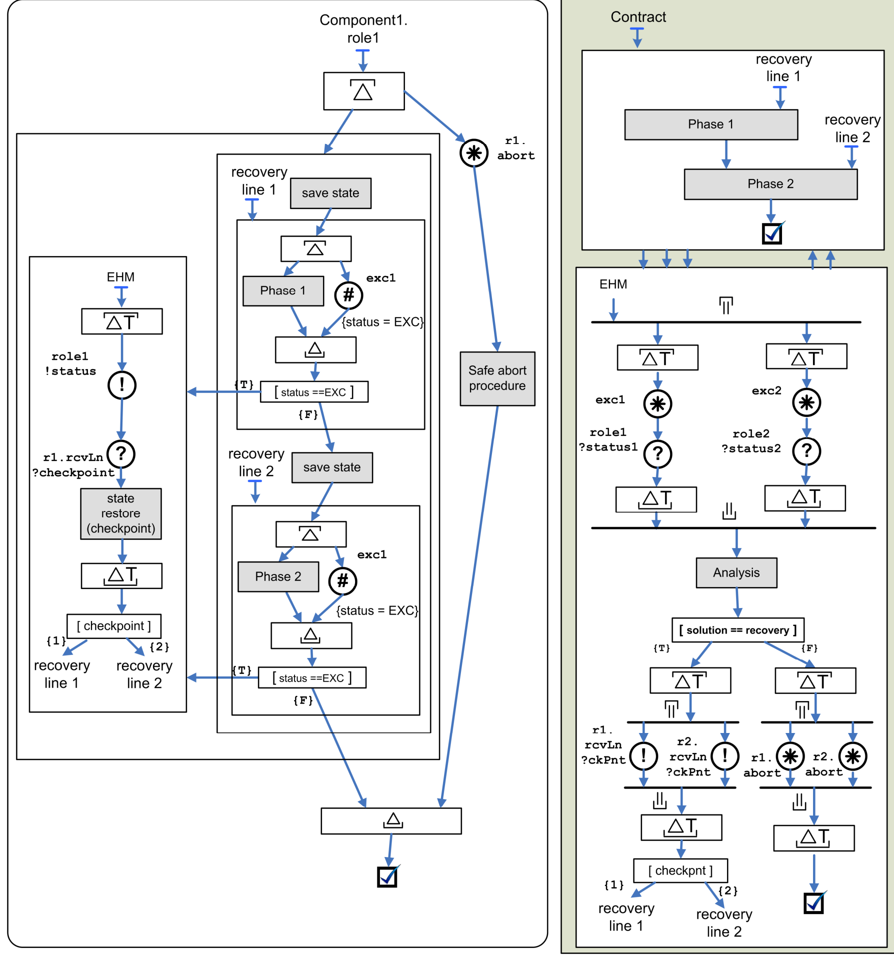


Figure 5-25 Checkpointing

Remarks

When interaction of several concurrently executing processes is guarded from faults using a checkpointing mechanism, a *domino effect* can occur. In such a *domino effect*, one process causes another process to rollback, the other causes a third one to roll-back, and so on, which can result in rolling-back the first process even further and so on. This makes asynchronous checkpointing of interaction unsuitable for real-time systems where execution must be predictable in the sense of time and memory requirements. The proposed design pattern relies on interaction contract as a manager that keeps the roll-back process of involved components synchronized. In this way, the "domino-effect" is avoided and as a consequence the execution is more predictable in the sense of time and memory requirements.

5.5 Conclusions

In this chapter, SystemCSP design methodology is illustrated by designing reusable interaction contracts. The design patterns presented here, illustrate that SystemCSP is a graphical notation that seems to be able to provide, in addition to the formal verification capabilities, intuitive and readable modeling of interactions in concurrent systems.

The patterns defined here are intended to be reused as basic building blocks in the design process. Creating symbols for design patterns in order to abbreviate their description does in fact extend the vocabulary of the language. Using symbolic abbreviations of design patterns increases, in the same time, readability, scalability and expressiveness.

Interaction contracts are as reusable as components. Defining patterns in the shape of the reusable interaction contracts yields a possibility to verify contracts, as abstract entities, in isolation from concrete systems. Instances of interaction contracts can then be used in actual systems, eliminating in that way the need to make new design of interaction among components in ad-hoc manner every time. Patterns are especially useful for the interactions that are used very often.

If a standard set of interaction contracts would be created for the most common patterns of interaction among components, and if a set of standard language symbols are created for every one of them, software designers would be able to significantly raise the abstraction level in specifying and communicating designs. The example illustrating this is the usage of inter-component function call patterns and watchdog pattern in specification of the several other patterns presented in this chapter.

Some of the patterns are concepts often used in practice of software development, but rarely precisely defined and formalized. In that sense, since SystemCSP is directly translatable to CSP, this chapter also contributes to formalizing those patterns.

6 Production cell setup

If you want to learn to swim, jump into the water. On dry land, no frame of mind is ever going to help you.

Bruce Lee

The primary target application area of the SystemCSP is designing interactions in complex control systems. Purpose of this chapter is to test whether SystemCSP is a convenient way of capturing designs in control systems. The methodology used here is to design software for a complex control setup. This test case can be considered as a first step towards applying SystemCSP in real industrial settings.

The used setup is alike to realistic industrial applications regarding the number of controlled devices, complexity of interactions and performance. A *production cell* setup is one of the test cases used for illustrating various formal methods and design methodologies (Burns, 1998; Zorzo et al., 1999). Normally, in research communities, the production cell setup is used on the level of simulations. In this project, however, the decision was to really build the setup (see Figure 6-1).

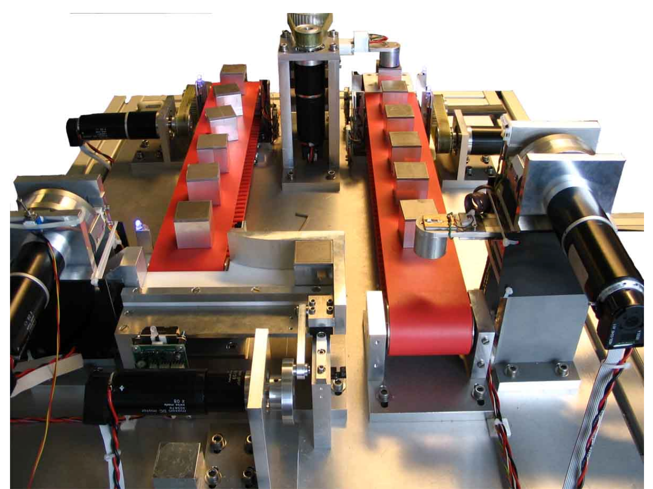


Figure 6-1 Photo of the Production Cell Setup

The advantage of having a real setup over just a simulation is that a real setup does in general raise important issues that were neglected in modeling and simulation phase. In a way, a real setup is the litmus test of the taken approach. In this case, the implementation of the setup started before the work on the SystemCSP graphical language. The setup was envisioned to be used for tests concerning the distribution, allocation optimization, testing fault-tolerance mechanisms and the validation of a simulation framework. However, during the design of software to control the setup, it was discovered that GML/CT approach was not really suitable for all aspects of the design. As a result, the SystemCSP approach emerged.

The focus of this chapter is validation of SystemCSP as a way to design software for complex control setups. Section 6.1 briefly describes the production cell setup

implemented in the context of this project, as the MSc assignment of van den Berg (2006). Section 6.2 introduces some other ways to design software that were used for controlling this setup. Section 6.3 presents a SystemCSP based design of the software that controls the setup. Section 6.4 is again SystemCSP design, but this time based on interaction contracts managing the synchronization among devices. Section 6.5 summarizes this chapter in a few sentences.

6.1 Production cell setup

A production cell setup in general has the following elements:

- a processing device,
- an input belt that carries the raw material to the processing device,
- an output belt that carries processed products to the storage place located at its end,
- a robotic device that transports raw material from input belt to the processing device, and
- a robotic device that extracts a product out of the processing device and places it on the output belt.

Our production cell setup is inspired by industrial molding machines as produced by the industrial partner in this project, Stork Plastics. Figure 6-2 depicts the mechanical design of the setup as captured in SolidWorks, a mechanical design tool.

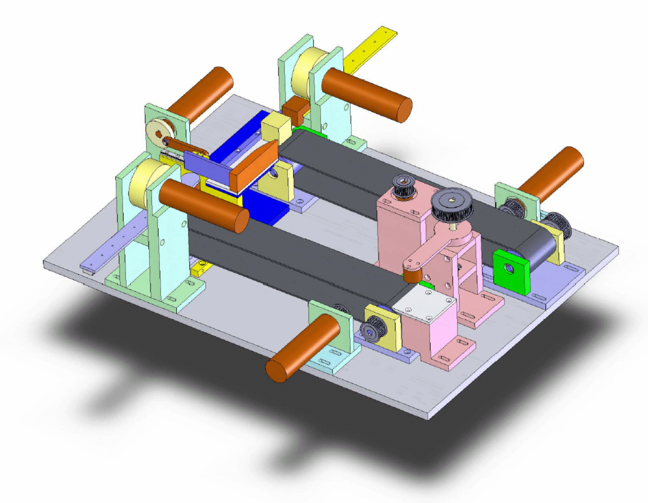


Figure 6-2 SolidWorks model of the Production Cell

A processing device inside our Production Cell setup is named the 'molding machine' or 'molder'. However the molding machine of this setup does not have anything to do with molding. The only similarity is that the processing device used in our setup does close a product exit door before it accepts raw material (metal blocks in our case) and then after some time opens the product exit door to allow the extraction device to pick-up the processed product (i.e. unchanged metal blocks in this case).

Purpose and properties

The setup is useful for testing and comparing different design choices regarding concurrency structure, distribution, fault tolerance, applied control algorithms, etc.

The setup consists of six devices that are capable to operate in parallel and that need to cooperate and synchronize their activities. Improper synchronization is visually observable from the behavior of the system.

The control of the setup can be distributed in an electronically reconfigurable way over several controllers that can be located on different nodes that are connected, for instance, via fieldbus connections.

The setup is made suitable for testing different fault-tolerant mechanisms, for instance, handling failures of a node or a network. If one controller is busy or fails at a certain moment of time, another controller should be able to take over its tasks almost instantly. In case when replicated controllers are located on different nodes, this requires switching of all I/O from one node to another. Van den Berg (2006) designed special hardware to allow this. This creates preconditions for experimenting with replicas distributed on different nodes and testing the response of the system on the node and/or network failures.

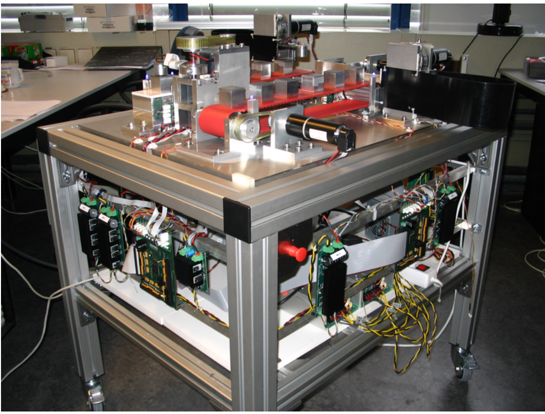


Figure 6-3 Setup is fixed to a cart

Special safety measures are taken in the mechanical design such that failures of the control system leave the underlying mechanical parts undamaged, allowing many experiments to be performed without degradation or change in performance and making the setup suitable for demonstration purposes. The complete setup is fixed

on a handcart (see Figure 6-3) allowing relatively easy transportation and out-of-the-lab demonstrations.

Detailed description

The setup consists of 6 mechanical devices that operate concurrently and need to synchronize their activities. Each mechanical device is actually an actuator performing one-dimensional movements. Devices are named according to their 'roles' in the system: feeder, molder, extraction device, extraction belt, rotation device and feeder belt (see Figure 6-4).

First, the molder is fed with raw material. The feeder is a device that pushes a block representing the raw material into the molder. This block can be pushed into the molder only after the door of the molder device is closed. The molding process is mimicked by keeping the door of the molder closed for a predefined time interval. After 'molding' is finished, the door of the molding device opens and the robotic arm of the extraction device is allowed to enter inside the molder and to pick-up the metal block (representing a product of the molding process) using its electromagnet. The robotic arm then uses a translational movement to deliver the box to the extraction belt. The extraction belt transports blocks from the extraction device to the delivery point at the end of the belt. The belt and the related one-place storage at its end introduce a flexible transport delay by providing temporary storage space, allowing in that way more flexible synchronization between machines.

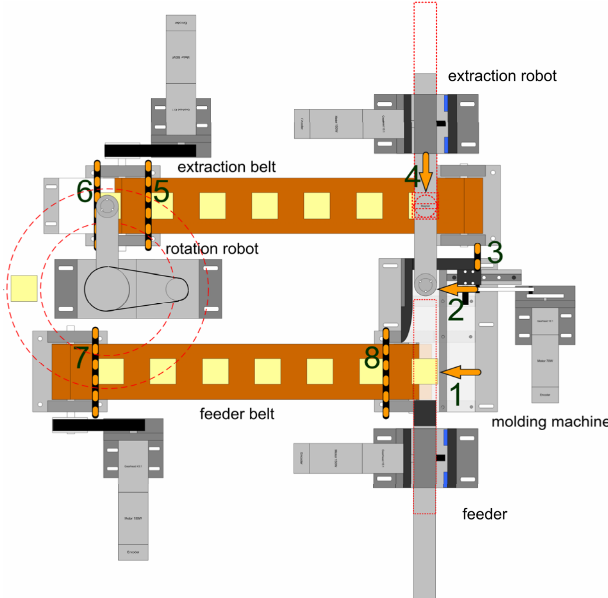


Figure 6-4 Scheme of the production cell

The one-place storage at the end of the extraction belt is the place from which blocks are reinserted into the system. The rotation device uses an electromagnet to

pick-up blocks and uses a rotational movement to transfer them from the storage place to the feeder belt. The feeder belt transports the blocks to the feeder. In that way, blocks circle through the system, making the setup convenient for demonstration and testing purposes.

In Figure 6-4, the positions of sensors are marked with numbers. The exact meaning of the sensors is given in Table 2.

Table 2 Measurement and consequence of each sensor

S#	Detection status	Consequence
S1	Block or feeder-arm detected	No block input possible from feeder belt
S2	Block in molder detected	No block input possible from feeder
S3	Molder door closed	No block extraction possible from molding machine
S4	Input space on extraction belt not empty	Completing extraction is not possible
S5	Block detected near the end of the extraction belt	The exact location of the block is known; a fixed end-motion profile can be initiated.
S6	Block detected at the drop-off point	No block input possible from extraction belt
S7	Input space on feeder belt not empty	Completing rotation is not possible
S8	Block detected near the end of the feeder belt	The exact location of the block is known; a fixed end-motion profile can be initiated.

Structural deadlock

The introduction of the rotational unit does introduce a cycle in the system structure. This kind of cycle is of course not present in a real production cell. However, in our case it is useful, because it creates a structural deadlock condition and thus creates an interesting research problem.

If the belts are used as buffers and moved one step at the time, we can assume that each belt is divided into N buffer spaces. Then a complete system will in fact represent a circular buffer with a size equal to 2N+3 spaces (2 belts with N spaces, plus a space in molder, feeder and storage), as illustrated on the left-hand side of Figure 6-5. The system cannot work for 2N+ 3 or more boxes, because it would mean that the circular buffer is full. This would be a deadlock situation caused by the structure of the system.

If the actions of the extraction device and the rotation device are split into parts allowing those devices to be used as buffers as well, then the system has $2N+5$ buffer spaces. In this case, system is in a deadlock situation when trying to use $2N+5$ or more blocks, as illustrated on the right-hand side of Figure 6-5.

For our Production cell setup, the size of the belts is chosen such that N is set to 6. Thus, deadlock takes place for 15 or more boxes in the first case and 17 or more boxes in the second case.

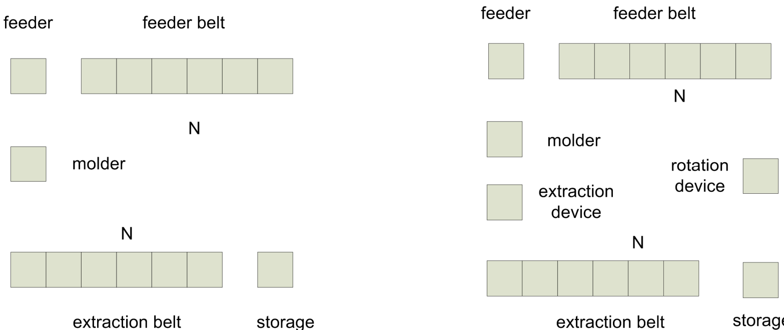


Figure 6-5 Production cell as a circular buffer with 2N+3 and 2N+5 buffer places

Due to moving the belt for one position only after there is a new block at its entrance, the minimal number of used blocks is equal to N. Less than N blocks will not fill the buffer of the belt and will therefore not be transferred further.

An alternative way of controlling the setup is to keep the belts running whenever that is possible. A belt then has to be stopped when both the storage position (located after the end of the belt) and the position on the belt immediately before that storage are occupied. In addition, depending on the product, it might be necessary to stop the belts prior to the delivery of a product to the belt.

In the case when belts are kept running whenever that is possible, it can happen that there is a block at the beginning and a block at the end of belt, while the rest of the belt is empty. If the same situation happens at the same time on the other belt and while all other buffer spaces in the devices are also occupied (3 or 5 as in the case of the step-by-step controller), then a deadlock situation occurs. Thus, deadlock can in this case happen already with 4+3 or 4+5 that is 7 or 9 boxes.

6.2 Other ways to design software

After the setup was implemented, several different approaches to software design were tested on the setup. Section 6.2.1 describes a time-based approach taken by the van den Berg (2006) to test whether the setup works as expected. Section 6.2.2 describes the follow-up work of Maljaars (2006) on controlling the setup using the GML/CT combination. Section 6.2.3 depicts the software design implemented in the scope of the ViewCorrect project by Huang and Groothuis (Huang et al., 2007). This approach is based on the POOSL modeling language.

The purpose of briefly illustrating those approaches is to obtain insight in points where SystemCSP can offer better design support than those other relevant approaches.

6.2.1 Time-table based approach

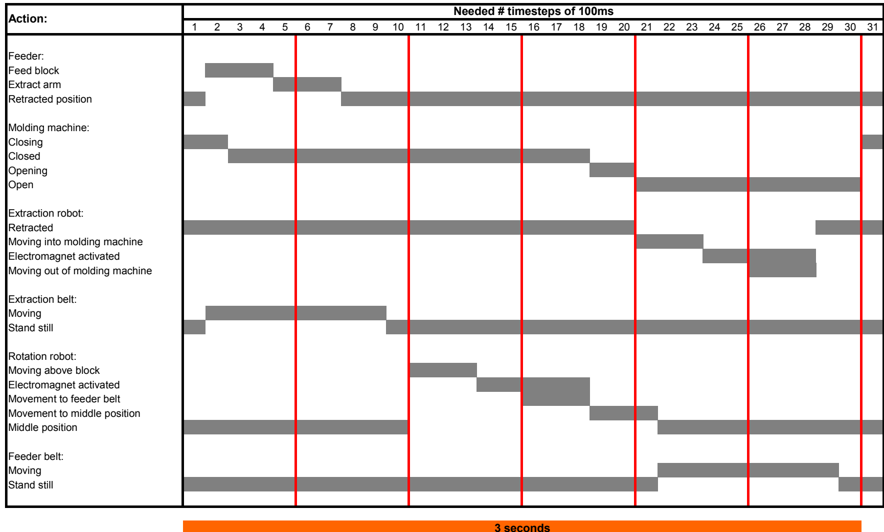


Figure 6-6 Basic timing scheme (van den Berg , 2006)

Van den Berg (2006) constructed the timing scheme given in Figure 6-6 as a trade-off between achievable actuator characteristics, a desired complete production cycle per block of about 3 seconds and taking precedence constraints into account. This timing scheme is divided into 30 time steps of 100ms each. Its main purpose was to derive achievable motion profiles and determine the characteristics of the actuators using simulation models. Van den Berg (2006), used it also as the basis for the software implementation.

The time-table based approach is often used in industry.

6.2.2 GML/CT library design

In the MSc thesis of Maljaars (2006), software to run the setup was designed using GML/CT approach. The design was entered in gCSP tool and the code generation facilities of the tool were used to generate source code.

Figure 6-7 depicts the system-level block diagram representing the process structure of the designed software. In Figure 6-7, all devices are composed in parallel and every device is represented via a parallel composition of Device_Motion_Profiles, Device_controller and Device_WritePWM process blocks.

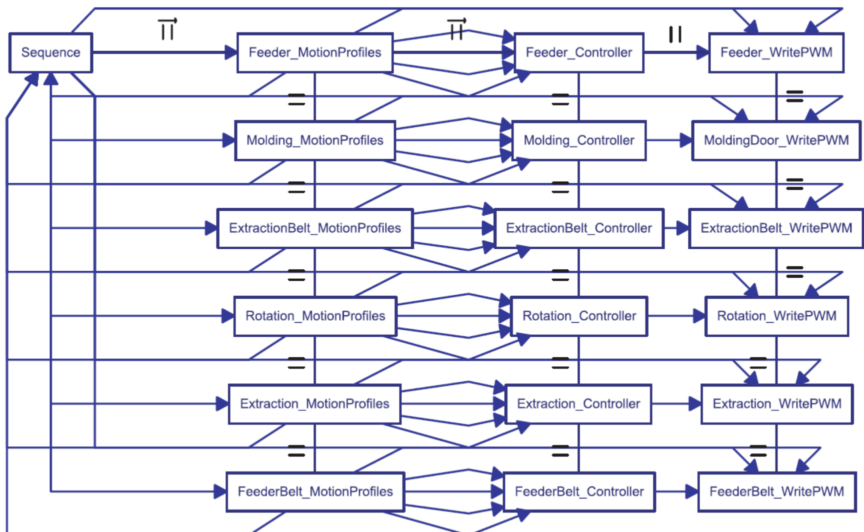


Figure 6-7 Block scheme (Maljaars , 2006) presenting process blocks on top-level

Figure 6-8 shows only the processes related to the rotation device of Figure 6-7. Note that this is in fact, the loop control layer.

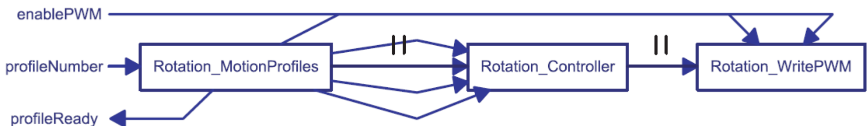


Figure 6-8 Rotation device (Maljaars , 2006)

All three processes from Figure 6-8 are specified using GML. Figure 6-9 depicts the GML specification of the Rotation_Controller process. It uses time channel to block until predefined time is reached, than it will read in parallel encoders and the reference values of position, speed and velocity, as calculated by the RotationMotionProfiles process. After that the value calculated in the previous cycle is written to the actuator and the Rotation_controller process is used to calculate values for next actuating action. The Rotation controller process is imported from 20-sim.

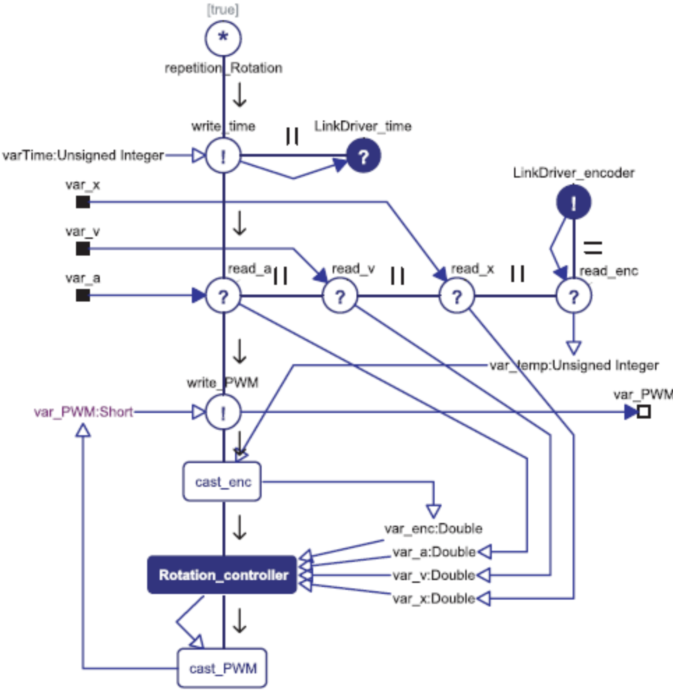


Figure 6-9 Rotation control process (Maljaars , 2006)

However, sequence controllers were not specified using GML. Reason was that sequence controllers are best expressed via state-machine like designs, and GML was found not to be suitable for such design. Instead, the UPAAL tool was used to specify the sequence controllers. The one for the rotation device is depicted in Figure 6-10.

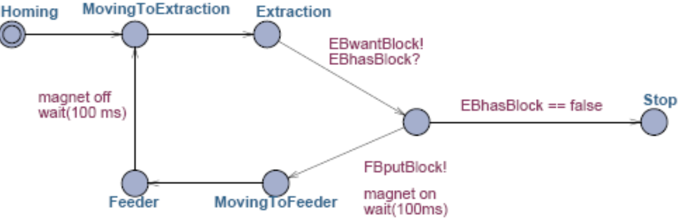


Figure 6-10 Sequence controller of the rotation device (Maljaars , 2006)

The lack of expressiveness of the GML approach, more precisely its inability to depict state-machine like designs that was observed during this project, was one of the reasons that triggered work on the SystemCSP graphical language.

6.2.3 POOSL

POOSL(Parallel Object-Oriented Specification Language) (Geilen et al., 2001; POOSL, 2007) is a general purpose modeling language based on process algebras and object-oriented concepts. The process algebra part comes from CCS, and is expressed via primitives that can be used to specify parallelism, non-determinism and time properties.

As in our approach, processes communicate via synchronous channels, and similar control-flow elements exist. Unlike SystemCSP, POOSL doesnot provide ways to visualize its control flow. Visualization is kept on the level of making block diagrams specifying the structure of the system via instances of process classes related via ports.

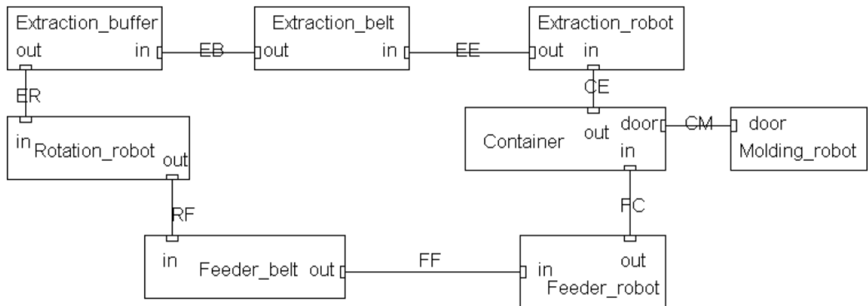


Figure 6-11 Block scheme of processes controlling the setup (Huang et al., 2007)

Figure 6-11 depicts the block diagram of the process instances involved in controlling the Production cell setup (Huang et al., 2007). Figure 6-12 illustrates the POOSL code for controlling the rotation device.

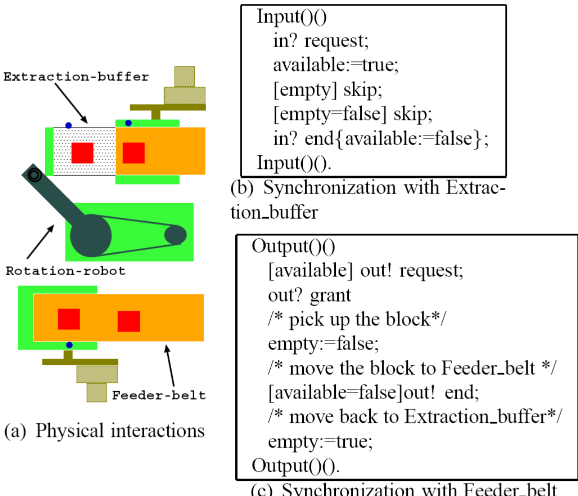


Figure 6-12 POOSL code for controlling the rotation device (Huang et al., 2007)

Compared to the SystemCSP approach, POOSL misses ways to visualize its control- flow elements, and could for instance benefit from using some SystemCSP-like visualization.

6.3 SystemCSP - Device-oriented design

6.3.1 System structure

Basic structure

A device in the production cell setup consists of a mechanical device and a software component that controls it. In Figure 6-13, the top-level of the software design made to control the production cell setup is specified in SystemCSP. The top level is a parallel composition of the software components representing the mechanical devices participating in the production cell. Internally, every software component that controls a device consists of a parallel composition of a sequence control subcomponent and a loop control subcomponent (see right-hand side of Figure 6-13).

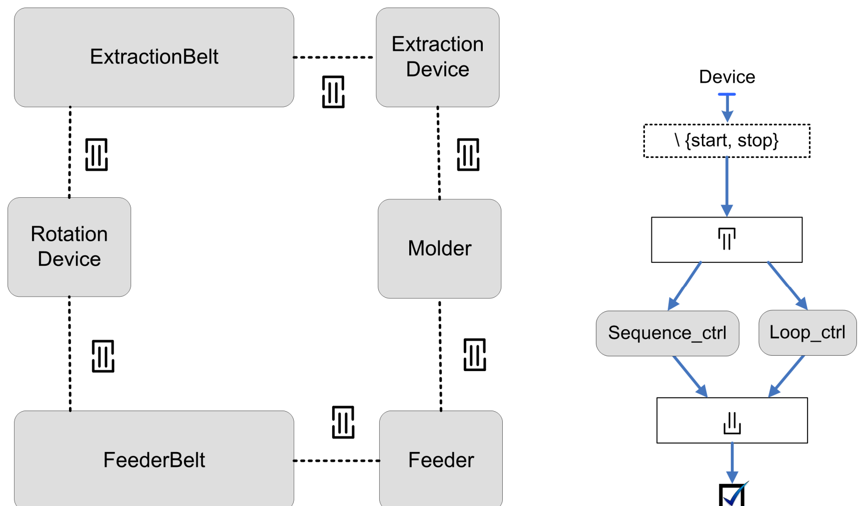


Figure 6-13 Structure of the Production Cell and the devices

Compared to design patterns given in section 5.3.1, the design given in Figure 6-13 does *not* contain interaction contracts. Instead, interaction between devices takes place directly among sequence control layers of the involved devices.

Each device in Figure 6-13 contains sequence control and loop control layers. In section 5.3.1, devices did in addition contain safety and supervisory control subcomponents and state data processes. The state data process was introduced in order to share state data of a device among its subcomponents. However, in this

case, sequence control and loop control components do not need to share any state data.

The safety layer is in the Production cell partly implemented in the hardware of the plant. Handling exceptional situations is merged into the sequence control layer.

A supervisory layer is not present in Figure 6-13. One way to add monitoring facilities is to follow the design pattern given in section 5.3.2.

Adding monitoring layer

The design of the system is extended with monitoring layer according to the pattern from section 5.3.2. This results in adding a central monitoring component in parallel to the device components (see Figure 6-14). Inside every device a `Logger` component is added. It is composed in parallel with the already defined parallel composition of `Sequence_ctrl` and `Loop_ctrl` components, as depicted in Figure 6-14.

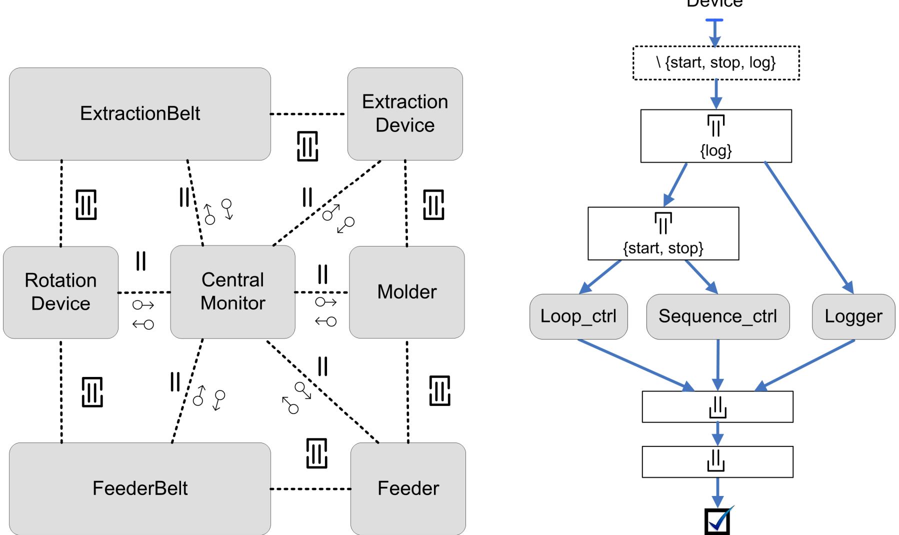


Figure 6-14 Device oriented software with central monitoring

The central monitor from Figure 6-14 does play the role of the supervision control interaction contract from section 5.3.1. Note in right-hand side part of Figure 6-14 that Loop_ctrl and Sequence_ctrl processes synchronize on ‘start’ and ‘stop’ events, but not on ‘log’ events. Each one of those two processes separately synchronizes with Logger component on ‘log’ events. In addition, events ‘start’, ‘stop’ and ‘log’ are not exported to higher abstraction levels in order to avoid synchronizing on those events with subcomponents of other devices.

In Figure 6-15, the interaction-oriented view is used to depict subcomponents of a device component, their compositional relationships and set of events on which they synchronize.

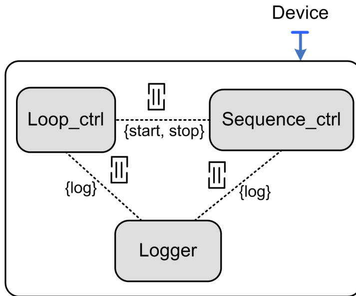


Figure 6-15 Interaction-oriented view of device structure

Assigning priorities

Let us consider the case when all software components presented in Figure 6-14 are executed by a single node. Loop_control subcomponents of all devices are real-time and thus need to be of highest priority. If loop_control processes of different devices are executed with different sampling periods, their priorities can be ordered using RM scheduling. The next priority level is reserved for supervision control components. Logger components do have the least priority.

The watchdog design pattern from chapter 5, could be used to detect timing faults in cases when periodic `Loop_ctrl` processes miss their deadlines.

As discussed in chapter 4, in order to decouple higher priority loop_controllers from the rest of the system, their interactions with the environment should go via *shared data objects*. An alternative way to decouple loop controllers from the rest of the system is to design their interaction pattern in such a way that it is not possible for them to wait on synchronization with the environment. One way to do this is if every iteration starts with checking out readiness of those events in a guarded alternative, where a SKIP guard is associated with normal execution. In that way, synchronization with events from the environment is attempted only when the environment is ready to engage in events. Next section deals in detail with the design of Loop controllers.

6.3.2 Loop controllers

Structural part

The structural part of the design related to the pure computation code (the sequential code without any events) is in SystemCSP best captured via UML class diagrams. In every device, a loop controller process controls the actual movement of the associated mechanical device using an instance of the DeviceControlUnit class (see the UML diagram in Figure 6-16).

An object of class `DeviceControlUnit` offers to the loop controller process, that contains it, an interface consisting of 5 functions. Once a motor is put in the running mode via the start() function, the loop controller is expected to periodically call the control_loop () function every sampling period. This function uses the

auxiliary private function calculate() to perform the actual control loop calculations. The stop() function is used to initiate stopping the motor. The setDirection() function is used to switch between forward and backward movement. isMovementDone() is a function used to check a flag keeping track of whether the motor has finished the specifed movement.

Two types of sensors (see Figure 6-16) are present in this setup: encoders and end-switches. Encoders are used to detect current position. The movements are limited via end-switches. Two types of actuators (see Figure 6-16) exist in the production cell system – a motor and a magnet. Movement of every device is performed by a motor. The extraction and the rotation devices, in addition, each have an instance of the Magnet class. The Magnet class offers only on() and off() functions.

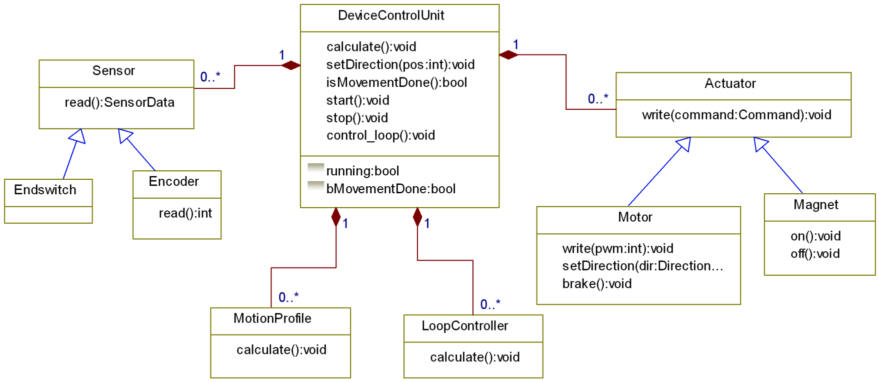


Figure 6-16 UML class diagram for actuators used in the Production Cell

The DeviceControlUnit internally contains one or more instances of MotionProfile and LoopController classes (see Figure 6-16). The function calculate() of the DeviceControlUnit will call in a row the functions calculate() of the objects implementing the motion profile and the controller object. The implementation of those functions can be generated from a CAD tool related to control engineering (e.g. Matlab, 20Sim, …).

As explained in Section 4.2.1, there are two ways to order sensor reading, calculation of the control loop algorithm and writing the calculated values to the appropriate actuator. The case of the sample-calculate-actuate pattern can be implemented like:

```
control_loop(){
      sensorValues =  sensor->read();
      reference = motionProfile->calculate();
      actuatorValues = controller->
                          calculate(sensorValues,reference);
      actuator->write(actuatorValues);
}
```

The sample-actuate-calculate pattern can be implemented like:

```
control_loop(){
      sensorValues = sensor->read();
      actuator->write(actuatorValues);
      reference = motionProfile->calculate();
      actuatorValues = controller->
                    calculate(sensorValues, reference);
}
```

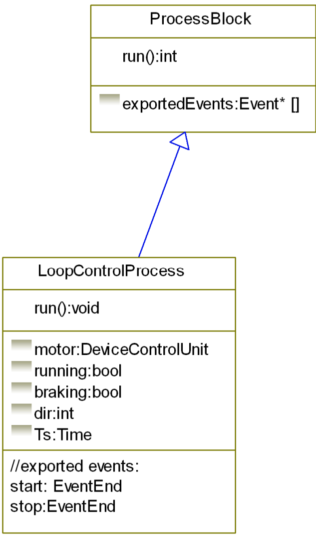


Figure 6-17 Loop Controller process

In Figure 6-17, the class defining LoopController process is depicted. This class is a kind of ProcessBlock abstraction. It internally uses an instance of `DeviceControlUnit` named `motor` to handle the device unit. In addition, it contains boolean flags `running`, `braking`, integer `dir` for keeping the direction, time `Ts` as a period of its activation, and 'start' and 'stop' event ends exported to interact with appropriate sequence control process of the same device. The behavior of the LoopController Process is specified in its run() function and depicted via SystemCSP diagrams.

Behavior definition 1 - Position based loop controller

In this variant, the motion profile is fixing the start and stop position. This means that the loop controller will internally use DeviceControlUnit::isMovementDone() to determine when a movement is done. Than, it will initiate the 'stop' event to signal the end of the movement to the higher-level sequence controller of the device. In Figure 6-18, an event synchronization pattern of the position-based loop controller is depicted.

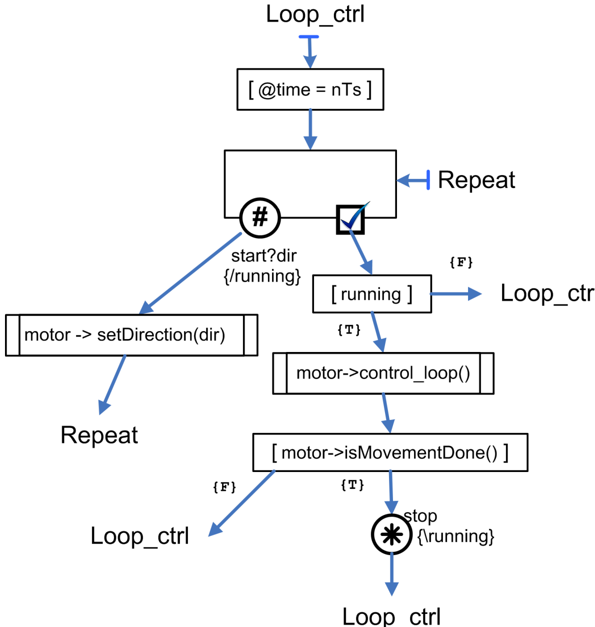


Figure 6-18 Loop controller when motion control unit initiates stop event

The loop controller is activated periodically with period Ts. After the occurrence of 'start' event, initiated by the sequence controller, the flag `running` will be set to the active level (note the rising "/" sign in front of it in the event related action field). The received parameter `dir` will be used to set the direction of the motor, which results in choosing the appropriate motion profile. In the standard body of the loop, when the flag `running` is set, the function control_loop() of the device control unit is called to read new sensor values, calculate actuator data and write them to the PWM unit.

The Loop controller given in Figure 6-18 assumes that the specification of the movement of a device includes fixed values for both start and end point. In the case of the belts, this implies moving the belts only for a fixed step. In that way, the length of the belt is divided into N buffer spaces filled with boxes one by one

Behavior definition 2 - Velocity based loop controller

The other solution is to keep the belts moving unless their movement is restricted by the needs of synchronization with neighboring devices. In this case, the loop controller needs to be different since initiating the stopping of the motor has become the responsibility of the sequence controller. The designed loop controller is depicted in Figure 6-19.

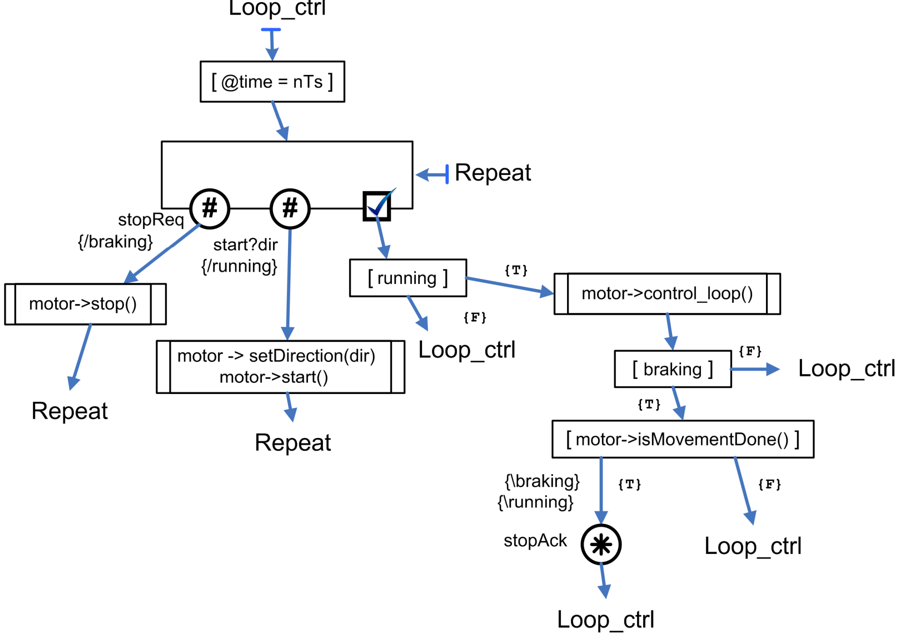


Figure 6-19 Loop controller when motion control unit accepts stop event

The event 'start' also is used to pass the direction as parameter. Immediatelly after this event, the flag `running` is set to active level (note the '/' sign before the flag name in the event-related action description field). Then the direction of the motor is set and the standard body of the loop is executed. The standard body of the loop is executed periodically with period Ts. If the `running` flag is set to true, the motor object (instance of the DeviceControlUnit abstraction) will calculate the next position and update the steering value for the actuator.

After the associated sequence controller issues the 'stopReq' event, the `braking` flag will be set to active level and the function DeviceControlUnit::stop() will be used to change the profile of the reference input to the one predefined for stopping the motor. In one of the next periods, the velocity of the belt will stabilize at zero, the belt will stop running, and the function DeviceControlUnit::isMovementDone() will return true. After that both 'braking' and 'running' flags will be reset (note the "\" sign) and a 'stopAck' event will be initiated to notify the associated sequence controller that the belt has stopped.

6.3.3 Sequence control

Basic blocks

A single motor, that can in general move forward or backward, drives any of the six devices of the production cell setup. In case of the belts, only forward movement is used. The extraction device and the rotation device have, in addition,

a magnet associated. The four basic processes repeating in different devices are therefore: Move_fwd, Move_bckwd, Magnet_on and Magent_off. Figure 6-20 depicts internals of those basic building blocks.

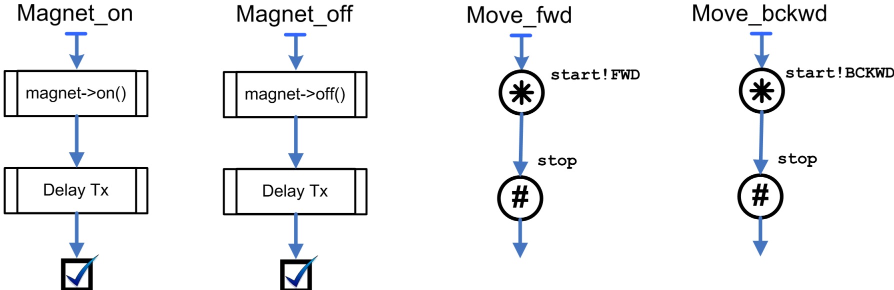


Figure 6-20 Basic sequence control blocks

Delay blocks are added to Magnet_on and Magent_off processes in order to allow the magnet to properly grasp/release a block before the next action (e.g. movement of the device holding the magnet) is allowed.

Dependencies

Table 3 lists all precedence constraints relating actions of the devices in the Production cell system. The first column in the table is an index of the table entry, the second column is the name and the state of the sensor (if any) that can replace explicit synchronization by detecting in the real world when the condition associated with the precedence constraint is fulfilled. The third column has subcolumns for the device and a condition that allows action to take place. The fourth column contains two subcolumns – one for the device and the other one for the action constrained by the condition of the third column.

Note in Table 3 that precedence constraint PC6 seems not to be strict in our setup. This is due to the construction of the setup where the extraction robotic arm is actually above door level. However, in a real molding machine the extraction device should not be allowed to attempt to enter into closed molder. In fact, this precedence constraint is important in our setup since it does imply the one that is more strict - a constraint that does not allow the extraction robotic arm after entering the closed molder to attempt to pick-up a block and carry it back.

Precedence constraint PC9 states that the molder should not close its door before the extraction device is in the idle position. In our setup it is not relevant due to the construction of the setup where extraction robotic arm is actually moving above the molder door level. However in a real production cell, depending on the homing position and the settling time of the positioning of the extraction robotic arm, it can be a safe solution to fulfill this condition.

#	sensor	Device :: condition		Device :: action	
PC1	¬(S1 & S8)	Feeder belt, Feeder	space available at the end	Feeder belt	can deliver block (move fwd)
PC2	S1	Feeder belt	delivered block	Feeder	can feed molder (move fwd)
PC3	S3	Molder	door closed	Feeder	can feed molder (move fwd)
PC4	¬S2	Molder	empty	Feeder	can feed molder (move fwd)
PC5	S2	Molder	block in molder	Molder	can mold
PC6	¬S3	Molder	door opened	Extraction device	can enter molder (move fwd)
PC7	¬S4	Extraction belt	space available	Extraction device	can deliver block (magnet off)
PC8		Extraction belt	not moving	Extraction device	can deliver block (magnet off)
PC9		Extraction device	not moving	Molder	can close door (move fwd)
PC10	¬(S6 & S5)	Extraction belt	space available at the end	Extraction belt	can move (move fwd)
PC11	S6	Extraction belt	delivered block	Rotation device	can pick-up (magnet on, move fwd/bckwd)
PC12	¬S7	Feeder belt	space available	Rotation device	can deliver block (magnet off)
PC13		Feeder belt	not moving	Rotation device	can deliver block (magnet off)

Table 3 Precedence constraints in the production cell

Precedence constraints PC8 and PC13 stand for the constraint that belts need to be stopped before a block is placed on them. In the general case of a production cell the optional presence of those constraints depends on the intended application usage. In our case, insisting on those constraints results in a slower speed of the production cycle, but yields a better aligning of the blocks and in that way decrease of the probability of blocks slipping off the electromagnet of the rotation device

due to the electromagnet being applied away from the center of the block.

Generic sequence control

Sequence controllers are usually state machines or sequences of activities repeated in cycles. In Figure 6-21, the sequence of actions performed by each of the devices is illustrated.

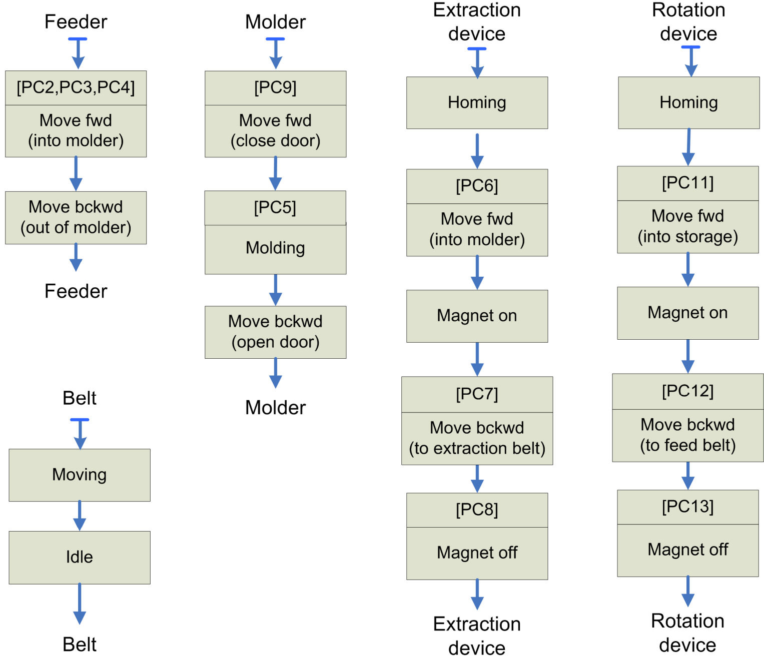


Figure 6-21 Sequence control with generic precedence constraints

For the belts, a simple finite state machine is depicted, consisting of only two states: moving and idle. The transitions from moving to idle and back are somewhat more complex due to precedence constraints PC1, PC2, PC12 and PC13 for the feed-belt and PC7, PC8, PC10 and PC11 for the extraction belt.

For the other devices the precedence constraints are made explicit as preconditions for the related actions according to Table 3.

The feeder is allowed to enter into the molder after preconditions PC2, PC3 and PC4 are satisfied. After feeding the block into the molder, it moves backward to the retracted position creating the space for the next block to be delivered by the feedbelt.

The molder is allowed to close the door after the precondition PC9 is satisfied. After closing the door, it will perform the ‘molding’ process and then open the door to allow the extraction device to pick-up the product.

The extraction device can move into the molder after precondition PC6 is satisfied.

There it will turn the magnet on to pick-up the block and start moving backward after precondition PC7 is satisfied. When it is situated above the extraction belt it needs to wait for precondition PC8 to turn the magnet off and deliver in that way a block to the belt.

When a block is in the storage place, indicated by PC11, the rotation device can enter the storage and pick-up the block by turning on its magnet. After precondition PC12 is fulfilled, it can move the block to the feeder belt. When it is positioned above the feeder belt, it still needs to wait for the precondition PC13 to be fulfilled before turning its magnet off and delivering in that way the block to the feeder belt.

An alternative, given in Figure 6-22, is not to allow the extraction and rotation device to be used as a buffer place.

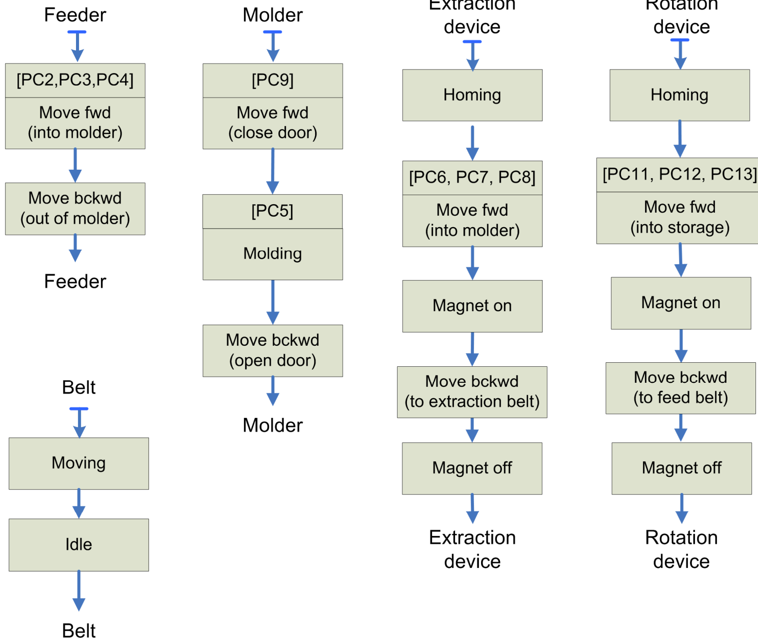


Figure 6-22 Case when the extraction and rotation devices cannot be used as buffers

Thus, the extraction device and the rotation device are not allowed to hold a block in air waiting until the preconditions for delivering the block are met. In that case, all three preconditions (PC6, PC7 and PC8) are met before the block transfer starts.

Time based schedule

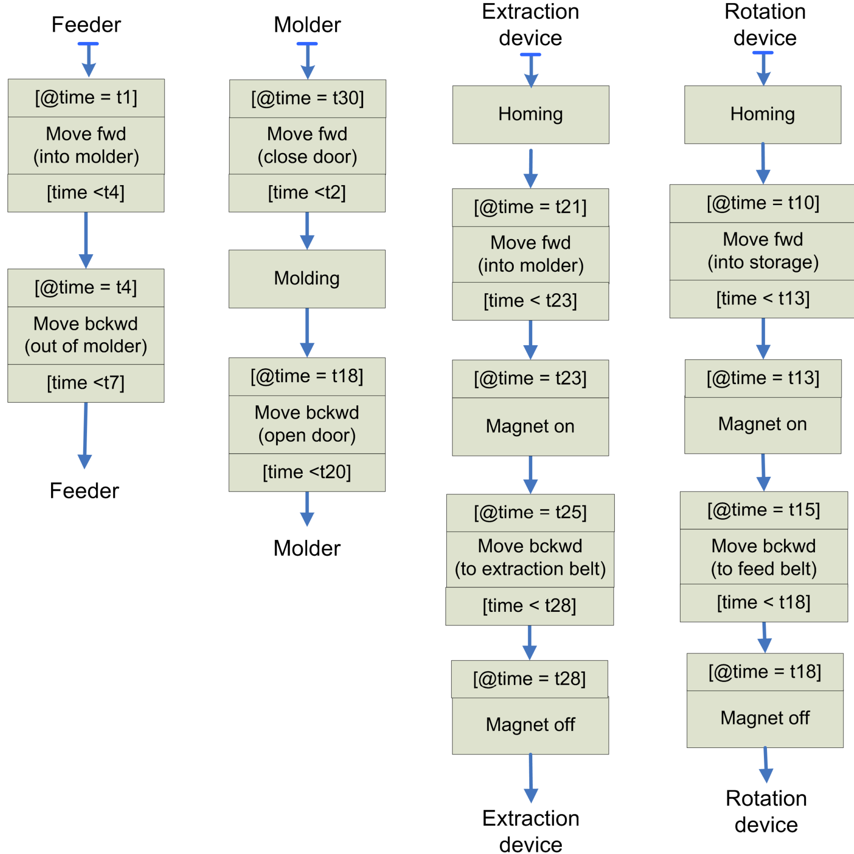


Figure 6-23 Time-based software design

One way to implement the software for production cell is, as in section 6.2.1, to associate starting-time and end-time constraint with each activity. In that way, preconditions that hide dependencies between devices are transformed into time constraints related to each action as it is in the SystemCSP diagram depicted in Figure 6-23 for the time table given in Figure 6-6.

The control-flow oriented part of SystemCSP is capable of depicting the time-based design in a clear and efficient way. This is, again, the case especially due to the possibility to associate a precondition field with any process and due to possibility to specify time constraints in the precondition field.

Sensor based sequence control

Feeder
[S1,¬S2, S3]
Move fwd (into molder)
Move bckwd (out of molder)
Feeder

Belt
Moving
Idle
Belt

Molder
[PC9]
Move fwd (close door)
Molding
[S2]
Move bckwd (open door)
Molder

Extraction device
Homing
[¬S3,¬S4, PC8]
Move fwd (into molder)
Magnet on
Move bckwd (to extraction belt)
Magnet off
Extraction device

Rotation device
Homing
[S6, ¬S7, PC13]
Move fwd (into storage)
Magnet on
Move bckwd (to feed belt)
Magnet off
Rotation device

Figure 6-24 Sensor based design of sequence controllers

Compared to the generic sequence control scheme from the Figure 6-22, in Figure 6-24 preconditions are validated using sensors. In this way dependency between devices is transferred into detecting, in the plant, the results of actions the devices have performed.

No sensors cover preconditions PC8, PC9, and PC13, but, as already explained, satisfying thepreconditions PC8, PC9 and PC13 can safely be omitted in our setup. Thus, it is possible to actually to design sequence controllers in a way that satisfies precedence constraints by relying only on sensor measurements.

Again, the control-flow oriented part of SystemCSP is capable to depict the intended design in a clear and efficient way. This is, again, the case especially due to the possibility to associate a precondition field with any process.

Event based sequence control

Another possible design choice is to replace every precedence constraint with event synchronization. This is illustrated in Figure 6-25 and Figure 6-26. In Figure 6-25, sequence control processes are depicted for the feeder, molder, extraction device and rotation device. The sequence controllers on Figure 6-25 still do match the generic sequence controllers from Figure 6-22. In this case precedence constraints are fulfilled via explicit event synchronization between devices. Note

that a precondition of the action marked as PCi on Figure 6-22 maps to accepting an appropriate event in Figure 6-25. The event is initiated by the device whose action will fulfill the precondition.

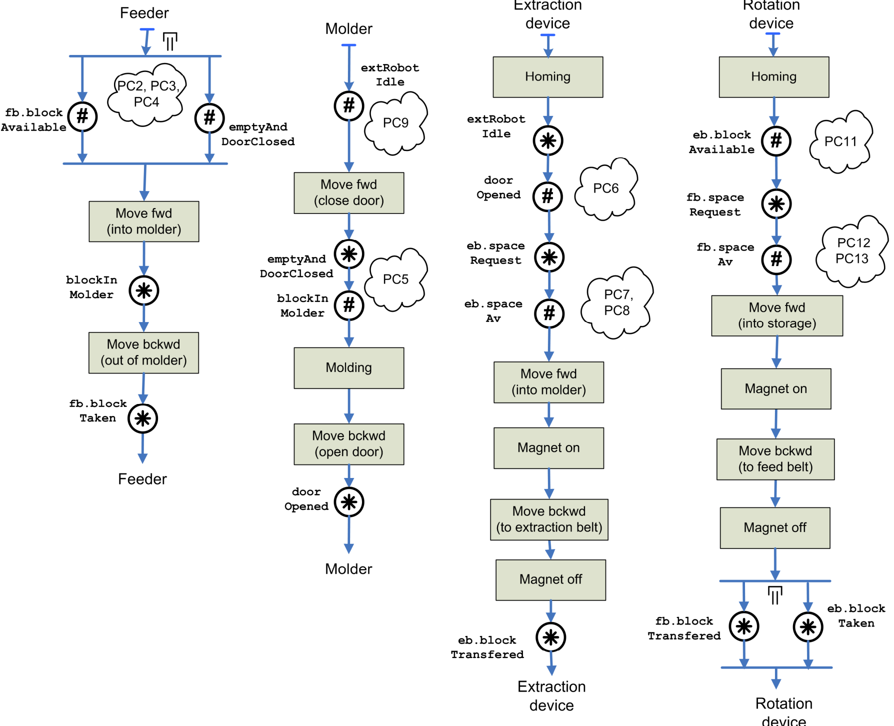


Figure 6-25 Event based solving precedence constraints

The feeder needs to wait until it has a block available (precondition PC2 is fulfilled by the event 'fb.blockAvailable'). Feeder at the same time needs to wait until the molder is ready to accept the block (event 'emptyAndDoorClosed' that maps to the preconditions PC3 and PC4). Since the order of the 'emptyAndDoorClosed'and 'fb.blockAvailable' events is irrelevant, the feeder waits for them in parallel allowing either to take place first. After the occurrence of both events, the feeder can move forward and push the block into the molder. After delievering the block, the feeder notifies the molder about that ('blockInMolder' event that maps to PC5) and then goes backwards to its home position. After this movement is performed, the event 'blockTaken' (that maps to PC1) is initiated.

The molder device needs to wait until the extraction device is stopped (PC9 satisfied via the occurrence of the event 'extRobotIdle') before it can close the door. Than event 'emptyAndDoorClosed' is used to signal to feeder both PC3 and PC4. After the feeder has pushed the block into the molder (event 'blockInMolder' that maps to PC5), the "molding" starts. When the "molding" time expires, the molder will open the door (event 'doorOpened' that maps to PC6).

The extraction device has its homing position in the middle of its path between the

entry point of the extraction belt and the molder. After it settles down in the homing position it does notify the molder about this (event 'extRobotIdle' that maps to PC9). Then it waits until the molder finishes the 'molding' process and opens the door (PC6 and related event 'doorOpened'). When the extraction belt is ready to accept block (PC7 and PC8 and related event 'eb.spaceAv'), the extraction device can enter the molder and pick–up the block, transfer it to the belt and notify the belt about block delivery using 'ev.blockTransfered' event.

The rotation device has its initial position in the middle of its path between the extraction belt and the feedbelt. It will wait in the home position until a block is available on the extraction belt (PC11 and related event 'eb.blockAvailable'). Then it will request space on the entry point of the feedbelt and when this request is granted (PC12, PC13 and related event 'fb.spaceAv') it can transfer the block from the extraction belt to the feed belt. When moving the block is finished and after the block is placed on the feed belt, the extraction belt can be notified that the storage space is empty and ready to receive the next block (event 'eb.blockTaken'). The feed belt can be notified that the block is delivered (event 'fb.blockTransfered').

There are two possible ways to realize the sequence control processes for the belts. Figure 6-26 depicts one for belts performing step-by-step movements and the Figure 6-27 the case when belts are kept free-running whenever possible.

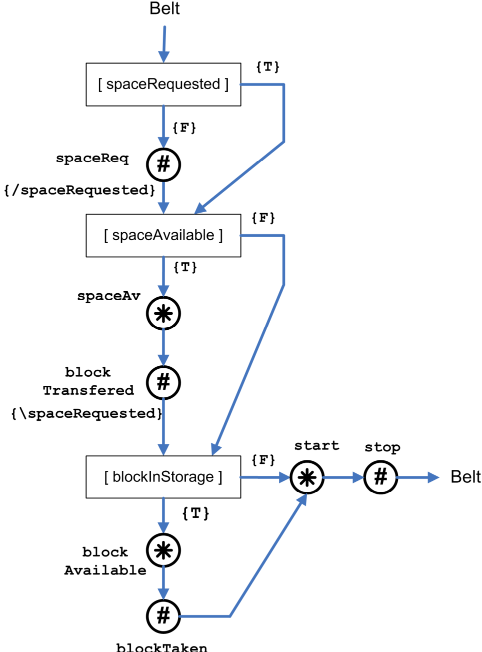


Figure 6-26 Belt sequence control for step-by-step case

In the case of the belt sequence controller in Figure 6-26, the belt is moved one step at a time. As explained in section 6.3.2, this kind of sequence controller is used together with the position-based loop controller. The sequence controller only

initiates the movement of the belt using the 'start' event, and the loop controller initiates the 'stop' event after the motion profile is done. The sequence controller accepts a 'stop' event and waits for the next request for space (see Figure 6-26). If the space at the beginning of the belt is requested, then in case the space is available (sensor S4 and associated precedence constraint PC7 in case of the extraction belt and sensor S7 and associated precedence constraint PC12 in case of the feeder belt), the acknowledgment event 'spaceAv' will be initiated. The sequence controller will then wait until block is delivered, satisfying in that way precedence constraint PC8 for the extraction belt, that is PC13 for the feeder belt. If space was not available, then if there is a block in the storage, action will be initiated for the next device in chain to take it away. If there is no block in the storage, then the belt can be moved for a single step.

The result of this sequence controller is that blocks are actually queued on the belt and moved only after a block is delievered to available space on the input part of the belt.

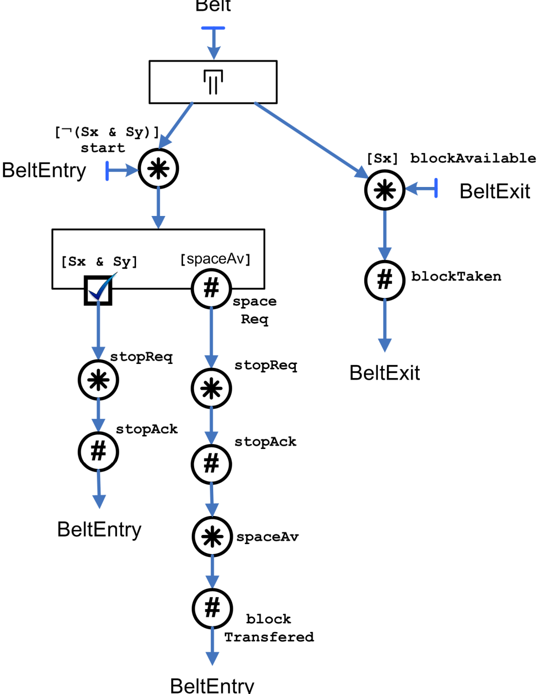


Figure 6-27 Belt sequence control for free-running case

The free running belt depicted in Figure 6-27 uses a velocity-based loop controller as described in section 6.3.2. This sequence controller needs to initiate both the 'start' event and the 'stop' event. The sequence controller is designed as a parallel composition of two repetitive processes: BeltEntry and BeltExit. BeltEntry process controls the movement of the belt and the BeltExit process is used to synchronize with storage on the belt exit point. BeltEntry process will allow the belts to start

whenever there is a place at its end, that is when sensors (Sx and Sy that map to S7 and S6 respectively for the extraction belt and S1 and S8 for the feeder belt) indicate that. When however, both places are occupied, the ‘stopReq’ event will initiate stopping of the belts and after the associated loop controller responds with an acknowledgment (‘stopAck’ event), the process BeltEntry returns to its starting point. Alternatively, belts are stopped (events ‘stopReq’ and ‘stopAck’) prior to the delivery of a block to the belt, that is after the device delivering blocks to the belt issues the space on the belt entry point and the space is available (note the logical guard spaceAvailable in the guarded alternative process in Figure 6-27).

6.4 Interaction contract based design

Obviously, interaction contracts bring in some overhead. However, using interaction contracts is considered to be much more structured approach than specifying the synchronization between devices in isolation. An interaction contract is a generic way to organize interaction. It can offer standardized interfaces that participating components are expected to full-fill, creating in that way reusable plug-and-play architecture. The real power of the interaction contract is in detecting and handling the exceptional situations that pass the boundaries of participating components or arise due to the interaction pattern. For instance, only in a centralized place that manages the interaction it is possible to efficiently detect and react on the violations of the time constraints that are imposed onto parts of the interaction handled by different participating components.

The design pattern presented in Section 5.3.1 is based on an interaction contract dedicated to the activity of managing a layer (sequence control, loop control, supervisory control and safety layer) or more layers of a complex control system.

Logical design choice is to first focus attention on introducing the sequence control interaction contract. As in section 6.3, it is possible to extend the designed system with a supervivision layer according to the design pattern given in Section 5.3.2. Again, the interaction contract for the loop control layer is implicitly present due to the coupling introduced via sharing the same processor node according to some priority scheduling scheme. The safety layer is partly implemented in hardware and partly merged into the sequence control layer.

The best starting point for defining the sequence control interaction contract is to reuse existing event-based interaction patterns of sequence controllers from Figure 6-25 and Figure 6-26, and to group related event synchronizations into interaction contracts.

As illustrated in Figure 6-28, three actions seems to be best suited to form the basis of this interaction pattern: 1) coordinating feeder belt, feeder and molder in the action of feeding the molder; 2) coordinating molder, extraction device and extraction belt in the action of extracting the product; and 3) coordinating the extraction belt, the rotation device and the feeder belt in the action of recycling the block.

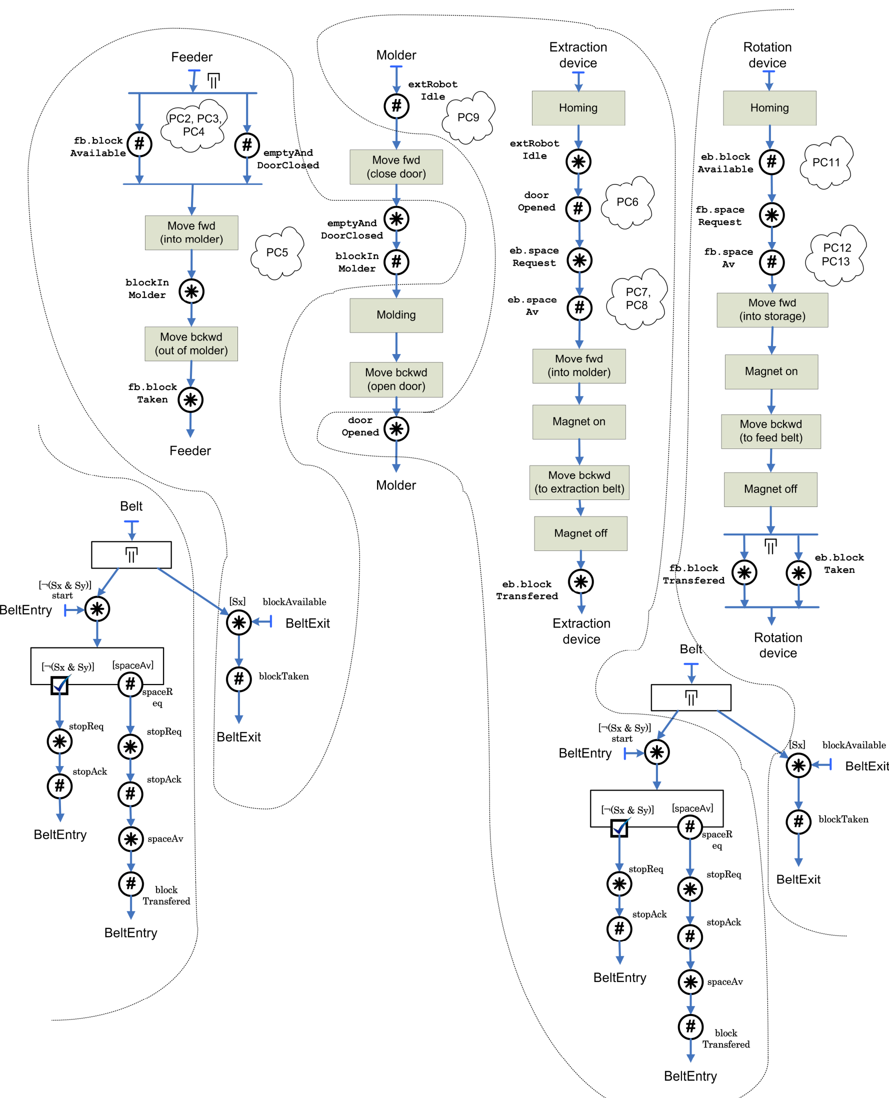


Figure 6-28 Splitting sequence control synchronization pattern in 3 interaction contracts

The block diagram representing the resulting interaction contracts and participating components is depicted on left-hand side of Figure 6-29. As in section 6.3, every device is again a parallel composition of loop controller and sequence controller process. (see the right-hand side of Figure 6-29).

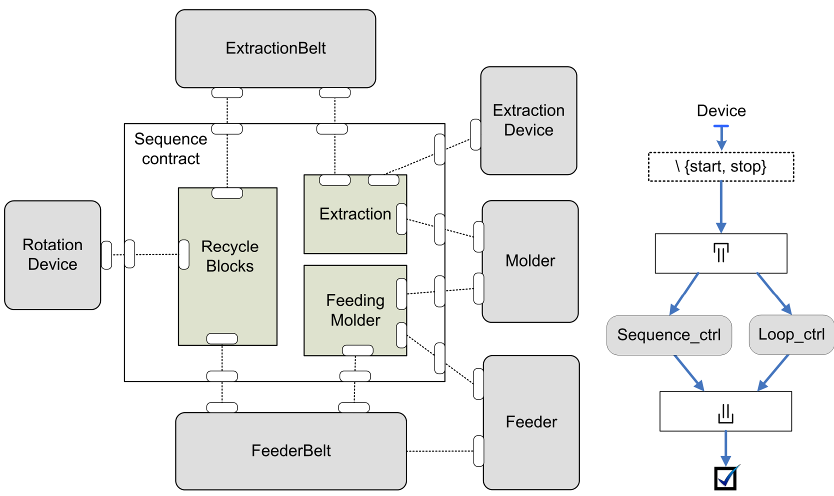


Figure 6-29 Interaction contract based design - top level

In Figure 6-30, interaction of the set of devices making the production cell setup is organized via sequence contract and supervision contract. Again, as it was done in section 6.3.1 according to pattern given in section 5.3.2, the Logger component is added inside each device in order to add support for the supervision layer.

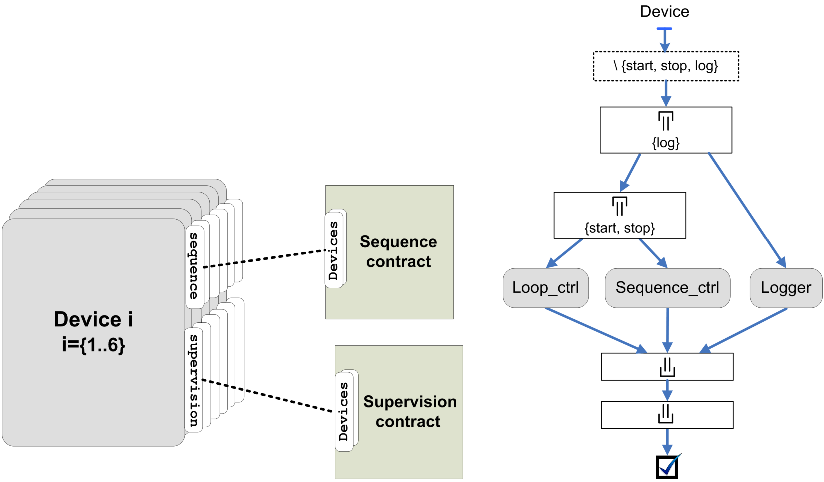


Figure 6-30 Extending system with supervision layer

Figure 6-31 depicts the interaction contract for feeding the molder. Event synchronization is similar as before. The addition is that now the interaction contract manager (named “Feeding molder” in Figure 6-31) takes control over managing the part of interaction.

The interaction contract manager does have sensors under its control and uses them to check the correctness of the readiness signals received from participating components. Thus, the readiness of participating devices is double-checked: via

sensors and via event communication. In case when erroneous state is detected, an exception is raised (exc1 or exc2 in Figure 6-31) and control is given to its subprocess specifying the exception handling mode (EHM in Figure 6-31). The EHM process can analyze sensor readings and communicate with involved devices to make diagnosys and suggest a way to remove erroneous state.

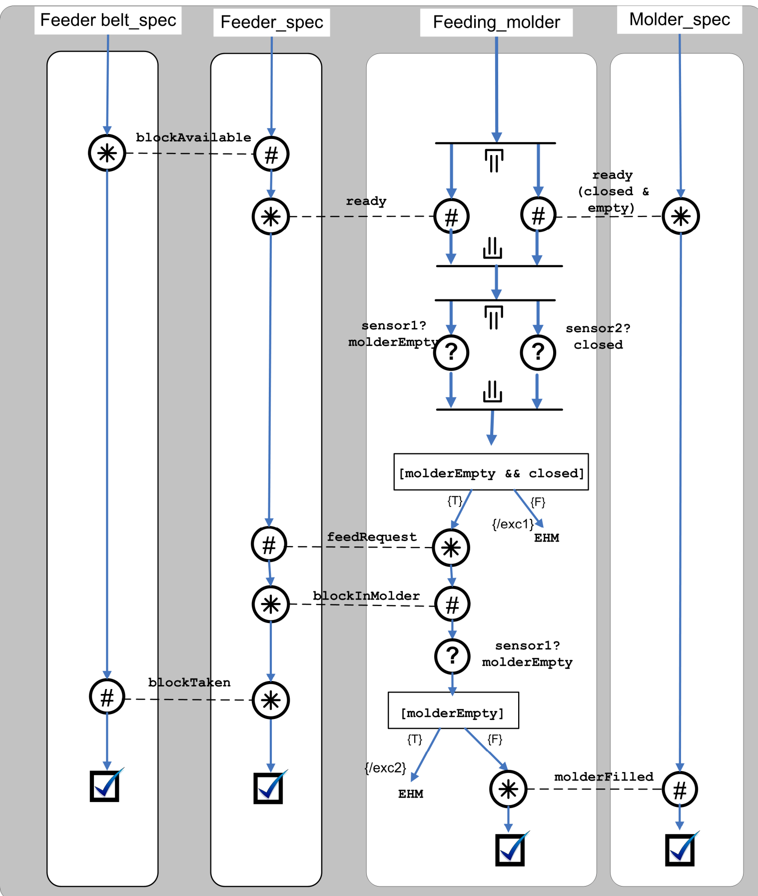


Figure 6-31 Interaction contract for feeding the molder

The sequence controllers inside devices are in fact implementing one or more roles defined by appropriate interaction contracts. Figure 6-32 illustrates the way sequence controllers are made out from implementations of role specifications from the interaction contracts. Note that the role implementations preserve event pattern defined in role, but add implantation and interactions that are irrelevant for the specification of the role in interaction contract.

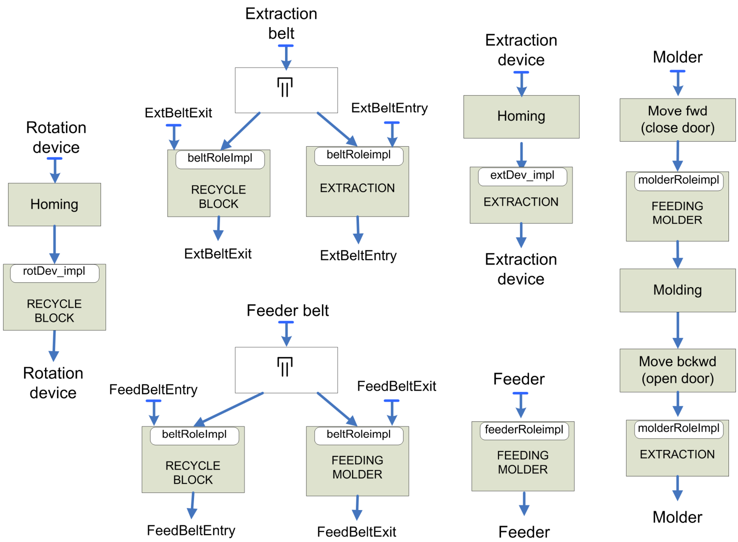


Figure 6-32 Sequence controllers as implementation of contract roles

6.5 Conclusions

The production cell setup is a big step towards an industrial-strength test of a design methodology. SystemCSP seems to be a convenient way to capture the interactions in the design specification of complex control systems. The first type of relevant interactions includes the ones inside the same control layer (e.g. interactions related to supervision and sequence control) spread across several devices. The second type of relevant interactions is the interaction between different control layers inside the same device (e.g. in proposed designs interactions between loop controller and sequence controller subcomponents of a device via 'start' and 'stop' events, or interaction of both with Logger component via 'log' event).

In fact, this setup was one of the triggers for the development of SystemCSP as a novel graphical language for the design specification of concurrent, component-based systems.

This chapter gives detailed description of the setup, explain some other ways used to design software for the setup and puts focus on ways to express different types of designs in the SystemCSP language.

In section 6.3.3, it is demonstrated that the control flow oriented part of the notation can be used to capture sequence controllers when interactions are implicit – as in designing the time-based schedule and sensor-based sequence controllers.

The most structured, flexible and reusable way to design control systems is based on creating interaction contracts. In fact, the introduction of interaction contracts

into SystemCSP was inspired by the paper dealing with coordinated atomic actions designed for some other production cell system (Zorzo et al., 1999).

Note that in addition to SystemCSP diagrams, UML class diagrams were used to define structure of objects used inside Loop Controller components. The possibility to combine SystemCSP with UML class diagrams was already introduced during the positioning of SystemCSP as given in section 3.7. The test case illustrates this in practice.

SystemCSP focuses on the interactions among components and does not include support for the domain-specific code generation (e.g. the contents of `calculate()` functions of `motionProfile` and `controller` objects). This is the task of CAD tools specific for control systems and physical system modeling domains. This is in accordance with positioning of SystemCSP as given in section 3.7.

The work on designing the software in SystemCSP for the production cell setup did in addition bring forth the need for additional elements that were added to the notation (action blocks, comment blocks and precondition/postcondition fields in process blocks).

Recommendation is to implement and test the proposed designs on the real setup. Performance measurements should be done in order to compare performance of different allocations of components to the hierarchy of execution engines (nodes, OS threads, UL threads, function-based concurrency). Performance should be thoroughly compared with the performance of the designs based on other approaches (e.g. the three approaches described in section 6.2).

Furthermore, the designs should further be extended with fault tolerance support. Design patterns from section 5.4 (e.g. replication and checkpoining) are expected to be useful in that procedure. Hardware support exists for distribution and dynamic reconfiguration of switching control over the interface to the devices among nodes. Suggested test case is the distribution of replicated components on several nodes and experimenting with reactions of system on node and network failures.

7 Conclusions and recommendations

If you always put limits on everything you do, physical or anything else, it will spread into your work and into your life. There are no limits. There are only plateaus, and you must not stay there, you must go beyond them.

Bruce Lee

7.1 Conclusions

7.1.1 Summary

The main contribution of this thesis is the introduction of SystemCSP, a novel graphical design language for specification of interactions in concurrent component based embedded control systems. SystemCSP was developed in the scope of the embedded control systems application area. However, SystemCSP is intended to be used in any kind of software/hardware development dealing with interaction of concurrent components.

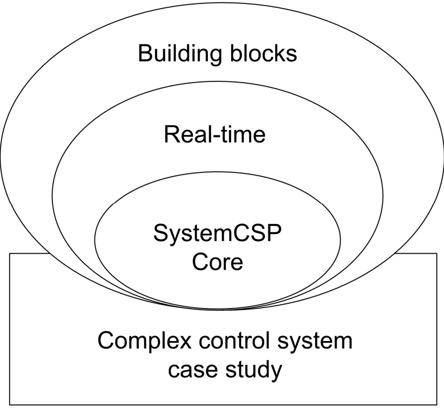


Figure 7-1 Main contributions of this thesis

As illustrated in Figure 7-1, the contribution of this thesis can be separated in several parts: introduction of core elements of SystemCSP notation, creating basis for real-time support, creating set of reusable design patterns intended to serve as basic building blocks in design, and using complex control setup to test the applicability of SystemCSP in control systems area.

SystemCSP core elements

SystemCSP is based on the CSP formal algebra, The main advantage of relying on CSP as underlying formal method is that the concurrency of an application is

designed in a structured way that is liable to formal checking. SystemCSP combines CSP with elements of modern component-based software engineering practice. In that way, it provides an intuitive and readable way to visualize concurrent component-based systems. Notion of *interaction contract* allows specifying, studying, analyzing (e.g. formal checking) interaction patterns in abstract way, in the isolation from the actual context of usage.

The notation has two essential viewpoints: control-flow oriented and interaction-oriented. The control-flow oriented viewpoint results in more readable final concurrency structure of the application. The interaction-oriented viewpoint is especially useful when the focus is on interactions of set of components designed in isolation from the rest of the system. System designs can be scattered in many interaction diagrams focusing on different aspects, and with the same components participating in more than one such diagrams. Still, firm, formally verifiable, relationships are preserved across interaction diagrams. Those relationships are reflected in the control-flow diagram describing the current abstraction level. In that way, incremental design of *control-flow diagrams* is possible by adding restrictions in different *interaction diagrams* throughout the process of system design. This is particularly useful in early stages of the design, when the focus is on interactions of sets of components in isolation from the rest of the system. At the end of the design process, *all* different diagrams converge into a single, formally verifiable, system that can be expressed via control-flow diagram.

The notation has been compared to relevant related graphical design specification languages, namely UML and GML. SystemCSP does incorporate some ideas from both. From its predecessor, GML, the idea of defining binary compositional relationships is adopted and used as a basis of interaction oriented diagrams. However, the philosophy behind the way of using binary relationships is changed and a set of more expressive binary relationships is introduced. SystemCSP seems to be in general more expressive and readable than its predecessor GML. UML has also strongly influenced SystemCSP. The comparison with UML illustrated that SystemCSP is capable to offer alternatives to most of the UML diagram types.

Real-time support

A separate chapter of this thesis is dedicated to time properties of SystemCSP based systems. First, the language elements for the specification of time properties are introduced. The proposed way of specification is inspired by previous work in the CSP community (Roscoe, 1997; Schneider, 2000). In second part of the chapter, implementation of CSP-based systems with real-time properties was investigated. The gap between concepts of CSP-based systems and the ones required by classic scheduling theories is identified. Two major directions are proposed as a way of handling this issue: (1) introducing design patterns that can fit CSP-based systems into requirements of existing scheduling theories and (2) constructing distinct scheduling theories for CSP-based systems.

Design patterns in form of reusable interaction contracts

SystemCSP design methodology is illustrated by creating set of design patterns in the form of reusable interaction contracts. Some of the patterns are concepts often used in practice of software development, but rarely precisely defined and formalized. In that sense, since SystemCSP is directly translatable to CSP, this work also contributes to formalizing those patterns.

Complex control system test case

SystemCSP was applied to design software for the Production Cell setup that was implemented in the scope of this project. The setup consists of several devices that operate concurrently and need to cooperate and synchronize their activities in order to achieve proper functioning of the overall system. Two ways of designing interactions were used: device-oriented and interaction contracts based. This case study proved usefulness of the notation and of design patterns related to structuring concurrency in control systems. The case study also provided feedback that was used to introduce several new elements to the notation.

Implementation issues

The appendices provide basic support for practical implementations of SystemCSP models. In Appendix A, the complete SystemCSP design domain is represented by an appropriate metamodel. This lays the foundation for a structured development of tool that will support the design methodology. Appendix B provides the design of a library providing support for software implementations of SystemCSP based models.

7.1.2 Evaluation

The problem statement given in section 1.3.1 did list key demands for the features of the SystemCSP language. The listed key properties are: mapping to some existing formal verification method, support for specification of time properties and ways to analyze them, support for modern notions of component-based development, expressiveness, readability, scalability, unambiguous interpretation and applicability in complex control systems.

Mapping to some existing formal verification method

This demand is satisfied by choosing CSP as a formal verification method and by associating the introduced notation elements with the elements of the CSP language. Mapping is specified in Chapter 3 by coding SystemCSP designs in CSP. The examples of this mapping given in Chapter 3 illustrate that mapping is simple and direct. Thus, SystemCSP designs is relatively easy to transform into CSP scripts. This mapping to existing formal method allows a user to formally verify his designs for properties like refinement and freedom of concurrency hazards (i.e. deadlocks and livelocks). The support for formal verification will be in prospective tool covered by code generation to CSP domain.

Support for specification and analysis of time properties

As explained in the summary section of this chapter, chapter 4 deals with time properties. The notation has fair support for specification of time properties. However, achieving real-time in practical implementation is more difficult problem. Some insight and possible approaches in real-time implementation and analysis are discussed in detail in chapter 4.

The key conclusion is that classical scheduling can be used in combination with SystemCSP, but that it imposes restrictions on some key properties. Briefly stated, replacing rendezvous channels with shared data object primitives is deleting some precedence constraints and thus extending set of possible behaviors (as expressed by traces). That can mean, as a consequence, that the implementation is no longer a refinement of the specification. Thus, although combination of classic scheduling and design patterns is possible, further research is advocated in the direction of inventing scheduling methods customized for CSP based systems. Two insightfull ideas are presented, but more as hints for further research than as mature solutions. Therefore, a recommendation for a prospective SystemCSP design tool is to (at least in beginning) relies on the combination of proposed design patterns and classical scheduling theories.

Support for component-based development

Component-based development support is discussed in chapter 2, and introduced in chapter 3. Support for component-based development includes introducing elements for specification of components, interaction contracts and different types of ports. Key issue is the introduction of interaction contracts as abstract reusable specifications of interactions. In that way, interactions can be formal checked without the need for concrete applications. Once checked, such an abstract definition can be instantiated in any suitable application context.

Expressiveness and readability

Expressiveness and readability are important properties of any visual notation. Those issues are most discussed in chapter 5, where a set of proposed design patterns was used as examples of SystemCSP designs. Lack of expressiveness and readability is easy to observe in a visual notation. However, the level of those properties present in a visual language is not easy to measure in an objective way. This is the case because the level of those properties also depends on the thinking process of a notation user. For instance, the thinking process of a notation user can be dominantly visual or dominantly textual, the set of chosen symbols might not match the intuitive meanings in the mind of the user and so on. The way the human mind operates and handles complexities is highly individual. Therefore, the approach taken was to provide separate problem statement, design in SystemCSP and its textual explanation and set of remarks both on patterns and notation sides. In that way, every reader can evaluate expressiveness and readability.

Scalability

Scalability is related to the way readability changes with increasing complexity of the designs. One powerful way of dealing with scalability is hiding part of the design in opaque process blocks and component symbols on the current level of abstraction and specifying them in a separate view. An additional vehicle for achieving scalability in SystemCSP designs is the introduction of the interaction-oriented part of the notation. This part of the notation enables specification of interactions among set of components in isolation from the rest of the system. The same component can participate in any number of such interaction-oriented diagrams. Binary compositional relationships of the component with other participating components specify its relative position in control flow oriented view. In that way, the control flow oriented view is built incrementally through adding restrictions in interaction-oriented views. The possibility to focus in this way on different interactions of the same component in a set of different diagrams containing part of control flow relationships relevant for that interaction, is a powerful way to achieve scalability and build systems in an incremental, iterative way.

Unambiguous interpretation

This demand is covered in several ways. First, the careful definition of the notation elements in Chapter 3. Second the formalizing the structural relationships between abstractions through the definition of the metamodel of the notation (see Appendix A). Furthermore, Appendix B defines software architecture that can be used as reference model for the implementation of SystemCSP designs.

Applicability to complex control systems

This property is tested by designing software for the complex control setup in Chapter 6. It seems that SystemCSP is good vehicle to specify synchronization among participating devices and also a good way to specify interactions present in each of the devices due to the existence of several layers (loop control, sequence control, supervision and safety). This case study has showed that SystemCSP is equally capable to express designs based on time-triggered architectures, as the ones based on implicit sensor events and the ones that use explicit event based synchronization between devices. Interaction contract oriented design seems to be offer very structured way of managing complex interactions.

Conclusion

All demands listed in section 1.3.1 are supported in the notation and evaluated throughout the thesis.

7.2 Recommendations and open issues

Tooling

Obviously, a tool is needed to support SystemCSP design process. The metamodels described in Appendix A can be used as a basis of a model database in the prospective tool. Recommendation is that some metamodeling framework, e.g. the Eclipse EMF framework (Eclipse, 2007), is used for specifying metamodels. The prospective tool should provide a design editor and set of basic viewers providing support for entering SystemCSP designs. It should be able to perform code generation to software, hardware and CSPm domains. The framework proposed in Appendix B should provide the support for proper execution of the code generated for the software domain. The tool should also provide facilities for coupling with most important tools in related domains. A simulation and run-time execution framework should exist and be able to feed the obtained data back to the tool. The tool should be able to perform a visualization of the current state in control flow. Debugging should be possible through special interfaces dedicated to the interaction of tool with a SystemCSP executable.

One of the issues was whether the existing gCSP tool based on the GML notation can be reused for building interaction-oriented part of the prospective SystemCSP tool. This question was raised due to similar basic concepts of GML and interaction-oriented part of the GML.

GML uses interaction-oriented elements to specify control flow, which restricts it to essentially specifying one big single diagram possibly divided in several separate views only via containment hierarchy. SystemCSP clearly separates interaction-oriented view from control flow oriented view. In that way, the same SystemCSP component can exist in many interaction-oriented diagrams and single control flow diagram.

gCSP tool is customized towards design philosophy of GML which is, as explained above, different from the one of SystemCSP. Due to this difference, reusing the existing tool is difficult and not recommended.

Mapping to hardware domain

Providing support for code generation from SystemCSP designs to hardware domain (e.g. VHDL code) is a necessary step for a seamless software/hardware codesign method. In designing ways to implement certain SystemCSP primitives in hardware, the software framework can partly serve as a role model. Useful lessons for this approach can be learnt from Handle-C (Celoxica, 2007). Handle-C is an occam-like programming language used in hardware target domains as an alternative to VHDL, Verilog and other hardware description languages.

Distribution

The SystemCSP notation is intended to be used in distributed systems. The production cell setup is built in a way that takes into account the needs for

distribution and dynamical reconfiguration (Van den Berg, 2006), and as such it is expected to be a useful test case for implementing and studying distributed SystemCSP-based designs for complex control systems. Distribution of control of the setup will allow testing different replication mechanisms and fault tolerance strategies.

The SystemCSP software framework is expected to be able to incorporate easily commercial real-time fieldbus drivers, and to provide set of real-time enabled drivers where they are needed and not available.

In the scope of subprojects of this project, several fieldbus interconnections were tested and their properties were evaluated. In one of the subprojects, real-time Firewire driver for real-time Linux was developed (Zhang, 2005; Zhang et al., 2005). However, this work preceded the creation of SystemCSP and although rich with interactions, it was not designed in SystemCSP.

Simulation

In the scope of this research an occam-based simulation framework for networked control systems (ten Berge, 2005; ten Berge et al., 2006) was designed. The framework internally uses the network simulator of TrueTime (Henriksson and Cervin, 2003; Henriksson et al., 2005) and was able to give some insight in the influence of different fieldbus parameters on the behavior of the overall system. In this simulation framework, the assumption was used that computation times are in general negligible compared to network delay times. This was a reasonable first approximation for the simulation framework intended to evaluate different fieldbuses. However, to make the simulation framework more realistic, the framework needs to be extended to deal with execution times of computations. In addition, incompatibility of the occam-like approach and the SystemCSP approach require significant changes in structuring that framework.

Software implementation

The software implementation framework as proposed in Appendix B is at the moment of writing not fully implemented. Missing parts need to be implemented and the whole framework needs to be rigorously tested before it is used in practice.

The design of software implementation as proposed in Appendix B, does introduce decoupling of the application from the execution engines architecture. In that sense, four possible layers are distinguished: nodes, operating system threads, user level threads and function call based concurrency. Recommendation is that OS processes are added as an additional layer. The significance of this layer would be that e.g. in C++ based systems, it would allow reusing components in binary form. In that way, it would provide enhanced possibilities for dynamic reconfiguration of the system.

Usage

In addition, further testing of SystemCSP language on various practical case studies is needed to provide the necessary feedback and to improve the notation.

A Metamodels

A.1 Metamodels as basis for code generation

A purpose of creating a metamodel is making a definition of the modeling domain. Such a definition can, for instance, be used as a basis of a structured way to capture models in tool implementations. Another advantage of using a metamodel is its potential to introduce a standard, tool independent, way of data interchange between different domains and tools.

The structure of a good metamodel reflects possible structural relationships and in that way restricts the possible ways of combining instances of abstractions and relationships in a model. However, what is a good way of defining the design domain metamodel, results usually in a poor performance in implementation of models in some target domain (e.g. software or hardware implementation). Hence, the differences between a class diagram defining a design domain metamodel, and a class diagram representing its implementation in some target domain, are often unavoidable in practice.

One structured approach to perform code generation using metamodels is described in (Milicev, 2002). There, metamodels are defined for both the source domain and the target domain. Intermediate metamodels can be introduced as a way to gradually and in a structured way proceed from a source domain metamodel to a target domain metamodel. For code generation from a SystemCSP model to its C++ source code implementation, that approach would look like as in Figure A-1.

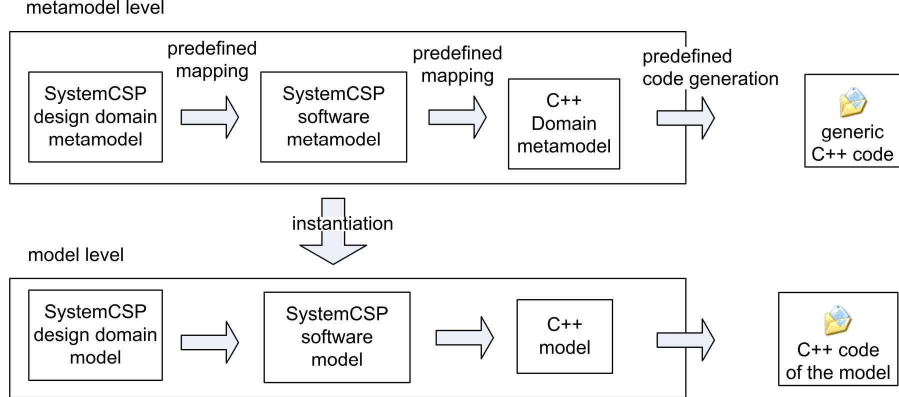


Figure A-1 Code generation process - from SystemCSP design to C++ software

Note that in Figure A-1 prior to the actual code generation, original model from the design domain is gradually transformed towards the final source code. This process takes place in stages defined via intermediate domains (SystemCSP software domain and C++ domain).

The SystemCSP design domain metamodel reflects directly the abstractions and relationships of the design space, as visible in graphical representation. However, in a practical implementation in a target domain a somewhat different set of abstractions and associated relationships is used.

The differences between design domain and software implementation models come from different requirements. While the source metamodel of the SystemCSP design domain puts focus on expressing abstractions and structural relationships as visualized in SystemCSP diagrams, the software implementation is focused on operational semantics and efficient implementation in the target domain (in this case C++). For instance, some of the abstractions are only needed during design - e.g. binary compositional relationships and also pairs of fork and join control flow elements that exist in design, are in implementation replaced with construct instances of the appropriate type. Some abstractions are less important and thus mapped to attributes of other abstractions, some new abstractions are needed to provide support for execution framework and again something that was less relevant and thus just an attribute of an abstraction in source metamodel might be represented with a standalone abstraction in the target metamodel.

The C++ domain metamodel defines abstractions representing notions in the C++ domain like classes, data members, member functions, formal parameters, association relationships and inheritance relationships. The definition of C++ domain metamodel is omitted here, since its detailed description is not relevant for the scope of this thesis. The code generation from the C++ domain metamodel to source files is hard-coded, since that part of the code generation process is not expected to change during tool lifetime.

In practice relying on a pipeline of metamodels, as the one in Figure A-1, can contribute to the flexibility of code generation process. Instead of hard-coding code generation facilities in the design domain model (which is often done in practice of tool development), decoupling between design domain and implementation domain allows for more structured customization.

The SystemCSP software metamodel is a place where the specification of target platforms and various customization options can be inserted. A customization can be e.g. specifying choice of operating system, choice of input/output devices, allocation of processes to nodes and routing of channels/events via networks and I/O interfaces.

Both design domain and implementation metamodels can contain auxiliary code for the model execution according to operational semantics of the involved elements. In this way, validation process is also divided in several steps – allowing separate validation of design and of its implementation and comparison of the results.

At the end, worth mentioning is that nowadays generic frameworks exist that provide some of the operations necessary for defining and maintaining models based on metamodels For instance, the Eclipse Modeling Framework (EMF) project (part of the Eclipse project) provides an UML-based meta-metamodel known as eCore. eCore is a generic metamodel used for defining metamodels.

eCore defines as its abstractions: classes, attributes and references and structural relationships possible between them. In this way, any metamodel can be expressed by using those basic abstractions and mapped to an appropriate UML class diagram. One can input metamodels directly in the form of an eCore model, or one can use annotated Java code, UML class diagrams, or model descriptions in XML. EMF is capable of transforming models from one of those representations to any other via the eCore model. Also, the EMF framework offers automated support for persistence to files, maintaining references and a system of notifications regarding changes in model elements. A graphical library exists that makes dealing with models easier.

A.2 Metamodel of SystemCSP design domain

In SystemCSP there are two basic types of graphical elements: nodes and connections. Nodes are for instance event ends, processes (component is a kind of process) and control flow operators. Graph connections (`GraphConnection` class in Figure A-2) are binary relationships between graph nodes (`GraphNode` class in Figure A-2), meaning that every graph connection relates exactly two graph nodes. Every node can have zero or more connections associated. Types of connections present in SystemCSP diagrams are: prefix operator (`PrefixArrow` class), interaction (or *event synchronization*) connection (`SyncConnection` class), refinement relationship (`RefinementRel` class) and binary compositional relationships (`BinaryComposRel` class). In addition, any graphical element (be it a node or a connection) can have a label (`Label` class) associated. All mentioned basic abstractions as well as described relationships between them are captured in the form of UML class diagram in Figure A-2.

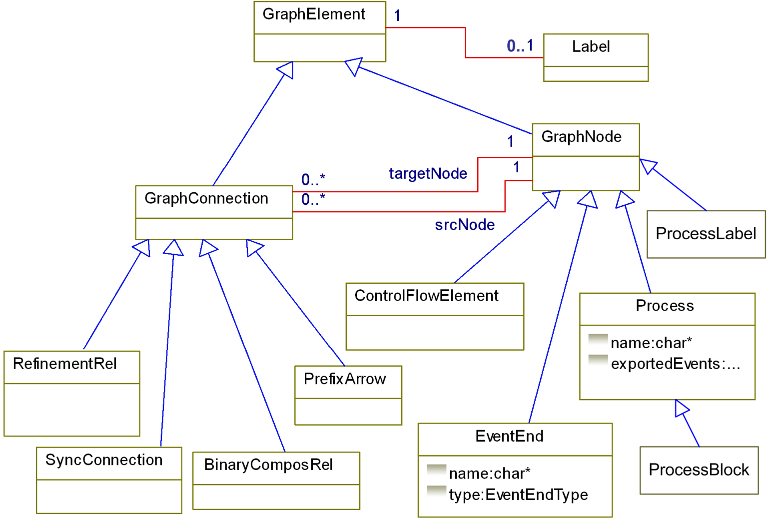


Figure A-2 Basic abstractions in SystemCSP design domain

Each connection in the graph relates two node elements. In some cases like for the prefix arrow type of connection, a connection has a direction that determines the order in which two related node elements are executed. In diagrams, the order is visible from the direction of prefix arrow. In the model, the order is represented via making distinction between source node and target node association ends.

Note that the design of the metamodel is somewhat influenced by its associated visual representation. For instance, a prefix arrow is seen as a connection that connects two node elements, although from a semantic point of view it is a control flow element equally as fork or join elements of various types.

A.2.1 Event ends

In Figure A-3 (originally appeared as Figure 3-4 in section 3.1.2), three cooperating processes are depicted.

```
CSPm:
P1 = ev1?x!y → activate_P3 → SKIP
P2 = ev1!w?z → ev2?q → SKIP
P3= activate_P3 → ev2!r → SKIP
```

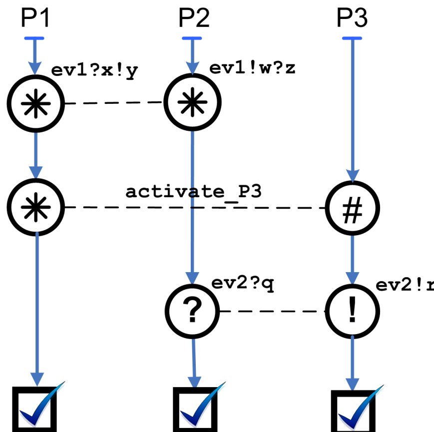


Figure A-3 Example that illustrates using event related abstractions

Let us try to identify abstractions and instances of abstractions present in this diagram.

First, there is a process entry label for every one of those processes. A choice is to create an abstraction of type `ProcessLabel`.

The process labels (see Figure A-4) can play one of the two roles: a *recursion entry point* and a *recursion process label*. The difference between those two ways of using process labels is in model made by introducing attribute ‘type’. The choice to use same abstraction for both purposes reflects the fact that the same symbol is used, and allows reusing of process labels. Process label does carry the name of the process, so a `ProcessLabel` abstraction has attribute `name`.

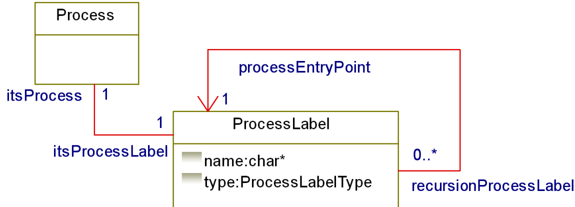


Figure A-4 Process labels

A process entry point is used to mark the named place at a control flow that can be used as a point to return to in a recursion. A recursion process label is a point in the control flow where a jump to the recursion entry point is made. As a consequence, for each recursion entry point, there can be any number (zero or more) of recursion point labels associated and for every recursion process label there can be exactly one recursion entry point to point to. Since `ProcessLabel` is a kind of `GraphNode`, it can be related to other `GraphNode` elements via `GraphConnection` elements (i.e. in this case via `PrefixArrow` elements).

SystemCSP diagram given in Figure A-3 contains set of different types of event end objects. So we define a class `EventEnd` with the attribute `name`. Event is in diagrams defined implicitly by the existence of more then one event-end with the same name. Thus, another class would be `Event`. The event ends can be of different types (INITIATOR, ACCEPTOR, READER, WRITER) so the event end also has the attribute `type`. One can further observe that on Figure A-3, event ends are related via dashed line connections. The abstractions representing that type of connections is in Figure A-2 already defined as `SyncConnection`. Furthermore, events 'ev1' and 'ev2' in the example from Figure A-3 are in fact channels used for data communication. Thus, a `Channel` abstraction is necessary as a special kind of `Event` abstraction, and a `ChannelEnd` abstraction as a special kind of a `EventEnd` abstraction. Next, a way is needed to specify set of communication flows on each channel. To make things configurable and flexible we choose to make an abstraction `DataFlow` and to chain such abstractions when needed. Further, the direction of every instance of the `DataFlow` abstraction can be set as either input or output, so we introduce an attribute 'direction'. Every instance of the `DataFlow` abstraction is associated with some component variable from which/to which it reads/writes data. Thus, an abstraction representing component variable is needed as well.

Note also that in accordance with the part of design domain metamodel defined in Figure A-2, a label can be associated with any graphic element. For instance, labels displaying channel names and data flows are associated in Figure A-3 with channel ends, while the label activate_P3 is associated with a `SyncConnection` object and not with event ends.

Instances of predefined EXIT event-ends (equivalent to specifying a SKIP process in CSP) are used for successful termination of processes P1 and P2 and P3.

Figure A-5 illustrates part of the object diagram that captures the instances and

structural relationships of abstractions as needed for the implementation of the model for the design given in Figure A-3.

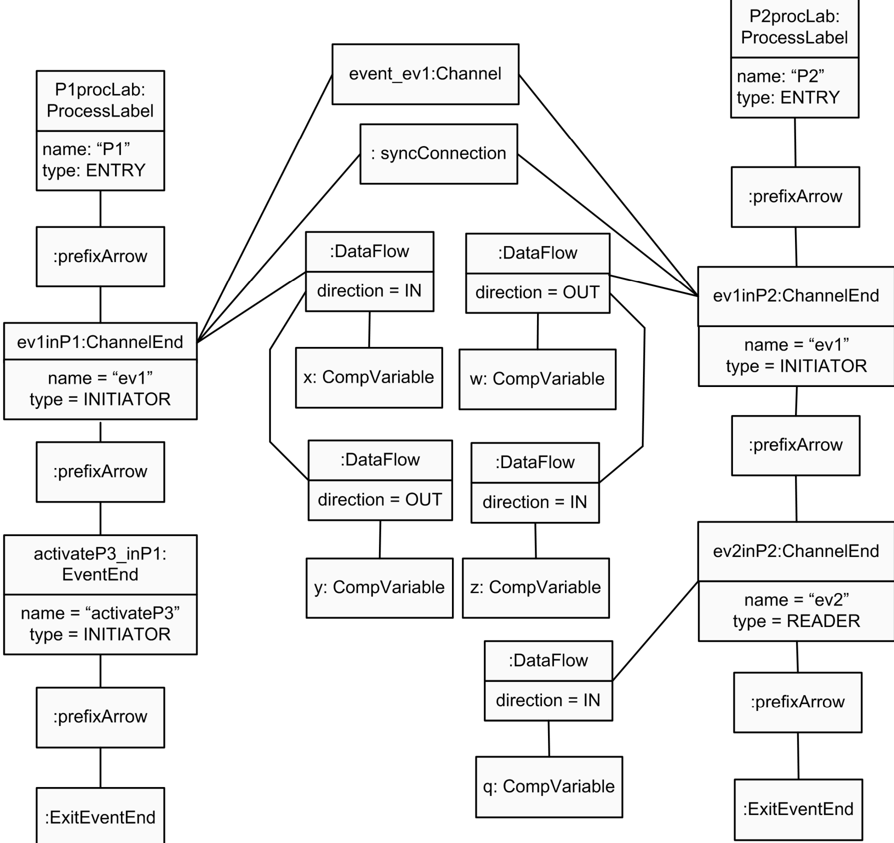


Figure A-5 Part of the object diagram capturing an example of the previous figure

Figure A-6 introduces class diagram that captures the abstractions and associated structural relationships derived from the previous reasoning. Any event-end can be associated to exactly one event, while each event has associated at least one event-end. This is in class diagram in Figure A-6 specified via multiplicity numbers on the ends of the association relating the `Event` abstraction with the `EventEnd` abstraction. A Channel-end is a special type of event-end that differs from an event-end by allowing the possibility for data communication. Every channel is related to exactly two channel-ends, and every channel-end has exactly one channel associated. Again, this is in class diagram expressed by specifying multiplicity numbers on the association ends. Data communication might consist of several data flows, and any particular data flow can either perform input of data or output of data. The direction of a data flow is determined by the `direction` attribute. In channel-end, a data flow from/to channel is always related to exactly one variable and for every variable there is zero or more related data flows.

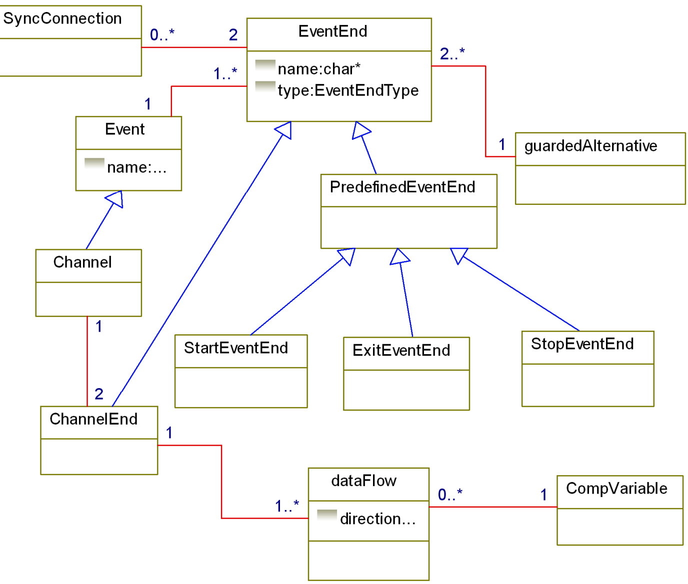


Figure A-6 Event related abstractions of the metamodel

A.2.2 Control flow elements

Elements with direct mapping to CSP

Figure A-7 is an example which contains a code block, 9 prefix arrows, 4 event-ends (1 predefined start event-end and 3 channels each with associated instance of the data flow entity), 5 process label instances (2 recursion entry points and 3 recursion process labels). Existence of the component-level variable `count` is also implied. The new elements compared to previous diagrams are: an instance of the IF control flow element and an instance of the guarded alternative control flow element.

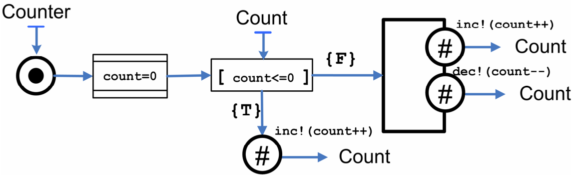


Figure A-7 Example that illustrates using control flow elements

Figure A-8 depicts a part of the metamodel introducing several control flow elements, some of them needed in design for making a model of the design from Figure A-7. IF control flow element allows conditional branching of control flow. In case the given condition evaluates to true, then the control flow branch (prefix arrow leading from this IF element to some other graphic node element) associated with the true value is followed, otherwise the other branch is followed. Note that

although the ‘conditions’ for IF control flow elements are expressions that evaluate to boolean, in the model belonging to design domain they are captured as strings. The *switch* control flow element is a generalization of the IF control flow element, allowing multiple switch conditions and associated branches. Every condition variable is associated with a branch to be followed when condition is satisfied.

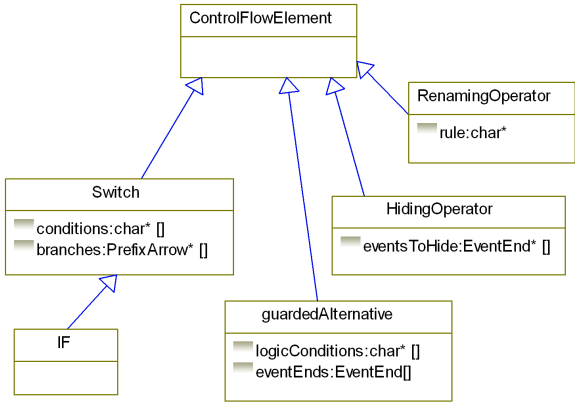


Figure A-8 Some control flow elements

Abstraction representing a guarded alternative contains data members for guarded event ends and for associated logical conditions. The guarded alternative control flow element participates in the synchronization mechanism on the behalf of the event-ends it is guarding.

The renaming operator specifies renaming rules as a set of strings containing pairs of old and new names. The hiding operator maintains the list of events that are hidden, i.e. not exported to a higher level of abstraction in the process hierarchy.

Start/exit control flow elements

In the example given in Figure A-9 (appeared as Figure 3-15 in section 3.2.1, we can first identify elements already mapped to the previously introduced abstractions: process label P1, process blocks P, T, Q and R, event-end ‘ev1’ and prefix arrow elements. In addition in this figure, some pairs of start/exit control flow elements appear. We can observe two pairs of START SEQ and STOP SEQ and one pair of FORK PAR and JOIN PAR control flow elements.

In this diagram instead of real nodes representing START SEQ and STOP SEQ control flow elements, the abbreviated form is used by associating their symbols with prefix arrows. In that way, instead of one block and two prefix arrows, only single prefix arrow is depicted, saving the space. In model however, all those elements do exist, they just have a special flag (‘hidden’) set to avoid visualization. Thus in this example there are in total more prefix arrow objects in model then visible in diagram.

P1= ev1 → (P;(T;(Q‖R)))

P1 ev1 P T Q R

Figure A-9 Example illustrating usage of abstractions related to basic CSP operators

Figure A-10 illustrates the FORK and JOIN control flow elements elements related to basic CSP operators. It also illustrates the fact that every pair of FORK and JOIN elements is in fact making a construct. The area with constructs is shaded, because it is not directly visible on diagrams.

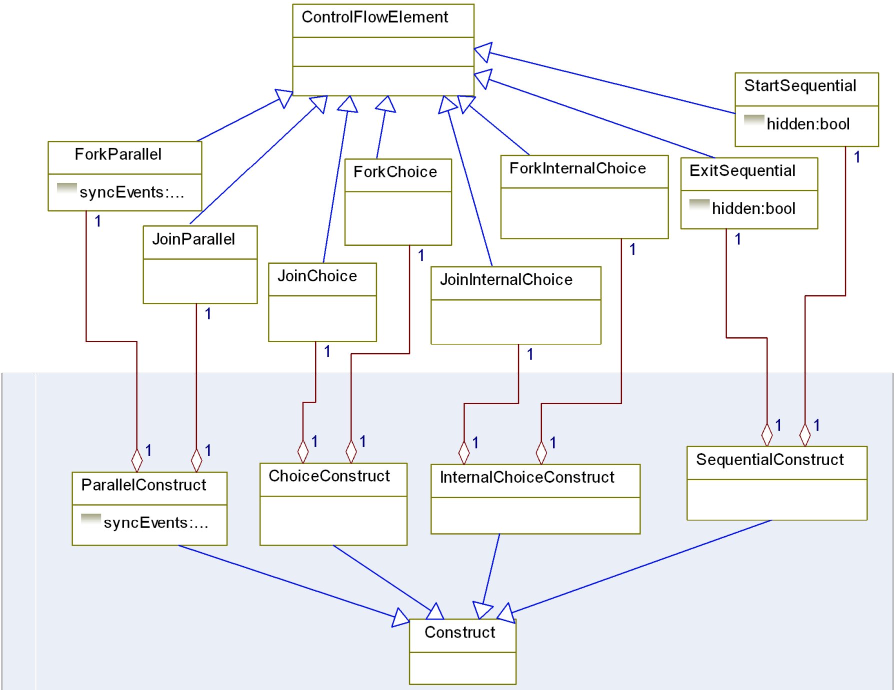


Figure A-10 Abstractions of control flow elements related to basic CSP operators

A pair of fork and join control flow elements of the same type, as the ones given in Figure A-9, maps to a basic CSP operator (Parallel, sequential, external, internal choice). Where in CSP expressions a scope of the operator is defined by brackets, in control flow based SystemCSP diagrams, instead of operator with scope limited by brackets, a pair of FORK and JOIN elements exists. Since in practice each fork/join pair is representing a single CSP operator, fork and join elements making a pair are related via the `Construct` abstraction of the appropriate type. `Construct` abstraction is not visualized directly on diagrams. However, it is essential abstraction for resolving grouping of fork and join elements and also for grouping binary relationships. In this way special types of constructs for each CSP

operator are introduced that contain exactly one fork and exactly one join element (see the multiplicity of the association). In addition, all types of constructs are types of a generic abstraction named `Construct`. `Parallel construct` needs to keep track of events on which its subprocesses do synchronize. From that reason, `ParallelConstruct` abstraction contains attribute `syncEvents` representing the list of events on which its subprocesses do synchronize.

Advanced fork/join control flow elements

Same functionality is in left side of the Figure A-11 specified via take-over operator and on the right side via timed interrupt operator.

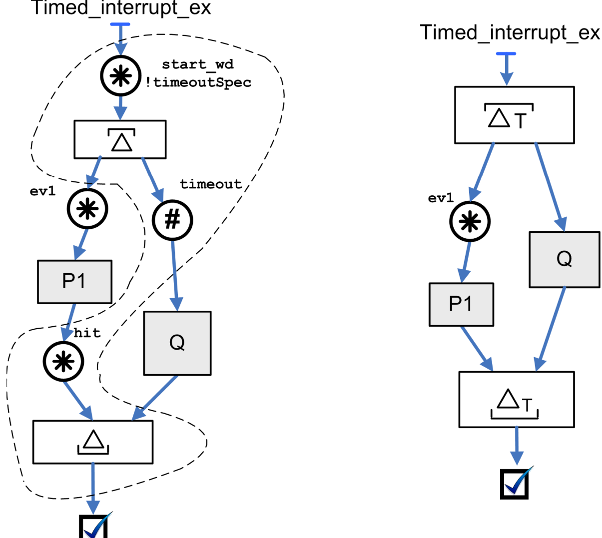


Figure A-11 Time related operators

Timed interrupt operator and timeout operator are built on top of take-over operator and timing subsystem supporting watchdog interaction contract, as explained in section 4.1.3. The question is how to represent those compound operators in the model. Left-hand side of Figure A-11 defines four instances of event-end abstraction, two process blocks, a pair of fork and join take-over operator, process entry label, prefix arrows and EXIT event-end. The one on right-hand side has only one instance of event-end abstraction and has a pair of instances of fork and join timed interrupt operator.

The model of timed interrupt should, however, allow for expansion from abbreviated form as in right-hand side of the Figure A-11 to the expanded one as the one on left-hand side of the Figure A-11. Thus, in case when abbreviated form based on derived time related operators (timeout operator and timed interrupt operator) are used, the associated model will need to have specified all data necessary to perform expansion to the equivalent representation based only on basic untimed operators. In model, the data needed for this transformation is kept in data members of instances of fork operator abstractions. Except for the model

transformation, this data is for instance needed when converting design to CSPm script.

Figure A-12 captures abstractions related to control flow elements of the SystemCSP design domain related to the take-over operator (equivalent to interrupt operator of CSP) and time related operators. Those operators are in SystemCSP diagrams represented by an appropriate pair of fork and join symbols. In the model, in addition, for each pair of fork and join operators, there is an instance of appropriate type of construct.

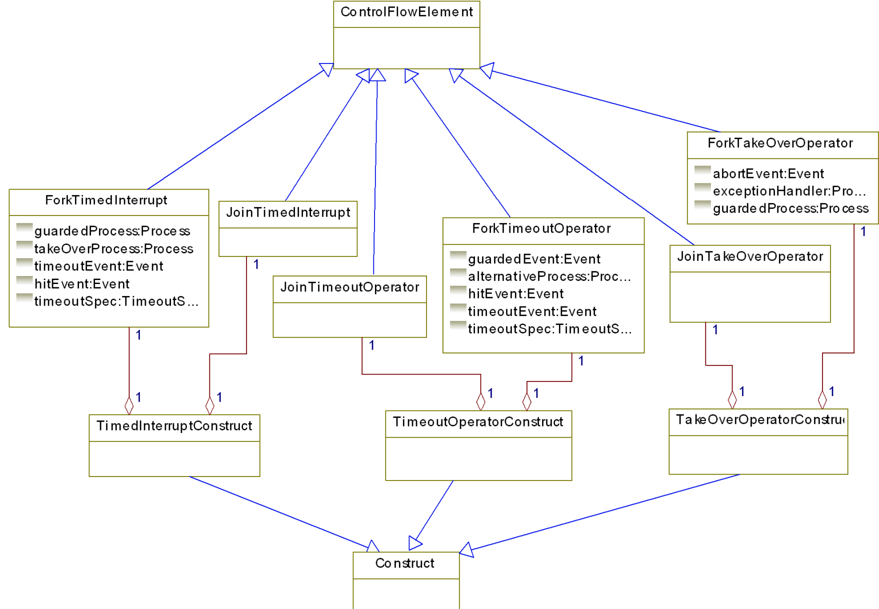


Figure A-12 Abstractions related to take-over and timed based operators

Fork control flow elements from Figure A-12 contain as properties their operands and additional elements needed for expanding the compound operator into a set of basic lower-level elements. For the takeover operator important parameters are guardedProcess, exceptionHandlerProcess and the event (abort event) that is used to initiate the take-over.

Part of timeout operator specification are its operands – the guarded event and the alternative process that will be executed in case that timeout expires before guarded event takes place.

Timed interrupt operator has two operands - the guarded process, and the take-over process which will takeover the execution of the guarded process when timeout expires.

For both time based operators, timeout specification needs to be specified. Due to building those operators on top of the watchdog design pattern, the `hit` and

`timeout` events are used as points of connection to the associated watchdog and underlying timing system.

A.2.3 Binary compositional relationships

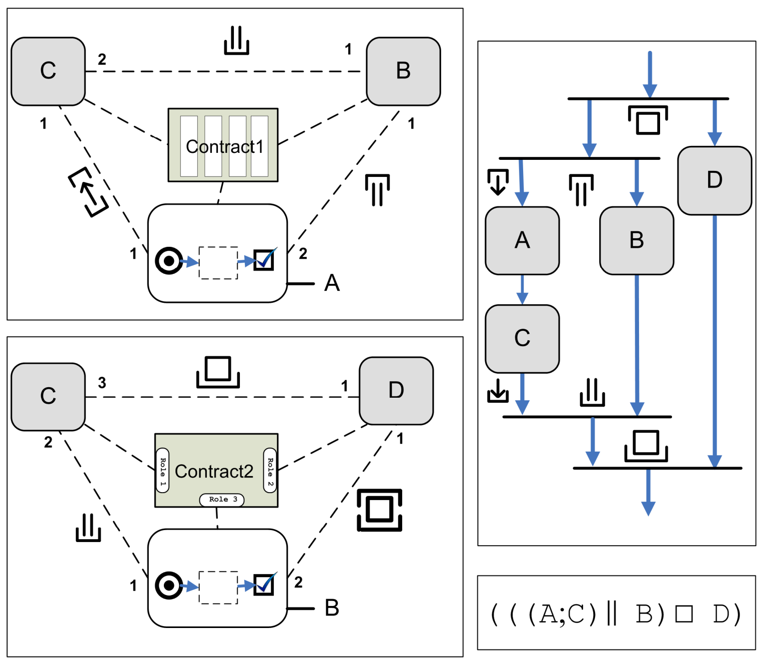


Figure A-13 Example of using binary compositional relationships

In Figure A-13 (repeated from section 3.4.2) two interaction-oriented views are depicted on left hand side and their control flow representation is depicted on the right-hand side. In interaction-oriented view, components are related via binary compositional relationships. Note that as depicted on the right-hand side of this figure, in finalized designs all binary compositional relationships are resolved into control flow elements and thus into appropriate constructs.

Figure A-14 illustrates the relation between binary relationships belonging to the domain of data flow oriented view of SystemCSP (a part of SystemCSP that origins in GML) and the appropriate construct abstractions as existing in final designs. Additional attribute defines the strength of the relationship, that is whether the relationship is of type: WEAK, START, EXIT or STRONG.

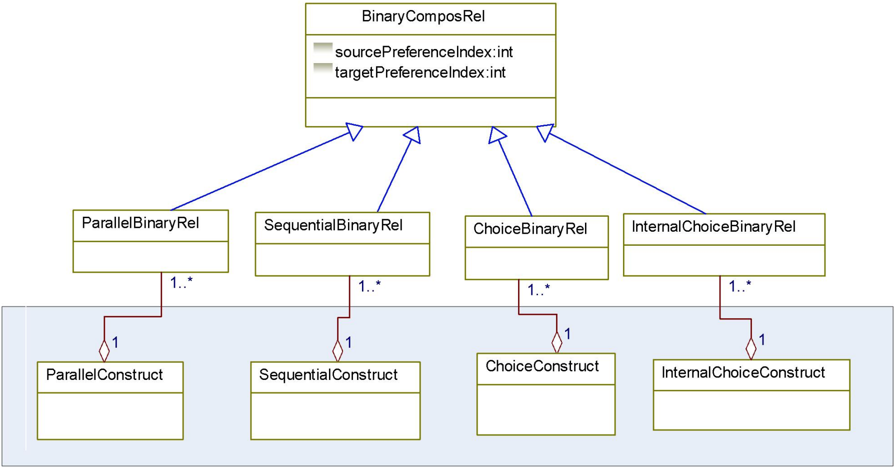


Figure A-14 Abstractions defined for binary compositional relationships of SystemCSP

A.2.4 Components and processes

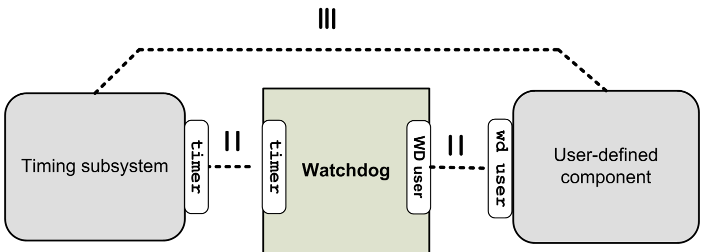


Figure A-15 Example with binary relationships and abstractions related to components

In the system depicted in Figure A-15, a user-defined component interacts with timing subsystem via the watchdog interaction contract. Role specifications and role implementation ports are created with names 'timer' and 'wd user'. Role implementations are associated with appropriate component instances, while role specification ports are related with the interaction contract. Compositional binary relationships of types parallel and interleaving parallel are specified between participating instances of components and interaction contract.

Figure A-16 introduces abstractions related to the part of SystemCSP notation related to components.

Basic unit of composition in CSP and SystemCSP is a process. In SystemCSP in addition, several special kinds of processes exist: process blocks, non-interacting processes, code blocks, constructs and components.

Non-interacting process is a special kind of process block that does not interact via events with its environment. In general case, it can internally contain subprocesses and event synchronizations. It has a symbol somewhat different then the one of

process block. A Code block is a special kind of non-interacting process that contains only pure computation code without any event synchronizations internally or externally.

Construct is a kind of process that composes involved subprocesses via one of the CSP operators and is object of further composition.

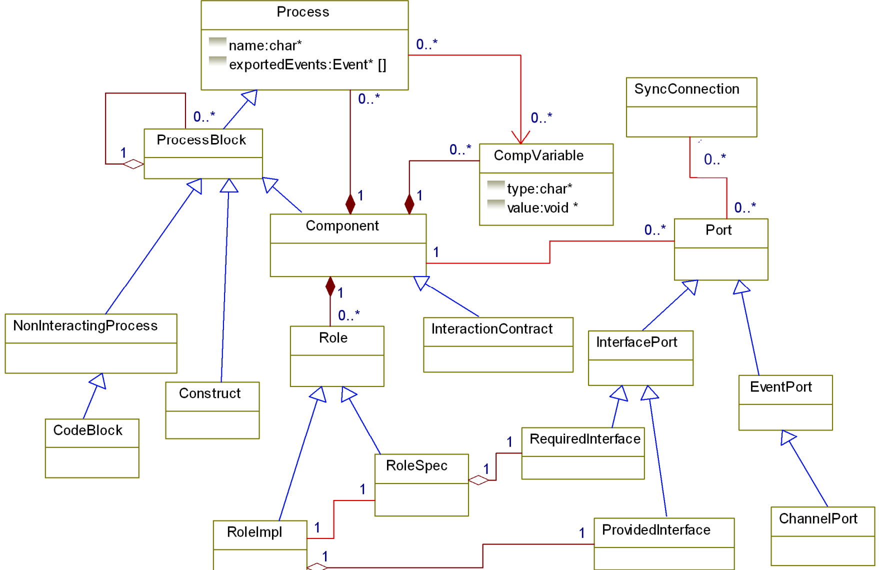


Figure A-16 Abstractions related to component based software engineering

Component is a kind of process that can own zero or more (note the multiplicity of the related "part-of" relationship) other processes (including process blocks, constructs and other components). Interaction contract is a special kind of component that is dedicated to managing interaction among some other components.

Variables defined on the level of the component (`CompVariable` class) are part of SystemCSP model. Those variables are owned by component and used by its subprocesses. Process blocks can in addition specify additional variables that are part of their description and not directly reflected with dedicated abstraction in the metamodel.

Every component is in contact with the rest of the world via access points known as ports. A port is in fact related to some interface (e.g. of a role or a function), or to some event/channel. Hence a difference is made between interface ports and event ports.Two kinds of interface ports exist: required interface and provided interface.

Both interaction contract and component can have both kinds of roles/ports. In interaction contracts, however, role specification and associated required interface

type of port is more common due to the function of interaction contract to relate components that provide different services.

A.2.5 Supervision elements

Figure A-17 displays abstractions related to the supervision of an application. As explained in chapter 3, associating supervision elements (logging and tracing points) with prefix arrow elements, allow us to view all supervision elements as a separate layer, which is normally hidden in visual representation, but can be displayed if required.

Every supervision point has ID unique in the scope of its parent component. Debugging point is used to specify points when the execution of the debugged application is halted. A logging point is a point in control flow where logging of the timestamp and of the chosen component's variables is performed. A tracing point reports only timestamp and ID and is used to track whether certain point in control flow was reached in actual execution and when.

For the point of view of logging, it is possible to define bit field that specifies which component variables are logged and which are not. If logging point does not specify such a bit field, then the default one defined on the level of the component is used. Note that properties like timestamp for supervision point and value for component variable are essentially implementation and not design issues. However, we make them part of the design domain metamodel in order to provide support for displaying data obtained as a feedback from the application executing on target.

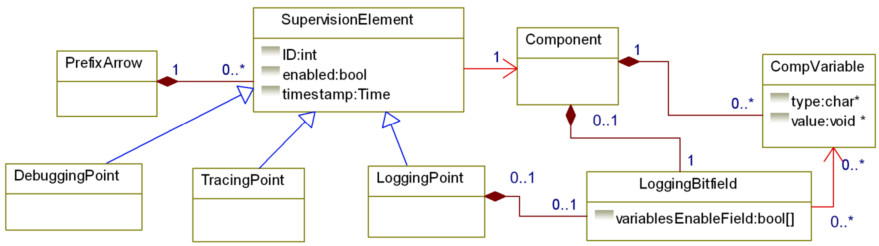


Figure A-17 Abstractions related to application supervision elements

B Software framework design

Section B.1 starts with a discussion focused on the possibility to reuse the CT library, developed at our lab, as a target domain framework for code generation. After discarding the possibility to reuse the CT library, the discussion about the basic design principles for a new library starts in Section B.2 with investigating practical possibilities for implementing concurrency. A flexible architecture is proposed that allows a designer to make trade-offs regarding the used structure of execution engines. A design of component internals is introduced, that allows subprocesses to access variables defined in parent components and offers a way to reuse processes in same way as components. Section further explains the way in which function-call based concurrency is applied to structure concurrency inside components. An example is given illustrating how this mechanism actually works.

Section B.3 explains a synchronization mechanism designed to handle CSP kind of events with any number of participants and with some of them possibly participating in several guarded alternative constructs. A special problem that was solved related to this was achieving mutual exclusion when event ends and the associated synchronization points are potentially scattered in different operating system threads or on different nodes.

Section B.4 introduces design of a mechanism that implements exception handling and of mechanisms that provide support for logging and tracing.

B.1 Why yet another CSP library?

In this section, we focus on a possibility to reuse the CT library, the occam-like library developed in our lab, as a framework for the software implementation of SystemCSP models. The CT library follows the occam model as far as possible. SystemCSP builds upon the CSP legacy. It does in addition introduce new elements related to the area of component-based engineering. However, those newly introduced elements are: 1) components and interaction contracts that both map to CSP processes and 2) ports that are just event-ends exported by such CSP processes.

In fact, SystemCSP defines auxiliary design time operators like the fork and join control flow elements and binary compositional relationships of FORK, JOIN, WEAK and STRONG types. Those auxiliary operators do exist only during the design process and are therefore after grouping, in mapping to CSPm target domain substituted with CSP operators, and in mapping to software implementations with constructs like the ones existing in occam and CT library.

Basic SystemCSP control flow elements and binary relationships do map to the constructs as it is the case in the CT library. However, since SystemCSP aims to correspond exactly to CSP, it cannot be implemented completely by occam-like approaches that do put only restricted part of CSP into practical use. Following text will explore those differences in more details.

In the CT library, like in its role-model occam, a `Parallel` construct spawns separate user-level threads for every subprocess. Synchronization points are defined by channel interconnections. The SystemCSP design domain allows both the CSP way of event synchronization (through a hierarchy of processes), and the occam-way with direct channel interconnections. Thus, a software implementation of SystemCSP designs needs mechanism for the hierarchical CSP way of event synchronization.

In SystemCSP, as in CSP, data communication over a channel can be multidirectional involving any number of data flows. The CT library, as occam, has only unidirectional channels. In addition, those channels are strongly typed using the template mechanism of the C++ language and as a consequence, they are not flexible enough to be reused in constructing the support for multidirectional communication. Thus, the channel framework of the CT library is not reusable.

The CT library implements the Alternative construct as a class whose behavior is based on the ideas of the occam ALT construct. The implementation of the Alternative construct (Orlic and Broenink, 2004) allows several different working modes (preference alting, PriAlternative, fair, FIFO), introduced to enable an alternative way to make a deterministic choice in case when more then one alternatives are ready for execution at the same time. The alting in CT library assumes that a channel can be guarded by some alternative construct only from one of the exactly two event-end sides (there can be either an input or an output guard associated with a channel). A guarded channel is just a channel with an associated guard. A guard is an object inside an alternative construct associated with a channel and a process. When a guarded channel is accessed by the peer process, the guard becomes ready and is added to the alting queue. The way in which guards are ordered in this queue, determines the working mode (preference alting, PriAlternative, fair, FIFO) of the alternative construct. An alternative construct is thus a single point where the decision of a choice is made.

The SystemCSP design domain makes a difference between an external choice and a guarded alternative operator and in that sense adheres strictly to CSP. Thus, an implementation is needed that can support both. Event-ends contained by a guarded alternative or the ones resolving the parent external choice operator need to delegate their roles in the process of CSP event synchronization to the related guarded alternative or external choice operator. In case when, in an event occurrence, any number of guarded event-ends can participate, the whole alting mechanism must be completely different then the one applied in CT library. This means that in fact for CSP event synchronization mechanism completely different implementation of alting needs to be implemented. Thus again in this respect too, the CT library is not useful.

Simple CSP processes, made out of only event synchronization points connected via the prefix and the guarded alternative operator, are often visualized using a *Finite State Machine* (FSM). With the guarded alternative of CSP, no join of branches is assumed, and the branches can lead to any other state. The occam/CT library choice (ALT construct) requires that all alternatives are eventually joined. Thus a natural FSM interpretation is not possible anymore. For SystemCSP, the

ability to implement FSM-like designs in a native way is especially important. Thus, implementation of the guarded alternative operator should not assume the join of branches.

In addition, it should be possible to use process labels to mark process entry points and allow recursions other then repetitions as in the SystemCSP design domain. Since in occam and the CT library, processes are structural units like components in SystemCSP, the use of recursion different then a loop is not natural there. A strict tree hierarchy of processes and constructs as basic architecture design pattern of occam and CT library is a misfit for our purpose. Thus, again the CT library does not meet the requirements imposed by SystemCSP.

In fact, instead of processes as structural units arranged in strict tree hierarchy, flexibility can be introduced by using classes for implementation of some processes (i.e. process blocks and components) and functions and labels for other processes. For instance, a single FSM-like design can contain many processes that in fact do only name the relevant points in control flow. Certainly, those processes cannot map to the occam notion of process. They are more convenient to be implemented as labels, while the whole finite state machine is convenient to implement inside a single function.

In addition, SystemCSP is intended to be used as a design methodology for design and implementation of component-based systems. This needs to be supported by introducing appropriate abstractions and also possibilities for easy reconfiguration, interface checking, and so on.

To conclude, the mismatch between the CT library and the needs of SystemCSP is to big to allow reusing the CT library as a framework for the software implementation of SystemCSP designs.

B.2 Execution engine framework

Brief overview of execution engines

Concurrency in a particular application assumes the potential for parallel existence and parallel progress of the involved processes. If processes are implemented in hardware, or if each of the processes is deployed on a dedicated node, these processes can truly progress concurrently. In practice, multiple processes often share the same processing unit.

Operating systems provide users with the possibility to run multiple OS processes (programs). Every OS process has its own isolated memory space and its own set of allocated resources. Within OS processes it is possible to create multiple OS threads that have their own dedicated workspaces (stack), but share other resources with all threads belonging to the same process. Synchronization in accessing those resources is left to the programmer. OS synchronization and communication primitives (semaphores, locks, mutexes, signals, mailboxes, pipes…)(Tanenbaum, 2001) are not safe from concurrency related hazards caused by bad design. OS

thread context switch is heavyweight, due to allowing preemption to take place at any moment of time.

User-level threading is an alternative approach that relies on creating a set of own threads in the scope of a single OS thread. Those threads are invisible to the underlying OS-level scheduler and their scheduling is under the control of the application. The main advantages compared to OS threads are the speed of context switching and gaining control over scheduling. The use of Operating System calls from inside any user-level thread is blocking the complete OS thread with all nested user-level threads (blocking operating system call problem).

Another approach is to implement concurrency via function-calls, where the concurrent progress of parallel processes is achieved by dividing every process into little atomic steps. After every atomic step, the scheduler gets back control and executes the function that performs the next atomic step in one of the processes. There is no need to dedicate a separate stack for every process. Steps are executed atomically and cannot be preempted. A function-calls based approach is often used to mimic concurrency in simulation engines. There is even an operating system (Portos (Chrabieh, 2005)) that is based on scheduling prioritized function calls.

Discussion

SystemCSP structures concurrency, communication and synchronization using primitives directly coupled to appropriate CSP operators. To implement concurrent behavior, it is possible to use any of the previously described approaches.

The CT library is based on user-level threading. Every process in the CT library that can be run concurrently (i.e. every subprocess of the (Pri)Parallel construct) has a dedicated user-level thread. A scheduler exists that can choose the next process to execute according to the hierarchy of Parallel/PriParallel constructs. As in occam, rendezvous channels are the basic communication and synchronization primitives. Possible context switching points are hidden in every access to local channels.

The first important issue related to the SystemCSP framework is what type of execution engine is best to choose. Actually, the optimal choice depends on the application at hand and is a compromise between the level of concurrency, the communication overhead and other factors. The best solution is, therefore, to let the designer choose the type(s) of execution engines on which the application will execute. A way to do this is to separate the application from the execution engines, and to let the designer map the components of his application to the underlying architecture of execution engines.

Four- layer execution engine architecture

An application is in SystemCSP organized as a containment hierarchy of components and processes. A component is the basic unit of composition, allocation, scheduling and reconfiguration. Inside every component, contained components, processes and event-ends are related via CSP control flow elements

(sequential, parallel, choice …). While a subprocess is inseparable part of its parent component, a subcomponent is independent and can for example be located on some other node.

As a result of the previous discussion, flexible execution engine architecture is proposed, that allows the user to adjust the level of concurrency to the needs of the application at hand. The execution engine architecture is hierarchical, based on four layers: node/OS Thread/UL thread/component managers. Any component can be assigned to any execution engine on any level in such a hierarchy.

The class diagram given in Figure B-1 defines the hierarchy of the execution engines. In the general case, inside an operating-system thread, a user-level scheduler exists, which can switch context between its nested user-level threads. Inside a user-level thread is, in the general case, a component manager that can switch between the contained components. Every component has an internal scheduler that will use a function-call based concurrency approach to schedule nested subprocesses.

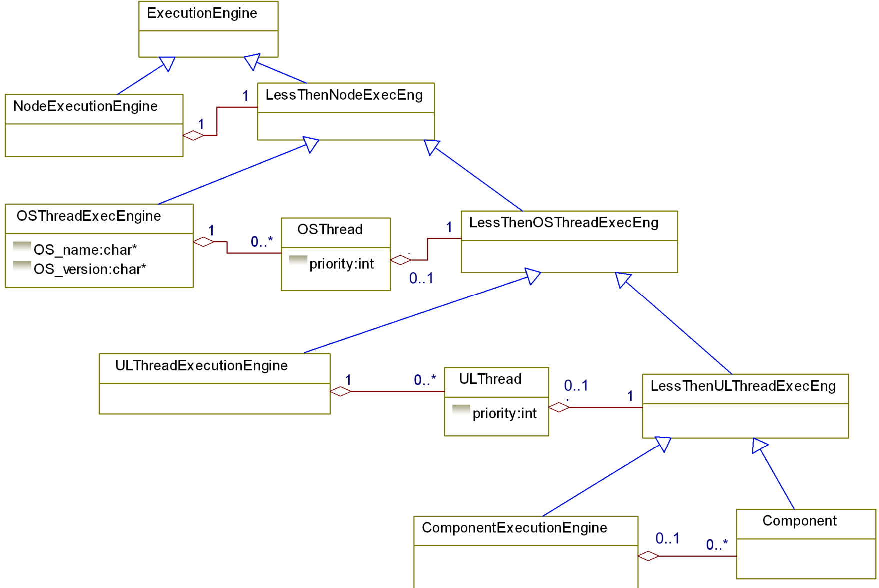


Figure B-1 Class diagram of the 4-layer execution engine framework

Internalizing the scheduler inside every component allows more flexibility in the sense that some levels in the 4-layer architecture can be skipped. The concurrency of the node execution engine can be delegated to operating system threads or to user level threads or to component managers or it can execute a single component directly without providing support for lower-level execution engines. It is even possible to have a single component per node. Similarly operating system threads can execute a set of user level threads, or a component manager or a single component. A user-level thread is able execute just a single component or a set of

components via the component manager. The possibility to choose any of those combinations is actually reflected in Figure B-1.

The OS thread execution engine is in fact representing the scheduling mechanism of the underlying operating system. Therefore, in the design domain this class contains the name and version number of the used operating system as attributes. In software implementation, there is no matching class since implementation is provided by the underlying operating system. The OS thread class in the software implementation domain does have a dedicated subclass for every supported operating system. In that way, the portability is enhanced by isolating platform-specific details in the implementation of subclasses. Auxiliary abstract classes LessThenodeExecEng, LessThenOSThreadExecEng and LessThenUL-ThreadExecEng are.introduced to enable the described flexibility in structuring the hierarchy of execution engines.

Allocation

An allocation procedure as the one depicted in Figure B-2 (below here), is a process of mapping components from the application hierarchy of components to the hierarchy of execution engines.

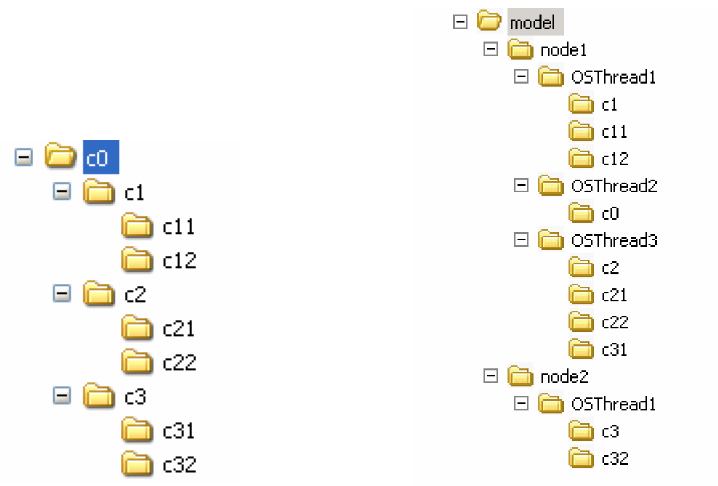


Figure B-2 Allocation = mapping from application to execution engines

The criteria for the choice of the execution framework and for the allocation, is setting the proper level of concurrency while optimizing performance by minimizing overhead. Two components residing on different nodes can execute simultaneously. Two components allocated to the same node, but to different operating system threads can be executed simultaneously only in the case of multi-core or hyper-threading nodes. Communication overhead between two components is directly proportional to the distance between the execution engines that execute them.

Control flow (as specified by parallel, sequential and alternative constructs) is decoupled from its execution engines. As a result, components can be reconfigured more easily. A component can be moved from one (node, operating-system thread, user-level thread) execution engine to another. Components can be dynamically created, moved around and connected to interaction contracts. On dynamical reconfiguration, checking compatibility of the interface required by the interaction contract with the interface supported by the component is done.

Priority assignment

CSP is ignorant of the way concurrency is implemented. Concurrency phenomena involving parallel processes interacting via rendezvous synchronizations are the same regardless whether concurrent processes are executed on dedicated nodes, or sharing CPU time of the same node is done according to some scheduling algorithm. However, temporal characteristics are different in these two cases. The most commonly applied scheduling schemes are based on associating priorities with processes. In real-time systems, achieving proper temporal behavior is of utmost interest. Therefore, in real-time systems priorities are attached to schedulable units according to some scheduling algorithm that can guarantee meeting time requirements.

In addition to the PAR (parallel) construct, in occam a prioritized version of the parallel construct, the PRIPAR construct, was introduced. It specifies parallel execution with priorities assigned according to the order of adding subprocesses to the construct. However, on transputer platforms only two priority levels were supported. Additional priority levels were sometimes implemented in software (Sunter, 1994).

Following occam, the CT library introduces a PriParallel construct with the difference that inside one PriParallel up to 8 subprocesses can be placed. While all subprocesses of a Parallel construct have the same priority, priorities of processes inside a PriParallel are based on the order in which they are added to the construct. This allows for a user-friendly priority assignment based on the notion of the, more or less intuitive, relative importance of a process compared to the other processes. The `PriParallel` construct is as any other construct also a kind of process, and as such it can be further nested in a hierarchy of constructs. This leads to the possibility to use a hierarchy of `PriParallel` and `Parallel` constructs to create a program with an unbounded number of different priority levels. Note however, that priority ordering, of all processes in a system, if defined in this way is not necessarily a strict ordering, but rather a set of partial orderings. If only PriParallel constructs were used, a set of partial orderings results in global strict priority ordering.

As in execution-engine architecture issues, where the conclusion was that flexibility can be achieved by separating hierarchy of components belonging to the application domain, from the hierarchy of execution engines, the similar reasoning applies to specifying priorities. The PriParallel construct of the occam-like approaches is hard-coding priorities in the design, where a intuitively priority assignment is related to the execution of processes on the real target architecture.

Priority values are in fact the result of a trade-off due to temporal requirements that belong to the application domain and processing time that belongs to the domain of underlying architecture engines. Therefore, the choice is *not* to follow the occam-like approach. Priorities belong to the execution engine framework and not to the application framework. Instead of relative priorities in each Parallel construct, a component from application hierarchy of components can be mapped to the execution engine of appropriate priority.

Every operating-system thread has a priority level used by the underlying operating-system scheduler to schedule it. Every user-level thread has its own priority level which defines its importance compared to the other user-level threads belonging to the same operating-system thread. In this way, a 2-level priority system exists and any component can be assigned to the pair of operating-system thread and user-level thread with appropriate priority levels

Note that the priorities specified on higher levels in an execution engine hierarchy overrule the ones specified on lower levels. This is the case because a higher-level execution engine (an operating-system execution engine) is not aware of the lower-level schedulable units (e.g. a user-level thread).

A problematic situation occurs when two components of different user-level thread priorities are allocated to two different operating-system threads of the same operating-system thread priority. In that case, it can happen that advantage is given to the component that has a lower user-level thread priority. In case when such a scenario should be avoided, two components with the same operating-system thread priority should always be in the same operating-system thread. In other words, this problem is avoided when there are no operating-system threads of the same priority on one node.

An additional issue is priority inversion that happens when a component of higher priority interacts with one of lower priority via rendezvous channels. Chapter 4 deals with those issues in more details.

Components, processes and variables

The UML class diagram in Figure B-3 illustrates the hierarchy of classes related to the internal organization of components. Every component has an internal scheduler that can handle various schedulable units (construct, processes, guarded alternative operators and event ends).

Variables are in SystemCSP defined in the scope of the component they reside in, and should be easily accessible from subprocesses of that component. A subprocess is allowed to access the variables defined in its parent component, but subcomponent cannot – because a subcomponent can be executed in a different operating-system thread or even on a different node. Instead of defining actual variables, the process class does define references to these variables (see Figure B-3). Those references are in the constructor of the process associated with real variables defined in the scope of the component. In this way, subprocesses can access variables defined in components without restrictions; Component

definitions are divided into smaller parts that are easier to understand and processes become as reusable as components are.

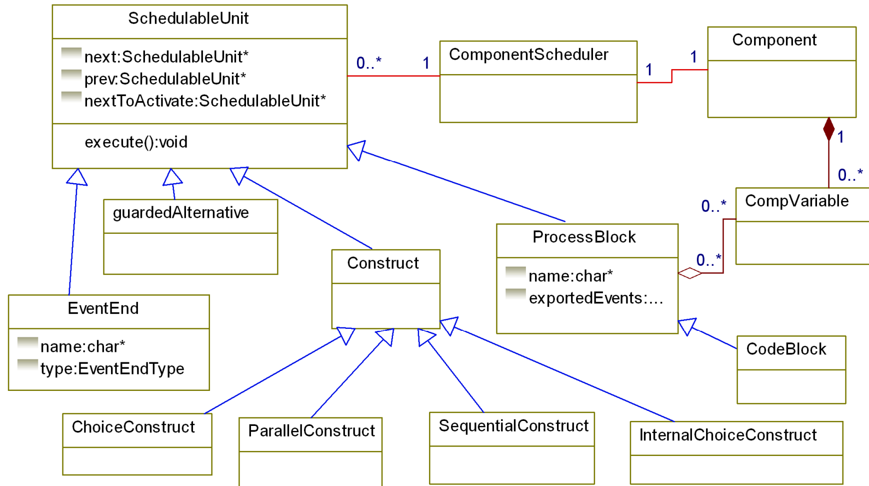


Figure B-3 UML class diagram illustrating scheduling units

Subcomponents that are executed in different execution engines do have associated proxy subporocess in their parent component (see Figure B-4). In that way, the synchronization between the remote subcomponent and its parent component is done indirectly via that proxy process. The Proxy process and remote subcomponent synchronize on start events and termination events via regular channels.

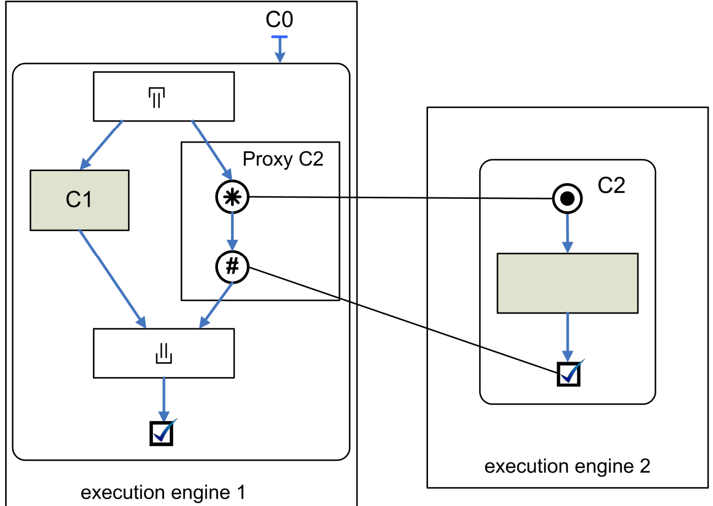


Figure B-4 Using proxy processes to relate remote subcomponents

Function call-based concurrency inside components

The class diagram in Figure B-3 defines that each component contains an internal scheduler. The dispatcher of a component is in its execute() function. It will use a scheduling queue (FIFO or sorted queue) to obtain the pointer to the next schedulable unit ready to be executed.

Every schedulable unit inside a component is implemented as a finite-state machine that performs one synchronization and computation step per each function call, and subsequently returns control back to the component scheduler. The current place where the schedulable unit stopped with its execution is remembered in its internal state variable. When the schedulable unit is activated a next time, it will use this value to continue from where it had stopped. Every schedulable unit does have associated a pointer to the next schedulable unit to activate when its execution is finished. This is either its parent construct or the next schedulable unit in sequence (if the parent is a sequential construct).

Every construct exists inside some parent component. Constructs (Parallel, Alternative and Sequential) as well as channel/event ends are designed as predefined state-machines that implement behavior expected from them.

For instance, a simplified finite state machine implementing the Parallel construct would have two states: one with forking subprocesses (the FORK state in code snippet bellow), and one waiting for all subprocesses to finish (JOIN state in the code snippet). In reality, a mechanism for handling errors and exceptional situations requires one or two additional states.

```
Parallel::run(){
   switch(state){
      case FORK:
         parentComponent->scheduler->add(subprocesses);
         state = JOIN;
         result =0;
         break;
      case JOIN:
         if(finishedCount == size)
         {
            state = FORK;
            finishedCount=0;
            parentComponent->scheduler->add(next);
            result =1;
         }
         break;
   }
   return result;
}

Parallel::exit() {
   finishedCount++;
      if(finishedCount ==size) parentComponent
                                ->scheduler->add(this);
}
```

The subprocesses use the exit() function to notify the Parallel construct that they have finished their execution. Since all subprocesses are in the same component and executed in atomic parts in function-call based concurrency manner, there are no mutual exclusion hazards involved.

When a construct finalizes successfully its execution, it returns a status flag equal to 1 or higher. For its parent it is a sign that it can move to the next phase in its execution by updating its state variable. In case of a guarded alternative, the returned number is in the parent process understood as the index of the branch to be followed and it is used to determine the next value of the state variable.

Thus, the system works by jumping in a state-machine, making one step (e.g. executing a code block or attempting event synchronization or forking subprocesses), and then jumping out. This might seem inefficient, but actually also in the user-level thread situation, a similar thing is done: testing the need for a context switch is hidden in every event attempt. Only performance testing can show which way is actually more efficient under what conditions. Recursions that are used to define auxiliary, named, process entry points are not implemented in a separate class. Instead they are naturally implemented using labels.

Let us use the example given in SystemCSP (Figure B-5), and also in CSPm code above the figure to display how its software implementation would look like in this framework.

```
Restricted_program_use = Program \ {install, uninstall}
Program = install → StartMenu
StartMenu = openProg  → UseProg | uninstall  → Program
UseProg = closeProg  → StartMenu | openDoc → Work
Work = updateDoc → Work | saveDoc  →Work
          | closeDoc → UseProg | closeProg → StartMenu
```

Figure B-5 SystemCSP design used as an example for software implementation

The code is as follows:

```
Program(){
 switch (state){
  case START:
     status = install->sync();
     if(status == 0) return;
     elseif(status == 1){
        Installation();
        state = START_MENU;
     }
     else state = ERROR;
    break;
  case START_MENU:
      status = guardedAlt_StartMenu->select();
      if(status == 0) return;
      elseif(status == 1) {
          InitializeProg();
          state = USE_PROG;
     }
   else if(status== 2) {
       UninstallProg();
       state = START;
```

```
    }
    else state = ERROR;
    break;
  case USE_PROG:
    status = guardedAlt_UseProg ->select();
    if(status == 0) return;
    elseif (status == 1) {
        SaveDocs();
        state = START_MENU:
    }
    else if (status == 2) {
        LoadModel();
        state = WORK;
    }
    else state = ERROR;
    break;
  case WORK:
    status = guardedAlt_Work->select();
    if(status == 0) return;
    elseif(status == 1) {
        UpdateModel();
        state = WORK;
    }
    elseif(status == 2) {
        SaveChanges();
        state = WORK;
    }
    elseif(status == 3) {
        SaveDocDlg();
        state = USE_PROG;
    }
    elseif(status == 4) {
        SaveDocs();
        state = USE_PROG;
    }
    else state= ERROR;
    break;
  case ERROR:
       printf(" process P got invalid status ");
    break;
  }
```

In the constructor of the class defining this process, objects for the contained event ends and constructs are instantiated. For instance, the guarded alternative named StartMenu is on creation initiated using the offered event ends (openProg and uninstall) as arguments:

```
guardedAlt* StartMenu = new guardedAlt(openProg, uninstall);

EventEnd* openProg = new EventEnd(parentESP);
```

Code blocks are defined as member functions of a class that represent the process in which they are used. Code blocks that are used in more then one subprocess are usually defined as functions on the level of the component. Note that all code blocks (even a fairly complex sequential OOP subsystem that contains no channels, events and constructs) will be executed without interruption. Their execution can only be preempted by the operating-system thread of higher priority. As explained, user-level scheduling and function-call based execution engines are not fully

preemptive. Thus, the events that need immediate reaction should be handled by operating-system threads of higher priorities.

B.3 Implementing CSP Events and channels

Event ends are schedulable units implemented as state machines. They participate in the synchronization related to the occurrence of the associated event. This includes communicating their readiness to upper layers and waiting till the event is accepted by all participating event ends. This section describes in more detail how precisely this synchronization is performed.

Event synchronization mechanism

CSP events use the hierarchy of constructs for synchronization. An event end can be nested in any construct and it has to notify its parent construct of its activation.

In Figure B-6, component C0 contains a parallel composition of components C1, C2 and C3 that synchronize on events ‘a’ and ‘b’. Component C2 contains a parallel composition of C11 and C12 that synchronize on event ‘a’. The guarded alternative located in component C21 offers to its environment both events ‘a’ and ‘b’.

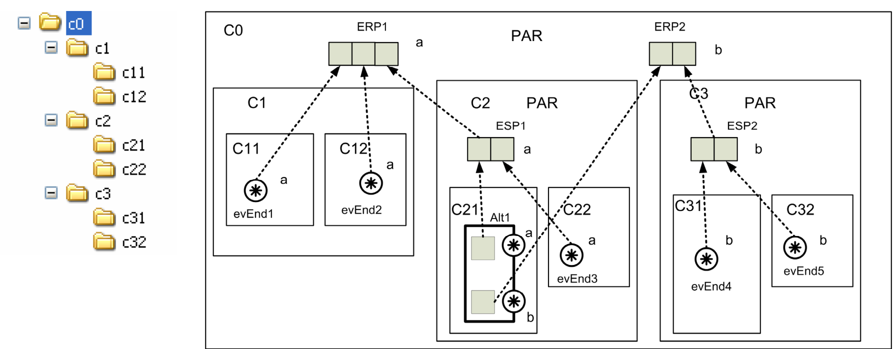


Figure B-6 Hierarchical synchronization of CSP events

Every process needs to export not-hidden events further to its environment, that is to a higher level synchronization mechanism. Every construct in the hierarchy must provide support for synchronizing events specified in its synchronization alphabet. This synchronization is done by dedicated objects – instances of the ESP (EventSynchronizationPoint) class (see Figure B-7). The event-end will actually notify the ESP object of its parent construct about its readiness. A guarded alternative offers a set of possible event ends and thus instead of signaling its readiness to its parent construct, it can only signal conditional readiness.

An ESP will, when all branches under its control are ready (conditionally or unconditionally) to synchronize on the related event, forward the readiness signal

further to its parent ESP. When an event is not exported further, that construct is the level where the event occurrence is resolved. In that case, instead of an ordinary ESP object, a special kind of it exist (Event Resolution Point or ERP class) that performs the event resolution process. If some event ends are only conditionally ready, the ERP object will initiate a process of negotiation with the nested guarded alternative elements willing to participate in that event. When all event ends agree on accepting the event, ERP will notify all of them about the event occurrence.

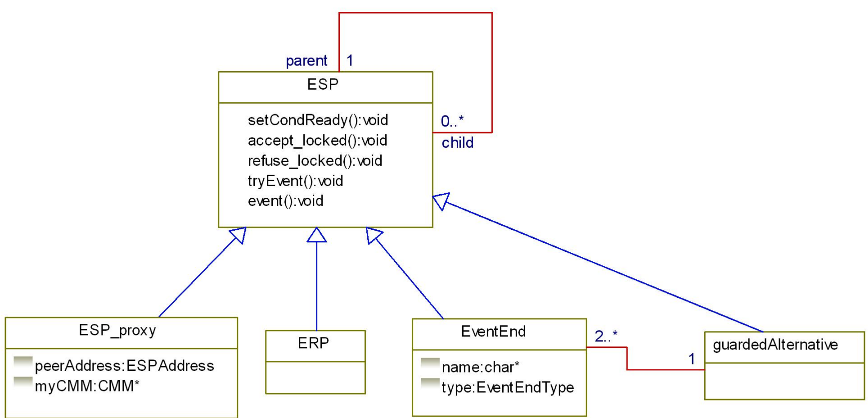


Figure B-7 Event synchronization point classes

When on the top-level, in ERP, all fields, representing readiness of the associated branches, are ready or conditionally ready, a procedure of negotiation with sources of conditional readiness starts. This action results in every participating guarded alternative being asked to accept the event. If not previously locked by accepting negotiation with some other ERP, the queried guarded alternative will respond by accepting the event conditionally and locking till the end of the negotiation process. The attempt to start negotiation with already locked guarded alternative results in a rejection. In that case, the conditional readiness of the guarded alternative is canceled for that event and the negotiation process stops. When all guarded alternative constructs participating in the negotiation process have accepted the event (and are locked - rejecting other relevant events attempts), the ERP declares that the event is accepted by notifying all participating event ends (including the guarded alternatives) about the event occurrence. However, after one of the involved guarded alternatives has rejected the event acceptance, the event attempt did not succeed and all involved guarded alternatives are unlocked. Guarded alternatives unlocked in this way do again state conditional readiness for those event ends for which it might have been canceled during the negotiation procedure.

The class hierarchy defining types and relationships between event synchronization points is illustrated in Figure B-7. For every type of the negotiation message, the ESP class declares a dedicated function. In case of local synchronization, a parent and the related children ESPs communicate via function calls. In case that

synchronizing parent/child ESPs are residing in different OS threads or nodes, the ESP_proxy abstraction is used.

In the table below, the list of exchanged messages is specified as an illustration of an attempt to synchronize participating event-ends in a scenario based upon the example from Figure B-6.

Table 4 One synchronization scenario

source	destination	message
evEnd1, evEnd2	ERP1	Ready
ALT1	ESP1	Conditionally Ready
ALT1	ERP2	Conditionally Ready
evEnd3	ESP1	Ready
ESP1	ERP1	Conditionally Ready
evEnd4	ESP2	Ready
ERP1	ESP1	Try event
ESP1	ALT1	Try event
evEnd5	ESP2	Ready
ALT1	ESP1	Accept_locked
ESP2	ERP2	Ready
ERP2	Alt1	Try event
ALT1	ERP2	Refuse_locked
ESP1	ERP1	Accept_locked
ERP1	ESP1, evEnd1, evEnd2	event
ESP1	ALT1, evEnd3	event

Solving the mutual exclusion problem

Let us assume that allocation of the application hierarchy from Figure B-6 to the hierarchy of execution engines is performed as in Figure B-8. Clearly, simultaneous access to variables, which is possible in the case of distributed systems and operating-system thread based concurrency, must be prevented while implementing the previously explained event synchronization mechanism.

Event synchronization is more or less a generalization of the synchronization process used for channels. Let us therefore use channel synchronization as an example to show where the simultaneous access can cause problems.

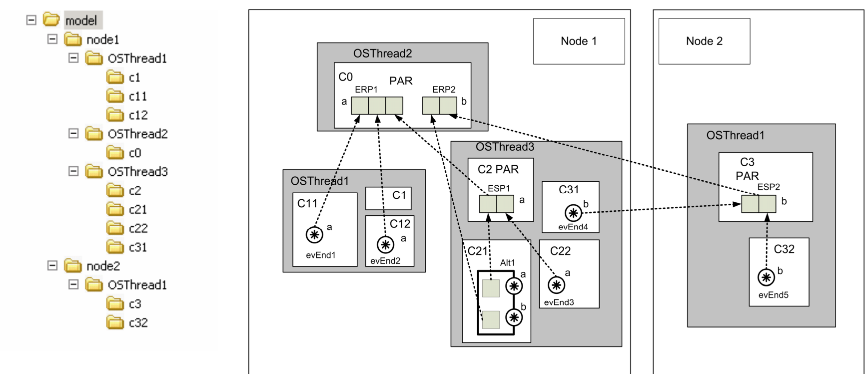


Figure B-8 Synchronization of event ends allocated to different execution engines

In CT, a channel is a passive object. The process that first accesses the rendezvous channel will be blocked (taken out of the scheduler) and the pointer to that process thread is preserved in the channel. The process thread that arrives secondly will then copy the data and add the blocked process (one that has arrived first) to the scheduler. In CT, there is no problem of simultaneous access because the whole application is located in single OS thread.

In the SystemCSP framework, due to the possibility of using several OS threads as execution engines, protection from simultaneous access needs to be taken into account in order to make safe design.

Problematic points for channel communication when truly simultaneous access is possible are: (1) making the decision who arrived first to the channel and (2) adding the blocked process/component/user-level thread to its parent scheduler that can be accessed simultaneously from many OS threads.

Constructing a custom synchronization mechanism using flag variables is complex and error-prone. Besides, it is highly likely that such mechanism will fail to be adequate in case of hyperthreading and multi-core processors.

Using blocking synchronization primitives provided by the underlying operating systems causes the earlier mentioned problem of blocking all components nested in an operating-system thread that makes the blocking call. Besides unpredictable delay, this introduces additional dependency that can result in unexpected deadlock situations. It also does not provide a solution for an event synchronization procedure in case the participating components are located on different nodes.

If non-blocking calls, to test whether critical sections can be entered, are used, the operating-system thread that comes first can do other things and poll occasionally whether a critical section is unlocked. However, this approach makes things really complicated. For instance, the higher priority operating-system thread needs to be blocked so that the lower priority one can get access to the CPU and be able to access the channel. To block only the component, which accessed the channel and

not the whole operating-system thread, one needs later to be able to reschedule it. For safe access to the scheduler from the context of another operating-system thread, another critical section is needed.

The previously discussed attempts to solve the mutual exclusion problem do apply only for processes located in different OS threads, but on the same node. In essence, from the point of view of the mutual exclusion problem, an operating system thread is equally problematic as synchronization with parts of a program on another node. Thus, it is convenient if the solution for both problems relies on the same mechanism.

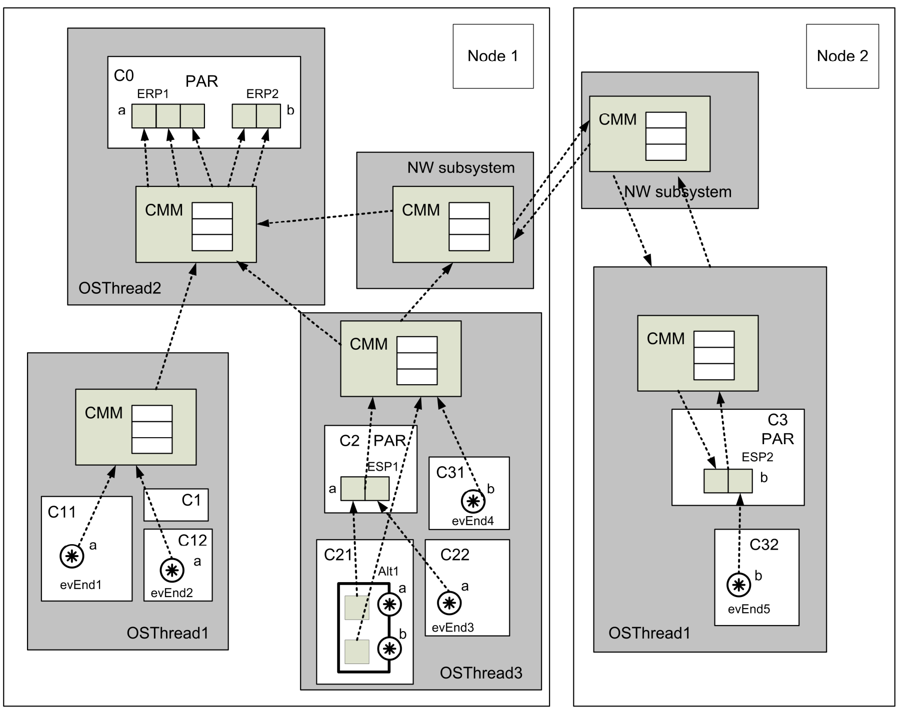


Figure B-9 Using message queue based CMM

What we propose is that every operating-system thread has an associated message queue (operating systems provide message queues as a way to have non-blocking communication between operating-system threads). Thus, every OS thread, that interacts with other OS threads, will contain a control message manager (CMM) component that dispatches control messages (like event ready, event conditionally ready, try event, event accepted and similar) to message queues of other operating-system threads and transforms the received control messages to the appropriate function calls. For synchronization between nodes, networking subsystem can be located in a dedicated operating system thread that has a similar CMM component. This CMM will use the networking system to dispatch control messages to other nodes and will dispatch control messages received from other nodes to the message queues associated with CMMs of appropriate operating-system threads.

ESP_proxy (see Figure B-7) communicates messages and addresses to local CMM, which further transfer it to the peer's CMM. The peer's CMM will then deliver the message by invoking direct function calls of appropriate ESP objects.

Channels capable for multidirectional communication

Channels are special types of events where only two sides participate and in addition data communication is performed. As such, channels can be implemented in a more optimized way then events by avoiding the synchronization through hierarchy. Similar optimizations can be done for barriers with always fixed participating event ends, shared channels (any2One, One2Any) and simple guarded alternatives where all participating events are channels that are guarded only on one side.

One of the requirements (imposed by CSP as opposed to occam) for channels is that data communication can contain a sequence of several communications in either direction. A design choice made here is to separate synchronization from communication. To achieve flexible multidirectional communication, the part dealing with communication is further decomposed to pairs of sender and receiver communication objects (TxBuffer and RxBuffer) instead of using the template C++ language mechanism to parameterize complete channels with parameters specifying transferred data types, only RxBuffers and TxBuffers are parameterized. In this way flexibility is enhanced. Every channel end will contain an array consisting of one or more TX/RxBuffer objects connected to their pairs in the other end of the channel.

Since TxBuffers and RxBuffers contain pointers to the peer TxBuffer<T>/RxBuffer<T> objects, checking type compatibility of connected channel ends is done automatically at the moment of making the channel connection. This is convenient in case when connections between components are made dynamically during run-time. Otherwise, design time checks would be sufficient. Decoupling communication and synchronization via Tx./RxBuffers is also convenient for distribution.

Distribution/networking

The CMM based design with control messages is straightforwardly extendable to distributed systems. In a distributed system, compared to operating-system thread based concurrency, besides control messages, also data messages are sent. Every node has a network subsystem with a role to exchange data and control messages with other nodes. The network subsystem takes control over RxBuffer and TxBuffer objects of a channel-end from the moment when the event is attempted, and returns control to the OS thread where the channel end is located after the data transfer is finished. This is done by exchanging (via the CMM mechanism) control messages related to location, locking and unlocking of data.

Of course, distributed event resolution comes with a price of increased communication overhead due to network layer usage. But, the task of the execution framework is to create conditions for this distribution to take place and the task of the designer of a concrete application is to optimize its performance by choosing to

distribute on different nodes only those events whose time constraints allow for this imposed overhead.

B.4 Other parts of the software design

Exception handling

In SystemCSP, exception handling is specified by the take-over operator related to the interrupt operator of CSP. The take-over operator specifies that when an event offered to the environment by the process specified as second operand (exception handler) is accepted, the further execution of the process specified as the first operand (interrupted process) is aborted.

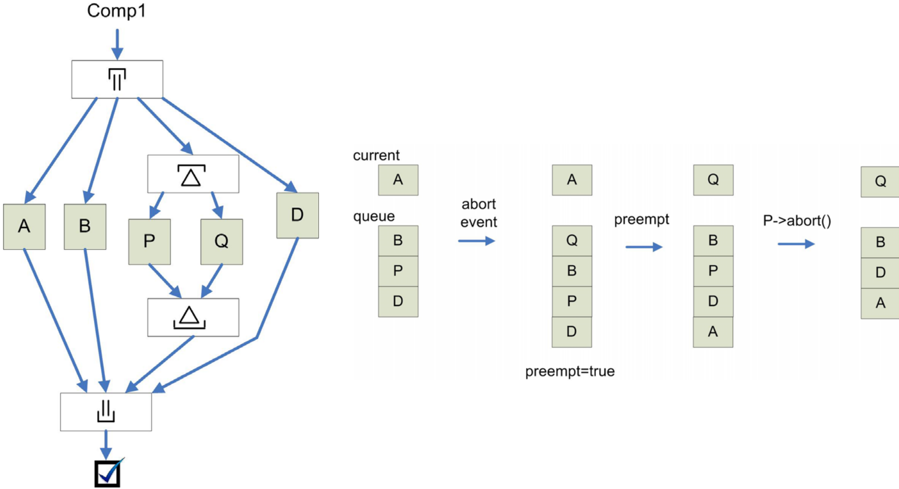


Figure B-10 Example used to explain the implementation of take-over operator

Upon the abort event (see Figure B-10), the exception handler process is added to the scheduling queue of its parent component. Since the exception handler is a special kind of process recognizable as such by the scheduler, it is not added to the end of FIFO queue as other, 'normal' processes, but at its head. The preempt flag of the component manager is set to initiate preemption of the currently executing process. In that way, the situation where the exception handler needs to wait, while the interrupted process might continue executing, is avoided as much as possible.

As illustrated in Figure B-10, the preempted process is appended to the end of FIFO queue of the component scheduler. If the preempted process is in fact the interrupted one then it will be taken out from the FIFO queue later during the abort procedure.

The first step in the interrupt handler process is calling the abort() function of the interrupted process. The default version of abort() will cancel the readiness of all event ends for which the aborted process has declared readiness or conditional

readiness. If the process is in the scheduling queue, it will be removed from there. Further, if the process is a construct, abort() will be invoked for all its subprocesses.

This exception handling mechanism does not influence the execution of other components that might have higher priority than the component where interrupted process resides.

Logging

In this framework, the design choice is to allow logging only for the variables defined on the component-level. The main reason is obtaining a very structured and flexible way of logging that allows on-line reconfiguration of logging parameters. Thus all data constituting the state of the component should be maintained in the shape of component level variable. Every component can have a bit field identifying which of its variables are currently chosen for logging. The interface is defined that allows human operators to update this bit field at any time and thus change the set of logged variables.

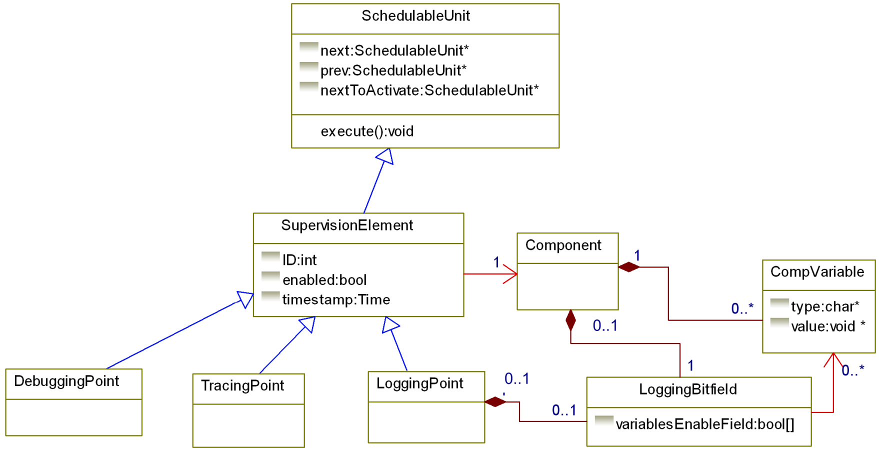


Figure B-11 Supervision elements

Logging points are predetermined in design by associating them to prefix arrows and define optionally visualized layer added on top of the design. In implementation however prefix arrows do not exist, while logging points are inserted to the appropriate location in execution flow, as defined by the position of prefix arrow in the design.

Any logging point, either uses set of variables set for logging on component level using the described bit field mechanism, or defines its own bit field with set of variables to log. The operator is via the NodeManager allowed to inspect logging points and update their bit fields. Every logging point has a tag (or ID) unique in scope of its parent component, that is used to uniquely identify it. On the target side of the application, this tag can be a pointer to the object implementing the

logging point. On the operator side of the application this tag is mapped to the unique ID of the logging point as specified in the system design.

The reason to opt for this kind of logging is predictability. The logging activity is considered to be part of the design and all the needed resources (e.g. CPU time, memory, network bandwidth and storage capacity) can be preallocated. Logging points can in design be inserted in such a way that it is possible to reconstruct change of every variable during the time.

Tracing

Tracing is an activity similar to logging. The difference is that instead of data, the information communicated to the human operator is the current position in execution flow of the application. Control flows leading to error states are always traced. Errors that are not fatal for the functionality of the system are logged as warnings. Other tracing points can be used for debugging or for supervising control. As it is the case for logging, the tracing is here considered to be part of the design and as such performed in predefined points of the execution flow.

SystemCSP defines a circle with a big T inside as a symbol of tracing point. Again it is associated with prefix arrow element, defining in that way the precise position of a tracing point. Every tracing point has a tag (or ID) that is unique per component and communicated to the operator to notify the occurrence of control flow passing over a tracing point. In addition, every function entry/exit is a potential tracing point.

B.5 Conclusions

This Appendix introduces design principles for the implementation of a software architecture that will support SystemCSP designs. This Appendix started with explaining the reasons to discard the possibility to reuse the CT library as a framework for software implementation of SystemCSP models. The rest of the text introduced the design principles for the implementation of the framework infrastructure needed in the software domain to support the implementation of a models specified in SystemCSP.

One of the main contributions of this Appendix is the decoupling application domain hierarchy of the components (related via CSP control flow elements and parent-children relationship) from the execution engine framework. In addition, this framework is constructed to allow maximal flexibility in choosing and combining execution engines of different types. In this way, flexible and reconfigurable component-based system is obtained. The priority specification is related to the hierarchy of execution engines and has thus become part of the deployment and not application design process.

Another significant contribution of the text presented in this Appendix is solving the problem of implementing the mechanism for synchronizing CSP events in a way that is safe from mutual exclusion problems and is naturally suited for

distribution. Besides that, the text describes and documents the most important design choices in the architecture of the SystemCSP software framework.

Recommendation for future work is to fully implement everything presented in this Appendix. Furthermore, a graphical development tool is needed that will be capable to generate code. The described software framework would be used as a basic infrastructure that supports the proper execution of generated code.

References

Allen R. J. (1997), “A Formal Approach to Software Architecture”, Doctoral thesis, Carnegie Mellon University, 231 pages, ISBN:0-591-64744-3, 1997.

Amerongen J. van, Breedveld P.C. (2002), "Modelling of Physical Systems for the Design and Control of Mechatronics Systems", *IFAC Professional Briefs*, published in relation to the *15th triennial IFAC World Congress*, International Federation of Automatic Control (http://www.ifac-control.org), Laxenburg, Austria, pp. 1-56, 2002.

Avizienis A., Laprie J.-C., Randell B., Landwehr C. (2004), "Basic Concepts and Taxonomy of Dependable and Secure Computing,", *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11-33, January-March, 2004.

Berg L. S. van den (2006), “Design of a Production Cell setup”, Master’s thesis, MSc-Report 016CE2006, Control Laboratory, University of Twente, July 2006.

Berge, M.H. ten (2005), “Design Space Exploration for Fieldbus-based Distributed Control Systems”, Master’s thesis, MSc-Report 029CE2005, Control Laboratory, University of Twente, August 2005.

Berge, M.H. ten, Orlic, B., Broenink, J.F. (2006), “Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach”, Proc. IEEE International Symposium on Computer Aided Control Systems Conference, Munich, Germany, pp. 434-439, ISBN 0-7803-9797-5, 2006.

Beugnard A., Jezequel J-M., Plouzeau N., and Watkins D. (1999), “Making components contract aware”, *IEEE Computer*, Volume 32, Issue 7, pp. 38-45, IEEE Computer Society Press, Los Alamitos, CA, USA, ISSN:0018-9162, July 1999.

Booch G. (1993), “Object-Oriented Analysis and Design with Application”, 2nd [rev.] ed., The Benjamin/Cummings series in object-oriented software engineering, Benjamin-Cummings Publishing Company, ISBN: 0-8053-5340-2, 1993., cop. 1994.

Booch G., Rumbaugh J., and Jacobson I. (1999), “The Unified Modeling Language reference manual”, The Addison-Wesley object technology series, 550 p, AddisonWesley, ISBN: 0-201-30998-X. 1998, cop. 1999.

Boosten, M. (2003), “Formal contracts: Enabling component composition”, in Proceedings of *Communicating Process Architectures* 2003, editors J. F. Broenink and G. H.Hilderink,, IOS Press, pages 185–197, University of Twente, Netherlands, 2003.

Broenink J. F. and Hilderink G. H. (*2001*), “A structured approach to embedded control systems implementation”, Proc. *IEEE International Conference on Control Applications*, pp. 761-766, September 5-7, México City, México, 2001.

Burns, A. (1998), “How to Verify a Safe Real-Time System. The Application of Model Checking and a Timed Automata to the Production Cell Case Study”,

Technical report, Real-Time System Research Group, Department of Computer Science, University of York, 1998

Buttazzo, G.C. (2002), “Hard real-time computing systems: Predictable Scheduling Algorithms and Applications“, The Kluwer international series in engineering and computer science, 379 p, Kluwer Academic Publishers, 1997, cop. 2002.

Buttazzo, G.C. (2005), “Rate Monotonic vs. EDF: Judgment Day“, *Real-Time Systems*, 29 (1): p. 5-26(22). Jan 2005.

Cassandras, C.G. and Lafortune S. (1999), “Introduction to discrete event systems”, 848 pages, Hardbound,,Kluwer Academic Publishers, ISBN 0-7923-8609-4, September 1999.

Celoxica (2007), Handel-C documentation at *http://www.celoxica.com/*, 2007.

Cervin, A. and Ekerz J. (2006), “The Control Server Model for Codesign of Real-Time Control Systems”, in Hans Hansson (Eds.): ARTES – A network for Real-Time research and graduate Education in Sweden 1997–2006, Department of Information Technology, Uppsala University, Sweden, March 2006.

Chrabieh, R. (2005), “Operating System with Priority Functions and Priority Objects”, Technical report, available at: www.portos.org/doc/whitepaper.pdf, 2005.

Crichton, C., Cavarra, A., and Davies, J. (2002), “A pattern for concurrency in UML”, Technical Report RR-01-22, Oxford University Computing Laboratory. http://web.comlab.ox.ac.uk/oucl/research/areas/softeng/FASE2002.pdf, 2002.

Crnkovic I. and Larsson M., editors. (2002), “Building Reliable Component-BasedSoftware Systems”, 454 pages, Artech House publisher, ISBN: 1580533272, 2002.

Davies, J. J. (2003), “Concurrency and Refinement in the Unified Modeling Language”, *Formal aspects of computing*, volume 15, issue 2-3, 2003.

Dijjkstra E. (1972), “The Humble Programmer”, ACM Turing award Lecture, published in the Communications of the ACM, 1972.

Eclipse (2007), EMF framework, available at URL: http://www.eclipse.org/modeling/emf/

Eddon G., Eddon H. (1998), “Inside Distributed COM”, 552 pages, Microsoft Press, ISBN:1-57231-849-X, 1998.

Eker J., Janneck J. W., Lee E. A., Liu J., Liu X., Ludvig J., Neuendorffer S., Sachs S., and Xiong Y. (2003), “Taming heterogeneity—the Ptolemy approach”, *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, vol: 91, issue 1, pages :127-144, ISSN: 0018-9219, January 2003.

ESI (2006), Boderc: “Model-based design of high-tech systems, A collaborative research project for multi-disciplinary design analysis of high-tech systems”, Embedded System Institute, available at URL: http://www.embeddedsystems.nl/site/projects/boderc/

Fleming P.J. (1988), “Parallel processing in control: the transputer and other architectures”, IEE control engineering series, 243 pages, ISBN: 0-86341-136-3, Peregrinus on behalf of the Institute of Electrical Engineers, 1988.

Formal Systems (Europe) (2005), *FDR2 User Manual*, at URL: http://www.fsel.com/software.html

Fidge, C.J. (1993), “A formal definition of priority in CSP”, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, Issue 4, p. 681-705, September 1993

Formal methods (2007), at URL: http://vl.fmnet.info/

Gamma E., Helm R., Johnson R., and Vlissides J. (1994), *Design Patterns:Elements of Reusable Object-Oriented Software*. Addison Wesley.

Garlan, D., Monroe, R.T, and Wile, D. (2000), “Acme:Architectural Description of Component-Based Systems”, in: Leavens, G.T., Sitaraman, M. (Eds.), Foundations of Component-Based Systems, pp. 47-67. Cambridge University Press, 2000.

Geilen M., Voeten J., van der Putten P., van Bokhoven L., and Stevens M. (2001), “Object-oriented modelling and specification using SHE”, *Journal of Computer Languages*, 27, 2001.

Harel D. and Politi M. (1998), “Modeling Reactive Systems with Statecharts: The STATEMATE Approach”, McGraw-Hill., 1998.

Henriksson D. and Cervin A. (2003), "TrueTime 1.13-Reference Manual,", Technical report, Department of Automatic Control, Lund Institute of Technology, Lund, 2003.

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

Hilderink, G.H., Broenink, J.F., Vervoort, W.A., Bakkers, A.W.P. (1997), "Communicating Java Threads", in: 20th *World Occam and Transputer User Group Technical Meeting*, pp. 48-76. Enschede, The Netherlands, 1997.

Hilderink G.H. (2003), "Graphical modelling language for specifying concurrency based on CSP", in: *IEE Proceedings: Software*, IEE, pp 108-120, Volume 150, Number 2, April 2003, ISSN 1462-5970, 2003.

Hilderink G. H. (2005a), “Managing Complexity of Control Software through Concurrency”, Doctoral thesis, pages ix - 352, ISBN 90-365-2204-8, University of Twente, 2005.

Hilderink G. H. (2005b), “Exception Handling Mechanism in Communicating Threads for Java”, in *Proceedings of Communicating Process Architectures 2005,* IOS Press, 2005.

Hilderink G. H. (2006), “Software Specification Refinement and Verification Method with I-Mathic Studio”, in *Proceedings of Communicating Process Architectures 2006,* IOS Press, 2006.

Hoare C. A. R. (1978), *Communicating sequential processes*, Communications of the ACM, 21(8), 1978.

Honeywell (2007), MetaH project, available at URL: http://www.htc.honeywell.com/metah/

Huang J., Voeten J., M. Groothuis M., Broenink J. and Corporaal H., A model-driven design approach for mechatronic systems

INMOS (1988), *occam 2 Reference Manual*, International Series in Computer Science, ed. C.A.R. Hoare, Prentice Hall.

JCSP (2007), University of Kent, *The JCSP Homepage* at URL:http://www.cs.kent.ac.uk/projects/ofa/jcsp/

Jovanovic D.S., Orlic B., Liet G.K.and Broenink J.F.(2004), "gCSP: A Graphical Tool for Designing CSP systems", in: *Communicating Process Architectures 2004*, 5-8 Sept 2004, Oxford UK, pp 233 - 251, 1586034588, 2004.

Jovanovic D.S., Orlic B. and Broenink J.F. (2005), "On issues of constructing an exception handling mechanism for CSP-based process-oriented concurrent software", in: *Communicating Process Architectures 2005*, Eindhoven, 18 - 21 Sept. 2005, IOS Press, pp 29 - 41, ISBN: 1-58603-561-4, 2005.

Jovanovic, D.S. (2006), "Designing dependable process-oriented software - a CSP-based approach", p. vi +264, ISBN 90-365-2334-6, 2006.

Kahn G. and MacQueen D. B. (1977), “Coroutines and networks of parallel processes”. In B. Gilchrist, editor, *Information Processing*. North-Holland Publishing Co., 1977.

Keding H. (2004), Systems Verification, presented at ASCI winter school 2004

Kock E. A. de, Essink G., Smits W. J. M., Wolf P. van der, Brunel J.-Y., Kruijtzer W., Lieverse P., and Vissers K. A. (2000), “Yapi: Application modeling for signal processing systems”, In *37th Design AutomationConference (DAC’00)*, pages 402–405, Los Angeles, CA, 2000.

Lange C.F.J., Chaudron M.R.V., Muskens J. (2006), “In practice: UML software architecture and design description”, *IEEE software*, vol. 23, pp. 40., 2006.

Lee E.A. (2006), “The problem with threads”, *IEEE Computer*, vol.39, No. 5, pp. 33-42, May 2006, available online as U.C. Berkeley EECS Department Technical Report UCB/EECS-2006-1

Magee J. and Kramer J. (1999), “Concurrency: state models & Java programs”, 2nd ed., ISBN: 13 978-0-47009355-9, ISBN: 10 0-470-09355-2, John Wiley and Sons Ltd., 1999.

Maljaars, P. (2006), "Controllers for the Production Cell Set Up", Master's thesis, MSc-Report 039CE2006, Control Laboratory, University of Twente, December 2006.

Marwedel P. (2003), "Embedded system design", Kluwer Academic Pubilshers, Dordrecht, Netherlands., 2003.

McConnellS.(2003), *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers*. Addison-Wesley Professional.

Meyer B. (1992), *Applying "Design by Contract"*, Computer, IEEE, Volume 25, Issue 10.

Milner R. (1989), *Communication and Concurrency*, Prentice-Hall International Series in Computer Science,Prentice-Hall International, Englewood Cliffs.

Milicev, D. (2002), "Domain Mapping Using Extended UML Object Diagrams," IEEE Software, Vol. 19, No. 2, March/April 2002, pp. 90-97

Nienaltowski P., Meyer, B. (2006), "Contracts for concurrency", *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages* CORDIE'06, York, United Kingdom, pp 27-49, 4-5 July 2006.

Ommering R. van (2004), "Building Product Populations with Software Components", PhD Thesis, Rijksuniversiteits Groningen, ISBN 90-74445-64-0. 2004.

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

Orlic, B., Broenink, J.F. (2006a), "*SystemCSP - Visual Notation*", in: Communicationg Process Architectures 2006, Edinburg, edited by Welch, P.H., Kerridge, J., Barnes, F.R.M., pp. 151-177, ISBN 1-58603-671-8, 2006

Orlic, B., Broenink, J.F. (2006b), "*Interacting Components*", in: Communicationg Process Architectures 2006, Edinburg, edited by Welch, P.H., Kerridge, J., Barnes, F.R.M., pp. 179-202, ISBN 1-58603-671-8, 2006

Orlic, B. and J.F. Broenink (2007a), CSP and Real-TIme: Reality of Illusion? in: Communicating Process Architectures 2007, A. McEwan, S. Schneider, W. Ifill and P. H. Welch (Eds.), IOS Press, Guidford, UK, pp. 119-147, ISBN: 978-1-58603-767-3, 2007.

Orlic, B. and J.F. Broenink (2007b), Design Principles of the SystemSCP Software Framework, in: Communicating Process Architectures 2007, A. McEwan, S. Schneider, W. Ifill and P. H. Welch (Eds.), IOS Press, Guildford, UK, pp. 207-228, ISBN: 978-1-58603-767-3, 2007.

OPC (2007), at URL: http://www.opcfoundation.org/.

Ouaknine, J. and Worrell J. (2003), "Timed CSP = closed timed epsilon-automata",*Nordic Journal of Computing*, 10(2): pp99-133, 2003.

POOSL (2007), at URL: http://www.es.ele.tue.nl/poosl/

Ptolemy (2007), Ptolemy II project, available at: http://www.ptolemy.eecs.berkeley.edu/

Pullum L. L. (2001), "Software Fault Tolerance Techniques and Implementation"Artech House, 2001.

Roscoe A. W. (1997), "The Theory and Practice of Concurrency", Prentice Hall.

Scattergood B. (!998), "Tools for CSP and Timed CSP", D.Phil Thesis, Oxford University, 1998.

Schmerl B. and Garlan D. (2004), "AcmeStudio: Supporting Style-Centered Architecture Development", In Proc. 2004 *International Conference on Software Engineering*, Edinburgh, Scotland, 2004.

Schneider S. (2000), "Concurrent and Real-Time Systems: The CSP approach", Wiley, 2000..

Szyperski C. (1998), "*C*omponent Software: Beyond Object-Oriented Programming", Addison-Wesley and ACM Press, ISBN 0-201-17888-5, 1998.

Sunter, J.P.E. (1994), "Allocation, Scheduling and Interfacing in Real-time Parallel Control Systems", PhD thesis, University of Twente: Enschede, Netherlands, 1994.

Sutter H. (2005), "The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software", Dr. Dobb's Journal, 30 (3); available at: http://www.gotw.ca/publications/concurrency-ddj.htm.

Tanenbaum, A. (2001), "Modern Operating Systems", Prentice Hall, 952 pages, 2001.

UPPAAL, (2007), at URL: http://www.uppaal.com/

Welch P.H. (1989), "Graceful termination - graceful resetting", in *Applying Transputer-Based Parallel Machines, Proceedings of* 10th *Occam User GroupTechnical Meeting,* pages 310-317, Enschede, Netherlands, IOS Press, Amsterdam, 1989.

Welch, P.H., May M.D., and Thompson P.W. (1993), "Networks, Routers and Transputers: Function, Performance and Application", IOS Press, Netherlands, ISBN 90-5199-129-0, February 1993.

Welch, P.H. and Wood D.C. (1996), "The Kent Retargetable occam Compiler", in *Parallel Processing Developments* -- Proceedings of WoTUG 19, p. 143 -166, IOS Press: Nottingham, UK, 1996.

Wittenmark B., Nilsson .J., Törngren M. (1995), "Timing problems in Real-time control systems", in Proceedings of the American Control Conference, Seattle, June 1995.

Yeung W. L., Schneider S. A., and Tam F. (1998), "Design and verication of distributed recovery blocks in CSP", Technical Report CSD-TR-98-08, Royal Holloway, University of London, 1998.

Zhang, Y. (2005), "Real-Time Network for Distributed Control", Master's thesis, MSc-Report 031CE2005, Control Laboratory, University of Twente, August 2005.

Zhang Y., Orlic B., Visser P.M. and Broenink J.F. (2005), "Hard Real-Time Networking on Firewire", in: Proc. 7th *Real-Time Linux Workshop*, Lille, pp 1-8, 3-4 Nov 2005.

Zorzo A. F., Romanovsky A., Xu J., Randell B., Stroud R. J., and Welch I. S. (1999), "Using coordinated atomic actions to design safety-critical systems: a production cell case study", in *Software: practice & experience*, vol. 29(8), pp. 677-697, ISSN: 0038-0644, 1999.

Curriculum Vitae

Biography

Born on 11th February 1976. in Bor, northeast Serbia.

June 1994 - March 2001: Electronics group, Faculty of Electrical Engineering, University of Belgrade, Yugoslavia; Average grade: 8.62 (of 10)

01. May 2001 – 31. December 2001: software developer in Research and Development department of Informatika AD company, Belgrade

11. March 2002 – 10. March 2006: Faculty of Electrical Engineering, University of Twente, Netherlands; Research assistant (Ph.D. candidate) at the Control Engineering group

01. July 2006 – 01. October 2007: Faculty of Electrical Engineering, University of Twente, Netherlands; staff member at the Control Engineering group

Research areas of interest:

Real-time control systems; formal methods (CSP and other); graphical modeling languages for design specification; concurrency patterns (multithreading, synchronization, occam constructs…); metamodeling; component-based system architectures; real-time operating systems / kernels; industrial distributed systems interconnected via CAN, FireWire, Ethernet, Profibus, USB; simulation frameworks; fault tolerance; design patterns.