

Eila Niemelä

# A component framework of a distributed control systems family

V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s  
V T T P u b l i c a t i o n s

VTT PUBLICATIONS 402

# **A component framework of a distributed control systems family**

Eila Niemelä

VTT Electronics

*Academic Dissertation to be presented, with the assent of the Faculty of Science,  
University of Oulu, for the public discussion in the Auditorium L10, Linnanmaa,  
on January 21st, 2000, at 12 o'clock noon.*



---

TECHNICAL RESEARCH CENTRE OF FINLAND  
ESPOO 1999

ISBN 951-38-5549-X (soft back ed.)

ISSN 1235-0621 (soft back ed.)

ISBN 951-38-5550-3 (URL: <http://www.inf.vtt.fi/pdf/>)

ISSN 1455-0849 (URL: <http://www.inf.vtt.fi/pdf/>)

Copyright © Valtion teknillinen tutkimuskeskus (VTT) 1999

#### JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT  
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT  
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland  
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU  
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG  
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN-90571 OULU, Finland  
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Leena Ukskoski

Libella Painopalvelu Oy, Espoo 1999

Niemelä, Eila. A component framework of a distributed control systems family. Espoo 1999. Technical Research Centre of Finland, VTT Publications 402. 188 p. + app. 68 p.

**Keywords** software engineering, component framework, component-based development, distributed control systems, software configuration management

## Abstract

A component framework is based on a software architecture, a set of components and their interaction mechanisms. This thesis examines the component-based software development by reviewing the requirements for a component framework development, proposing a model of a component framework of a distributed control systems family and demonstrating results with cases drawn from the control systems families.

The product families of the machine control systems, process control systems and manufacturing systems are studied to set the requirements for the component framework. Three main problems are discovered. A lack of appropriate modelling methods prevents describing product features and variability at the software architecture level. Interoperability and adaptability of software components that are required in the integration phase are inadequate in most cases. Furthermore, integrators and maintenance staff also need support software for extending and upgrading systems.

The component framework of a distributed control systems family introduces two dimensions: tiers and elements. The three tiers of the component framework define the subsystems in the first tier, integration platform in the second tier, and the product family in the third tier. The tiers explain the domain, technology and business viewpoints of the framework correspondingly. The elements define the product features, software architecture, components and their interaction mechanisms. The development and utilisation of the component framework have three main tasks, described as the viewpoints of the component-based software development, concurrent software engineering and software configuration management.

The development of the component framework is presented by the development of the reuse assets: the product features, product-family architecture and soft-

ware components. The architecture styles, key-mechanisms, services and components of each tier are depicted. The framework mixes agent, layered, client-server and rule-based system architectures and their mechanisms to provide a coherent solution for software flexibility and stability required by the product families.

The results are analysed as regards the evaluation criteria, set for the component framework as the result of the problem analysis. Variability and adaptability are examined at the architecture and component level, as well as the interoperability of tiers, services and applications and interchangeability of product features and components.

The adaptive approach restricts the affects of the changes in the business, technology and application domain to the corresponding tiers that provide their own mechanisms for adaptability. The integration tier could be reused community-wide, whereas the subsystem tier is domain-specific and the product-family tier is always an organisation-dependent solution.

# Preface

This research was carried out during the years 1994 to 1998 at the Technical Research Centre of Finland (VTT Electronics) in the KIURU, DYNAMO and ARTTU projects.

The major part of the research of component-based software development was laid during the KIURU project during the period 1994–1995. The results were refined in the DYNAMO project by the idea of dynamically configurable application components in 1996 and the extended services of the integration platform in 1997. The reconfiguration was combined with the feature-oriented domain analysis method, developed in the KOMPPI project, and further refined in the ARTTU project during the period 1997–1998 to deal with the reconfiguration of fine-grained product features through the integration platform. The results of these projects are applied and further refined in several research and contracts projects in VTT Electronics.

I wish to thank my supervisor at the University of Oulu, Dr. Ilkka Tervonen, for guiding my dissertation efforts. I am most grateful to Dr. Jan Bosch (University of Karlskrona/Ronneby) and Dr. Olli Ventä (VTT Automation) for providing encouraging critique as the nominated reviewers of this thesis. I would also like to thank Dr. Veikko Seppänen (VTT Electronics) and Dr. Mike Mannion (Napier University) for their advice and guiding comments.

I would like to thank my colleagues at VTT Electronics and co-operation partners of other organisations. Mr. Harri Perunka and Mr. Tuomas Ihme have provided inspirational support for carrying out this work and co-authored several of the papers included in this thesis. I would also like to thank my colleagues involved in the case studies of the above-mentioned research projects: Mr. Juha Marjeta, Mr. Mikko Holappa, Mr. Arno Tuominen, Mr. Jouni Heikkinen, Mr. Tomi Korpipää and Mr. Jukka Koutaniemi who have given the remarkable support by implementing the prototypes of the component framework. I am also grateful to the co-operation partners, especially Mr. Pekka Huuskonen of Sisu Tractors Oy, Mr. Jani Granholm of Oy Mercantile Fastems Oy, Mr. Tapio Niemi of Oy ABB Transmit Ab and Mr. Juhani Meriluoto of Tomra Systems Oy who have given the exemplars for the research projects.

Tekes, VTT Electronics, and several industrial companies have financed the research projects behind this work. The work has also been financially supported by the Foundation of Emil Aaltonen. I am grateful for their support.

Finally, I wish to express my deepest and warmest gratitude to my family, Pekka, Tanja, Katja and Juha, for their support and patience during the period of this research.

Edinburgh, September 1999

Eila Niemelä

# Contents

<b>ABSTRACT .....</b>	<b>3</b>
<b>PREFACE .....</b>	<b>5</b>
<b>LIST OF ORIGINAL PUBLICATIONS .....</b>	<b>10</b>
<b>LIST OF ABBREVIATIONS.....</b>	<b>12</b>
<b>1. INTRODUCTION.....</b>	<b>16</b>
1.1 THE NEED FOR COMPONENT FRAMEWORKS .....	16
1.2 COMPONENT FRAMEWORK ARCHITECTURES .....	19
1.2.1 Definitions .....	19
1.2.2 Applied approaches .....	22
1.3 SCOPE OF THE RESEARCH.....	24
1.3.1 Application areas .....	24
1.3.2 Research areas .....	27
1.4 PROBLEM STATEMENT.....	29
1.4.1 Research problem .....	29
1.4.2 Research assumptions.....	31
1.4.3 Research methods and results.....	32
1.5 OUTLINE OF THE DISSERTATION.....	37
<b>2. PROBLEM ANALYSIS .....</b>	<b>38</b>
2.1 COMPONENT-BASED DEVELOPMENT OF A PRODUCT FAMILY .....	38
2.2 PROBLEMS IN THE COMPONENT-BASED SOFTWARE DEVELOPMENT .....	40
2.2.1 Domain analysis .....	42
2.2.2 Feasibility studies .....	44
2.2.3 Software architecture .....	45
2.2.4 Component design and implementation .....	47
2.3 PROBLEMS IN INTEGRATION.....	48
2.3.1 Interfaces .....	50
2.3.2 Interoperability .....	50
2.3.3 Adaptability .....	52
2.4 PROBLEMS IN UPGRADING SYSTEMS .....	54
2.5 SUMMARY .....	55



<b>3. A COMPONENT FRAMEWORK OF DISTRIBUTED CONTROL SYSTEMS .....</b>	<b>58</b>
3.1 AN OVERVIEW OF THE COMPONENT FRAMEWORK .....	58
3.2 THE SUBSYSTEM TIER.....	61
3.3 THE INTEGRATION TIER.....	65
3.4 THE PRODUCT FAMILY TIER .....	70
3.5 SUMMARY .....	73
<b>4. DEVELOPMENT OF THE COMPONENT FRAMEWORK .....</b>	<b>74</b>
4.1 DEVELOPMENT OF REUSABLE ASSETS.....	74
4.2 FEATURES OF A PRODUCT FAMILY .....	77
4.2.1 Defining product features .....	77
4.2.2 Validating features.....	82
4.3 A PRODUCT-FAMILY ARCHITECTURE .....	87
4.3.1 A layered architecture.....	88
4.3.2 An agent architecture.....	91
4.3.3 A client-server architecture.....	97
4.4 SUMMARY .....	108
<b>5. EXPERIENCES WITH THE DEVELOPMENT OF THE COMPONENT FRAMEWORK.....</b>	<b>110</b>
5.1 DIVERSITY OF THE PRODUCT FAMILIES AND USED TECHNOLOGY .....	110
5.2 JUSTIFICATION FOR THE SELECTED ENVIRONMENTS.....	111
5.3 VARIABILITY OF A PRODUCT FAMILY.....	113
5.3.1 Defining and managing product features.....	113
5.3.2 Mapping variability to the architecture and components.....	114
5.3.3 Configurability of a product family .....	118
5.4 INTEROPERABILITY OF DISTRIBUTED CONTROL SYSTEMS .....	122
5.4.1 Interoperability between tiers .....	122
5.4.2 Interoperability of services .....	123
5.4.3 Interoperability of applications.....	125
5.5 ADAPTABILITY OF THE COMPONENT FRAMEWORK.....	126
5.5.1 Portability of distributed control systems .....	126
5.5.2 Flexibility of the component framework .....	127
5.5.3 Extendibility of distributed control systems .....	128
5.6 SUITABILITY FOR THE CONTROL SYSTEMS DOMAIN.....	129
5.6.1 Operability and simplicity .....	129
5.6.2 Adaptation of COTS and legacy software .....	130

5.6.3	Heterogeneous methods and tools .....	131
5.6.4	Impacts on the development process .....	131
<b>6.</b>	<b>RELATED RESEARCH .....</b>	<b>133</b>
6.1	APPROACHES TO THE COMPONENT-BASED SOFTWARE DEVELOPMENT	133
6.1.1	Feature-oriented software reuse.....	134
6.1.2	Architecture-oriented software reuse.....	138
6.1.3	Language-oriented software reuse .....	144
6.2	MECHANISMS FOR DYNAMIC CONFIGURATION .....	147
6.2.1	Changing product features .....	147
6.2.2	Replacing applications and components.....	148
6.3	FRAMEWORKS .....	150
6.3.1	Domain-specific frameworks.....	150
6.3.2	Integration frameworks.....	151
6.3.3	Product-oriented frameworks .....	154
6.4	SUMMARY .....	157
<b>7.</b>	<b>INTRODUCTION TO THE PAPERS .....</b>	<b>159</b>
7.1	COMPONENT-BASED SOFTWARE ENGINEERING .....	160
7.1.1	Paper I: Domain analysis by re-engineering application software ....	160
7.1.2	Paper II: Product features and component-based software.....	161
7.1.3	Paper III: Development of software components .....	162
7.2	SOFTWARE ENGINEERING OF DISTRIBUTED SYSTEMS .....	162
7.2.1	Paper IV: An integration platform of real-time distributed systems..	163
7.2.2	Paper V: An integration platform of a product family.....	163
7.2.3	Paper VI: An ORB as an integration platform.....	164
7.2.4	Paper VII: Dynamic configuration of architectural components	165
<b>8.</b>	<b>CONCLUSIONS AND FURTHER RESEARCH .....</b>	<b>167</b>
8.1	ANSWERS TO THE RESEARCH PROBLEMS .....	167
8.2	TOPICS OF FURTHER RESEARCH .....	172
	<b>REFERENCES .....</b>	<b>174</b>

## PAPERS I–VII

*Appendices of this publication are not included in the PDF version.  
Please order the printed version to get the complete publication  
(<http://www.inf.vtt.fi/pdf/publications/1999/>)*

## List of original publications

This thesis includes the following eight original publications and a submission:

- I Ihme, T., Niemelä, E., Salmela, M., Seppänen, V. Object-oriented re-engineering of embedded software. *Mechatronics* 1995. Vol. 5, No. 1, pp. 76–86.
- II Kalaoja, J., Niemelä, E., Perunka, H. Feature modelling of component-based embedded software. *Proceeding of 8<sup>th</sup> IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, Los Alamitos: IEEE Computer Society, CA, 1997. Pp. 444–447.
- III Niemelä, E., Taramaa, J., Seppänen, V. Integration of prototyping and target system development for embedded machine control software. *Proceedings of 14<sup>th</sup> IASTED International Conference on Modelling, Identification and Control*. Anaheim, California: International Association of Science and Technology for Development – IASTED, 1995. Pp. 12–17.
- IV Ihme, T., Niemelä, E. Adaptability in object-oriented embedded and distributed software. In: *Special issues in object-oriented programming*. Muhlhäuser, M. (ed.). *Workshop reader of the 10<sup>th</sup> European Conference on Object-Oriented Programming ECOOP'96*. Heidelberg: Dpunkt Verlag, 1997. Pp. 29–36.
- V Niemelä, E., Perunka, H., Korpipää, T. A Software Bus as a Platform for a Family of Distributed Embedded System Products. In: *Development and Evolution of Software Architectures for Product Families*. van der Linden, F. (ed.). *Lecture Notes in Computer Science*, No. 1429. Germany: Springer-Verlag, 1998. Pp. 14–23.
- VI Niemelä, E., Holappa, M. Experiences from the use of CORBA. *Proceedings of the 24<sup>th</sup> EUROMICRO Conference*. Los Alamitos: IEEE Computer Society, 1998, Vol. 2. Pp. 989–996.

- VII Niemelä, E., Marjeta, J. Dynamic configuration of distributed software components. Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98, July 1998, Brussels. TUCS General Publication No. 10. Weck, W., Bosch, J., Szyperski, C. (eds.). Turku, Finland: Turku Centre of Computer Science, October 1998. Pp. 61–72.

The author of this thesis is the principal author of the other papers except Papers I and II. However, the author's effort has been essential for these papers as providing application examples and evaluation support as well as being a co-author.

## List of abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
ActiveX	Microsoft's component model
ADL	Architecture Description Language
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CACE	Computer Aided Control Engineering
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CAM	Computer Aided Manufacturing
CAN	Controller Area Network
CASE	Computer Aided Software Engineering
CBSE	Component-Based Software Engineering
CDL	Component Description Language
CIM	Computer Integrated Manufacturing
CMM	Capability Maturity Model for software processes
COM	Component Object Model
COMI	Configurable Module Interface
CORBA	Common Request Broker Architecture
COTS	Commercial off-the-shelf software
CSE	Concurrent Software Engineering
DA	Domain Analysis

EC	Embedded Controller
ECA	Event Condition Action concept
EDLC	Evolutionary Domain Life Cycle
EPC	Embedded Personal Computer
FMS	Flexible Manufacturing System
FODA	Feature-Oriented Domain Analysis
FODAcom	FODA in communications
FORM	Feature Oriented Reuse Method
GUI	Graphical User Interface
IDL	Interface Description Language
ISO	International Standards Organisation
ITU	International Telecommunication Unit
JODA	Joint Object Oriented Domain Analysis
LAN	Local Area Network
LON	Local Operating Network
MES	Manufacturing Execution System
MFC	Microsoft Foundation Classes
MIDL	Microsoft's Interface Description Language
MIL	Module Interconnection Language
MSC	Message Sequence Chart
MTL	Message Transmission Layer
MVC	Model View Controller pattern
ODM	Organisation Domain Modelling method

ODP	Open Distributed Processing
OLE	Object Linking and Embedding
OMG	Object Management Group
OMT	Object Modelling Technique
OOA	Object-Oriented Analysis method
OODB	Object Oriented Database
OPC	OLE for Process Control
ORB	Object Request Broker
OS	Operating System
OSACA	Open System Architecture for Controls within Automation Systems
OSI	Open Systems Interconnection Model
OTS	Off-the-shelf software
PAC	Presentation, Abstraction, Control architectural pattern
PC	Personal Computer
PDM	Product Data Management
PFM	Product Feature Modelling
PLC	Programmable Logic Controller
QFD	Quality Function Deployment
QoS	Quality of Service
RMI	Remote Method Invocation
ROOM	Real-time Object-Oriented Modelling
RPC	Remote Procedure Call

RSEB	Reuse-driven Software Engineering Business
RT	Real-time
RTDBMS	Real-time Database Management System
RTSA	Real-time Structured Analysis
SCM	Software Configuration Management
SWE	Software Engineering
UML	Unified Modelling Language



# 1. Introduction

The life cycle of embedded systems produced by the electronic and software industry is continuously shortening due to the acceleration of technologies and cutting time-to-market. Real-time and embedded systems are integrated into the products in many technology areas, for instance in different kinds of automation systems controlling production and machines.

Historically software reuse focused on reusing code. The use of design methods and CASE tools encouraged reuse of software designs and thereafter, reuse of software architectures and components (Prieto-Diaz 1987; Seppänen 1990; Tracz et al. 1993). There is no mutual understanding of, what software architecture and a component are, and they have different definitions according to the context they are set (Szyperski 1997; Clements 1996; Bass et al. 1998b). Nevertheless, the effectiveness of software reuse depends on how the different viewpoints of a software architecture and components are taken into consideration and committed by their stakeholders. Architectural styles and patterns, product-family architectures, connectors and configuration of components, for example, are different approaches to reuse software architectures and components (Shaw 1995; Bushmann et al. 1996; Bass et al. 1998b; Brown & Wallnau 1998; Bishop & Faria 1996).

A component framework is based on a software architecture, a set of components and their interaction mechanisms. This thesis examines component frameworks applied to the distributed control systems domain. We review requirements for a component framework development, propose a model of a component framework of a distributed control systems family, and demonstrate results with cases drawn from the families of machine control systems, process control systems and manufacturing systems.

## 1.1 The need for component frameworks

Software reuse can be considered from the viewpoints of the organisation, the development process, technology, and products. The organisation sets the long-term objectives by specifying business goals and market segments. The development work has organisational and process aspects that define the work-

allocation, used engineering methods and tools, whereas technology provides alternatives that are limited by the requirements of the distributed control systems family.

Distributed control systems are automated systems that are controlled by embedded microcomputers and programmable logic, as well as personal computers and workstations. Distribution modularises and adjusts the level of automation, and networks, such as field-buses and local area networks, are used for connecting heterogeneous subsystems. Subsystems are developed in concurrent engineering processes, each of which is concentrating on its own aspects, e.g. mechanics, electronics, and software (Rossak et al. 1994). Due to the complexity of the software of distributed systems, the software is normally decomposed into smaller, less complex parts, which are allocated to engineering teams and sub-contractors. Therefore, there is a need for a development method that systematically guides and supports the development of software components, which are interoperable but can be produced independently. Independence of a software component does not only assist in allocating resources but also helps to integrate a system through carefully designed interfaces and guidelines how to use them.

The use of third party components is remarkable in the development of distributed control systems. In process control systems, for example, over 80 % of software components are third party components and the system development by integration is the most general manner in the engineering practice (George & Kryal 1996). Many software development problems arise in the integration phase, often due to missing standard interfaces, resulted from their poor performance, lack of knowledge and training time required for their use. However, the diversity of communication protocols and media is a problem that control engineers are aware of and there is ongoing research into more flexible solutions (Siegel 1996; Cysewski et al. 1998; Polze & Sha 1998). Therefore, the most complicated parts of the distributed software, that is the communication of distributed applications should be standardised and designed as common communication rules used by each application.

Software architecture is often poorly designed and misunderstood by application developers because a uniform architecture description language is missing or it should be adapted to existing modelling methods (Robbins et al. 1997; Clements 1996; Perry & Kramer 1998). Development methods need adaptation to the ap-

plication domain and training to use them. Software architecture is usually described as functional blocks and interconnection between them. Non-functional requirements, called also quality requirements, are defined as constraints in hard real-time and safety critical systems. Designers or, at the latest programmers, make their own decisions how quality requirements are implemented and therefore, the system may have different kinds of policies, for example in error handling and resource allocation. This may lead to an unbalanced and inefficient solution and produce additional problems in integration and maintenance.

Systems themselves are complicated, but it is more complicated to design a software architecture that can be utilised for a product family of distributed systems. Seldom do application developers have such a knowledge or understanding about a product family that they can define, which features should or should not be changed in the systems family. Especially, if the developers work in several co-operating organisations, reasons behind design decisions are often blurred (Dolan et al. 1998).

Diversity in product features, resulted from the needs of market segments, has brought out plenty of problems in software development: inability to describe product variants clearly, to understand descriptions of product variants, and to maintain product variations. There is a lack of modelling methods and techniques for deriving commonality and variability of a product family. The methods are needed for separating generic and specific parts of the software and for documenting them completely (van den Hamer et al. 1998). A method for customising the software architecture and components for a product variant has been lacking. Digre (1998) proposed a component description language (CDL) as a solution for defining semantic contracts for business object components that utilise the CORBA framework as an execution environment. However, it supposes that the domain framework already is available. Customising and configuration is essential when the cost-effectiveness of the software development is considered. The development of software architecture for a product family is time-consuming and expensive, and repayment can happen only by reusing it, in the best case several times during the life cycle in installing and upgrading a distributed control system.

As a summary, in the distributed control systems domain there is a need for the means to develop software to provide a coherent solution for the marketing,

system designers, application development, systems integration, and maintenance. A component framework provides an architecture, a set of components and their interaction mechanisms, and therefore, a component framework has been seen as a concrete and promising means to increase software reuse and share reusable assets among the stakeholders.

## 1.2 Component framework architectures

### 1.2.1 Definitions

*Software architecture* is an abstract and overall design description of a system integrating different issues that are separate but have a contrary influence on each other (Szyperski 1997; Bass et al. 1998b). Component-based software architecture is a structure of the system including software components, the externally visible properties of those components and relationships among them (Bass et al. 1998b). Stakeholders, i.e. people and organisations that are interested in the development of systems, have different concerns that they wish the system to provide. Therefore, the software architecture seeks out for balances between understandability, functionality, and economy and provides the basis for independence and co-operation of software components. A layered architecture is a traditional bottom-up software architecture that decomposes software hierarchically.

Brown and Wallnau (1998) summarises several definitions of *a software component* that diverge from each other concerning the size, autonomy and context of a component to some extent. The large-grained nature of a component emphasis on a business concept and defines a component as a replaceable part of a system, a dynamically bindable package of one or more programs that are managed as a unit. Szyperski (1997) underlines the context of a component and defines a software component as a unit of composition with contractually specified interfaces and explicit context dependencies only. The context defines the required interfaces and the acceptable execution environments of a component. A component is an independent unit of software that is deployed in the binary format and used as a third-party component.

Another viewpoint is taken when reusable design models are also seen as software components (Rossak et al. 1997; Gomaa & Farrukh 1997; Johnson 1997; Digre 1998; Bass et al. 1998b; Brown & Wallnau 1998). In this case, the focus is on the software reuse at the system development level and components are inseparable from its architecture. Because the software components of distributed control systems are not designed to be delivered as products themselves, reusable design models are also considered as software components in this thesis. However, subsystems are components that are deployed independently or they can consist of deployable commercial components, e.g., communication protocols. A component can be layered or it is located within a particular layer of a system's architecture.

A *framework* is a skeleton of a system with an integrated set of components that can be reused and customised (cf. Johnson 1997; Brugali et al. 1997). A component framework of a product-family defines the product features that are fulfilled with a dedicated and focused software architecture, a few key-mechanisms and a fixed set of policies for the mechanisms at the component level (cf. Szyperski 1997). A component framework of a product family has three tiers: subsystem, integration and product-family (Figure 1). The tiers of the component framework point out the views of its stakeholders; application developers, system integrators and system developers, correspondingly. Each tier embodies the component architecture using architectural styles and patterns. A style defines the types of components and a way of their runtime control and data transfer. An architectural pattern is a reusable design that realises a style or several styles.

A *subsystem framework* is a first-tier component architecture that defines the styles, patterns and components for a specific application domain. The tier may also be layered. The difference between tiers and layers is in their focus. A tier concentrates on systems' integration and layers are used for portability and modifiability within a tier. Layers are hierarchical and classified according to the degree of abstraction and generality. A layer hides its implementation details from the upper layer by providing a generic service interface. Component frameworks, first of all, are applied to application domains, for example monitoring systems and graphical user interfaces, and therefore, the first tier represents the domain viewpoint of a system and the tier may also be called a domain-specific framework.

A *component system architecture* is an architecture that consists of a set of platform decisions, a set of component frameworks, and interoperation design for the component frameworks (Szyperski 1997). A component system architecture is a second-tier component architecture that mediates between subsystem frameworks and is called *an integration framework*. The terms ‘platform’ and ‘core assets’ have the same meaning: they describe a set of software solutions that are shared with diverse products (Clements & Northrop 1998). The integration framework focuses on the technology viewpoint of distributed systems and the integration of applications. According to the definition, the Object Request Broker (ORB) of CORBA (OMG 1995) is the first-tier component architecture, when it mediates between components, whereas the ORB represents the second-tier, if it integrates subsystems produced by component frameworks. On the contrary, CORBA services and facilities are layers that utilise the services of the ORB and provide services for the vertical CORBA domains, the top layer.

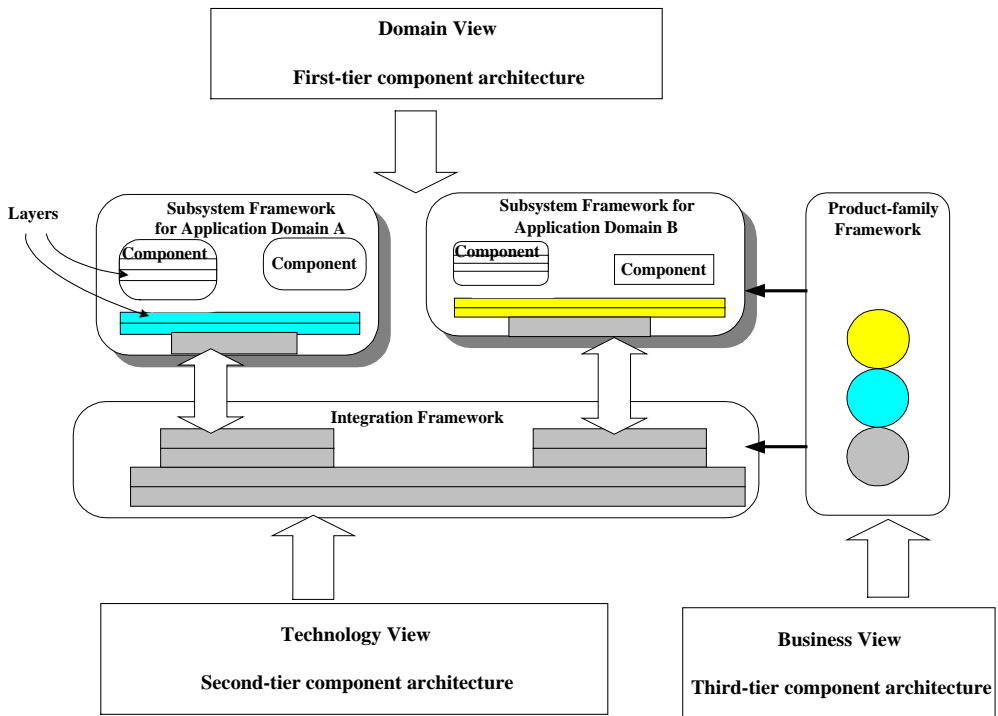


Figure 1. Tiers of component architectures.

A *product-family* is a third-tier component architecture. It focuses on the variability and commonality of a systems family and represents a business viewpoint of the distributed systems. It is an overall design of the systems in a product family. Generalising and abstracting a product-family architecture captures important aspects of the product family and enables the creation of an individual product architecture from a product-family architecture (Perry 1998). Producers of product-lines and product-families are more market-oriented rather than customisation-oriented (Dolan et al. 1998). *Product-line architecture* refers to products that are produced by different production lines and intended to satisfy different market segments. However, control systems are always customised at some level according to the customers' and end-users' needs, and therefore, a *product-family* describes better the characteristics of the control systems domain. A product-family may be more difficult for customisation-oriented system-producers due to indistinct differences between products and a great deal of variations. However, component-based software can also be used for customised control systems, if the flexibility of components could be provided for changing parts of software.

A product-family has the software architecture that every system of the product family is dealing with. Thereafter, the component architecture of a product family is an architecture, which mediates between component system architectures within the second-tier. In other words, a component architecture that supports a product family is a third-tier component architecture. In our context, the third tier is more abstract than the component system architecture that provides its implementation. However, the tier can be called a *product-family tier* according to its focus.

The term 'component framework' is used in two meanings. A singular means a descriptive framework for the tiers of the component software architectures, as it is used in stating the research problem and defining the model of the component framework in Chapter 3. The tiers are instantiated as component frameworks and a plural is used to refer to these instantiations.

### **1.2.2 Applied approaches**

The X-model defines software reuse by two processes: the software development for reuse and the software development with reuse (Hodgson 1991). There are

three main reuse-approaches; a feature-oriented approach, an architecture-oriented approach and a language-oriented approach, that are closely twisted round the development of a component framework of a product family. However, each approach has a diversity of methods that highlight organisational, process, technical and product aspects in different way.

Feature-oriented domain analysis (FODA) is a model-driven approach for requirement analysis that describes functional features of a system (Kang et al. 1990). However, the distributed control systems also have quality, e.g., real-time and safety requirements that have to be defined. The management of the product features of distributed systems is also difficult, owing to the large amount of features that have to be shared among development teams.

Software architecture recovery represents the architecture-oriented approach that aims at identifying architectural patterns in the code and to define, whether matched code can be associated with a component or with the infrastructure (Mendonca & Kramer 1998). An integral hierarchy and diversity models are also used to describe an extendible software architecture for product families (van den Hamer et al. 1998). In this case, variation points are included inside components and feasible combinations of component variants are expressed by explicit connections between interfaces. The model does not describe the guidelines of how to define and manage the component connections and how to add or change components of an existing system.

Agent architecture is a possibility to achieve flexibility at the architectural and execution level (Wooldridge & Jennings 1995; Fisher 1995; Lejter & Dean 1996). Intelligent agents with different kinds of distribution control protocols, e.g., request-response and peer-to-peer strategies are applied to fulfil the adaptability requirements of control systems. Using agents, the focus of the architecture is on the negotiation-based communication. Although the systems' flexibility, achieved by loosely coupled software agents, provides possibilities for software reuse through extendibility, other techniques are required for defining and supporting software reconfiguration.

Port-based objects, as a language-oriented approach, form the basis of a programming model that provides specific guidelines to create and integrate software components, which are designed to be dynamically reconfigurable (Stewart



et al. 1997). The approach was used for robot control systems and it supports the domain characteristics in respect of real-time and configuration aspects, which are integrated with the operating system. However, most control systems also use commercial software, and so far a special framework can be reused quite restrictedly.

Domain-specific frameworks, mostly based on object-orientation, are believed to be the answer to reduce a product's time-to-market and provide the opportunity to respond to new and rapidly changing markets (Schmidt & Fayad 1997; Codenie et al. 1997). A domain framework is based on a reusable software architecture defined by collaboration contracts between classes and a set of variation points, where the framework can be customised. Therefore, a domain-specific framework is a first-tier component architecture with the support for a product-family tier. The collaboration contracts define the rules that customisation must obey. In this case, the software reuse is based on reusable object classes and the systems should be constructed by using the selected domain-specific framework. In the control system domain, systems are based on concurrent development processes and commercial components that are integrated together. Therefore, a domain framework alone is not enough but provides a partial solution for the software reuse of distributed control systems.

Object-orientation is a key-technology in the development of product-family architectures and software components. A domain framework that is a combination of patterns and components provides a practical way to manage and share reusable assets in a focused domain. Therefore, a component framework is examined as a means to carry out reusable assets for a distributed control systems family in this work.

## **1.3 Scope of the research**

### **1.3.1 Application areas**

The research is focusing on the characteristics of the three levels of the distributed control systems: manufacturing execution systems, process control systems and machine control systems (Figure 2). The levels may be considered as hierarchical layers that use the operational services and data produced by the lower

layers (Törngren & Wikander 1992; Ferguson 1995). In this context, each layer represents different kinds of timing requirements and implementation technologies.

An exceeded timing requirement that always causes an error in hard real-time systems, is occasionally accepted in soft-real time systems. Embedded controllers (EC) with hard real-time requirements are typical for machine control systems. Embedded personal computers (EPC), programmable logic controllers (PLC) and personal computers (PC) with mixed or soft real-time requirements are used in process control systems level. A process control system, in the used context, is a system that controls several machines and their co-operation. It can be a cell control system in a manufacturing system or a repayment control system that manages the repayment process of bottles and cans. The manufacturing execution systems are mission critical systems that mostly utilise commercial software components, for instance databases and protocols, and PCs and workstations as execution environments.

Manufacturing execution systems (MES) are task-oriented systems that are responsible for making optimised execution plans carried out by process control systems (Greenwood 1986; Ljung 1986). Manufacturing systems handle and manage state information and adapt their control by performing transactions that may consist of the operations performed in several process control systems. Therefore, the level is called the transaction level. The transaction level can consist of soft real-time requirements, but concurrent transactions that have to be performed according to the ACID properties (Atomicity, Consistency, Isolation, and Durability) are more typical (Barry 1994; Kappel & Vieweg 1994). Computer integrated manufacturing (CIM) systems and flexible manufacturing systems (FMS) have the same kinds of properties. They both are highly computerised and they control the workflow that produces products. However, FMS can produce different kinds of products at the same time and they also control the tools used in the production line. The local-area network is nowadays used as a distribution medium in manufacturing systems.

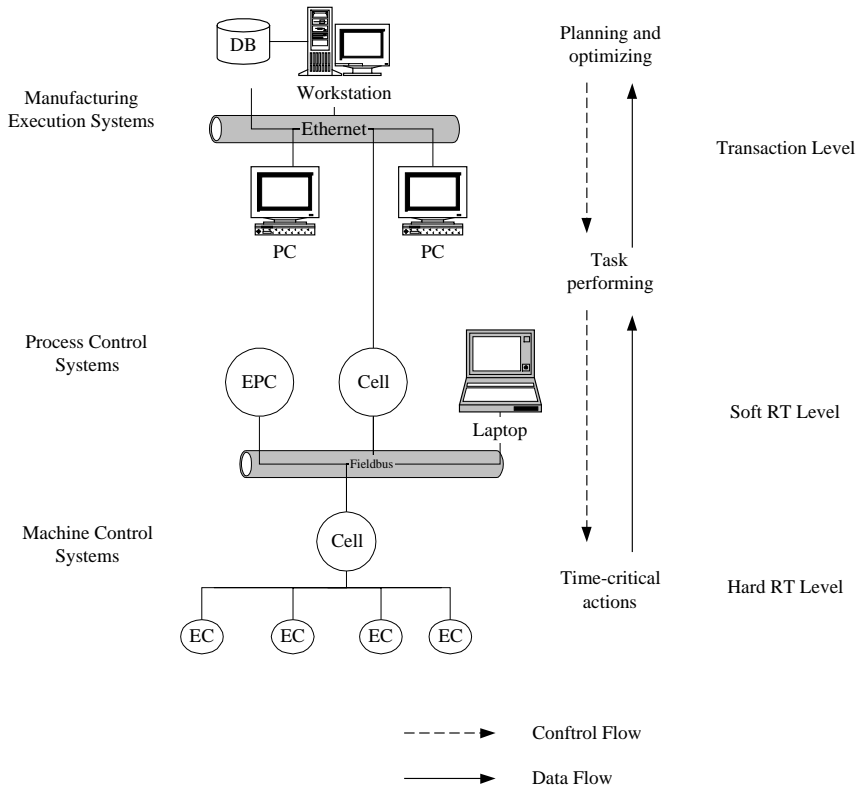


Figure 2. Control system domains.

Process control systems perform a single phase of a workflow in a defined manner and time. The external circumstances of a control system are known and the variables that affects to its behaviour are changes in the controllable objects or machines, which perform the commands the process control system dispatches to them. In the most cases, process control systems require smooth balancing between unpredictable events and continuous operating and therefore, the process control level is considered here as a soft real-time level, however, they can also have hard real-time constrains. Local area networks and field-buses are used as distribution media in this control level.

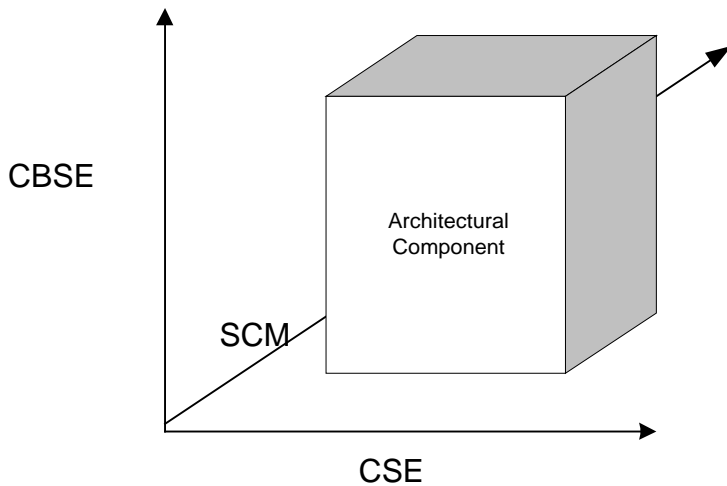
The most accurate timing requirements are set for the control systems that operate in ever-changing world. These systems have to respond to external and inter-

nal events by scheduling tasks according to their priorities and available resources. A machine control system is an exemplar at the hard real-time level. Machine control systems are based on controllers that co-operate through the distribution media, e.g. field buses, and perform the distributed operations tightly-coupled with data and its timestamp (Kopetz et al. 1989; Törnngren & Lind 1994).

Numerical controls, PLCs and robot control systems are out of the scope, as well as Computer Aided Manufacturing (CAM). The component framework is based on the software engineering practice used in networked embedded systems, i.e. embedded controllers, EPCs and the systems that they are connected to.

### **1.3.2 Research areas**

The development of a component framework supposes knowledge of the technology, products, and their development process. System integrators' and maintenance engineers' knowledge is also required to understand how the component framework is going to be used. Therefore, this research concerns the software engineering research area from several viewpoints (Figure 3). Firstly, the component-based software engineering (CBSE) highlights requirement analysis and software architecture and components. Requirement analysis is based on feature-oriented domain analysis performing conceptual knowledge of the control systems domain that pays attention to the products and their markets, i.e. the business beside products. Software architecture and recovery are based on the knowledge produced by the analysis. Reverse- and re-engineering are used to develop component-based software and architecture. Distributed systems are constructed by incremental integration from subsystems developed by concurrent software engineering (CSE). This special feature is noteworthy and affects software architecture and components, and above all the features, which are widespread in the systems and require sophisticated methods to be managed and traced by the software configuration management (SCM).



*Figure 3. Views of the research area.*

Product data management (PDM) examines how information technology can assist to produce complex products effectively. The view of PDM is on methods and tools used at the organisation level, whereas we consider the product features and the way they can be mapped to the product-family architecture and software components. We also study what kinds of mechanisms systems need for managing variability.

Concurrent and incremental software development has usually been considered as a part of software development process without studying how the characteristics of the development process affect the product itself. This clear separation between process-oriented and product-oriented software engineering has led to unrealistic anticipations of the effectiveness of software reuse. The concurrent development is considered as regards software reuse in this research.

Reconfiguration is required due to the domain characteristics. It has to be integrated to the systems and considered at a product-family level while a component framework is developed. Traditionally, software configuration management is a part of the software development process, not a run-time feature of a system as it is considered in this work. The management of product features and the mechanisms incorporated into a product provide a part of application management in the change-oriented configuration management (Taramaa 1998). In our

case, the software configuration management is an integrated property of a product that supports the evolution aspect of distributed systems.

Although, the process-oriented viewpoint is mainly out of the scope of the thesis, the concurrent software development and software configuration management are the research areas closely connected to the development of a component framework.

## **1.4 Problem statement**

### **1.4.1 Research problem**

Based on the previous discussions of applied approaches and the scope of this work, the research problem is defined as follows:

What kind of a component framework supports the development and evolution of distributed control systems' family?

The control systems domain, the development of a component framework, and the development and maintenance of control systems set different requirements for the component framework. Based on the different views to describe requirements for a component framework, the research problem can be stated as sub-problems that mirror technical (Q1, Q2), process (Q3) and organisational (Q4) viewpoints:

- Q1. What requirements does a control systems family set down for a component framework?
- Q2. What kind of a component framework supports the evolution of a distributed control systems family?
- Q3. How should the component framework of a product family have to be developed?
- Q4. How does a component framework of a product family change the component-based software development?

Our tentative approach to the research problems is as follows:

- Marketing aspects of systems products are reflected on the end-users' requirements, the existing knowledge of marketing area and systems products. The research problem is reduced to finding such a modelling method that facilitates explicit requirement definition as product features and their evolution.
- Product features are classified according to the type of a system family and its constraints to the development process and the execution platform. The classification assists in analysing the essential features and constraints of a product family and defining the style of software architecture and components.
- A part of the software architecture and components stays stable over 5–10 years in spite of variability inside a product family and the evolution of the system products. By isolating the common software from the variable software, a generic software platform for a systems family can be developed.
- The software platform, designed according to the needs of the development and maintenance, provides sufficient flexibility that the software platform stands up to the changes in implementation technologies and the needs of end-users.

The hypothesis can be summarised as follows:

A component framework of distributed control systems has to be based on the features of a product family which are mapped into application components and the software platform that supports the incremental development and maintenance of the distributed control systems.

The problem analysis and the component framework presented in Chapter 3 answer the first two research problems. Practical experiences, presented in the thesis and papers, answer the two latter questions.

### 1.4.2 Research assumptions

Existing systems and domain experts' knowledge are necessary to collect domain knowledge. If documentation is not up-to-date, reverse engineering is used to bring back the designs of existing code. Re-engineering is required to improve architecture and components of existing systems for software reuse. However, the reverse-engineering and re-engineering do not emphasise why domain analysis is required and for whom its results are intended. Software components and architecture are quickly ruined and software reuse seems to be ineffective, unless the software development process is improved at the same time.

The focus of architecture recovery is on improving software architecture systematically. It emphasises the evolutionary aspects of software architecture, but it ignores those who are using the architecture, and therefore, the results of the recovered architecture are not utilised as extensively as possible. The architecture recovery is a time-consuming and an ineffective way to develop a product-family architecture. This is due to undocumented design decisions of existing systems and high-level abstraction that hampers the understanding of the specifications of a product-family architecture.

Application frameworks focus on components that are reused by means of inheritance and polymorphism in common and special application areas, and the frameworks are used as existing resource for application development by changing class hierarchies and interfaces. System-level software architecture and the means to upgrade systems are rarely considered. This is a repetitive problem when distributed systems, which have a graphical user interface based on an application framework, have to be upgraded or new features have to be added to them.

We believe that the viewpoints of different stakeholders of the software architecture are essential to develop a component framework successfully. A product-family architecture, derived from the features of a product family or families, takes account the aspects of marketing staff. The product-family architecture is the basis of a component framework, but it has to be developed for the users of the component framework, that is for system integrators and maintenance staff. We believe that if the different viewpoints are combined, and the product features are kept as a leading thread in the framework development, the component



framework is enough of an efficient and effective means of software reuse and investments for it are acceptable.

### **1.4.3 Research methods and results**

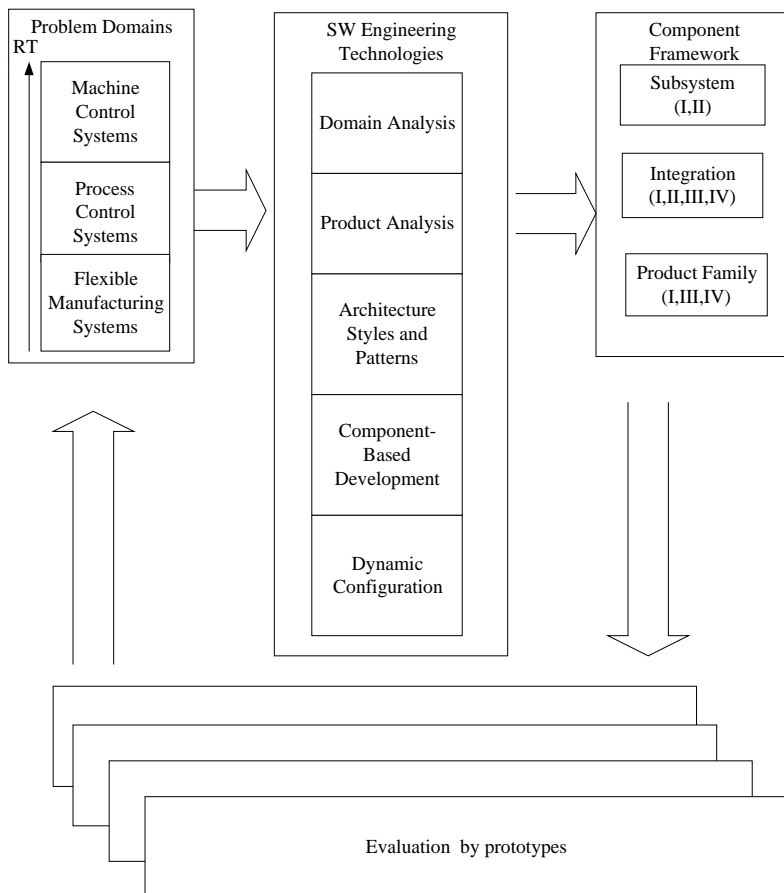
Several research methods were used in the different phases of the research. The first phase was mainly analytical, and industrial case studies were the way to obtain domain and product knowledge, to understand the problem domain. An analytical method was used to describe the requirements and features of the application domains and products. As information, we used literature, the results of interviews and inspection meetings. Literature of control systems, information systems and software engineering were used. Several interviews of industrial partners and domain experts were made and descriptions were inspected with the industrial representatives.

In the second phase, engineering methods, for example FODA, ROOM and OMT, were used to create a component framework that was applied to construct the prototypes of the case studies. The significance and the relevance of the framework were measured and analysed as regards the requirements of the application domains: machine, process, and manufacturing systems. Both quantitative and qualitative analyses were made.

During the third phase, a generic model of the component framework was formulated and analysed by reflecting the results of the case studies (cf. Glass 1995; Pfleeger 1997). Evaluation criteria for the trials are derived from the results of the analytical phases of the case studies. The state variables are classified into the four main categories according to the characteristics of the product family, used methods and tools, development process, and end users. Response variables are derived from the requirements of the prototypes and they are defined as results of the problem analysis in Chapter 2.

Figure 4 depicts the followed research strategy summarising the problem domains, software engineering technologies, parts of our solution and the application domains, where the constructed prototypes were evaluated. Arrows indicate the process and its repetition. Problem domains represent the three levels of timing requirements and the size of the systems. Machine control systems have hard real-time requirements, process control systems are mainly soft-real time

systems, and flexible manufacturing systems represent mission critical systems. The size of the systems is contrary to the timing requirements, i.e. hard real-time systems are mostly small systems. Although the size and the timing requirements of control systems are different, the application domains may be physical layers of a manufacturing system. Therefore, the component frameworks can be applied in these layers, as they are applied in the case studies. Correspondingly, Papers I, III and IV concern the machine control systems domain, Paper V the process control systems domain, and Papers VI and VII the manufacturing systems domain.



*Figure 4. The applied research strategy.*

The software engineering technologies define the areas used in the development of the prototypes of the component frameworks. In addition to the above-mentioned papers, Papers I, II and III focus on used techniques: re-engineering and feature modelling combined with reusable software architectures. The tiers of the component framework are called the subsystem, integration and product-family tier according with their focus and use. Correspondingly, the numbers of trials, referred in parenthesis in each tier in Figure 4, mirror the focus and results of each case study, which are described in detail in Table 1. It also gives a brief description of the applied approach and the research contribution of each case study.

Table 1. A summary of the trials to develop a component framework.

Description	Results and main contributions
<p><b>Trial I: The KIURU project 1994–1995</b></p> <p><i>Goal:</i> Software architecture and components for interoperable subsystems.</p> <p><i>Approach:</i> Domain analysis and re-engineering were used for transforming slightly from RTSA to object oriented technology. The ROOM method and ObjecTime tool were applied for modelling and simulating component-based distribution architecture and a component framework for subsystems. Prototyping was used for validating components and subsystems incrementally.</p>	<ul style="list-style-type: none"> <li>▪ <i>Subsystem:</i> A layered component-based software architecture with responsibility-driven classification of components and their interfaces.</li> <li>▪ <i>Integration:</i> System and application development processes were isolated. The dispatcher pattern applied for a generic communication component of the subsystems.</li> <li>▪ <i>Product family:</i> Mode-based configuration of the communication components.</li> </ul> <p><i>Contribution:</i> A layered component architecture for a sub-systems family in the distributed machine control domain.</p>
<p><b>Trial II: The DYNAMO project 1996</b></p> <p><i>Goal:</i> Software architecture and mechanisms for dynamic configuration of architectural components.</p> <p><i>Approach:</i> Mechanisms for dynamic configuration was constructed by using the OMT method and C++ on the QNX operating system and its messaging services. Photon was used for developing presentation components of the applied PAC architectural pattern.</p>	<ul style="list-style-type: none"> <li>▪ <i>Subsystem:</i> Agents as independent medium-grained architectural components that separate application logic from its presentation.</li> <li>▪ <i>Integration:</i> An integration frame was applied as a part of location-independent architectural components. Configuration mechanisms were implemented as a part of the integration framework.</li> <li>▪ <i>Product family:</i> A tool for the on-line configuration of architectural components</li> </ul> <p><i>Contribution:</i> A component framework for dynamically configurable applications using layers and PAC agents in the MES domain.</p>

Table 1. A summary of the trials to develop a component framework continued.

Description	Results and main contributions
<p><b>Trial III: The DYNAMO project 1997</b></p> <p><i>Goal:</i> Software architecture and components for different product variants.</p> <p><i>Approach:</i> The features of the product family was described by the OMT method and allocated to the default and optional properties of components. A simulation model that included the integration platform and independent applications was used to validate the software architecture.</p>	<ul style="list-style-type: none"> <li>▪ <i>Subsystem:</i> Reactive agents as location-independent logical subsystems.</li> <li>▪ <i>Integration:</i> An application-specific software bus as an integration platform. The centralised data management and configuration support was implemented as a part of the framework.</li> <li>▪ <i>Product family:</i> Configuration through the platform by a configuration tool. The development of the applications and the software platform was isolated.</li> </ul> <p><i>Contribution:</i> A component framework of a product family by means of the independent components and data centred repository architecture styles in the domain of process control systems.</p>
<p><b>Trial IV: The ARTTU project 1997–1998</b></p> <p><i>Goal:</i> Component-based architecture and platform with the COTS and OTS.</p> <p><i>Approach:</i> A features model and scenarios described the functional and behavioural features of a product family. CORBA RPC was combined with Event-Condition-Action (ECA) concept. The ECA executive acted as a centralised propagation point and an externalised binding mechanism for product features.</p>	<ul style="list-style-type: none"> <li>▪ <i>Subsystem:</i> A dispatcher as a component of the subsystem framework.</li> <li>▪ <i>Integration:</i> An integration framework by applying commercial components, a CORBA and an OODB, with an ECA executive and an adapter for the legacy systems.</li> <li>▪ <i>Product family:</i> The management of the product features was implemented by the rule database and ECA executive.</li> </ul> <p><i>Contribution:</i> A component framework of a product family using the client-server and rule-based styles in the FMS domain.</p>

In the last trial, a model of the component framework of a distributed control systems family was created by synthesising and analysing the results of the earlier phases. The influences of the different control domains on the component framework are analysed by reflecting to the experiences received in the case studies. This thesis includes the results of the last trial.

## **1.5 Outline of the dissertation**

Chapter 2 analyses the problems that the software developers have in the development and maintenance of distributed control systems. These problems are taken as the requirements for the component framework presented in Chapter 3 that describes a component architecture for each tier of the component framework, the subsystems, integration, and product-family tier.

The development and evaluation of the component framework are studied in the following two chapters. The tiers of the component framework are presented and demonstrated by the developed prototypes.

Related work is reviewed and compared to the results of the thesis in Chapter 6. An introduction to the included papers is given in Chapter 7 and Chapter 8 draws the conclusions of the thesis.

Papers I to VII are presented in appendices.

## 2. Problem analysis

This chapter studies the problems that appear in the development of component-based software of distributed control systems. The aim is to define critical factors for successful software reuse and set the requirements for the development of a component framework. The definition of the key factors is based on empirical analysis in the projects related to the product families of distributed machine control systems, automatic repayment control systems and flexible manufacturing systems. The discovered problems are emphasised by other findings of component-based software engineering and control systems, as well as the literature used in the development of the prototypes. This study is used as requirement analysis for the component framework that is presented in Chapter 3.

### 2.1 Component-based development of a product family

The development of a product family is based on (Soininen 1997):

- a basic product that can easily be modified, extended and customised, or
- a core-product that itself is not a product but embodies the core technology and know-how of the development organisation.

In the former case, the benefits from a product family are achieved by improving the development process. In the latter case, the dominant factor is the management of technology and interfaces of the core-product. In the distributed control systems, the core product covers in-house components and commercial components that require special expertise to develop and apply.

The component-based software development of distributed control systems can also be seen as two information flows, the one for managing reuse assets and the other for product data management (Figure 5). Software, mechanical and electric engineers produce components that are used while constructing products. Although not all components are software, the people in the production-chain are involved in software architecture and its components, because all components have to match together. Product managers who are responsible for the deliveries of the systems need to know the product data, the whole information produced

during the development, from the order to the delivery. Therefore, marketing people are also involved in the product features that are realised by using the product family architecture.

Component-based software engineering is divided into domain and application engineering (Bass et al. 1998b; Sodhi & Sodhi 1999). Domain engineering produces the reuse assets that are utilised in the application development. From the viewpoint of the system development, component-based software engineering has the system and component development processes and an integration and configuration process for assembling a system from the reusable assets.

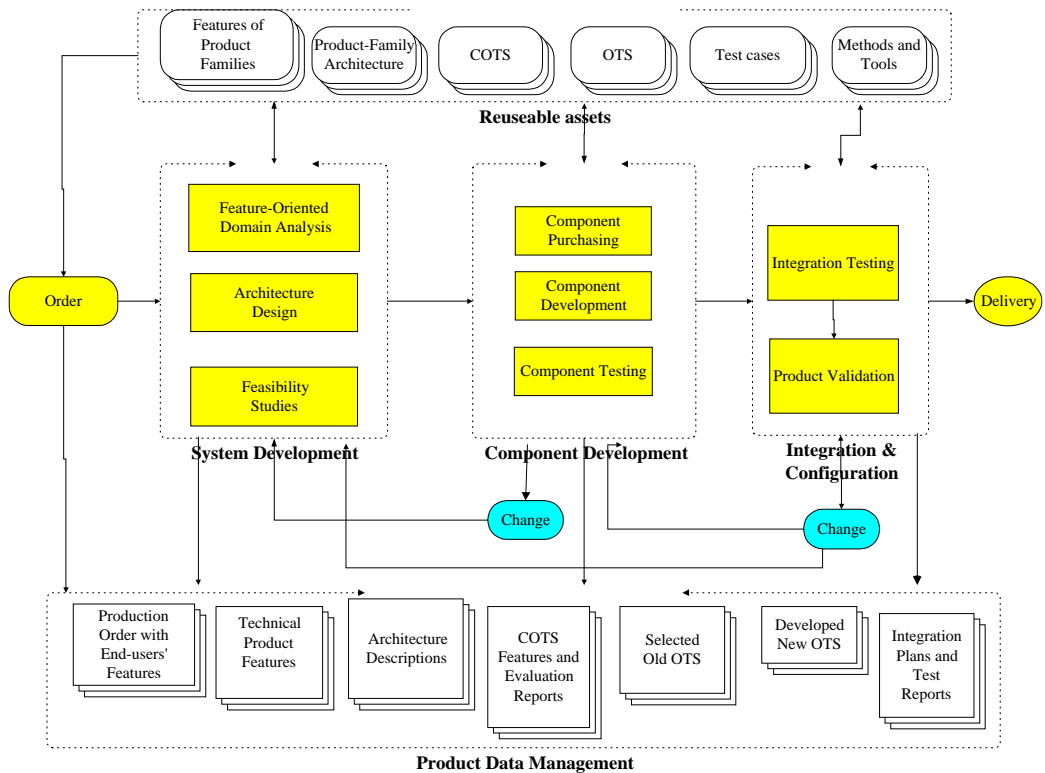


Figure 5. Component-based software engineering.



## 2.2 Problems in the component-based software development

Problems in the component-based software development of distributed systems can be classified to product, technique, process and organisation dependent problems. The component-based software development aspires to components that are used in a product family. Therefore, the software development consists of the problems that result from the shortening life cycle of products and the aim to reduce the time-to-market. Software engineers should also have knowledge of existing and forthcoming features of product variants to be able to design a product-family architecture. If there is no formal way to share information between marketing and software development staff, the information mismatch leads to improper products and lengthening production time. Software engineers also have difficulties to figure out the differences between variants and versions. (Bosch 1998). Variability describes differences in a product family and versions are different instantiations of the same variant. Variants mirror the dissimilarities between products and versions reflect the evolutionary aspect of components (Figure 6). Nevertheless, variants are not stable, but their life cycles are longer than versions’.

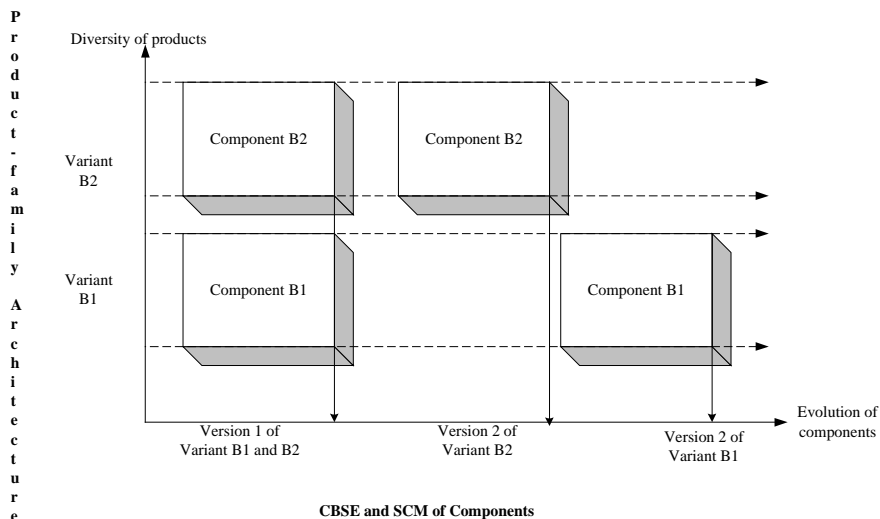


Figure 6. Variants and versions of a component.

Technical difficulties of gathering and reusing the domain knowledge appear while knowledge, spread by ad-hoc manners, is tried to be formalised by using semiformal models that do not adhere to strict rules and lead to semantic mismatch. Several modelling methods, architectural description languages (ADLs) and tools are proposed and evaluated in industrial case studies (Jacobson et al. 1997; Clements 1996; Griss et al. 1998). However, tools may be inappropriate or they are unable to be customised according to their end-users' requirements (Bosch 1998).

Process-dependent problems appear as an inability to trace and manage variability between products and deliveries due to an immature software development process and software configuration management. It has been estimated that the maturity of the software development process should be at CMM Level 2, at least concerning the software configuration management (Bass et al. 1998a). The mapping between component versions and variants should be managed so that the previous products can be re-created easy and dependably.

Nevertheless the software configuration is mature enough, the organisation may have problems in the utilisation of commercial components and in the allocation of the software development to development teams and subcontractors. Researchers have paid less attention to these complex problems that arise from crossing organisational boundaries. Dolan et al. (1998) have discovered the stakeholders of a product-family architecture, but he ignores the integrators' aspect of commercial components and services. The evaluation is simpler in the case of COTS, but most control systems require special components that are produced by subcontractors. Therefore, they accompany an important factor in the component-based software development of distributed control systems.

In the following chapters, we shed light on the problems encountered in the software development of the distributed control systems families and possible approaches to solve them.

### 2.2.1 Domain analysis

The following aspects are essential in the domain analysis of distributed control systems:

- distribution degree,
- functional and quality requirements of the products,
- the commonality and variability of a product-family, and
- skills of the development team.

The distribution degree is a technical factor that affects used methods and technologies. Distribution is understood as a distribution of data, functions, and control (Lawson 1992). Data can be collected into a centralised database or it can be distributed by federative databases or intelligent agents (McCarthy & Dayal 1989; Hawryszkiewicz & Rose 1995). A functional architecture, which is often used in control systems, decomposes functionality of a system to several communicating subsystems. Functions and data may be distributed, but control is still kept centralised. The solution is less error-prone than distributed control, but may be ineffective and inflexible (Kappel & Vieweg 1994). Autonomous subsystems implemented as agents that have a common goal are more flexible, however, the design is more complicated and time-consuming. Although the distribution is only one factor, it is a fundamental one, which affects other requirements, for example extendibility, scalability and safety.

Traditionally, system requirements are defined as functional requirements and other requirements are defined as a set of constraints. Feature-oriented domain analysis (FODA) introduced functional features from the end-users' perspective (Kang et al. 1990). FODACOM is an enhanced model-driven approach for requirement analysis that describes end-users as actors and their interactions by a use-case model, in which extension and parameter points can be defined (Dionisi Vici et al. 1998). In the enhanced RSEB method, use-case models describe the capabilities of a system for the users and system engineers, and on the contrary, features models assist re-users and domain engineers (Jacobson et al. 1997; Griss et al. 1998). However, the common problem is how to keep the features

model simple, understandable, and consistent. The management of product features is more important in the development of large systems, owing to collaboration among several development teams.

Above-mentioned feature modelling methods consider functional properties of systems and the variability inside them. Quality requirements, i.e. non-functional requirements, are seldom included in the domain analysis or explicitly described. Bellay and Gall (1998) have categorised architectural properties, such as safety and variance that are beyond design descriptions, and therefore, represented in an implementation instead of a design. The essence is that quality requirements should have to be described in software architecture. In practice, safety and reliability requirements, however, are defined in later design phases and implemented as aspects of software components (Kiczales et al. 1997). Time-threads are scenarios that capture behavioural requirements as causality flows in relation to activities and components. Although time-threads only can be made explicit in software architecture design, when partitioning of components is clear, they keep the focus on performance and robustness issues in the analysis phase (Buhr & Casselman 1993).

Quality Function Deployment (QFD) is a method for analysing the quality of design decisions in the development process (Day 1993). The main tool, the House of Quality matrix plays the central role in defining the link between the customers' needs and the technical requirements. In addition to the customers' needs and technical features, it consists of the analysis of the customers' satisfaction, correlation between needs and features, correlation between features and comparisons with competitors' products. The structure of the matrix is not fixed but it can be modified according to the need of the organisation.

The engineering of control systems is nowadays based on the use of CAD and CAE tools that are heavily affecting how design results can be transferred between concurrent development processes of a control system. The main problem is that engineering tools do not assist engineers to understand each other, and therefore, domain knowledge is ineffectively reused (Seppänen et al. 1995). The key-point in the software development is to get a balance in work-allocation, team-members' responsibilities and supporting methods and tools. Therefore, design methods have to be adapted to the skills of the development team and their working manners. If the skills of developers do not meet the requirements

of the selected method, the desired benefits of new technology are not attained (Macala et al. 1996).

This work focuses on technology-independent methods and existing tools to describe and validate product features.

### **2.2.2 Feasibility studies**

Distributed control systems are heterogeneous systems combined by using different hardware and software technologies as execution platforms. The diversity of used technology produces problems for the component-based software by demanding a set of feasibility studies. Because the results of the feasibility studies are knowledge that should also be reused in the design and implementation of software architecture and components, they concern:

- commercial off-the-shelf components,
- legacy systems and components, and
- software architecture styles and patterns.

The feasibility of commercial components, e.g., operating systems, communication protocols, databases, and GUI development tools need to be studied. COTS can save time and money, but their use requires guidelines for certifying (Tran & Liu 1997; Voas 1998). Interfaces of COTS components may be complicated and too slow, also they could restrict programming languages, component models and software architecture styles. The quality of COTS components may be weak or a producer can be unreliable. Because commercial components are mostly black-box components, delivered in the binary format, their quality is hard to find out. If COTS is a key-component, i.e. a component of the core-product, its dependability and safety have to be able to be evaluated thoroughly. Black box testing is proposed to be used in defining the risks of commercial components (Voas 1998; McGraw 1998). Testing takes time, and therefore, the selection of COTS presumes a quite long-term commitment. The same problem is with legacy systems and legacy software, however, their quality is better known.

Existing systems and experiences are the basis of the feasibility studies. Architecture styles and design patterns are collected and documented experiences to solve the same kind of problems (Shaw 1995; Shaw & Garlan 1996; Schmid 1996b). However, their suitability has to be evaluated as regards product variants and used technology. Architecture styles and patterns may be technology-dependent, for example object-orientation, and require professional skills that may be missing. Therefore, the use of new technologies provides challenges that development teams have to accept and be committed.

Although the feasibility studies are done in practice, their results are poorly documented and their effects on the other requirements, defined in the domain analysis phase, are not considered. Thus, domain and feasibility analysis are iterative.

The aim of the work is to discover how COTS, legacy software and architectural styles and patterns can be used in the development of a component framework.

### **2.2.3 Software architecture**

The software architecture of a product family balances conflicting requirements. Architecture is based on the results of the domain and product analysis and feasibility studies by matching them to the style, structure and flexibility. The problems in software architecture design of a product family concern:

- available documentation of software architectures,
- architecture description languages,
- validation and verification techniques of software architectures, and
- selection of techniques for component design.

Although the same software architecture is used for product variants, the software of a new product is tailored from the former rather than made by selecting components, that is a product family is based on a basic product. The problem is in the non-identified differences between products. Because the code is mostly

the only up-to-date and available information, the identification of variation is difficult and time-consuming.

Contradictory requirements, which have to be balanced in a software architecture, are difficult to prioritise, and therefore, several design cycles are needed. Distribution and product variants bring additional difficulties to the software architecture design and supporting tools are necessary. Modelling languages consider behavioural aspects of systems, contrary to ADLs that concentrate on representing components and their interconnections. Although modelling languages may also support software architecture design, their capability to describe variability and contextual dependencies is mostly inappropriate (Selic et al. 1994; Burns & Wellings 1995; Bass et al. 1998b). However, modelling and prototyping have proved to be valuable while discovering a balanced solution for quality-in-use and real-time requirements (Savola et al. 1995; Alonso et al. 1998). Duenas et al. (1998) also defined an evaluation model of quality requirements and proposed that the common metrics of internal quality have to be adapted to ADLs. On the other hand, the software architecture of a control system is described in practice at the information level without an ADL. Engineers also prefer target systems as validation environments to the simulation tools.

The black-box and white-box techniques are common design techniques in component design. The black-box method with parallel components is useful, if clear differences between component variants are defined. The white-box technique relies heavily on the properties of the object-orientation, i.e. inheritance and dynamic binding. However, it is a noteworthy approach when the scope of a domain is quite small and a great deal of frequently changing variations has to be managed. The weakness of the technique is that developers have to have intimate knowledge of the internal structure of object classes (Fayad & Schmidt 1997).

The characteristics of the software development process affect software decomposition. Dependencies of components dictate how the work has to be allocated to subcontractors and team-members. The communication of the engineering teams should be intensive and formal, if architectural components have a great deal of dependencies, whereas loose dependencies make it possible to use different design methods and tools. If subcontractors are forced to use a method or a tool that they have no experience in, problems may arise. Therefore, a uniform

design method can seldom be used in the development of distributed control systems.

This work examines the means to define a product-family architecture by using heterogeneous design methods and tools.

### **2.2.4 Component design and implementation**

The interface description and the scope of a component define its reusability. Components can be classified to architectural components and those that are reused in the development of architectural components. Because the context, for which the component is designed, defines the interfaces, the component designers have to know the software architecture, i.e. where the component is desired to be used (provided interfaces) and what its dependencies are to other components (required interfaces). However, this kind of information is often lacking or incomplete. The problems concerning the component design and implementation are mostly some of the follows:

- the lack of interface descriptions,
- the diversity of component models, device and communication platforms, and
- technology-dependent testing tools.

While object-orientation is preferred, an environment-independent definition language, such as the IDL can be used. Most control systems are modernised gradually and they use several programming languages, e.g., C, C++, and graphical programming languages. Although it is unrealistic to suppose that the IDL could be used in all cases, modern languages can be used parallel with traditional ones.

CORBA, COM and Java components are the three main component models. The CORBA model is language-independent and its focus is on object-orientation at the enterprise level. On the contrary, Microsoft's COM model is designed for the desktops and is based on C. Java components, as also COM components, are language-dependent. However, all these models can be applied in control sys-



tems in the area where they come into their own (Polze & Sha 1998; Santori 1997; Lange 1998). The problems, caused by heterogeneous languages and component models, are attempted to be solved by bridges that require more computing resources and may be inappropriate for real-time and embedded control systems.

Distributed control systems normally uses more than one device and communication platform, which brings out the need for portable applications and services. Product families can be classified to the four classes according to the common property: technology, the objective, the use (a set of devices) and generation (Soininen 1997). The objective of a control system is to control, e.g., machine, process or manufacturing systems, and these subsystems may be used together. However, the used hardware can be totally different. The problem can be solved by isolating differences, for example by leaving out the product that uses totally different technology, or in some case, selecting an architecture that is able to support the diversity of hardware platforms.

Software components have to be verified that they are consistent with the architecture and their implementations fulfil the quality requirements. However, tools are often technology-dependent, and customisable testing and simulation tools for heterogeneous engineering environments are not available.

The focus of this work is to find out the contribution of the IDL for heterogeneous distributed control systems and a practical testing technique for software components.

## **2.3 Problems in integration**

In the system integration phase, software components with syntactical and semantic differences prevent their inter-operating temporarily or permanently. Temporal differences can be avoided by using adapters, filters, type casting, polymorphism, or a wrapping technique. Semantic differences are aspects that are designed according to some policy. The policy defines how some restricting features, i.e. aspects, are implemented all over the system architecture. Kiczales et al. (1997) define an aspect as follows:

*“A property that must be implemented is an aspect if it can not be cleanly encapsulated in a generalised procedure. Aspects tend not to be units of the system’s functional decomposition, but rather to be properties that affect the performance or semantics of the components in systematic ways.”*

Reliability, performance and safety requirements are allocated to strategies, through which restricted resources and unordinary situations are managed. Aspects are properties of software components that are based on the design decisions, made under software architecture design and they can not be changed or are difficult to change afterwards. However, the selected policy determines how easily or not the properties of components will be changed in the system integration. Examples of policies implemented as aspects are:

- Communication policies applied to communication media.
- Scheduling, synchronisation and allocation policies used to computing resources.
- Security and synchronisation policies applied to data and states.
- Memory allocation policies used to manage memory resources.
- Error handling policies for exceptions.

These aspects have different kinds of realisation in the hard real-time, soft real-time and transaction levels and the scalability and extendibility indicate if these aspects have been taken into consideration in the development of a component framework.

Co-operation of distributed systems may be seen as a set of aspects, which define policies and mechanisms for interactions between components. Communication policy defines how the distribution media can be used. The synchronisation of data and states describes how the balance is kept between computing systems. Resource allocation policy defines how computing resources is managed. Lack of co-operation is found in problems with component interfaces, interoperability, and adaptability.

### 2.3.1 Interfaces

System integrators have two kinds of problems in interface technology: problems with user interfaces and subsystems. The features of graphical user interfaces are the first and the most impressive properties of the systems. While trying to fulfil the end-users' needs, developers have encountered enormous problems with the use of GUI tools and application frameworks. Tight coupling between GUI and application logic, non-portable software, and lack of configuration support are the main reasons. Reconfiguration is also needed to customise user interfaces for new end-users and environments.

Although industrial device and system producers mostly use standard interfaces, the variety of standards makes it almost useless. Interfaces may be defined and documented correctly, but additional conversions are needed. Lack of infrastructure results in the conversion code being multiplied, and therefore, computing and memory resources are lost. The reason for this is the software architecture that is defined and maintained independently of the component development. A hot-spot architecture and building blocks with export and import interfaces are proposed as a solution (Schmid 1996a; van der Linden et al. 1995). In the former case, the interfaces of subsystems are described as sub-classes from the base class. It resolves the variability problem at the application level, if the analysis produces a base class with a generic interface for a set of subsystems. However, it does not resolve the diversity at the communication level, that is the diversity of protocols and media. The latter solution introduces a binding interface that links interfaces at different layers. The approach supposes that the interface is known in the development phase, and therefore, it provides flexibility for software development, but not for integration.

This work aims at providing the means to integrate subsystems in a systematic way and isolate GUI components from the application logic.

### 2.3.2 Interoperability

*Semantic differences* between components appear as interoperability problems that may concern data, functionality, states, communication and timing. The development of predictable and reliable software for real-time control systems is difficult due to differences in timing, communication, and state information. On

the other hand, data and functional consistence are the characteristics of process and manufacturing systems.

A control system observes the state of environment at a specific interval and decides on the base of acquired state information, which actions have to be taken. Therefore, it is important that the hard real-time and safety critical system supports periodic execution of tasks with minimal jitters. That is known as *time-driven approach* (Eriksson et al. 1995). In this approach, the communication resources are split up slices that are allocated to the processes, which communicate periodically. In distributed systems, this is reflected on the interoperability of networked subsystems that have beforehand designed slices of time to send messages, that is the messages are time-synchronised.

However, all communication is hard to design beforehand. Distribution makes it more complicated because control systems may have subsystems that are not control systems and their characteristics are based on *asynchronous and synchronous messages*. The approach is called communication-based synchronisation (Törngren & Backman 1993; Gyllenswärd & Eriksson 1994). Thus, the important relationship between components is temporal. How could temporal behaviour be predicted, if this kind of description does not exist at the component level? In practice, engineers execute tests in the target system. Because timing requirements are heterogeneous, the temporal behaviour of the whole system is not necessary to be predicted, but only as concerns the hard real-time level. The problem is simpler, if the hard real-time level can be isolated from the other parts of the system and its predictable behaviour could be designed as communication policies.

*Exception handling* is another aspect of interoperability. If exception handling is not defined systematically, all subsystems handle exception according to their own principles. Although exceptions can be refined and their scope and context can be determined, it may lead to unexpected behaviour at the system level. Integration work often consists of the adaptation of exception policies that reliability can be guaranteed at the system level. Centralised controllers, for example an error logger and exception handler, are their practical solutions and Event-Condition-Action (ECA) rules with a rule processor is a solution to implement an event-controller (Dittrich et al. 1986; Kappel et al. 1994).

Some functions are common to all systems and they have remarkable effects on the systems' co-operation. The functions that deal with timing and the system's state, for example, are shared with subsystems and co-ordination is based on *shared knowledge* that may be allocated to one's responsibility. The master-slave control architecture is based on one's authority, on the contrary, agents are based on shared authority and autonomy. The philosophies behind the approaches are opposing and can not be used in the same system without adaptation. The trend is to transfer from master-slave systems to agent-based systems, in which a global clock is replaced with a reference clock and periodic time correction (Genesereth & Ketchpel 1994; Gergeleit & Streich 1994).

In large control systems, as manufacturing execution systems, shared data may bring out integration problems that result from weak designs, heterogeneous databases and their different access policies. Data consistence checking and fault-tolerance are responsibilities of databases, and they can be dealt with using different kinds of policies. These differences have to be faded by adaptation. Non-interoperability also appears in the formats of stored data that have to be filtered or converted and the data presentations that need to be scaled according to the viewers and roles of end-users. If interoperability of data management has not been paid attention to in the software development, redesign and re-implementation have to be done in the integration phase (Kappel & Vieweg 1994). All in all, interoperability depends on interfaces, their completeness, usability, performance, and reliability.

This work attempts to find out an architectural solution that enables to control semantic differences in communication and the management of exceptions, state and data.

### **2.3.3 Adaptability**

Incomplete and changing requirements are the main reason why adaptability is needed in the integration phase. The last one is the most usual reason in large control systems, in which changes are made in the installation and introduction phase. The reason for incomplete requirements is the fact that all exceptional situations are hard to imagine beforehand and systems are designed with the thought 'normal operation'. Therefore, new situations arise in the installation phase, and the features have to be added and changed. In component-based

software, changes might not have influences on other components or their influence should be able to estimate and manage. Changes may cause performance problems and reallocation. Therefore, software adaptability concerns location-independence, environment-independence, extendibility, and flexibility.

Because software reuse attempts to reduce redesign, re-implementation and re-testing, applications should have to be isolated from the network topology and the execution environment, i.e. an operating system, protocols and hardware. They are preconditions for the systems' extendibility that is needed in integration and upgrading. Extendibility embodies features, components, and subsystems. Thus, the integration needs to be supported by management and configuration interfaces at the architectural level (Ran & Xu 1996).

Flexibility is required in component interfaces when the attributes of an interface, i.e. parameters, operations or bindings, are changed. Parameterisation, type casting and also polymorphism and dynamic binding are solutions, if they are supported by the language. A filter, connected to the interface, performs a data conversion. Adaptation between components concerns interoperability and is called inter-component adaptation (Pryce & Crane 1996; Lycett & Paul 1998).

Legacy software needs other kinds of adaptation. Dissimilarity between subsystems is often in protocols and data formats and applications may use different architecture styles that need to be adapted. An adapter and wrapper adjust a legacy application to a new software architecture and its interconnection rules. Glue software, such as a wrapper, decreases performance, but it is a viable solution in proportion to the size of a component and re-development effort. This is called inter-framework adaptation (Mowbray 1997; Lycett & Paul 1998).

Adaptation can be done statically or dynamically. Traditionally, software configuration management is aimed at static configuration. In some cases, the stopping of a distributed control system brings out significant economic losses and therefore, dynamic adaptation is needed (Lim 1996).

This work focuses on the mechanisms and interfaces demanded of the component framework for reconfiguring product features, services and applications of distributed control systems.

## 2.4 Problems in upgrading systems

In this thesis, maintenance problems are considered from the viewpoint of the development of a component framework, i.e. how the maintenance point of view should be paid attention to in the development phase. The justification for our approach is as follows:

- The installation and instruction phase of a manufacturing system or a complex machine control system can take weeks depending on the size, complexity and quality of a system. If the system software supports reconfiguration and allows for making changes in software according to the beforehand fixed guidelines and rules, lower maintenance costs and better customers' satisfaction are achieved.
- Due to the lack of maintenance professionals, upgrades should be able to perform by less experienced staff. This provides that the systems are supported with appropriate maintenance tools.

According to our experiences, the evident parts of software that are changed during the life cycle of a distributed control system are the following:

- *User interface.* User interfaces need to be customised according to end-users' expertise and situation action of the work. Information, in different forms, has to be propagated to several places, for example to a worker, a foreman and management. The location and shape can also change, which can result from the changed workflow or physical structure of a control system, for example a cell or store is added.
- *Communication.* Network communication media, protocols, data conversions, routing and message definitions need to be changed while connecting a control system to an other system.
- *Environment interface.* Interfaces to the surrounding environment should be able to be changed easily because data acquisition and control devices could get broken or be modernised.

- *Physical components.* Although the hardware is attempted to be kept stable, an upgrade can cause performance problems and new hardware is required. Due to physical extensions and changed working manners, a controllable target may change. In most cases, this means a new system.

This work aims at a flexible solution for managing the predictable changes in user interfaces, communication network, and system’s environmental interfaces.

## 2.5 Summary

The problem analysis is summarised as essential requirements of the development of a component framework (Table 2). The requirements illustrate the overall tendency to manage the changes that get their origins from marketing needs, evolution of implementation techniques and systems themselves. Justifications of the requirements reflect the problems discovered in the application domain.

*Table 2. Key requirements of the component framework.*

<b><i>Requirement</i></b>	<b><i>Justification</i></b>	<b><i>Discussed</i></b>
Start from the business issues of a product-family.	New features spring up from marketing issues and affect the repayment-time of the component framework.	Section 2.2, Papers II, VI
Analyse functional and quality requirements as regards product diversity.	Diversity of requirements is needed to get a balanced product-family architecture.	Section 2.2.1, Papers IV, V, VII
Consider the consequences of the distribution degree as regards the other requirements of the products and their development.	Philosophy behind the distribution is essential in the software architecture design and allocation of the development work.	Section 2.2.1, Papers IV, V
Describe the strategies for extensions, replacing and scaling.	Strategies guide in design and implementation of components and keep the framework flexible.	Section 2.4.1, Papers IV–VII
Make feasibility studies of styles, policies and COTS.	Evaluation and risk analysis assist to select the appropriate development approach.	Section 2.2.2, Papers II, VI



<b><i>Requirement</i></b>	<b><i>Justification</i></b>	<b><i>Discussed</i></b>
Utilise architecture styles and design patterns.	The styles and design patterns guide in making design decisions and are tested reusable assets.	Section 2.2.3, Paper VII
Prioritise contradictory requirements and evaluate risks with different priorities.	The prioritised requirements assist to discover a balanced architecture.	Section 2.2.3, Paper IV
Select the communication policies for component interactions.	Classification of interfaces guides the implementation.	Section 2.3.1, Papers IV, V
Isolate user interfaces from the application logic.	User interfaces are changed frequently and have to be substitutable components.	Section 2.3.1, Papers V–VII
Define timing, communication, exception and data access policies.	Adjustment of policies is difficult to make afterwards.	Section 2.3.2, Papers III, IV, VII
Define adaptability for location, environment, extensions and component interfaces.	Adaptability increases the capabilities of a system to be evolving.	Section 2.3.3, Papers V–VII
Describe items and mechanisms for reconfiguration.	Reconfiguration is a software component of the framework.	Section 2.4, Papers VI–VII
Evaluate and select modelling methods and tools as regards the product-family architecture.	Graphical representations give support to transfer information between team-members and evaluate possible solutions.	Section 2.2.3, Papers I–III
Evaluate and adapt techniques and tools for the use of engineering teams.	Products are multi-technological, and heterogeneous and legacy technology is used concurrently with new ones.	Section 2.2, Papers I–III

Evaluation criteria of the component framework are derived from the above-mentioned requirements (Table 3). Because the component framework oughts to solve the technical problems concerning a set of distributed control systems, its ability to solve the problems of variability and distribution are the most important. The suitability of the component framework is evaluated concerning the

process issues that have been important in the case studies of the control systems domain.

*Table 3. Evaluation criteria for the component framework.*

<i>Criteria</i>	<i>Description</i>
Variability	The ability to define and manage product features and variants.  The ability to map variability to the product family architecture.
Configurability	Ability to select and change product features.
Interoperability	Interoperability of tiers, services and applications.  Ability to manage syntactic and semantic differences.
Portability	Independence of applications and services, hardware independence, installability.
Flexibility	Ability to change functionality, user interface, system's topology and hardware.
Extendibility	Ability to add applications and services.
Suitability	Operability and simplicity.  Ability to use commercial and legacy software.  Ability to use heterogeneous methods and tools.  Required expertise and the amount of developers.  The costs of the framework development.  Ability to share the framework among stakeholders.

### **3. A component framework of distributed control systems**

The component framework, described in this chapter, has been driven by the needs of industry that are presented as the requirements in the previous chapter. With the intention of maximising software reuse, the component framework focuses on adaptive design and attempts to meet the challenges that the changing requirements entail to the software development. In addition, business and organisations also evolve and bring about needs to change the properties of systems and their development and maintenance.

Evolution also reflects the component framework that is presented. The same kind of progress is evident, if the component-based software development is applied to the engineering practice. The history of how the ideas have progressed, therefore, is described briefly as motivation to the component framework. The tiers of the framework are described in more detail, and they tend to put the things described in Chapters 4 and the papers in a context that could be used outside of this thesis.

#### **3.1 An overview of the component framework**

The component framework was devised by synthesising and generalising the frameworks developed and applied in the case studies and described in Papers IV–VII. In the first phase, adaptive and component-based software design, presented in Papers I, III and IV, were applied to the machine control systems domain. The software architecture included three layers: system, node, and communication layers. A generic communication layer acted as an integration platform providing a transparent communication mechanism for the subsystems. In addition to the communication layer, subsystems also had a node layer with application-specific data and operations, and a system layer with system-level data and a configuration mechanism.

In the next phase, dynamic configuration was applied to architectural components of distributed control systems (Paper VII). An agent architecture was applied to logical subsystems that may be physical subsystems or twin processes in

the same computing environment. Transparent communication and on-line configuration support provided the integration services and generic interfaces for the subsystems.

These two approaches were combined in an application-specific software bus that was used in the automatic repayment systems (Paper V). In this phase, system-level data was integrated as a part of the integration framework, as well as the configuration support software. A configuration tool enabled to change functionality, communication, and data. Agents and the simplified interface technique were applied to the subsystems. The integration framework was based on client-server and agent architectures.

In the last experiment, the on-line configuration support of product features was developed and applied to flexible manufacturing systems (Paper VI). A commercial CORBA implementation was used as the integration framework, which was extended by an ECA executive and a commercial OODB as the components of the configuration management and a data propagation mechanism. Behavioural patterns of the system were configured through a rule database. Moreover, the software architecture style was shifted to the client-server and rule-based system architectures.




The overview of the component framework (Figure 7) describes the main elements of the component framework and its different views (cf. Figure 1). As defined earlier, a component framework has a dedicated architecture, some key-mechanisms and a set of policies how components use these mechanisms. These elements describe what a component framework is, and product features define for what a component framework exists.

The tiers of the component architectures emphasise the above-mentioned views in different ways. The first tier that is applied to logical subsystems focuses on the characteristics of the application domain. The term ‘subsystem’ is used to emphasise independence and interchangeability of the applications. The second-tier emphasises the technical means to provide an open platform for interoperable subsystems, and therefore, it is called an integration framework. A product family characterises the business view of the third tier. The development and utilisation of a component framework have three main viewpoints, described as CBSE, CSE, and SCM in Figure 3. Each viewpoint is dominant in different tiers

and has different aims and design rationale. The complementary viewpoints of the elements are presented in parentheses.

<i>Element</i> → <i>View/Tier</i> ↓	<i>Product Features</i>	<i>Architecture &amp; Mechanisms</i>	<i>Components &amp; Policies</i>
Domain/Subsystem	CBSE (SCM)	CBSE	CBSE (CSE, SCM)
Technology/Integration	CSE (SCM)	CSE (CBSE)	CSE (CBSE, SCM)
Business/Product-family	SCM	SCM (CBSE)	SCM (CBSE)

<b>Viewpoints</b>			
-------------------	---	---	--

*Figure 7. An overview of the component framework.*

Component-based software engineering (CBSE) principles are essential in the subsystem tier, whereas concurrent software engineering (CSE) focuses on the integration tier. Software configuration management (SCM) gets its origin from the features of a product family, but requires components and policies of the subsystem and integration tiers to use mechanisms in the product-family tier.

The software architecture of a tier balances the contradictory requirements, and it should have to provide solutions for the requirements, set by the related viewpoints. Therefore, each tier may have to be described and developed in different ways.

The next chapters give an overview of the elements of each tier and the relationships between tiers and the viewpoints of software engineering.

## 3.2 The subsystem tier

CBSE is the dominant viewpoint that deals with the subsystem tier and it focuses on:

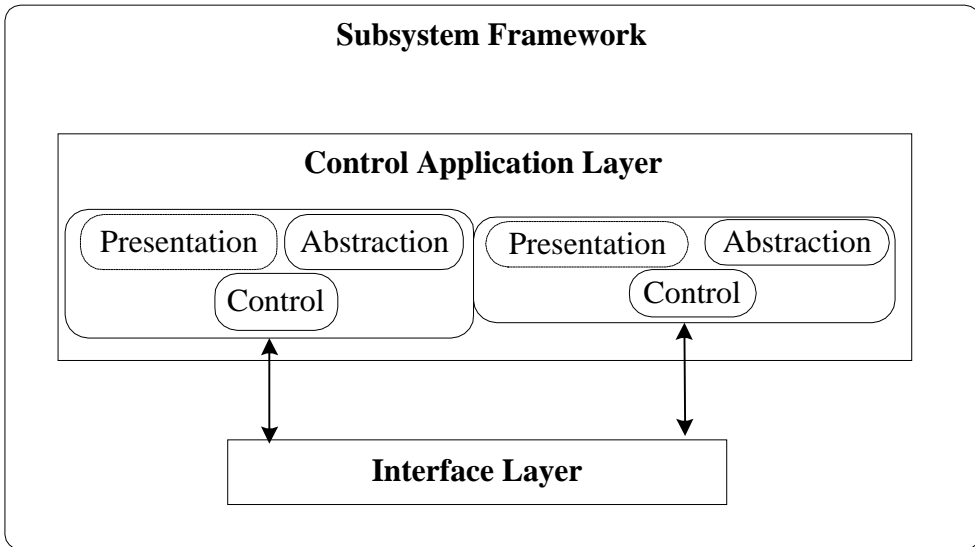
- Speeding up the application development.
- High-quality products by reusing application knowledge.
- Easy connected subsystems for networked control systems.
- Adaptable solutions for different kinds of control tasks.

These issues are the main goals that a component framework, used for developing subsystems, should have to fulfil. Speeding-up the application development provides that the knowledge of application area and earlier products is high, in a reusable form and that there is a means of its utilisation. High quality is the aim that is achieved by using earlier tested and reliable artefacts, such as software architectures and components. Compatibility highlights the ability to develop application concurrently (CSE) and the ability to use applications easy in larger contexts, for which they are not designed from the first. Adaptability focuses on forthcoming needs by providing the techniques to change the purpose and properties of applications by software configuration management (SCM). The subsystem tier attempts to meet the above-mentioned goals by providing such architecture, mechanisms, and components that fulfil these product-oriented and organisational aims.

The subsystem has the two main layers that are the interface layer and the control application layer (Figure 8). The control application layer consists of one or more application-specific software agents that may also be layered. An agent consists of two or three components. An abstraction component defines data structures of an agent. A control component controls the behaviour of an agent and its communication. A presentation component, if needed, defines formats and operations to describe state and data of an application agent. The architectural pattern is known as the PAC (Presentation, Abstraction, and Control) pattern (Buschman et al. 1996). In our case, the components of an agent are loosely

coupled and the presentation component may be allocated to a different node than the abstraction component and control component (Paper VII).

The interface layer assures adaptability of subsystems. It provides a standard interface for communication and configuration and connects subsystems to the integration tier. All inter-agent communication is directed through it to the integration platform (Paper V). In order to be interchangeable, application agents also have a configuration interface. If on-line configuration is necessary, a subsystem may be configured in two ways. Parameterisation is used to change the internal variables of a subsystem, e.g. the message types or controlling interval. Reconfiguration is made under the control of the integration platform. If an application agent is updated or a new one added, a sophisticated negotiation-based interface is required (Paper VII). In that case, the agent decides how to react to an in-coming configuration request. If the request is accepted, the interface prepares the agent into the appropriate state and monitors whether configuration is performed correctly.



*Figure 8. The component architecture of subsystems.*

The purpose of the interface layer is to provide an environment-independent solution for portable applications. A physical implementation of the interface

layer has appropriate components from the interface layer of the integration framework, for example the interfaces of an operating system, a communication medium, and environment devices. These components are logical parts of the integration framework, but they may be allocated in different ways to the physical subsystems. Therefore, only application-dependent components of the interface layer are included in the subsystem tier.

The subsystem tier is a domain specific framework and its architecture style is selected according to the characteristics of the control domain. An agent architecture is preferred to client-server architecture in hard and soft real-time systems, and in manufacturing systems that need on-line configuration support. The layers provide adaptability and portability as regards environment-independence and location-independence.

A more comprehensive description is given in Table 4, which represents the solutions and their intentions according to each element of the component framework.



Table 4. Elements of the subsystem framework.

<i>Element</i>	<i>Solution</i>	<i>Intention</i>
<i>Product feature</i>	<ul style="list-style-type: none"> <li>• Data-based configuration by parameterisation</li> <li>• On-line configuration by a negotiation-based interface</li> </ul>	<ul style="list-style-type: none"> <li>• Adaptability for variability inside components</li> <li>• Adaptability for updating and adding application agents</li> </ul>
<i>Architecture</i>	<ul style="list-style-type: none"> <li>• Interface layer for portability and adaptability</li> <li>• Agents embody the roles of applications</li> <li>• Layered agents for intelligent, autonomous agents</li> </ul>	<ul style="list-style-type: none"> <li>• A uniform interface simplifies application development</li> <li>• Systematic application development due to layers, agents and components</li> <li>• Increased reuse by isolating domain and technology-dependent parts by the layers</li> </ul>
<i>Mechanism</i>	<ul style="list-style-type: none"> <li>• A mediator for communication and configuration</li> <li>• A bridge for isolating applications from their environment</li> </ul>	<ul style="list-style-type: none"> <li>• Centralised communication and a centralised changing point increase system's flexibility</li> <li>• Portable applications</li> </ul>
<i>Component</i>	<ul style="list-style-type: none"> <li>• Classification of components and their responsibilities according to the PAC pattern and the roles of agents in the system</li> <li>• Location-independent agents and presentation components</li> </ul>	<ul style="list-style-type: none"> <li>• The responsibility-driven approach increases modularity</li> <li>• Decomposition supports variability inside agents and layers</li> <li>• Separated allocation of applications simplifies the development and increases systems' flexibility</li> </ul>
<i>Policy</i>	<ul style="list-style-type: none"> <li>• Configuration and communication policy of applications</li> <li>• Message-based communication for loose coupling</li> <li>• Local error handling policies</li> </ul>	<ul style="list-style-type: none"> <li>• Speeding up application development, increasing quality, and compatibility</li> <li>• Easily connected and configured subsystems</li> <li>• Enables to balance network load</li> </ul>

### 3.3 The integration tier

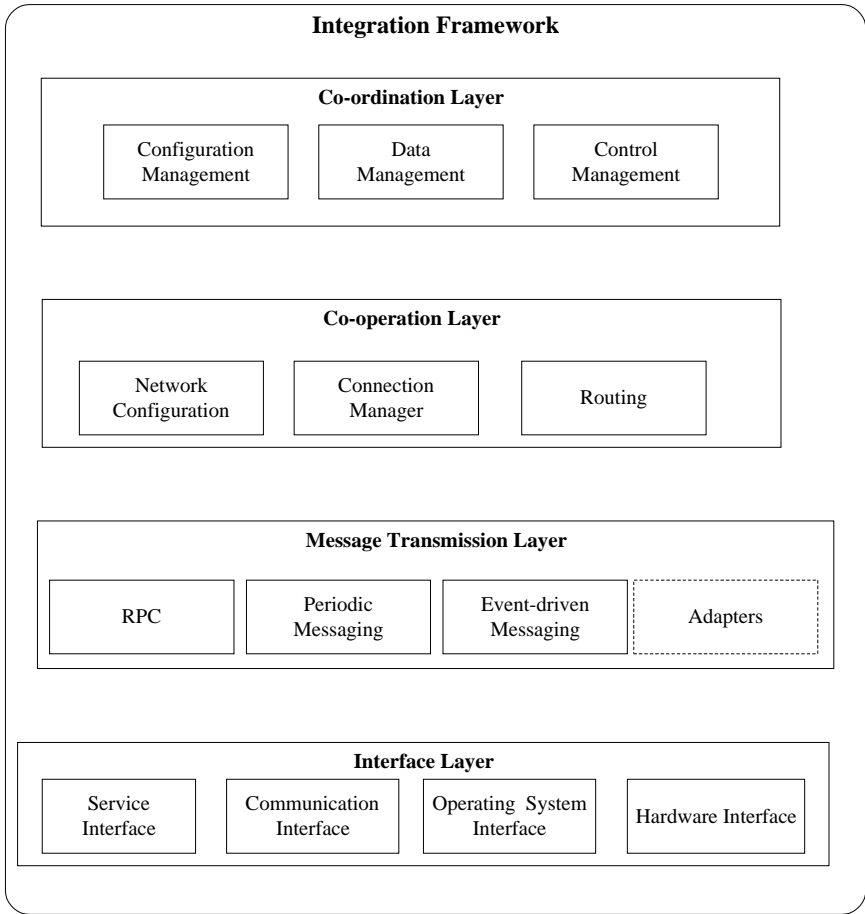
Concurrent software engineering (CSE) is the dominant viewpoint that has an influence on the development of an integration tier. This means that the tier has to support the concurrent development of subsystems and incremental integration of a distributed system. The tier can also be defined as the infrastructure, a uniform development and operation-support environment that provides a set of generic services to the application components and implements a set of common functions defined by the architecture (Rossak et al. 1997). The integration tier provides a generic platform for the subsystems developed according to the principles of the subsystem tier. Although CSE is dominant in developing and using the tier, CBSE affects its architecture and components, as well as that SCM attempts to manage the changes required due to evolution. The intention of the integration tier is to

- Provide a stable, application-independent distribution-platform.
- Support incremental software development.
- Be capable of adapting applications for interoperability.
- Be adaptable to different implementation technologies.
- Provide adequate services for applications.
- Allow variability of the used services.

The integration tier acts as a mediator that decouples applications and provides a transparent distribution platform by the four layers: the interface layer, message transmission layer, co-operation layer, and co-ordination layer (Figure 9). The first three layers represent technology aspects and may have several configurations due to the variability of implementation technology and required services. The co-ordination layer is an application specific layer and provides the services required for the data and control management at the system level. The co-operation layer has mechanisms for binding components and routing messages and uses application-specific data, but its functionality is application-independent.

The interface layer provides flexible connections to communication media, hardware, operating systems, and applications (Papers IV and VII). The name of the layer is the same as in the subsystem tier, but its purpose is also to provide a service interface for applications and the interfaces to COTS and hardware for the upper layers of the integration tier. It hides the complexity and heterogeneity of the implementation technology so that technological changes do not affect the means in which the services are used. On the contrary, the interface layer of the subsystem tier provides a standard way to connect an application to the integration platform. A hot-spot and a bridge pattern (Schmid 1996b; Gamma et al. 1994; Buschmann et al. 1996) provide mechanisms for achieving adaptability, also required if legacy systems and COTS components are wrapped for the integration platform (Papers VI and VII).

The services of the message transmission layer may be implemented in three ways. Firstly, the layer may be implemented utilising the services of operating systems (Papers V and VII). Secondly, a commercial component as an ORB and its RPC mechanism for message transmission may be used, if it is appropriate for the applications (Paper VI). The third way is to develop an own communication layer (Paper IV), which is the most appropriate solution for hard real time systems. In the latter case, both the periodic and event-driven message passing services are required. Periodic messaging is a time-driven communication mechanism for real-time purposes. Event-driven messaging provides loosely coupled applications without hard response times. The used technique depends on the required performance and heterogeneity of operating systems, communication media, and protocols. Adapters are additional components of the message transmission layer in heterogeneous environments (Paper VI).



*Figure 9. The component architecture of the integration platform.*

The co-operation layer includes a binding mechanism, a connection manager that manages the binding information defined by the configuration service (Paper VII). The layer also includes knowledge of how the messages, transferred by the message transmission layer, have to be routed to the subsystems (Papers V and VII). Application agents send information without knowing its receivers and therefore, the bindings define the producers and consumers of information. Physical bindings are determined in the integration phase while allocating applications. The co-operation layer provides the mechanisms required to interconnect subsystems, that is the mechanisms of the network configuration software.

Although the integration tier is mostly application-independent, the data management service includes application-specific, system-level data, required for the co-ordination and reconfiguration of subsystems (Papers IV and V). System-level data is produced during the system development (Papers IV, V and VI). Filters may be used for adapting data to a uniform format required for interoperability. The configuration mechanisms that process the configuration rules defined in the product-family tier are the link between the integration tier and the product-family tier.

The architectural style of the integration tier is a combination of the client-server and layered architecture. However, an intelligent agent, which provides versatile routing services, for instance, pier-to-pier, multicasting and broadcasting, with several routing algorithms, may be applied in the co-operation layer of complex networked systems (Papers V and VII).

The integration tier is the glue that hides the diversity of used technology and connects together old and new systems. Therefore, the integration tier has to be ported to different hardware and software platforms and integration support for legacy systems is also required. Adapters and filters as common services provide a uniform solution, but in most cases, the use of wrappers may be more practical for adapting legacy software for the integration tier, for instance, wrapping existing applications by the PAC interface-wrapper.

The summary of the integration framework depicts the used solutions of each element and their meanings (Table 5).

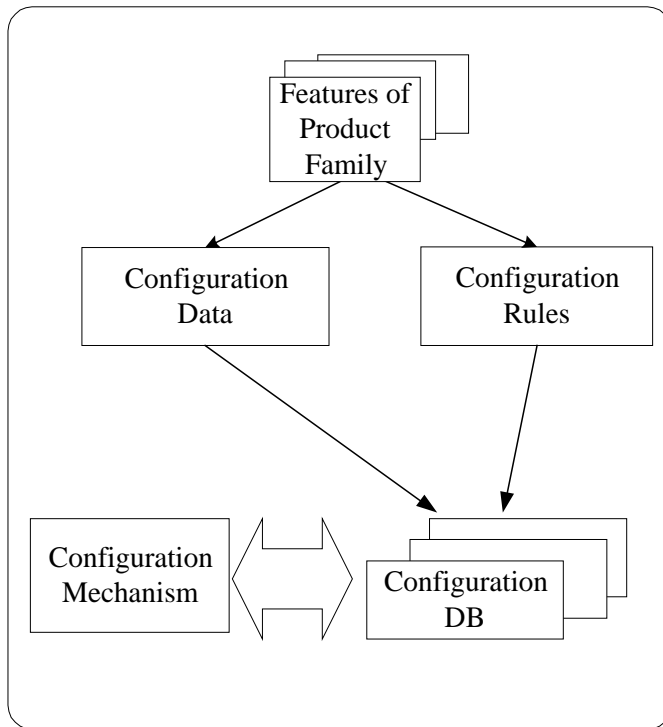
Table 5. Elements of the integration framework.

<i>Element</i>	<i>Solution</i>	<i>Intention</i>
<i>Product feature</i>	<ul style="list-style-type: none"> <li>• Optional components of the services</li> </ul>	<ul style="list-style-type: none"> <li>• Selectable components for required services</li> </ul>
<i>Architecture</i>	<ul style="list-style-type: none"> <li>• System-dependent co-ordination layer</li> <li>• Co-operation layer for interoperability</li> <li>• MTL for message transmission</li> <li>• Interface layer for portability and adaptability</li> <li>• Hot spot for interchangeable protocols</li> </ul>	<ul style="list-style-type: none"> <li>• Separation of concerns: application, technology, responsibility, change-frequency</li> <li>• Layered architecture for management and evolution</li> <li>• Client-server architecture for application purposes</li> <li>• Patterns for achieving flexibility</li> </ul>
<i>Mechanism</i>	<ul style="list-style-type: none"> <li>• Configuration manager</li> <li>• Connection manager</li> <li>• Separated components for periodic and event-driven messaging</li> <li>• Dispatchers for periodic message-passing, routing and task ordering</li> <li>• Adapter for integrating heterogeneous systems</li> <li>• Bridge for environment-independence</li> <li>• Filter for data translation</li> <li>• Wrapper for unifying interfaces</li> <li>• ORB as MTL</li> </ul>	<ul style="list-style-type: none"> <li>• Support for incremental software development</li> <li>• Location-independent applications</li> <li>• Isolated real-time layer in messaging</li> <li>• Reusing mechanisms</li> <li>• Adaptation for interoperability by using design patterns</li> <li>• COTS as a distribution platform</li> </ul>
<i>Component</i>	<ul style="list-style-type: none"> <li>• Service interface for subsystems</li> <li>• Standard interfaces for OS, hardware and communication media</li> <li>• Customising layers by the selection of components</li> </ul>	<ul style="list-style-type: none"> <li>• Stability of the platform</li> <li>• Portability of the framework</li> <li>• Restricted flexibility and variability inside layers</li> </ul>
<i>Policy</i>	<ul style="list-style-type: none"> <li>• Communication policies: periodic, event-driven and RPC</li> <li>• Centralised error-handling services.</li> <li>• Several allocation policies for the services</li> </ul>	<ul style="list-style-type: none"> <li>• Loosely coupled subsystems</li> <li>• Co-ordination point for system-level error handling</li> <li>• Distribution degree is freely selectable</li> </ul>

### **3.4 The product family tier**

The product family tier represents the characteristics of the business domain. SCM focuses on product features and their management, representing the viewpoints of marketing and maintenance. The marketing staff is interested in reducing time-to-market and cost, but maintenance staff are interested in easy and correct configuration. Therefore, the principles, used in CBSE, affect the product features and their management.

Product features define the semantics of applications and therefore, they are also called semantic components (Bergmans 1998; Digre 1998). A product family has varying properties that are described in the features model of a product family. Therefore, the features model is the essential knowledge for producing a correct configuration of a system product. The product family tier produces and organises the product-knowledge into a form that can be used in upgrading a system and adding features to an existing system (Figure 10). Configuration data identifies and describes the features that can be configured only in the manners that are described as the configuration rules. Data and rules are stored in a configuration database, which is a component of the product family tier. Configuration mechanisms with user interfaces are the other part of the support software. The mechanisms are the components that are implemented as a part of the integration tier.



*Figure 10. Integrated support of a product family.*

The tier consists of the definitions for relevant product features, rules to use them and configuration tools to carry out legal combinations (Papers II and V) (Table 6). The features of a product family are defined during the development of a product-family architecture, but the features of a system are defined using the subsystem and integration frameworks. Therefore, the product features are fragmented and mostly mapped to the components developed by the subsystem and integration frameworks. Nevertheless, the components of the third tier are needed to define how the features are implemented and how they can be used. The implementation of the product-family tier varies due to the variability of the product family and the reconfiguration support that is needed for the integration and maintenance.

The semantic information is required in the co-ordination and co-operation layer of the integration tier and in the way in which the applications co-operate



through the integration platform. Although the product-knowledge is all over the system, they are controlled through a point, the configuration interface and mechanisms, for example an ECA executive that processes Event-Condition-Action (ECA) rules. Reconfiguration tools are intended to be used by the system administrator because overall understanding of the product-family is required.

The summary of the product family framework depicts the used solutions of each element and their meanings (Table 6).

*Table 6. Elements of the product family framework.*

<b><i>Element</i></b>	<b><i>Solution</i></b>	<b><i>Intention</i></b>
<i>Product feature</i>	<ul style="list-style-type: none"> <li>• Features model of a product family</li> </ul>	<ul style="list-style-type: none"> <li>• Variants and permitted combinations</li> </ul>
<i>Architecture</i>	<ul style="list-style-type: none"> <li>• Semantic components described as rules, data and functions</li> <li>• Components for managing semantic components</li> </ul>	<ul style="list-style-type: none"> <li>• Centralised support software integrated as a part of a system product</li> </ul>
<i>Mechanism</i>	<ul style="list-style-type: none"> <li>• Database for configuration data and rules</li> <li>• ECA executive</li> <li>• Configuration files</li> <li>• Configuration user interface for on-line modifications through the configuration database</li> </ul>	<ul style="list-style-type: none"> <li>• Product knowledge stored in reusable format</li> <li>• Activation mechanism for behavioural features</li> <li>• Activation mechanism for feature variables</li> <li>• Integration and maintenance support</li> </ul>
<i>Component</i>	<ul style="list-style-type: none"> <li>• Configuration management as default components of the integration platform</li> </ul>	<ul style="list-style-type: none"> <li>• Simplified on-line configuration support for incremental development, integration, and maintenance</li> </ul>
<i>Policy</i>	<ul style="list-style-type: none"> <li>• Restricted variations</li> <li>• Integrated product data management</li> </ul>	<ul style="list-style-type: none"> <li>• No conflicting features</li> <li>• User-friendly change management</li> </ul>

## 3.5 Summary

The component framework of distributed control systems is depicted as three tiers that are used for different purposes. The subsystem tier focuses on the characteristics of the control systems domain and provides a conceptual architecture for agents that are autonomous as regards control, location, and evolution. Agents have standard interface, communication and error handling policies, but in other concerns agents are flexible and illustrate the dynamics of the control systems domain. Layers and responsibility-driven components provide inherent flexibility for applications. The framework supports the concurrent application development and its interface layer connects subsystems to the integration tier.

The integration tier focuses on the development of distributed systems. The tier provides a common infrastructure for logical subsystems produced according to the subsystem framework. The tier hides technological implementation from the services provided for location-independent applications. The integration tier supports integration-engineers that develop distributed systems from building blocks, application agents and common system-level services and provides configuration mechanisms for the product-family tier.

The product family tier provides a systematic means of integrating marketing and maintenance viewpoints to the system development. The use of the tier assumes that the first two tiers are available and that the product features and their use could be defined. The tier represents the highest level of reuse and the knowledge is product and market dependent, not technology or domain-dependent.

Although the tiers are described independently, their implementation is closely interlocked with each other. Due to selected distribution degree and technology, the tiers may have different implementations. Performance and memory size can enforce an allocation model, where physical controllers have only the control components of application agents and the presentation and abstraction components are centralised into a node with appropriate memory and performance properties. Distribution of the co-operation and co-ordination layers that requires more sophisticated decision-making and communication than the centralised one, can not be avoided in large distributed systems.

## 4. Development of the component framework

In this chapter, we describe how the component framework, depicted in the earlier chapter, may be developed. The development of a component framework is presented as the development of the reuse assets, i.e., the product features, the product-family architecture and software components. Descriptions and techniques, used in each phase, attempt to illustrate the useful approaches experienced in practice.

Usually, distributed control systems are constructed in association with several industrial partners. A developer of a distributed control system may have one or both of two roles: to develop subsystems and to develop distributed systems by integrating third-party components and subsystems, produced by subcontractors. The integration platform is important for system developers and the subsystem tier for the developers of applications and subsystems. Subsystems may also form a product family, and therefore, the importance of a product family does not depend on the role that the partner plays in the development process of a distributed system. In addition, the product family is the prerequisite for successful component framework development, because it defines the requirements and properties of the software architecture and components, the essential elements of the component framework.

### 4.1 Development of reusable assets

The development of a product family focuses on the development of

- a basic product that can easily be customised, or
- a core-product which is used as a part of the products.

On the base of the focus, the development of reusable assets is divided into domain-oriented re-engineering and domain engineering (Figure 11). The main difference between these approaches is that domain-oriented re-engineering is the interest of the software developers, on the contrary to the domain engineer-

ing that is a planned activity of an organisation. However, re-engineering is also a partial activity of domain engineering.

Re-engineering is mainly a bottom-up technique and always necessary when no reuse asset exists yet. In this case, reusable assets are mined from existing systems and domain experts' knowledge. The re-engineering process produces a domain model including domain concepts, domain architecture, and reusable components. Re-engineering integrates domain analysis, the process of identifying, collecting, organising, and representing relevant information in a domain, with analysis and design methods such as RTSA or OMT (Paper I). Step-wise architectural design recovery is also proposed for identifying reusable software components from the code of existing systems and used COTS (Bratthall & Runeson 1998). Domain-oriented re-engineering may be seen as a first step toward more comprehensive domain engineering.

Domain engineering is mainly a top-down technique to develop reuse assets. When there is already experience of the development of reusable components, the development and management of reuse assets can be planned and done in a systematic way. Business plans and marketing modules are the initial information of domain engineering, as well as all documentation of existing products and experiences from the product development. Marketing modules, if available, represent the features of existing systems, the commonality and variability of a product family from the customers' viewpoint. The business plan gives a vision of what kinds of features ought to be demanded in the near future and indicates the anticipated changes of the product features.

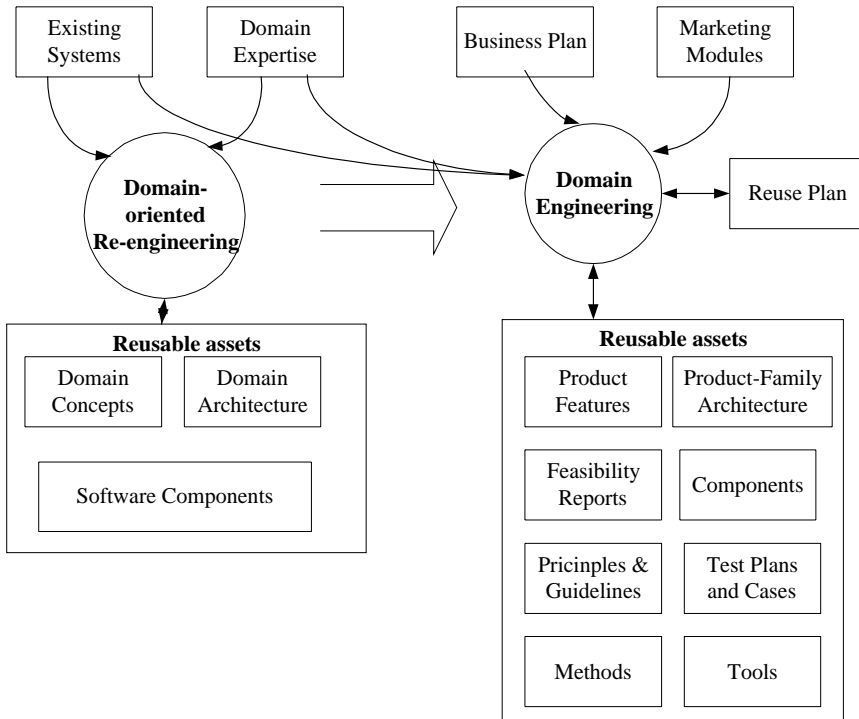


Figure 11. Development of reusable assets by domain-oriented re-engineering and domain engineering.

Domain engineering that consists of domain analysis, architecture and component design and implementation produces the following reuse assets:

- a features model of a product family,
- a product-family architecture,
- feasibility reports of COTS,
- software components of a product family,
- principles, guidelines, methods and tools for domain engineering, and
- test plans and cases for component and integration testing.

The rest of this chapter concentrates on the first four reuse assets. Methods, principles and testing techniques are given as examples, as used in the case studies.

The following chapters also emphasise the stakeholders' roles in the development of a component framework for a product family (Table 7).

*Table 7. Stakeholders of the component framework.*

<i>Stakeholder</i>	<i>Main interests</i>
Marketing staff, customer, end-user	Product features
Domain analyst	Product features, product-family architecture
Application developer	Components and connections, common services
System integrator	Common services, integration support
Maintenance staff, customer, end-user	Reconfiguration support

## **4.2 Features of a product family**

The features of a product family are conceptual, semantic models that describe the intended purpose of the products. Changes in the market segments and end-users' needs are reflected immediately in the product features.

The definition of product features includes two phases, specifying and validating features. Product features should be defined and reviewed in co-operation with the product managers and marketing staff. However, if there is no commitment by these people, the work falls on the domain analyst. Unfortunately, this seems to be the situation in most cases in practice.

### **4.2.1 Defining product features**

We developed the product feature modelling method (PFM) that enhanced FODA by defining typed feature-blocks, default features and mutual exclusion of child features in the parent feature (Paper II). However, the need of a special

modelling method is not essential in small systems. For instance, the OMT method and tables were sufficient for the product family of the automatic repayment systems (Paper V). In this case, the product family had three main product variants and the varying features could be allocated to the subsystems directly. Large systems, such as manufacturing systems, require a feature modelling method to define and manage:

- features of physical environments,
- features of user interfaces,
- operational features (functionality, behaviour, timing), and
- relationships between different features.

Most changes concern the features of the physical environment and user interfaces, and therefore, they are defined separately. The following models are required:

- a features model for describing the structure of product features,
- use-cases and scenarios for describing the intended use of features, and
- a time-thread model for timing requirements.

In the first phase, the features model describes the structure of the product family as logical subsystems, which are actors in use-cases and scenarios. The three models are defined concurrently, and they require several iterations.

The features model is a hierarchical tree that may include combined features, single features, or lists of the alternative features (Figure 12). The features model is semiformal and changed by adding, removing features and structuring them in a new way.

A feature is typed. The type may be mandatory, optional, or conditional. Mandatory features are required in all product variants, as in the maintenance UI in the example. Optional features are selected into known product variants, e.g. the





A scenario describes the intended use of features and the connections between combined features. A scenario is an semantic component of a system that is to be mapped to a component or several components. Use-cases may be used to describe interactions between actors at a generic level, but scenarios are more useful for sketching and illustrating the intended behaviour of the systems.

Scenarios are hierarchical. A mandatory behavioural feature combines functional and informational features and forms the main scenario. An optional feature is described as a sub-scenario (Figure 13). Thus, the variation points between the mandatory and optional levels can also be taken into consideration in the scenarios. Scenarios can be combined and described as a use-case with actors, which form an abstract domain architecture. More fine-grained variability, for instance, parameterisation is described at the architecture and component level.

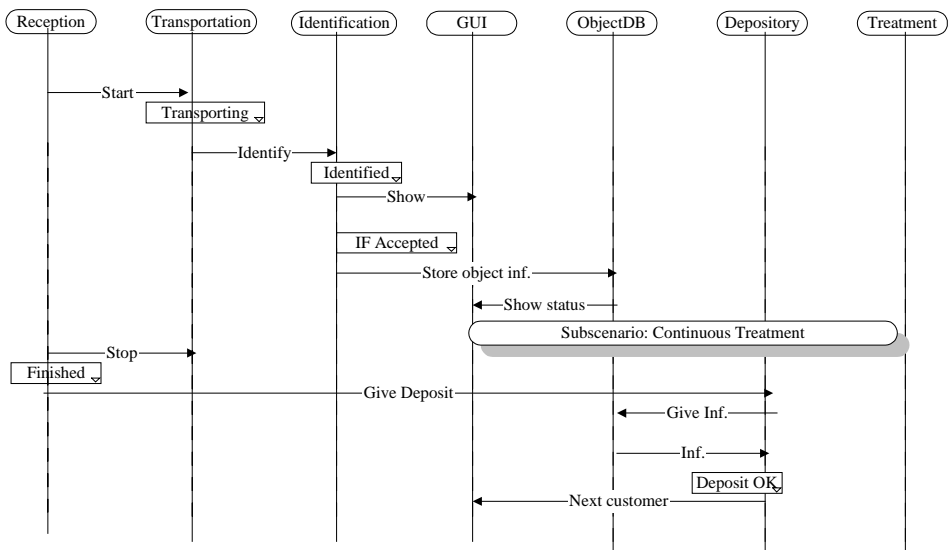


Figure 13. Mandatory and optional features in a scenario.

Time-thread models are scenarios that represent slots for operations, potential components and the timing requirements for a chain of operations (Figure 14). The example describes two time-threads with different timing requirements, and potential components and their operations that are marked as codes, for example the operation F1 in the Velocity Sensor component. Codes inside components are functional features defined in the features tree.

Time-threads are used to describe the behaviour of time-critical systems, for example machine control systems. Although other quality requirements, e.g., safety and robustness, are also important, they were not modelled but defined in a textual form, if they were included in the scope of the case study.

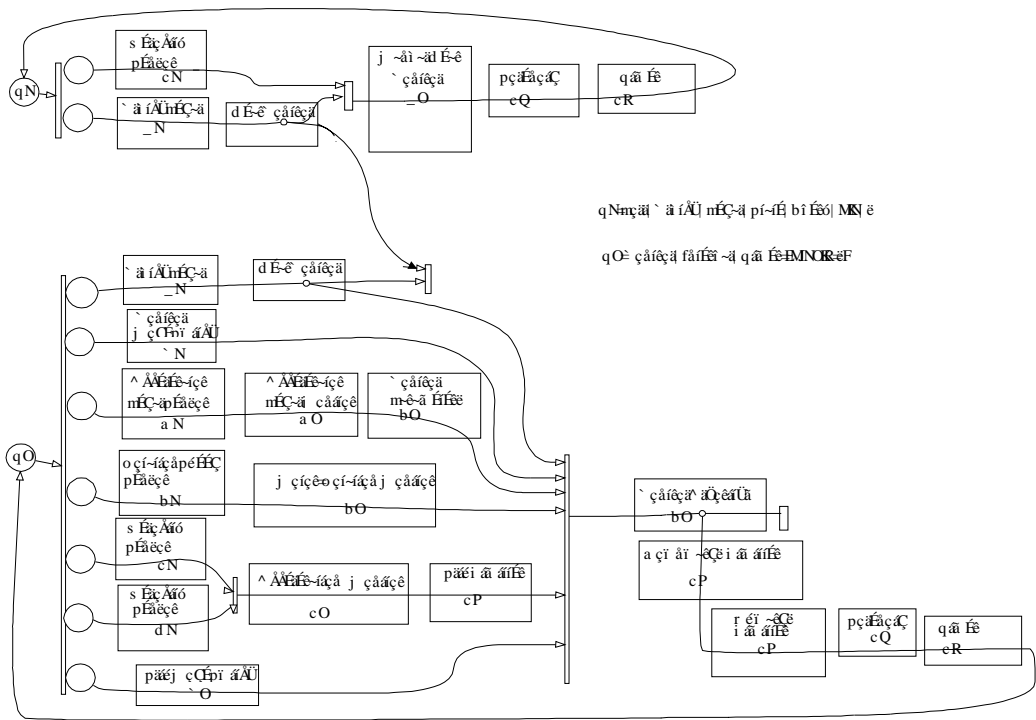


Figure 14. A time-thread model.

Iteration of the feature, scenario and time-thread models gives the first proposal of mapping features to components. Comprehensive feature-component-mapping is done in the architecture and component design phase. Time-threads can be utilised as guidelines in defining communication and allocation.

Table 8 summarises the representations of product features and the intended use of each model.

*Table 8. Models of product features.*

<i>Model</i>	<i>Representation</i>	<i>Purpose</i>
<i>Feature</i>	Structured tree	Classifying and managing features
	Required relationship	Pre- and post-conditions of a feature
	Leaf in the tree	Functional or informational property
<i>Scenario &amp; use-case</i>	Single scenario	Semantics of the functional features for actors inside and outside the systems
	Hierarchical scenario	Variability in behaviour, separation of mandatory and optional features
	A use-case	A combined set of features describing behaviour at the architecture level
<i>Time-thread</i>	Slots	Operations or components in a timing chain, pre- and post-conditions in timing
	Join and fork	Variability in a timing chain

#### **4.2.2 Validating features**

We used the QFD method to check that the features were appropriate before starting their development. The same method was applied to evaluate potential COTS components.

In the first phase, the stakeholders' needs are defined and classified into five categories: usability, functionality, quality, conformance, and adaptability (Figure 15). Usability includes the needs concerning the use of a system, for example ease to install and maintain. The functional and quality needs describe desired operational properties and their quality, e.g., safety, performance, reliability and robustness. Conformance means required standards the products have to meet and the constraints of software execution environments. Adaptability focuses on changes and the importance of evolutionary aspects of the products. Each category may have constraints.

The strength of each need is defined concerning the stakeholders, including sales and marketing, end-users, developers, integrators, maintenance, and training staff. Because products are targeted at different marketing segments, the differences between segments are also analysed. The total value of a need illustrates its significance. Each feature also has a weight-factor that defines its importance in relation to the quality of the whole product.

In the next phase, the features that relate to a need are analysed and classified. A feature can have a negative influence on a need and a positive influence on the others as the features F1 and F2 in Figure 15. A feature that has a positive influence, e.g., the feature F3 in Figure 15, on the important needs such as N1 and N3, should be a mandatory feature, if it is not only the interest of a special group. Analysis of the features is made for each need respectively.

Each decision about a feature that has contradictory influences is specific and generic rules can hardly be given. However, the following principles may be useful:

- Mandatory features fulfil needs that are important for most stakeholders.
- Mandatory features should not have negative impact. This leads into a compromise because a feature can have a negative influence on a quality issue that is not important in some market segment but extremely important in the other segments.
- Optional features satisfy needs that are important only for some group of end-users or marketing segments.

- Conditional features meet needs that are important for a special group of stakeholders, for instance maintenance staff.
- Features that meet less significant needs are matched to optional features.
- Features that have contradictory influences are optional or conditional features.

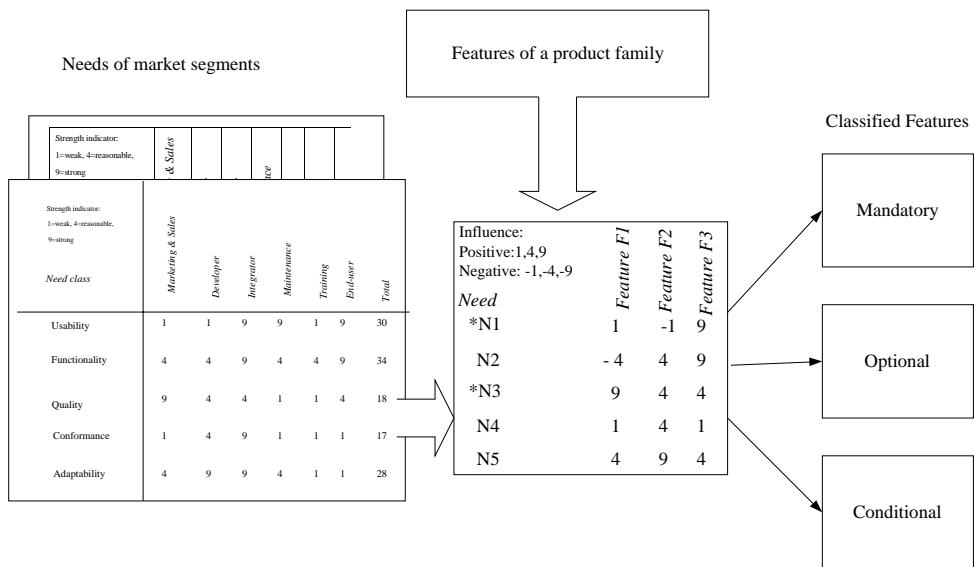


Figure 15. Analysing features by QFD.

In our case, the only knowledge was the experience of the developers and product managers who estimated the impacts of the product features on the customers' needs. Estimation was made by teamwork, combining the existing experience without analysing markets and customers' satisfaction.

The other approach to validate product features is clustering them into two groups: the features of the core-product and the features of variants (Soininen 1997). Grouping assists to manage a multitude of features. The quality function of a feature can be classified, according to the Kannon model (Day 1993), into three classes: features of basic, expected and surprising quality.

After clustering, the product family is analysed in two phases (Soininen 1997). First, each product variant is analysed as regards the quality and differentiation in the market segment. The result indicates the goodness of the specification of a variant. Secondly, the product family is analysed as regards the competitive products, which produces information about the superiority of a product family. The approach has been applied to embedded products and validation was made by measuring customers' satisfaction, achieved by interviews and monitoring. The approach may be applied to software as well. In this case, the features of the core-product are mandatory features of basic and expected quality. Variants have optional and conditional features of expected and surprising quality that would be analysed as regard each market segment. Lastly, all features would be analysed in respect of the competitive products.

In addition to the QFD method, simulation and prototyping with a target environment are potential techniques for validation of functional and user-dependent properties. Simulation models with skeletal components are useful while testing user-interfaces and architectural partitioning. Validation of a new control algorithm may require an expensive test-bench. However, CACE and CASE modelling environments such as LabVIEW, Prosa/om and ObjecTime, can be integrated and used in the development of software architecture and components (Papers III and IV). Table 9 defines the tasks of modelling, prototyping and developing reusable components, depicted as an integrated development environment in Figure 16. The first three tasks concern feature modelling.

Table 9. Validating features by prototyping.

<b>Task</b>	<b>Result</b>
1. Define the features of a product family using a CASE tool.	<ul style="list-style-type: none"> <li>• Behavioural and functional properties: event scenarios, time-threads, a features model.</li> </ul>
2. Design control algorithms using the CACE tool.	<ul style="list-style-type: none"> <li>• Tested functional primitive components for applications.</li> </ul>
3. Connect and test the CACE models with the target environment using data acquisition boards.	<ul style="list-style-type: none"> <li>• Performance results of components in one environment.</li> </ul>
4. Convert graphical CACE models to object classes and communication models using the CASE tool.	<ul style="list-style-type: none"> <li>• Architectural components with functional properties.</li> </ul>
5. Refine the software architecture and components by using the CASE simulation tool, e.g. Objec-Time.	<ul style="list-style-type: none"> <li>• Refined and validated software architecture of a product-family.</li> <li>• Tested architectural components with functional, behavioural and timing properties.</li> <li>• Alternative distribution models.</li> </ul>
6. Develop the control software using the reusable software components and test in the target environment.	<ul style="list-style-type: none"> <li>• Reusable assets.</li> <li>• Tested real-time properties.</li> </ul>

The refinement of software architecture, in the phase 5, can include contradictory requirements concerning, for example, flexibility and performance that have to be balanced. Therefore, the product features also have to be validated in clusters, i.e., the mapping of features to the architectural components.

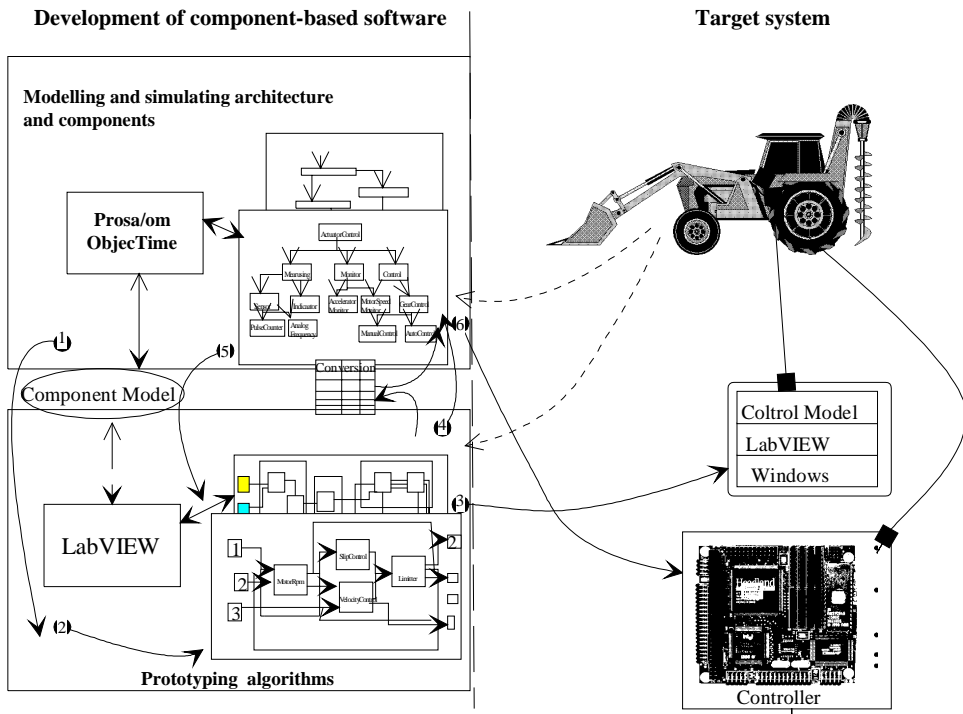


Figure 16. CACE and CASE tools as an integrated development environment.

### 4.3 A product-family architecture

Software architecture refers to the architecture of a specific system, to an architectural instance. An architectural style, however, defines constraints on the form and structure of a family of architectural instances (Garlan & Perry 1995). Styles range from abstract architectural patterns such as the client-server and layered architecture, to concrete architecture such as the ISO OSI communication model. Styles are high-level reusable assets that get their realisation through software architectures, components, and component frameworks. Therefore, the architectural style of a component framework has to be frozen before starting the development of the component framework. Product features models and scenarios, produced in domain analysis, guide in decision-making. A style defines the vocabulary of components, connectors, and structural and semantic constraints. Architectural styles and patterns are generic contrary to a product-family archi-



ture that is a specific solution by combining a set of styles and patterns. Therefore, modelling methods and tools used in product-family architecture design have to match with the used styles.

### **4.3.1 A layered architecture**

A layered architecture of the component framework is intended to support maintenance and evolution of the systems. Layers separate application-dependent parts from technology-specific software, for example the coordination layer from the message-transferring layer. Layers also separate frequently changed components from more stable components, for example the message transfer layer from the network interface layer.

A layer has a structural and behavioural description. The structure describes the components and their connectors inside and outside a layer. Figure 17 illustrates the MTL in the machine control systems family, defined by the ROOM method. The layer has three components that have their own behavioural description. The structure represents the connectors, called ports, of the layer and each component, for example the Message and EventMessage ports. Connectors that are not connected to other components of the layer, or inter-layer connectors, e.g., Control and Exception ports are controlled and connected by the behavioural description of the layer, which performs join and fork operations inside a layer.

Each port has a protocol that defines the direction of the port, the used communication manner, synchronous or asynchronous, and the format of the data. Ports are classified to the four main classes: configuration, control, event, and data interfaces (Paper IV). Exceptions are normally event-based, but error-logging results are transferred through data-exchange interfaces. Configuration and InterLayerControl ports are control connections to the upper layer. Data and Event are the basic data-exchange and event-state interfaces to the upper layer. On the contrary, Message and Communication Exception interfaces are the connectors to the lower layer.

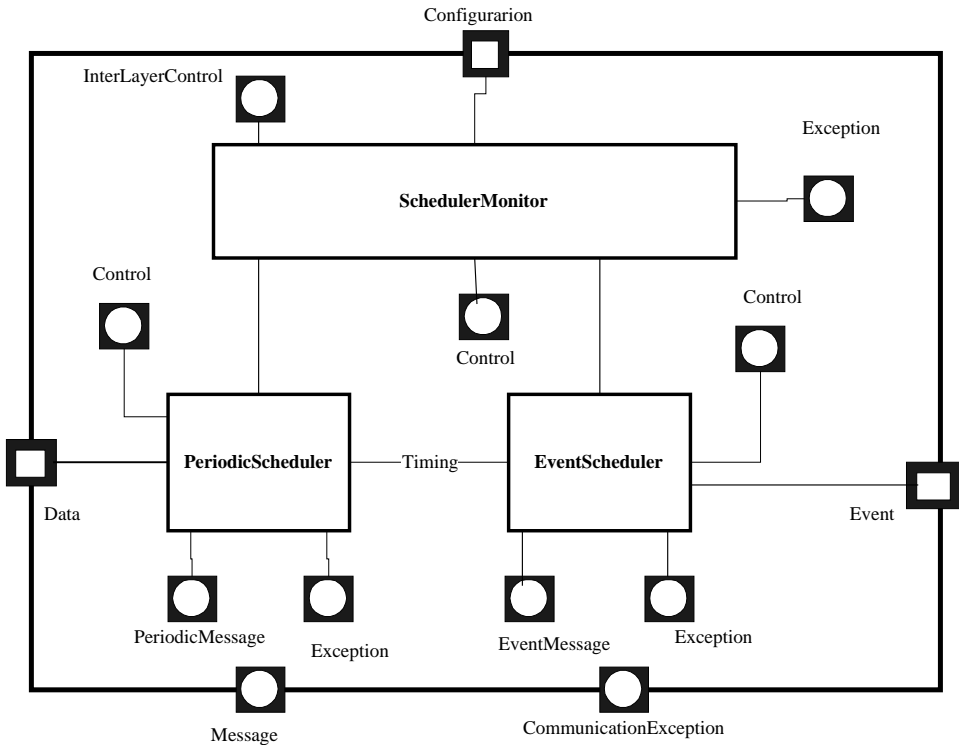


Figure 17. A structural description of a layer.

Each layer has a monitor component that controls the state of the layer. Network and environment components of the interface layer have two monitoring components, one for each other. Inter-level control may be steered to the monitor component or it can be forked to components at the layer level as it has been done in Figure 17. The former centralises control to the monitoring component and the latter allocates control to member-components of the layer. In the latter case, components are more autonomous, and the layer can be configured more easily for product variants by selecting appropriate components. Respectively, exceptions are handled locally by each component, and a notification is sent as a state-event through an exception connector. A typical illustration of the behavioural description is given in Figure 18 that defines the behaviour of the Scheduler-

Monitor, in the case of localised control and exception handling. The monitor component can be kept simple, its only duty is configuration of the layer.

The summarised description of the interface layer as an example lists the main components, connector classes and structural and behavioural constraints (Table 10).

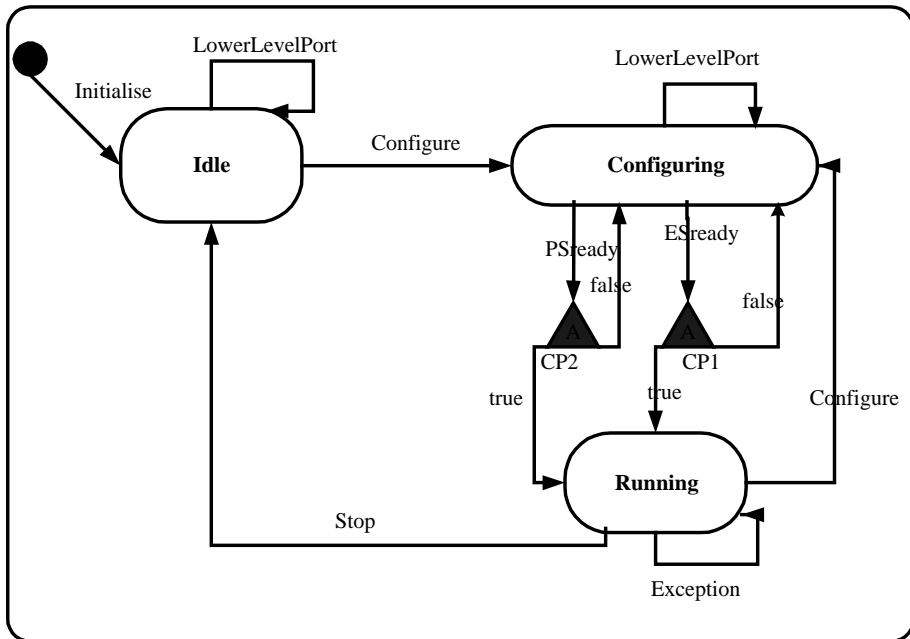


Figure 18. Behaviour of the monitor component.

Table 10. A definition of the interface layer.

<i>Components</i>	<ul style="list-style-type: none"> <li>• NetworkInterface: message transferring, network configuration and monitoring</li> <li>• EnvironmentInterface: data acquisition, control sending, and monitoring</li> <li>• OSInterface: a bridge to operating-system services</li> <li>• ServiceInterface: provided services for applications</li> <li>• ErrorHandler: optional, local error handling</li> </ul>
<i>Connectors</i>	<ul style="list-style-type: none"> <li>• Classes: configuration, control, state-event, and data-exchange connectors</li> <li>• Role: intra- and inter-level connectors</li> <li>• Type: input, output and bi-directional connectors</li> <li>• Protocol: asynchronous inter-layer communication, synchronous and asynchronous intra-layer communication</li> </ul>
<i>Structural constraints</i>	<ul style="list-style-type: none"> <li>• Configuration connectors for co-operation and co-ordination layers</li> <li>• Control and data-exchange connectors for the upper layers</li> <li>• The Bridge pattern for isolating interface classes from their implementation</li> </ul>
<i>Behavioural constraints</i>	<ul style="list-style-type: none"> <li>• Errors are handled inside the layer, or dispatched to the co-ordination layer (variability concerns)</li> <li>• Restricted state-events to the co-operation and co-ordination layers</li> <li>• Synchronised configuration with the co-ordination layer</li> </ul>

### 4.3.2 An agent architecture

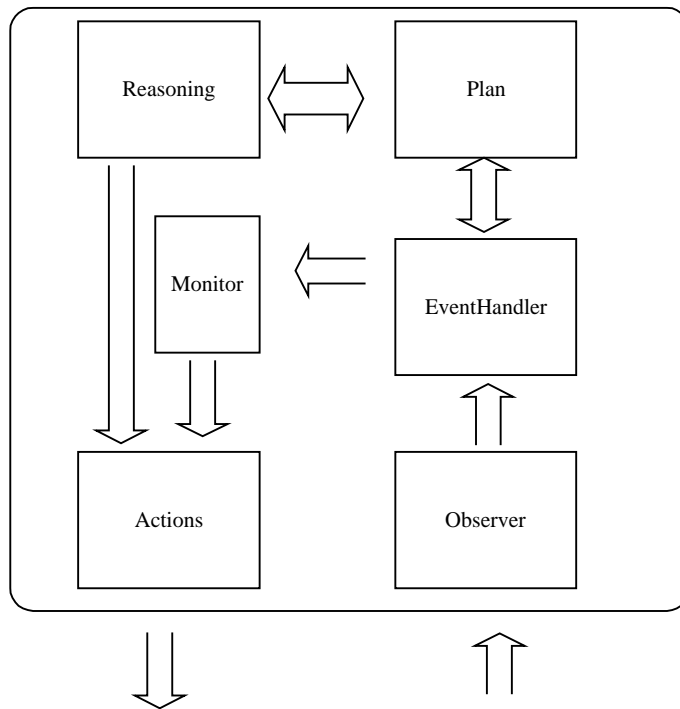
The subsystem tier consists of software agents, but they may also be used in the co-ordination and co-operation layer of the integration framework. The aims at applying agents are:

- to outline responsibilities of software components,
- to restrict control-treads among components, and
- to get coherent structure, behaviour and interface descriptions for container components.

Agents may be reactive or intelligent. A reactive agent is an autonomous operational unit without knowledge of the common goal of the system. A reactive agent may be a surrogate that has a substitute in the real world and is stimulated by events occurred in the environment. State-based controllers and dispatchers are typical exemplars of reactive agents.

An intelligent agent has knowledge and alternative ways to perform operations. An agent is active by requesting, responding, activating and connecting to other agents, and it makes decisions on the base of its knowledge and received information. An intelligent agent has higher-level responsibility than a reactive agent, and it may use several communication manners. Events and signals carry state-based information. Message-based communication is dealt with request-reply couples, directed and undirected propagation messages. In some situation, commands may be necessary. However, communication is asynchronous because of autonomy. An intelligent agent has mechanisms to acquire and handle varying data on the basis of knowledge stored as states and rules. An agent may act, for example, as a co-ordinator, a connection manager, a message router, an allocation optimiser, or an event scheduler.

An agent is logically formed from the six parts (Figure 19). An observer is a data acquisition mechanism to environmental and internal events. An event-handler identifies and deals with events by accepting, refusing and propagating. It may also make low-level decisions, by combining events and data before transferring it for monitoring and reasoning. The third part is the plan that defines the goal and the means to achieve it, i.e., the relevant events and operations within the known period or context. The reasoning part uses the plan as rules to activate and perform actions. If the action is distinct, and there is no alternative way to perform it, the monitor activates the action directly. The plan and reasoning of a reactive agent are simple, and mostly implemented by state-machines and data structures.



*Figure 19. The parts of an agent.*

The main structure of a software agent has three components: an abstraction, a control, and a presentation component (Figure 20). An abstraction component defines data required for the plan and the rules, used in identification and reasoning. The abstraction component also includes data for configuring connections, i.e. logical names, and communication partners or groups.

The content of the abstraction component is used by the control component that has the only access to the data of an agent. The plan and reasoning of a reactive agent is implemented as data structures of the abstraction component and hard-coded state-machines of the control component (Papers V and VII). However, the rules can also be defined by means of separate ECA rules (see section 4.3.3) that the control component activates by triggering events (Paper VI). The interface layer and a common mechanism, e.g. an ECA executive, provide the agent interface, the observer and event-handler mechanisms. The interface mecha-

nisms are configured by event identification numbers and the propagation plan defined by the abstraction component or ECA rules (Papers V and VI).

The presentation component is optional, most reactive agents do not need presentation components. Intelligent agents that inform and collaborate with users have presentation components, which can show the same information in situation-specific ways, i.e., the same information in several places in different formats. Therefore, the presentation component is logically a part of the agent, but loosely connected to the control component through the agent interface (Paper VII). Flexible coupling assists evolutionary changes and location-independence of user interfaces.

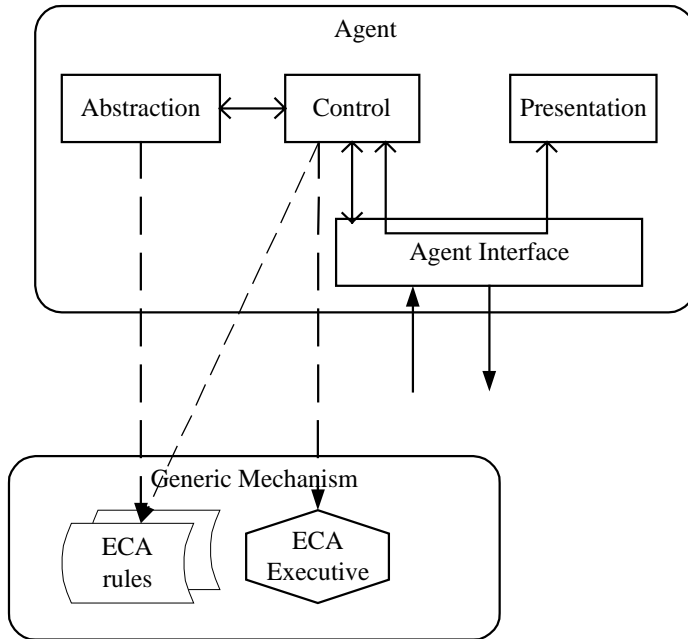
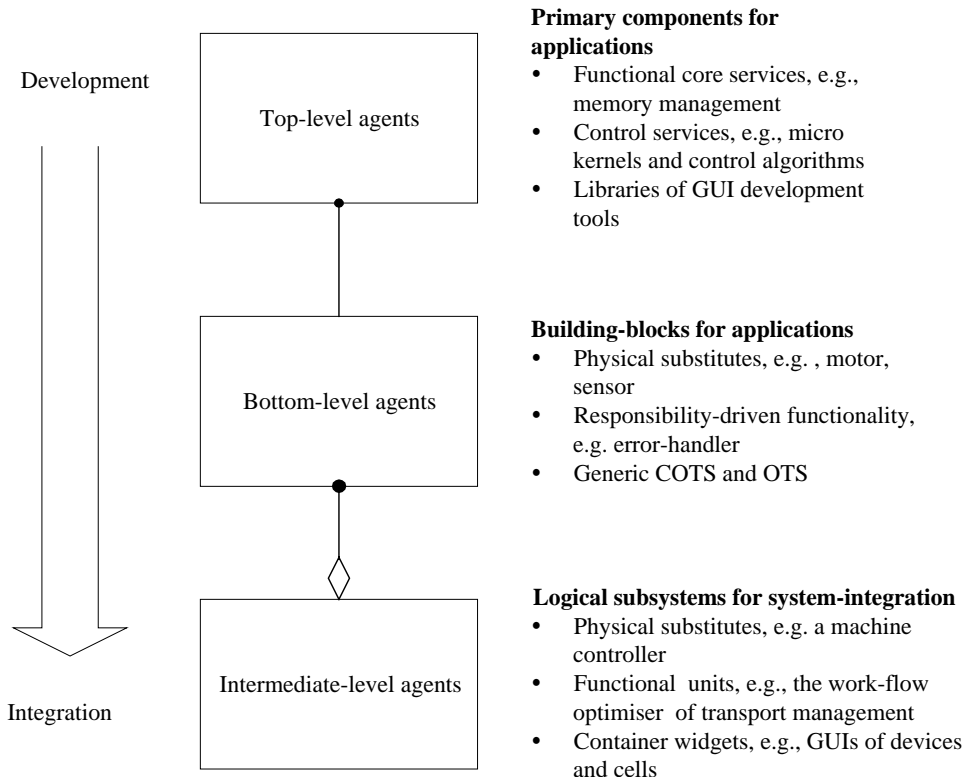


Figure 20. Components of an agent.

Agents can belong to the bottom, the intermediate or the top layer. Top-level agents are fine-grained operational components, used by the bottom-level agents to achieve their actions, for example, for error-handling and sensor-data acquisition. Bottom-level agents are building blocks for the medium-grained agents of the intermediate level. Medium-grained agents perform activities reasoned from

functional responsibilities, product variations and organisational aspects. Top-level and bottom-level agents are reusable units of the software development, but medium-grained agents are autonomous units, integration-oriented components and their development may be allocated among the team-members and subcontractors (Figure 21).



*Figure 21. Layered agents.*

The characteristics of a medium-grained agent are summarised in Table 11.



Table 11. The definition of a medium-grained agent.

<p><i>Components</i></p>	<ul style="list-style-type: none"> <li>• <i>The abstraction component</i> includes structural and behavioural information. The component defines the logical name of the agent, the group the agent is belonging to or its logical connections to other agents. The behavioural information defines the states of the agent and internal sub-states and operations. Internal state-information is separate and restored while reconfiguring, contrary to the main states of the agent that are generic for all agents.</li> <li>• <i>The control component</i> is the connection point to other agents and the only access to the content of the abstraction and presentation components. The control is centralised in the component, but common mechanisms, e.g. an ECA executive, may be used as outside services.</li> <li>• <i>The presentation component</i> is a user interface of the application-logic part of the agent. The component has tight logical coupling with the control component, but it is physically independent.</li> <li>• <i>The interface component</i> defines messages and their handling, used by the control and presentation components.</li> </ul>
<p><i>Connectors</i></p>	<ul style="list-style-type: none"> <li>• Classes: configuration and communication connectors</li> <li>• Role: inter- and intra-agent connectors</li> <li>• Type: input and output for abstraction, bi-directional for control and presentation components</li> <li>• Protocol: synchronous for abstraction, asynchronous for presentation and control components</li> </ul>
<p><i>Structural constraints</i></p>	<ul style="list-style-type: none"> <li>• The PAC design pattern, optional presentation component</li> <li>• The Bridge pattern for isolating the interface component from the operating system</li> <li>• The intermediate level only has configuration connectors.</li> <li>• Inter-agent communication through the communication connectors, a special interface of control and presentation components.</li> </ul>
<p><i>Behavioural constraints</i></p>	<ul style="list-style-type: none"> <li>• Communication and configuration have separate message definitions and mechanisms.</li> <li>• The interface has a synchronisation mechanism for on-line configuration.</li> </ul>

### 4.3.3 A client-server architecture

Client-server architecture is the upper-level architecture that intermediates between tiers, the subsystem tier that requires services from the integration tier. The focus is on integration and therefore, the architecture describes provided services and their interfaces. Because the integration framework is the platform of a product-family, it is also varying. Therefore, the framework has mandatory and optional services. The classification of the services into the mandatory and optional application-specific services and generic services gives an idea of categorised services. The integration framework provides the mandatory services, and variant-dependent services are offered as optional software packages that are loosely connected to the integration platform (Figure 22).

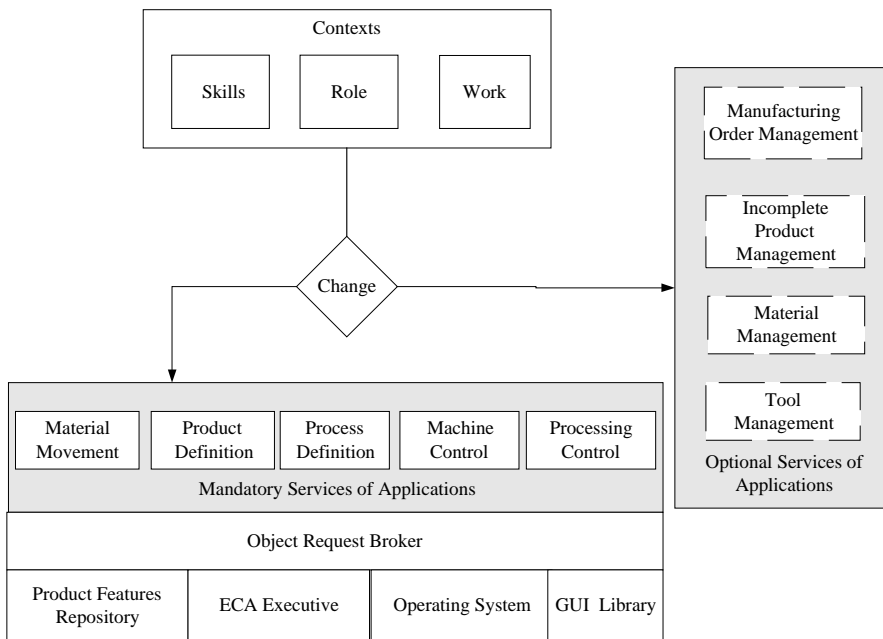


Figure 22. Mandatory and optional services.

The integration framework has the four layers with different combinations of services (cf. Figure 9). The services of each layer and their appropriate use are described in the following sections.

### **Services of the message transmission layer**

The MTL consists of one message-passing service or all three, the RPC, the periodic and event-driven messaging. Adapters of legacy systems are varying with the need of adaptation. A commercial RPC is appropriate for mission critical systems. The event-driven messaging is used in soft RT systems, and periodic and event-driven messaging are jointly used in hard RT systems. The principle of each communication service is different. The RPC highlights sequential functionality, and concurrent processing is the aim of asynchronous events and periodic messaging. However, the latter is focusing on time-driven messages (Figure 23). The connectors of the periodic and event-driven messaging are depicted in Figure 17 and RPC based messages are defined by IDL as two-way or one-way calls.

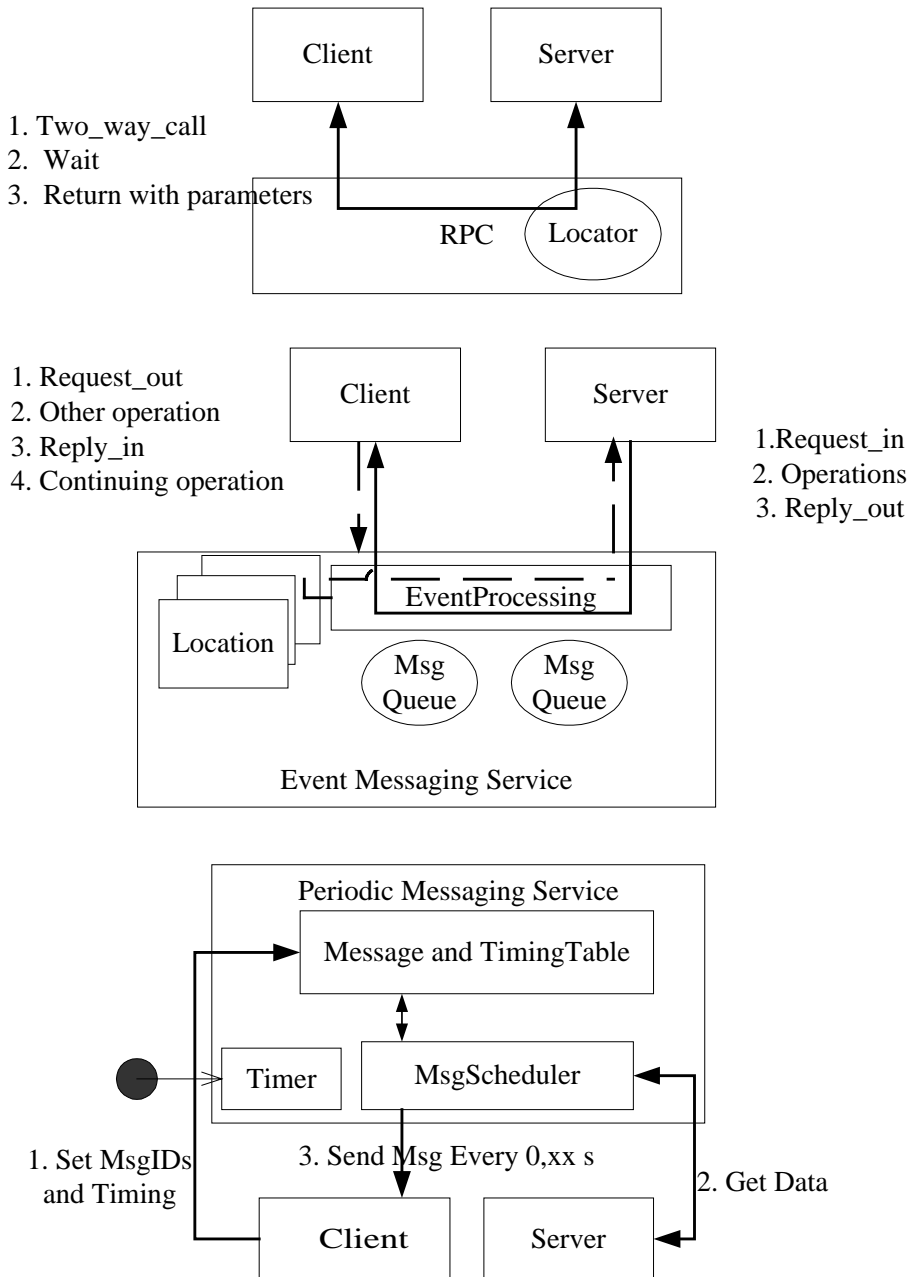


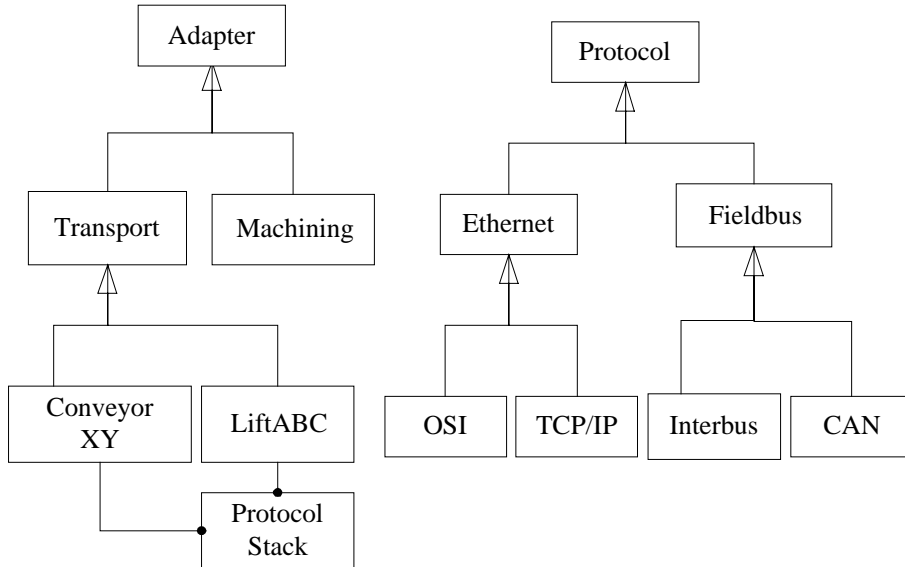
Figure 23. Principles of the message transmission services.

The RPC is the simplest communication manner for application developers, but clients and servers are tightly coupled in respect of timing and control. Therefore, it is appropriate for mission critical systems, but not for real-time systems. Location-independence is provided by an internal interface repository and a locator service. One-way remote calls simulate the message-based communication, but a server and its clients ought to use the similar interface.

Event-driven messages provide loose coupling as regards timing, interface and location. However, it presumes that location information is defined and available. The interface is generic, the message identification number is the only coupling information between clients and servers.

The periodic-message service does not have a locator. The service presumes that every client defines the messages it is interested in, and the message definitions are loaded to the nodes during the start-up. However, configuration can also be done dynamically using event-driven configuration messages. All periodic messages are coded in a common way and receivers ought to be intelligent enough to pick out the messages, which they are interested in, from the communication channel. Therefore, clients and servers have to design in a beforehand-defined manner and binding is done in the development phase.

A flexible adapter adapts legacy systems, which do not use any of the provided messaging services (Figure 24). In the example, the adapter class defines a standard service interface to dispatch messages through the RPC based ORB (Paper VI). The messages of transporting and machining devices are derived from the *Adapter* class and further converted to the form of physical machines. Correspondingly, filters are used to match the data from machines to a generic data format understood by the other components. Filters and conversions are primary components that are used as building blocks. The protocol stack, the lower part of the MTL, may have one or more instances of classes defined by the *Protocol* class hierarchy.



*Figure 24. A flexible adapter for product-variants and legacy systems.*

Commercial components can be used at the MTL to substitute one service or the whole layer. The message queue service of the QNX operating system was used as a message-passing service of the automatic repayment systems (Paper V). Dynamic connections between clients and servers were also based on the message queue service (Paper VII). If message passing services for a LON fieldbus and serial communication are needed, the off-the-shelf component and COTS are integrated to the MTL (Figure 25). The co-ordination layer configures the services, used by the routing service. The FMS product family used the RPC as a message-passing service, and a commercial ORB provided the whole MTL (Paper VI).

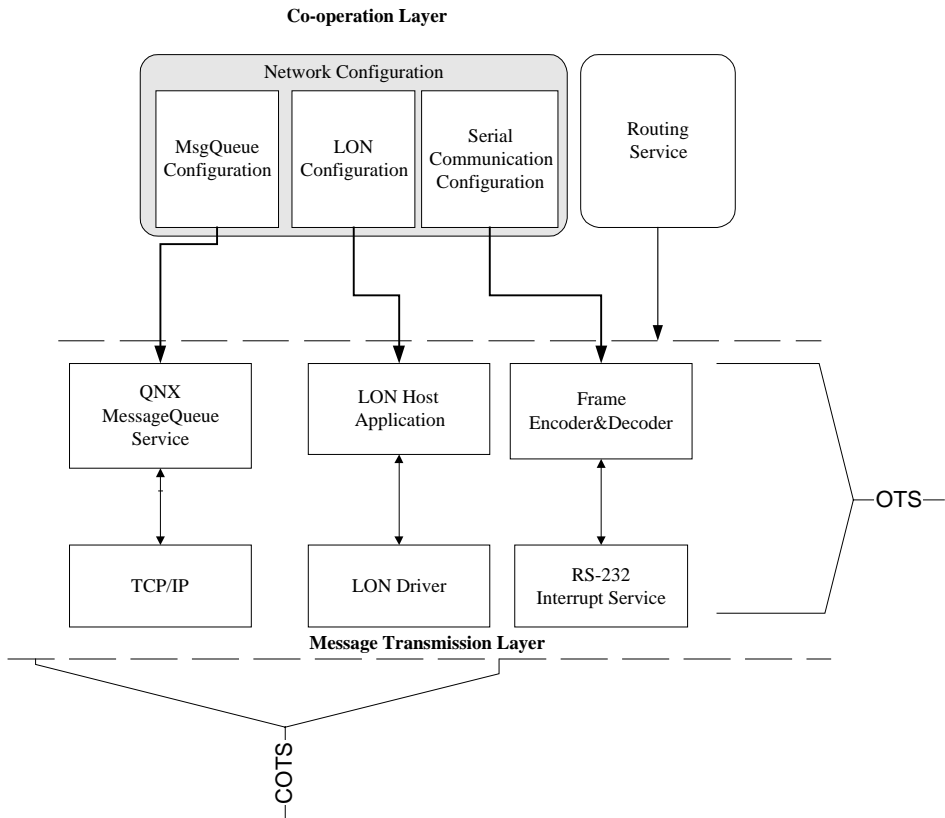


Figure 25. Reusable OTS and COTS in the MTL.

### Services of the co-operation layer

The co-operation layer consists of the network configuration, a connection manager and a routing mechanism. The network configuration service, as depicted in Figure 25, may configure commercial or off-the-shelf components. Configuration of periodic message passing is implemented as a dispatcher that changes the message and time table according to the system's state. The configuration is called mode-based configuration (Paper IV) and it provides a simple solution for run-time flexibility. In practice, the configuration is made by changing the content of the data structures of the configuration management, which may be allocated to one or each nodes of the system.

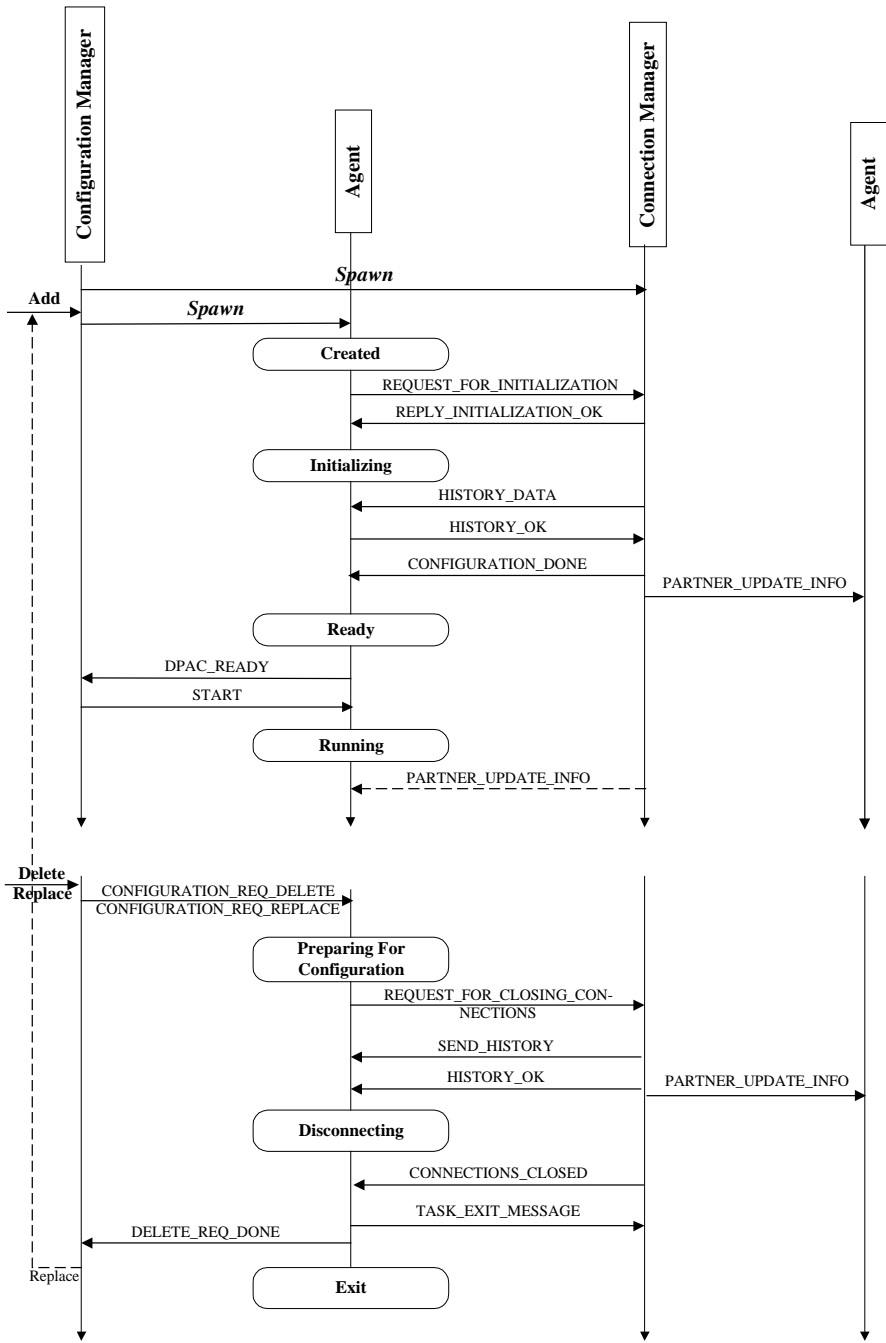


Figure 26. Dynamic binding of connections.



The connection manager deals with the bindings of agents by connecting and cutting connectors at run-time (Paper VII). Each agent defines its structural information, i.e., the logical name, the partners it is willing to be connected to or a group to which it belongs. The connection manager carries out the connections according to this information, when the system is ready for it. The behaviour of dynamic configuration depicts the negotiation-based communication and synchronisation between the configuration manager, connection manager and an agent (Figure 26).

If several message passing services are used, the messages need to be routed according to information produced in the network configuration (cf. Figure 25) and the connection manager. The network configuration manager configures the MTL for the purpose of the co-ordination layer and produces the available physical addresses. The connection manager maps the logical names to the physical names according to the allocation model, defined by the configuration manager of the co-ordination layer. The router is a mechanism that uses the connection information for routing messages to the correct addresses. A dispatcher is a typical implementation of a routing mechanism.

### **Services of the co-ordination layer**

The co-ordination layer consists of three management services: the data management, the control management, and the configuration management. Due to the characteristics of the layer, which is more application-specific than the lower layers, the needed services and their implementations vary according to the characteristics of a product family.

The data management service can be a component, a main-memory database, or a commercial database. In hard real-time systems, data management components are allocated to the nodes, and the consistency is dealt with exchanging system-level data between nodes (Paper IV). A centralised main-memory database is a more sophisticated manner for soft real-time systems, which handle objects and data associated with them (Figure 27). Due to the reasoning mechanism, the data management has characteristics of an intelligent agent. A commercial database is suitable for mission critical systems, e.g. manufacturing systems. The problem, arising from COTS, is the interface technique. A generic interface may be missing or it makes the system tight coupled with the COTS.

Figure 27 depicts the reasoning process in the repayment systems. The operation and command databases define static information for treatment operations and presentation formats. The processing of objects, inserted to the object database by the identification activity, uses the defined information for reasoning. Data and control are tightly coupled in this example, and therefore, the mechanisms and data are allocated near to each other.

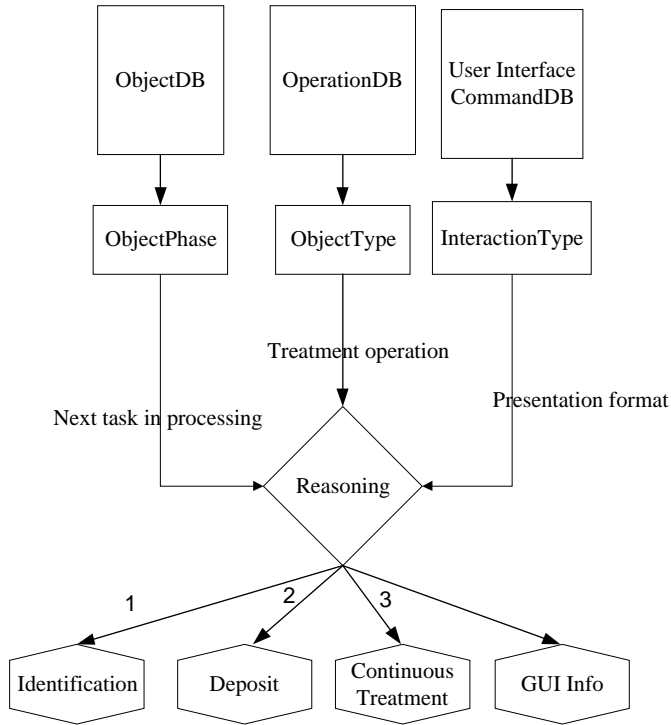
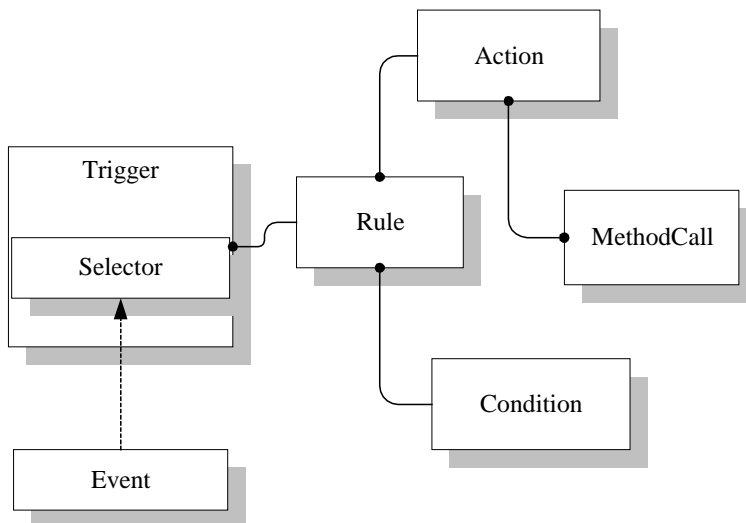


Figure 27. An example of data-based reasoning of the control and data management services.

The configuration management is a service provided for integration and maintenance. Depending on the needs of the product family the configuration service provides mechanisms and the user interfaces for configuring connections, defining the database, messages and their parameters, activating and deactivating features. The connections of software agents are configured by the definitions of the abstraction components, given through the user interface (Paper VII). A da-

tabase may be changed by a special user interface as it was done in the automatic repayment systems family. In the FMS family, the on-line schema modifier was utilised to construct an association changer for manipulating associations between the components of ECA rules (Figure 28). Configuration files and a special user interface were used to configure messages and their parameters. The used mechanism depends on what is changed and when it has to be changed.

The ECA concept is a way to use the rule-based architecture style, combined with the client-server architecture in our case study. The ECA concept defines how the rules have to be defined and how they are performed. A rule has a condition and action-list that are linked together by a trigger. An event, defined as a triggering event in the selector of a trigger, activates the execution of a rule. The ECA executive processes the rules according to their modes and priorities. The mode can be immediate, deferred, or separate. In the immediate mode, the rule is executed immediately after detecting an event. The deferred mode is used when the execution order of a rule is defined by its priority. The separate mode can be used if concurrent rule processing is necessary. In the case study, we used the deferred mode.



*Figure 28. The components of an ECA rule.*

The ECA concept is a centralised controller to manage data, control and configuration. The ECA with CORBA provide event-based behaviour on the RPC-based communication. The ECA was used to propagate GUI information and activate functional and behavioural tasks, e.g., to create a component and activate a semantic component, defined by a rule.

### **Service interface**

The service interface is a provided interface for application agents, and it is used for configuration and communication. The configuration is done by changing the messages the agents deal with, or by changing their physical locations.

The communication is message-based, either events or data. Control is not passed through the service interface. An event activates a service, e.g., the ECA executive observes an event and activates the operations associated with the event. When an agent has executed its operations, a state-based information is sent and stored by the data management service. The next operation is activated as a result of reasoning based on the updated state information. However, if an agent does not have the required information, it requests further information, which is passed as a data-exchange message.

Optional services do not affect service interface. They are activated by a generic triggering-event. The differences in the implementation technology of the co-ordination layer and the applications that utilise its services produce different kinds of service interfaces. However, the service interface should offer the required services of a product family, not services for all kinds of control systems.

A summary of the provided services is depicted in Table 12.

*Table 12. Categorised services and the their relationships.*

<i>Layer</i>	<i>Service</i>	<i>Relationships</i>
<i>MTL</i>	<ul style="list-style-type: none"> <li>• Periodic messaging</li> <li>• Event-driven messaging</li> <li>• Remote procedure call</li> <li>• Adapters</li> </ul>	<ul style="list-style-type: none"> <li>• Network configuration</li> <li>• Network configuration</li> <li>• Other services of the layer</li> <li>• Used messaging services</li> </ul>
<i>Co-operation</i>	<ul style="list-style-type: none"> <li>• Network configuration</li> <li>• Connection management</li> <li>• Routing</li> </ul>	<ul style="list-style-type: none"> <li>• Services of the MTL</li> <li>• Configuration management</li> <li>• Other services of the layer</li> </ul>
<i>Co-ordination</i>	<ul style="list-style-type: none"> <li>• Configuration management</li> <li>• Data management</li> <li>• Control management</li> </ul>	<ul style="list-style-type: none"> <li>• Connection management and other services of the layer</li> <li>• Data and control management: data and mechanisms</li> </ul>

## 4.4 Summary

Because a product-family architecture is the core competitiveness of an organisation, the development of the component framework was presented by means of a conceptual architecture that combines the selected architectural styles and patterns. A PAC agent, applied in the subsystem tier, is a combination of heterogeneous styles: event-based independent components, interpreters, and layers. Application agents are in the role of clients to the integration tier that provides services by layers and independent components. The product-family tier embodies a rule-based system that has the knowledge of a product family in formal rules and an engine for their processing. The implementation of the rule-based system is a part of the co-ordination layer in the integration tier that manages the application-dependent information. Applications are connected to the integration tier by a service interface.

The development of the component framework highlights the concerns of localisation, adaptation, timing, and controlling. Localisation defines how the functional responsibilities, connections and services may be identified and located.

An agent provides a coherent unit for localised functionality. A centralised connection manager is a mechanism for connecting communicating functional elements of a system. It may also be a central point for resource allocation. Separation of functional properties from the resources isolates two development processes, application and system development, from each other.

Because the systems are evolving, adaptability and adaptation are the cornerstones in software reuse. Adaptability is the ability to reuse software in different kinds of environments and products. Adaptation is the ability to change the existing software for different use. Filters and the Bridge pattern give reusable solutions for informational and structural adaptability. Run-time adaptability is achieved by dynamic bindings. Legacy software and COTS may be adapted by adapters and wrappers, by changing their functionality, interfaces and data format.

Schedulers and dispatchers are mechanisms for controlling time and order. Isolating data and mechanism software is inherently flexible, and therefore, less adaptation is needed in the future.

## **5. Experiences with the development of the component framework**

The development of the component framework consists of the four case studies briefly described in Chapter 1. In this chapter, we analyse the differences of the developed solutions as regards the evaluation criteria defined in Chapter 2. In the case studies, the component frameworks supported diverse product families, and the results and experiences of the development of these frameworks are used for justification.

### **5.1 Diversity of the product families and used technology**

The state variables of the case studies are classified to the two main categories: the product and process dependent factors. The used execution platforms and the end-users of the systems changed according to the product family but the development methods, tools and developers' expertise also varied.

The execution platform depends on what the systems are intended to control, how the tasks have to be distributed, and what kind of a distribution medium and operating system the product family uses. The latter two factors depend on the first, the type of system. The machine control systems, i.e., hard RT systems, were quite small embedded systems without any operating system using a CAN field-bus as a distribution channel. The material transferring system was a soft RT system that used the QNX operating system with the local-area network. The repayment systems family, soft RT systems, included four nodes, two embedded controllers, an embedded and a standard personal computer and used the QNX operating system in the PCs. LAN, LON and serial communication were used as distribution media. The mission critical system, i.e., the FMS family in our context, used LAN as a communication medium and Windows NT as an operating system. Commercial software, such as Orbix, Objectivity\DB and MFC, was used as the components of the integration tier. The subsystems of the machine control systems formed a product family, from which commonalties and variations were identified. The repayment systems family included three main variants and customisable features of each variant. The FMS family had two main variants and many customised features.

The role of end-users varied from customers of repayment systems and drivers of farming tractors to workers and controllers of a production line. The amount of concurrent end-users of the systems also varied from one to dozens.

In the case studies, we used object-oriented methods, OMT and ROOM, extended by PFM and scenarios. The implementation language was C++, except in the automatic repayment system and the GUI software of the material transferring system that were implemented by the C code. Component interfaces were defined by IDL in the FMS family. On the other cases, graphical and textual descriptions were used. Prosa/om, ObjecTime and Visio with a set of drawing stencils formed the tool-set.

The used development process was a combination of the waterfall and concurrent software engineering with incremental prototyping. The level of experience varied with the role of the team-members. A domain expert assisted with the product knowledge of the industrial partners made domain analysis and the basic architecture. Software engineers with moderate or novice skills designed and implemented the components in the two latter case studies, on the other hand, the more experienced engineers worked in the first two case studies. The author's contribution to these case studies was the analysis, design and implementation of the first case study, the machine control systems family. In the latter case studies, the author selected the used approaches and was involved in analysis and design, as well as supervising the development of the systems, mainly performed by undergraduates.

## **5.2 Justification for the selected environments**

In the case of the machine control systems, the main objective was to find out the means to define and validate the requirements of a subsystem in order to be able to allocate the development work to several subcontractors. The CAN bus was selected for a distribution media because it is generally used in the machine control systems. LabVIEW was used as the prototyping environment due to its support for graphical components and connectivity with a target system, which removed the need of an expensive test-bench. The ROOM method was selected because of its ability to describe and simulate concurrence and timing properties, required in the design of distributed systems.



In the second case study, the aim was to develop the mechanisms for the dynamic configuration of applications in a RT environment. The commercial components, the QNX, CORBA and the Photon GUI tool were selected on the base of their interoperability. The Orbix for QNX was evaluated but dismissed because of its unsuitability for RT systems. LAN was used as a distribution media in the final solution, but another field-bus, Profibus, was also evaluated but dropped due to its complex configuration procedure and less general use in distributed systems. Due to the Photon tool, C instead of C++ was used for implementing the GUI components.

In the case of the repayment systems, the aim was to develop an application-specific platform for the product family. The same commercial components as in the preceding case study were used, supplemented by the LON network due to distributed variables. The selected commercial component, a card reader, had only a serial communication channel that the platform also had to support. The analysis was made by the OMT method, but the platform was implemented by the C code, due to the used tools and commercial components.

In the FMS family, the aim was to find out a flexible software architecture and mechanisms for managing variability and extending systems. Object-oriented commercial components were used with the intention to get experiences with their maturity and suitability for the application domain, in which relational databases and structural analysis are normally used. However, two relational databases and three OODB were evaluated before Objectivity\DB was selected. The selected operating system, Windows NT, also reduced the alternatives. The following properties assisted to make the decision: modelling support for complex objects, ability to get the CORBA adapter, safety and fault tolerance support, and good references in the control systems domain. The promised CORBA adapter of the database restricted the CORBA implementation to Orbix. LAN was a natural choice for the distribution due to CORBA and the possibility to switch the OSI based connections of the PLC subsystems over to the TCP/IP based connections, recently being come to the market.

In the following chapters, the component framework is examined as regards the evaluation criteria, depicted in Chapter 2.5.

## 5.3 Variability of a product family

### 5.3.1 Defining and managing product features

The consistency and completeness of a product features model are important but difficult. The problem is considerable in small and medium size systems, as it was in the machine control systems and automatic repayment systems. In both cases, the features were clustered into subsystems and the variability was defined inside subsystems. Scenarios and message sequence charts assisted to define a feature completely and follow up the mapping of the features to the layers and components. Subsystems were mandatory or optional and used configurable services, which were the mandatory features of the product family. Moreover, the maturity of the product family facilitated to cluster the features to the subsystems. When the context of the features had been defined, standard CASE tools were adequate for the definition and management of features and no special tool was needed.

In large systems, as in the manufacturing systems family, this did not work. One way to manage the problem is to cluster the features to the services and thereafter, to components (Figure 29). The mandatory and optional services were defined (cf. Chapter 4.2.1) and the principles, such as defined in Chapter 4.2.2, support the management of the features, but a special tool support is still needed. The principles are used inside the scope of a service, an architectural component, and a primary component. After clustering and modelling the features they are formulated as rules and stored into a database, where they can be managed and their use is possible to be checked with an algorithm. The approach needs further studies, and there are still open questions, to which this work can not answer exhaustively. However, the viewpoint-oriented domain requirements definition method and the multi-modelling approach with concepts and features (Mannion et al. 1998; Simos et al. 1998), which have similarities with our approach, strengthen our belief in its correctness.

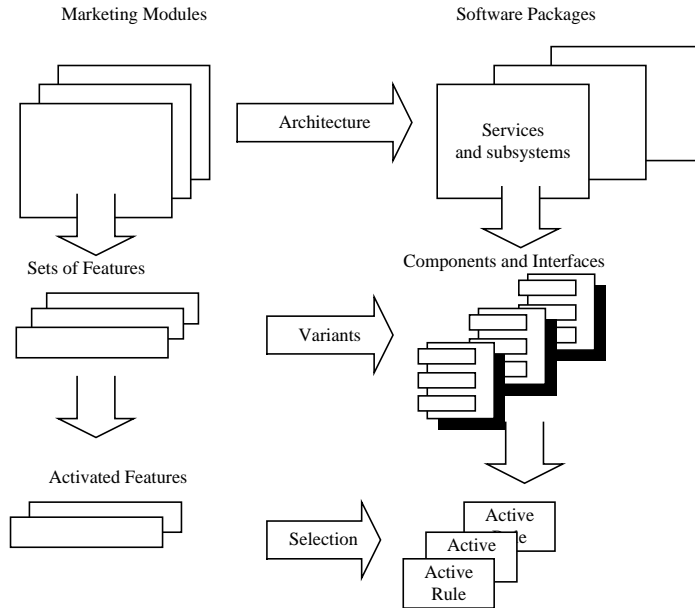


Figure 29. Defining the scope of variability.

The main weakness of our approach seems to be the lack of support for large distributed systems. Therefore, the following areas need further studies:

- A formalised feature modelling method with concepts.
- Tools for modelling and managing features and variants of large distributed systems.
- An explicit means to describe services and manage their relationships.
- Group-ware support that is necessary for the development of large distributed systems.

### 5.3.2 Mapping variability to the architecture and components

The product-family architecture is a conceptual architecture that has several realisations. Therefore, services and architectural components are mandatory, optional, or conditional. A service is a set of components that may be architec-

tural or primary components. Agents, in our context, are architectural components representing applications. Due to loose coupling and dynamic binding there is no problem to manage functional diversity of applications with black-box components at the architecture level. However, the services are more complicated owing to the relationships between components. Layering, separation of mechanisms and data, isolation interface technique and adapters provide flexibility needed for the diversity of services. However, they do not define how the services should have to be organised and configured for product variants. Therefore, the following approach has been applied in the case studies:

- **Scope.** A variable feature was targeted to a layer, a service or a component.
- **Quality.** The target entity was tagged by a basic quality, for example, a hard RT MTL, a soft RT error handler, a safety-critical brake control system and a DB with the fault-tolerance option.
- **Relationships.** The relationships were defined between layers and services, as depicted in Table 12 and Figure 30.
- **Mapping.** Variability was mapped to data, structure, behaviour and connections of the components by choosing an appropriate means to carry out the variants.

In the case of the machine control systems, the *variable services* were carried out by a generic communication service that was used in every node and configured according to the communication description. Each node also had a local configuration manager in the upper layer, one of them acting as a co-ordinator. The communication service was configured by data, which is a time and space saving means, and therefore, suitable for real-time systems.

The repayment systems required different configuration management for the network and co-ordination layer. However, the following principles were used in both cases:

- The upper layer configures the services of the lower layer, controlled by the monitoring component of the layer.

- A special service of the configuration management deals with the system-level configuration on the co-ordination layer.

The approach provides that a service defines its relationships to system-level data and the services on the underlying layer. Variability of the system-level data is defined in the features tree as informative leaves. The relationships of the services, as depicted in Figure 30, are structural and behavioural defining prerequisites. A connection manager dealt with the structural relationships in the MES family, and the ECA executive managed the behavioural relationships in the FMS family.

There are two main approaches to carry out the *variability at the component level*: by inheritance in white-box components and by parameters and connection points in black-box components. The use of the hot-spot design pattern and inheritance were the appropriate ways to implement variability, if changes could be anticipated. In this case, the structure, operations, and behaviour may be changed at the same time. The used approach also depends on the amount of dependencies and the use of a component. A component that has a few dependencies and is repeatedly used in the systems, is a black-box component, otherwise white-box.

Parameterisation was a useful technique to vary mechanisms and install surrogate components to match the real environment. The means was used, for example, for the sensor and valve components in the machine control systems family and for the device components in the FMS family. The parameters were fixed by instantiation, statically or through a user interface.

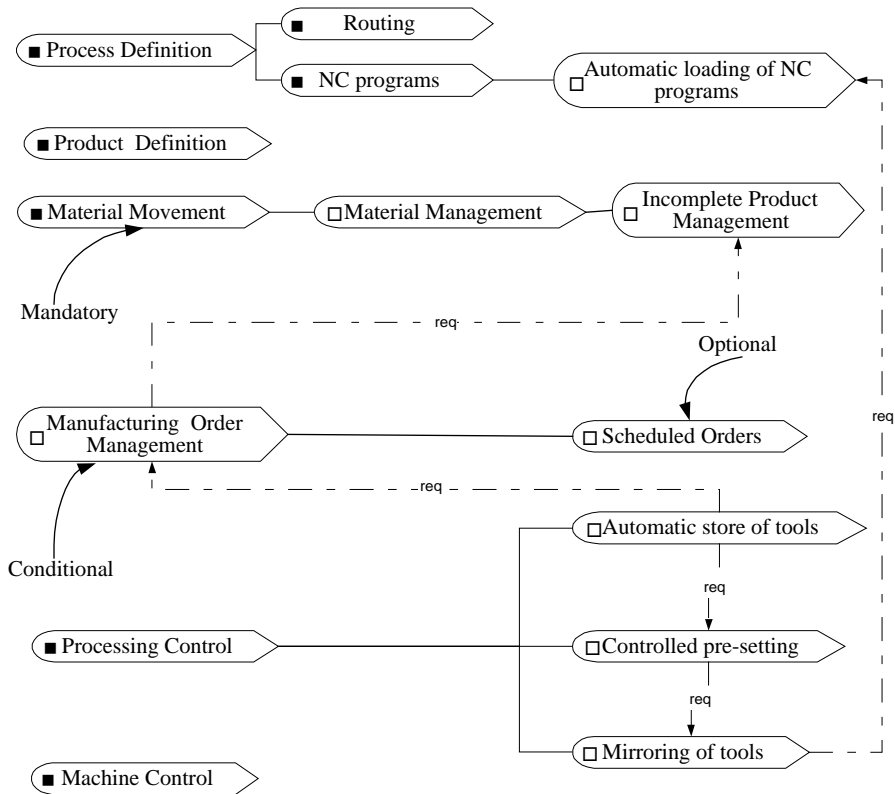


Figure 30. Relationships of the services in the FMS family.

Although the structure and functionality of an architectural component is the same, its *connections* may vary and filters are needed to adapt a component to its new context. Two connection points were provided: the connection manager to change the connections of the PAC agents and the ECA executive to change the connections of the internal components of the PAC agents and services. Although the connection manager does not enable to import a filter directly, a filter class can be added to the class hierarchy of the COMI interface. In this way, the internal state of a replaceable agent is able to be conformed to the state of the old one. The same can be carried out by a black-box component that the ECA executive connects to an agent. In the case studies, a filter was applied to converting messages from one protocol format to another, and it was implemented as

the methods of a class in the inheritance hierarchy of the adapter, depicted in Figure 24.

If behavioural variability was required in services and non-architectural components, ECA rules with the ECA executive was used. This way of managing variability is not straightforward, but it provides the most versatile means of reusing product features. Through the ECA rules, all aspects of a component -- structure, operations, data, behaviour, and connections -- may be changed at the same time or one at a time. Therefore, variability is the strictly beforehand-defined property of a component, and the ECA concept is a flexible solution to manage semantic components of the product-family tier. On the other hand, it has side effects on the execution time and memory requirements, and therefore, its use is restricted in RT and embedded systems. A faster implementation with a main-memory database and an optimised ECA executive in a node with necessary computing resources may be the solution to the problem.

### **5.3.3 Configurability of a product family**

The selection of product features is a hierarchical decision tree (cf. Figure 29). In the first phase, mandatory and optional services and logical subsystems are selected. In the second phase, conditional relationships of services and subsystems are considered and corresponding components are selected. In the third phase, the fine-grained features of the services and application agents are selected and possible new features are defined. The dependencies of new features have to be checked as regards the features tree of the product family. In the optimum case, the selection is based on black-box components, except fine-grained features that are implemented by an inheritance hierarchy and parameters.

ECA rules replace the white-box feature-selection by bringing it to a later development phase. In this case, the selection of fine-grained features is done in the installation and instruction phase by on-line changes in the associations of the rule-members. In large control systems, this way produces better satisfaction of the end-users that could smooth the system according to their work and working manners. However, if the features can be selected in the development phase and need not to be changed during the life-cycle of a system, the on-line configuration is unnecessary. Selecting black-box components and configuring the inte-

gration platform with data and operation through parameters give the reasonable flexibility as regards product features.

If on-line configuration is required, the connection manager and ECA executive provide the mechanisms to change applications and features. By the connection and configuration manager, an application agent can be moved and changed. Its presentation component can also be located to a different node than the control and abstraction components. However, their reconfiguration always has to be made at the same time.

The connection and configuration managers were used with the messaging services of the real-time operating system, QNX. The performance tests of the implementation showed that the execution times are too slow for real-time systems. Preparing a replacing agent ready to work before the connections of the old one are cut and reconnected to the new agent, shortens the execution time of the replace-operation. However, it may be an adequate improvement for soft real-time but hardly for hard real-time systems. Thus, the configuration of hard RT systems should have to be made during the start-up or using a faster messaging service and an optimised interface.

The ECA concept provides the following reconfiguration manners:

- to change the state of a trigger,
- to change the associations between triggers and rules,
- to change the conditions and actions of a rule,
- to change logical links between events and selectors, and
- to create a new rule chain for a new feature.

The first three are on-line, the rest off-line configuration manners. A simple user interface was developed for modifying triggers and associations on the fly. New rules were evaluated while constructing the prototype incrementally.



The state of a trigger activates a product feature or a set of features. The property is useful in the installation phase by allowing one to add new features on-line. However, it provides that the corresponding functionality is implemented in the services and applications.

By clustering a set of functional features to a rule, a standard way to react, for example, to errors can be defined and reused. If this behavioural pattern is needed to be changed, replacing the rule with a new one carries out the change and an on-line change in association between a trigger and rule is enough. Changing the components of a rule, its conditions and action-list, produces subtler tailoring capabilities for tuning a system.

The link between events and selectors provides a possibility to change the behaviour of the systems. An event can be connected to a new behavioural pattern or the existing behaviour can be activated by a new source. The former is needed, if the functionality of a system is changed. The latter is used, if the activator is changed, for example by changing a software or hardware component. However, recompiling is required in both cases.

The rule database represents the knowledge of a product family as a part of the core product, whereas the activated rules and associations between the components of the rules form the description of a product variant. When the product features model is constructed incrementally or updated, new features are added by defining a new rule. Respectively, the ECA rule is destroyed, when the feature is no longer relevant to the product family. However, the use of the ECA concept has the following limitations:

- The maturity of a product family and the development process must be high enough so that the product features could be identified and defined exactly by rules.
- Because of its influence on the size of memory and performance, lighter configuration manners are more suitable in small systems and the systems that need only limited customisation during the integration and installation phase.

- The rule database may be deployed as a component of the systems. In some cases, the new version of the rule database is enough for upgrading, but this requires the termination of a system and restarting the applications.
- A centralised configuration builder, which actually creates the object instances to the rule database, simplifies the implementation of rules and acts as a reusable software component. However, no automatic support exists and each change in the product features is reflected in the configuration builder that has to be updated manually.

The configuration tool of the automatic repayment systems provided a means of changing the identification manners of target objects, as well as the objects and operations of the continuous treatment (Figure 31). The data-based configuration activates and changes the behaviour of the system, but it is a special solution for restricted use. On the contrary, the ECA concept is a generic mechanism and provides the same functionality extended by the ability to reuse the components of existing rules, i.e., actions and conditions. However, the simple configuration tool was enough in the repayment systems family.

The configuration of the machine control systems was made by loading configuration data during the start-up. The same technique was also used for defining messages of the interacting applications in the repayment systems family. In most cases, this is enough for integrators, if a system is installed at the same time. On the contrary, large systems, such as manufacturing systems, are incrementally installed, and then the ECA concept is a powerful tool for fine-tuning a system's properties.

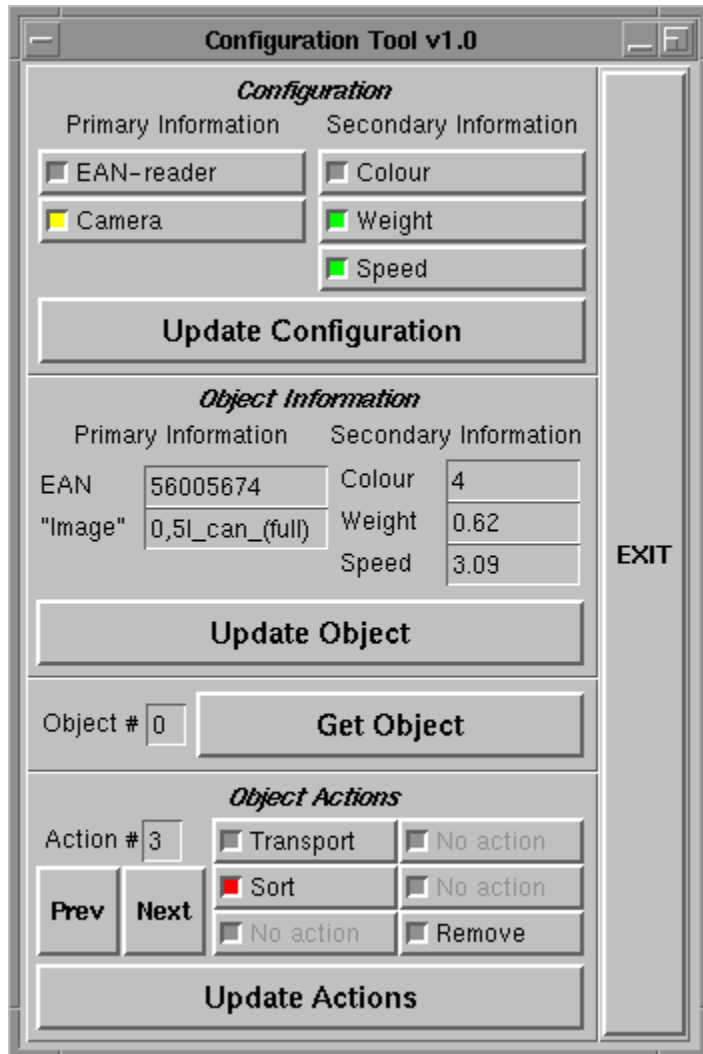


Figure 31. A configuration tool.

## 5.4 Interoperability of distributed control systems

### 5.4.1 Interoperability between tiers

The inter-framework interoperability is achieved by generic communication mechanisms, configured for the required use. In the machine control systems,

the periodic messaging service was configured by the message identifications and timing requirements by loading them to the configuration managers during the start-up. The local configuration manager selected a set of parameters and configured the communication mechanisms during the execution according to the state of the system at the time. The other service, to which the periodic message service was connected, was the data management service. In other words, the interoperability was achieved by implementing a generic communication mechanism, and the configuration managers controlled the co-operation of the applications.

The lightweight communication interface was the generic component for interoperability in the repayment systems. The interface, configured during the start-up, is suitable for real-time and embedded systems with limited memory. The performance of the communication depends on the underlying message transmission services, not the interface itself. The same approach was used in the material transferring systems, but it was implemented as a frame. The generic communication interface simplified the connections of applications and guaranteed the interoperability between subsystems and the integration platform.

The ORB as an integration platform, used in the manufacturing systems family, seems to be adequate for soft RT and mission critical systems, where the average frequency of operation calls is not high. Usually, the responses to the user-activated commands are desired to get within 200 ms in order to avoid user's negative feelings about the system. This restricts the allocation of applications and the way in which the ECA executive handles the GUI messages. Although the ECA concept provided event-based behaviour using the one-way calls, the interfaces of applications were defined by IDL and communication was RPC-like. Because no generic interface component could be used, the applications with the underlying integration framework, the CORBA ORB, were tightly coupled.

### **5.4.2 Interoperability of services**

The type of the control systems family had the greatest influence on the required services of the integration platform and their allocation policy. In the machine control systems, the co-ordination and co-operation layers were allocated to each physical subsystem, whereas the automatic repayment systems had the central-

ised ones. Due to performance requirements, the configuration and control management of the machine control systems was near to the messaging services that passed messages regularly, but shortly.

The services of the automatic repayment system consisted of the configuration management of application agents and the network, the memory management, and the message transmission layer with asynchronous message-passing to the LAN, LON and serial communication networks. The routing mechanism on the co-operation layer dealt with the messages to the proper addresses. The network interface layer with appropriate services integrated the services with the physical environment. Because there were no hard real-time constraints, commercial components were used without testing their performance. Due to missing documentation, an additional testing software was needed for discovering the interface of a commercial card-reader, connected by a serial communication channel.

The FMS family has an evident need of CORBA services as mandatory services. For example, the concurrence, transaction management, and persistence services are required. Clients have to be able to perform atomic operations within one or more server objects. Data consistency should also be ensured and the mapping of data objects into the database should have to be transparent. Unfortunately, only a few ORB vendors could provide these services. Therefore, our approach utilised commercial ORB as the building blocks that were fulfilled by off-the-shelf components, an adapter and ECA executive. However, they do not remove the need of above-mentioned services.

The ECA concept provides services for several purposes. Firstly, it facilitates to create and bind surrogate objects and their GUIs. Secondly, it controls the co-operation of application agents, if the communication sequence is implemented as a sequence of operations and stored as a method list in the database. Thirdly, the ECA executive was used for propagating event-based messages to several GUI components. In this case, the ECA concept acted as a dynamic binding mechanism performing the logical multi-cast communication. This is useful when information flows have to be re-ordered and controlled. It also provides a systematic way to pass information from the control components to the presentation components.

The other off-the-shelf service, applied in the manufacturing systems family, was the adapter (Figure 24) that provides the RPC compliant communication to the legacy systems of the lower physical layer. The adapter has three parts; a standard interface, derived from the responsibilities of the controlled machines, a filter component that formulates data to the required physical format and vice versa, and a protocol stack, which consists of the used protocols. The adapters may be implemented in the same server with corresponding machine control objects and protocols. However, separate servers are simpler and more useful if several protocols are required and physical changes are anticipated.

### 5.4.3 Interoperability of applications

Applications have to be syntactically and semantically interoperable. The interface technique and the principles of its use provided the *syntactic interoperability*. There was no problem, when the same interface component was used in every application (Papers V and VII). On the other hand, the use of the ORB required some additional principles (Paper VI). Because of the need of memory and memory management, the CORBA interface should not be used when an *a-part* relationship prevails between objects. A database, not CORBA objects, should have to manage the associations between data objects. Large systems also need a garbage collection policy that should be selected from the two alternatives: the client destroys the object when it is no longer needed, or the server cleans up the objects after specified time-outs. The selection depends on the timing requirements of the product family.

It is the responsibility of the co-operation and co-ordination layer to guarantee the *semantic interoperability* of applications. If the co-ordination consists of the controlling mechanism and the data that configures the mechanism, the flexibility of the systems can be provided and the connections of applications are interchangeable.

Defining and changing the co-ordination and co-operation layers of the machine control systems were complicated due to the strict timing requirements associated with the definitions of periodic messages. Therefore, the communication between subsystems had to be designed separately, and next, the message definition with their timing requirements were loaded to the co-ordination layer, i.e. the configuration manager, and further to the communication components. The

approach needed a special tool with simulation capabilities. The ObjecTime tool was selected for this purpose. After the architecture was modelled and simulated, a real-time subsystem was implemented. In the simulation, the interoperability was verified by a model of two subsystems and the CAN network. Although the timing requirements of the messages were easy to design for two nodes, it may be extremely difficult for a system with dozens of nodes.

In the repayment systems, the same architecture was applied, but the system had the centralised co-ordination and co-operation layers with soft real-time requirements, and therefore, the semantic interoperability was much easier to manage.

Although the ORB provides the syntactic interoperability of application objects, ORB-based systems without domain services have a co-ordination problem. In large systems, the chain of control spreads out into the distributed objects and therefore, the co-ordination of the objects is impossible to manage without the co-ordination layer.

## **5.5 Adaptability of the component framework**

### **5.5.1 Portability of distributed control systems**

The ability to move applications and services to another environment, shows the independence of software. In the case studies, application-independence was supported by the following manners:

- Applications are *uniform* software agents that apply the PAC pattern.
- The levels define the *scope and responsibilities* of agents: primary components at the top level, building blocks at the bottom level and logical subsystems at the intermediate level.
- A *standard interface* to the integration framework.
- *Dynamic configuration* for the agents at the intermediate level.

- Presentation components are *separated* from abstraction and control components.

The independence of services is more complicated. The services were targeted to the layers that attempted to reduce their dependencies by separating technology-dependent and application-dependent services. The Bridge pattern was used to separate technology-dependence software, for example the operating system, from the messaging service. However, if a service is transferred from one environment to another, some modifications and recompiling are needed.

In machine control systems, the implementation of surrogate components, e.g., a sensor component, with a separate hardware interface provided the hardware-independence of sensors and valves. The Bridge pattern was useful to manage the evolution of hardware interfaces and surrogate components separately. In the repayment systems, the separation was implemented by gathering the hardware-dependent parameters and software to a file that could be changed at the same time as the communication network.

### **5.5.2 Flexibility of the component framework**

Flexibility concerns how easily a user interface, system's topology and hardware can be changed. In the case studies, the following approach was applied:

- The allocation model is separated from the development of the applications.
- User interfaces are separated from the application logic.
- Separate interfaces and configuration support were applied to the network and interface devices.
- Built-in configuration support for applications and the network.

Due to the separate allocation phase, applications and subsystems can be easily changed. The connection and configuration managers provided the support to allocate applications freely, and therefore, the system's topology does not affect the applications. The ORB provided the corresponding support in the FMS family. However, the message-based communication is preferred to RPC-like com-



munication, because the former provides looser coupling. The interface could also be customised by parameters in the majority of cases.

By using PAC agents, user interfaces could also be allocated freely. A PAC agent is flexible enough, if a system needs only an instance of each GUI, but the presentation components may have to be allocated separately in large distributed systems. Therefore, the allocation of multiple instances of the same GUI was demonstrated with the ECA executive that propagated the information to the bound user interfaces (Paper VI). However, the same could be implemented by small changes in the connection and configuration manager.

Dynamic binding and transparent communication assure that the topology of the network does not affect the applications, whereas changes in the network affect the services of the integration platform. However, the changes are tolerable and manageable, if layers with components are used. Changes in interface devices are limited to the surrogate components or their interfaces only.

### **5.5.3 Extendibility of distributed control systems**

The ability of a system to add applications and services defines its extendibility. For applications, the following approach was applied:

- *Syntactic conformance.* Applications have a uniform structure and a generic interface component that connects them to the integration platform.
- *Independence.* Applications are location-independent. It is the responsibility of the integration framework to manage locations and bindings between applications.
- *Semantic conformance.* Applications are configured through the integration framework that stores and manages the definitions of semantic interoperability.

If applications fulfil these three requirements, a system is extendable as regards applications. Although the extendibility of services is much more complicated, the following approach was tried and tested in the case studies:

- *Placeholders* were reserved for the architectural components in the structural and execution architecture, i.e., a slot for the variants of a component in the structure and time-thread.
- An *externalised binding mechanism* was provided for the behavioural variants, i.e., a place to add and change the connections of components.
- *Alternative behavioural patterns* were stored as the ECA rules into the database of the integration platform, i.e., configuration knowledge as rules.
- The *associations* of fine-grained components were changed on-line by the configuration management tool.

Although this approach provided the ability to add a service by connecting it to the other services through the ECA executive and a set of rules, it does not provide the on-line configuration for a service but some features of a service. The approach also requires that the new service is syntactically and semantically interoperable with the existing ones. Therefore, the use of commercial software as a service presumes that its properties are defined, managed and configured in the same way as the other services’.

## **5.6 Suitability for the control systems domain**

### **5.6.1 Operability and simplicity**

PAC agents, applied to the repayment systems and material transferring systems, supported mapping the variability to the responsibility-driven logical subsystems. Layers provided the means of bringing together the components at the same control level. In this way, the PAC agents defined the conceptual architecture, whereas the layers formed the aggregation level of the architecture. Dynamic architecture concerned only the agents at the intermediate level. The architecture was workable in both cases, because there were only a few main variants and diversity could be targeted to the logical subsystems directly.

In the FMS family, the rule-based system architecture was combined with the client-server architecture, because functionality was more natural to define as

services than agents. The reason was that each service had several relationships to other ones, whereas an agent is an autonomous unit. However, the PAC pattern might also be applied to the FMS family, but the behavioural pattern of an agent should be defined by a set of ECA rules.

### **5.6.2 Adaptation of COTS and legacy software**

The interfaces of COTS were mostly unsuitable for our experiments. The COTS and off-the-shelf software, used in the repayment systems, had to be tailored. The LON-dependent software had to be customised regarding the interfaces of the operating system and hardware. The functionality of the software components were customised as regards the interrupt services, implemented by the supplier. Although tailoring was needed when the version of the operating system changed, there was no change in the API. On the other hand, for the serial communication software that was an off-the-shelf component, only a few configurations were made easily by changing the parameters. In the last case study, COTS was offered with a new interface that at the end of the project was still under development. The COTS itself was stable, but the supplier was not reliable.

The transaction service, needed for the manufacturing systems, was not available at first. Later, its high price and supplier-dependence should have restricted the selection of other COTS and made it a critical component in the systems. In the other case, the LON network configuration tool was a critical component. Systems had to be configured with an expensive tool that could not be used without expertise. According to our experiences, the critical factors in the use of COTS are (cf. Voas 1998):

- The provided properties of a component compared to its requirements.
- The quality of a component and the reliability of the supplier.
- The impact of a component on the product family.

The liability of the supplier and the impact of a component are more important when a component is intended to be used in a product family, not only in a system.

### **5.6.3 Heterogeneous methods and tools**

In the analysis phase, we used technology independent methods and the Visio drawing tool. The PFM, based on FODA, was suitable for small systems, but its expressive ability is limited. Therefore, the definition and management of the services of large distributed systems need more support for design and maintenance. The same concerns the QFD method that was applied to validating a small system and commercial components. However, these methods do not need the skills of special technology and therefore, they are appropriate for stakeholders' team-working.

Time threads and scenarios, on the other hand, proved to be useful in all cases. Although they are descriptions of object-oriented methods and used mainly by domain and software engineers, they are easy to understand, for example, by mechanical engineers and marketing staff, whereas the IDL descriptions can be used only among software engineers.

### **5.6.4 Impacts on the development process**

According to our experiences, both the product family and the development process have to be mature enough: product variants have to be identified and defined and the change management has to be guided. In the case of not being able to identify all product variants straight away, the scope of a component framework should be restricted to the application framework or the integration platform. In this way, we performed the case of the machine control systems and the FMS family.

To be successful, the modelling of product features has to be the responsibility of the domain experts. The developers of user interfaces are responsible for describing the features with the concept models of different roles and skills. Sub-contractors are responsible for defining the functional, safety and performance features of their components. The same requirements have to be set for the COTS suppliers, in addition to their reliability. However, the domain experts and marketing staff validate the product features.

The parts of the component framework were developed in several projects, in which ten software engineers worked for fifty man-months altogether. Each

project included familiarisation with the application domain and products. The technology that had been used was changed entirely three times. However, if the product family and the development process are mature enough, it roughly takes one to three man years to develop a component framework. The scale is due to the size of the systems and the amount of product variants and features.

The development and use of the component framework presume that the product managers and domain experts work together with the developers of the component framework. The product-family tier is shared by the domain analysts, product and marketing managers and its development requires all their expertise. Application developers, system integrators and maintenance staff share the integration platform, and their expertise with the experienced technology experts of distributed systems are needed for its development. The expertise of the application domain is important in the development and use of the subsystem tier. However, application developers can be much less experienced than the domain analysts that needs the knowledge of the products, technologies, and development process. Their responsibility is to define the principles, rules and guidelines that the application developers utilise in a production line. The development of reuse assets and systems are separate development processes, both of which are made incrementally and systematically, and therefore, an organisational change may have less influence on the quality of software.

## 6. Related research

The topic of this thesis concerns software engineering, but there are other research areas that have also dealt with it, for example computer science, artificial intelligence, and control engineering. We review different reuse approaches and the aspects related to the development of a component framework. CBSE focuses on product features, software architectures, and components. Dynamic configuration is near to architecture description languages and an enabling technology for the run-time configuration of component-based software. Concurrent software engineering focuses on processes and environments that support the management of reuse assets. We compare some environments and frameworks that support component-based software development to our component framework.

### 6.1 Approaches to the component-based software development

There exist various approaches to component-based software development in the literature. The following three main approaches, which have matured nearly all together, are prevailing: a feature-oriented approach, an architecture-oriented approach, and a language-oriented approach.

The idea of program families originates from the stepwise refinement development method and information hiding modules (Parnas 1976). Goullon et al. (1978) applied the principle of data abstraction to types and models for dynamic restructuring of module's implementation. Prieto-Diaz (1987) made the pioneering work of domain analysis by defining the faceted classification technique for finding components from code. Furthermore, module interconnection languages (MIL), the predecessors of IDLs, were the first step towards architectural languages that have directed the interest of a large research community back to product-family architectures (Prieto-Diaz & Neighbours 1986; Shaw & Garlan 1996; Bass et al. 1998b). However, the target of reuse has changed from code to product features and product-family architectures.

### 6.1.1 Feature-oriented software reuse

In the 1990s, Feature Oriented Domain Analysis (FODA) has become the most popular domain analysis method (Kang et al. 1990; Dionisi Vici et al. 1998; Griss et al. 1998). The method made a significant contribution to *model-driven analysis*, which uses several complementary views of a domain to convey complete information. FODA consists of two activities: context analysis and domain modelling. The context model describes the environments in which the applications will be used. Domain modelling produces entity-relationship models that depict developers' understanding of the domain entities. A features model firstly included only the *end-user's perspective*, but it has been enhanced with the *viewpoint of developers*. Data flows and finite-state machines illustrate the functional requirements of the applications. The strength of FODA is in its product-orientation and technology-independence. We enhanced and applied FODA for describing mandatory, optional and conditional features of product-families (Papers II, Chapter 4).

The Synthesis method defines the domain process that creates and manages *reuse-assets*, and the application engineering process that utilises the assets (Campbell 1992; Mannion et al. 1999). Contrary to FODA, Synthesis is a process-oriented method consisting of three activities, that produce a domain plan, definitions, and specifications. The method provides a systematic way to produce and document the scope, glossary, assumptions, and maturity of the domain. The main contribution of the method is that it considers *business and market needs* and *technical feasibility* in addition to the existing products in the domain. The practical feature modelling method (PFM), depicted in Paper II, combines the component-development process and enhanced FODA, later called PFM, together due to the need to automate feature-component-mapping.

Joint Object Oriented Domain Analysis, JODA (Holibaugh 1993), is based on the Coad/Yourdon Object-Oriented Analysis method (OOA) (Coad & Yourdon 1990). *Business and methodology planing* are the preceding phases of domain and application engineering processes. Domain analysis is a part of the domain engineering process that has three phases: prepare a domain, define a domain and model a domain. The preparing phase corresponds to the definition phase of Synthesis. Domain definitions that are refined during the domain modelling produces a subject diagram, whole-part diagrams, domain services, domain de-

dependencies, and a domain glossary. Software reuse is based on object classes and features are described by specialisation-hierarchies, denoting the *parts that are parameterised, added or changed*. JODA is a technology-oriented method that focuses on *reusable requirements*. Domain services that are visible to subsystems are listed as well as the dependencies of services that must be available from subsystems for the domain to meet its requirements.

The main contribution of JODA is the separation of domain engineering from application engineering and their interaction. JODA also emphasises business and methodology planning as a starting-point of domain and application engineering, as well as the reuse-asset library as a part of domain engineering. The method is a comprehensive, “pure” DA method and can be applied straightforward, if the OOA method is used. However, more often a method has to be adapted to the needs of an organisation, as OMT in Paper I.

The Organisation Domain Modelling method (ODM) emphasises organisational and non-technical issues, but builds directly on insights of methods such as the faceted classification and FODA (Simos 1995). ODM is a technology-independent process-oriented method that assumes that there is a set of legacy systems, which are reasonably stable. Therefore, the method is suitable for organisations that already have the well-defined software development processes.

ODM has three activities: domain planning, domain modelling, and component engineering. Business objectives and *stakeholders’ needs* are considered in a domain plan that focuses on the current and *future exemplars* and *other domains*. The domain model contains a lexicon, a set of concept models, and a set of features models, which reflect different views of the domain. A concept model defines semantic relationships between domain terms that are defined in the lexicon. A feature is an attribute of a concept that characterises the concept by defining it or distinguishing it from other concepts or instances of the concept. Features are *clustered within a concept* or between different concepts. Clusters reduce the number of combinations and guide the component development.

To sharpen the differences of features and concepts, Simos and Anthony (1998) applied the Sigma formalism in the model web clarifying the problems in the semantics of concepts and features. In the model web, certain models have the *status of concept models* because they are *central to the domain focus* estab-



lished by the stakeholders' interests. Other models, due to their form and correlation to the model-world play the role of *features models* within the web. In the FMS family, roles, skills and work played the attributes that gave the status of concept models. However, Simos and Anthony have *formalised* that which we attempted to describe informally.

The contribution of ODM with Sigma is its strength to describe features of large distributed systems with several stakeholders and "hot-spots" that are the second-level features, i.e. concept models, in the domain model. Formalism makes it possible to construct business level architectural components from the semantic models that non-programmers can develop and evolve to compose and customise applications. Digre (1998) has presented a concrete example with a component definition language and a business object component architecture. Our code-from-features-approach, presented in Paper II, proved to be proper for small and centralised systems, but not for distributed systems. The problem is the management of product features. Therefore, formalism and a corresponding tool are essential for the product features of large distributed systems.

The Reuse Driven Software Engineering Business (RSEB) is founded on Jacobson's use-case driven OO method (Jacobson et al. 1997). The method is organisation-oriented and defines a domain as a *business area*, in which a suite of applications realise one or more business processes (Mannion et al. 1999). RSEB has four processes. Object-oriented business engineering reengineers the target business processes that benefit from software reuse. *Use-case driven* object-oriented software engineering consists of software family engineering to engineer the application *family architecture*, component system engineering to develop components and application systems engineering to develop new applications from the application family architecture and components.

The contributions of RSEB are the mapping between business and SWE processes, as well as incremental development and variation points defined in behavioural patterns. However, we prefer scenarios to use-cases, because they give stricter understanding of the behaviour of the systems (Papers III and VI).

FODAcom (Dionisi Vici et al. 1998) is extended FODA that has been applied within a telecommunication business unit. The method uses three models in requirement analysis: an actor diagram as a context model, use-cases, and a fea-

tures model. Parameterisation and extension points are described in the use-case model, derived from RSEB. The extensions aimed at encouraging the reuse of requirements specifications and integration of definition processes within the domain projects. Griss et al. (1998) refined FODAcom further by integrating features to RSEB using use-cases and extended features trees. Actors and use-cases are used to express the commonalities and differences, but on the contrary to FODAcom, variation points are described in a features model with OR, XOR and required relationships. The latter requires an expanded view of the features model. Justification of these two models is that a use-case model is intended for system engineering and communication with end-users. The features model, on the other hand, is used in domain engineering for defining technical features. The features model is similar to our model, except for notation and required relationships, which are described in the same model in our approach. Furthermore, optional scenarios are also described in our approach.

Bergmans (1998) introduced a notation to describe concepts of a product-family architecture as a concept map. The notation bridges the gap between the informal nature of the domain analysis and the formalism required from an architecture description. Concepts are categorised into *aspect-of* relations, *is-a* relations, *semantic* relations, and *constraints*. An *aspect-of* relation defines the compositional structure of concepts, and an *is-a* relation represents a specialisation relation. A semantic relation represents certain domain-specific semantics, e.g., interaction or co-operation. The three categories of relations are interpreted through the variability of systems by defining the *allowed configurations* and deriving the *constraints* that can be positive, negative, and semantic. A positive constraint defines that a certain concept is valid only with another or other concepts. Negative constraints define combinations that are not allowed. Semantic constraints describe quality requirements, e.g. time, space and performance requirements. Relevant *quality requirements* tag the concepts and relations in the concept map. In PFM, positive and negative constraints are defined by required and exclusive relationships and semantic relationships by time threads. However, the main contribution of the notation is that it considers quality requirements, on the contrary to the others that are focusing on functional and behavioural features.

All of the above-reviewed methods have their strengths and weaknesses, but none of them alone provides comprehensive support for feature-oriented soft-

ware development. However, the trend of combining and formalising a DA method, for example ODM with Sigma formalism, could resolve the problems that product-families of distributed systems embody. An enhanced ODM strengthened with an incremental development tool, the business-view and variation points of RSEB and the semantic constraints of Bergsmans' approach could be the answer to the development of the semantic components of the product-family tier.

### **6.1.2 Architecture-oriented software reuse**

We categorise architecture-oriented software reuse to the five categories: architecture styles, architectural patterns, domain-specific architectures, product-family architectures, and reference architectures. The scope of reuse is different in each class. Styles are reused as principles when software architecture is developed and architectural patterns are selected. An architectural design pattern, for example a broker, is a macro architecture applied as a common reuse-asset in several business areas. A design pattern, for example a dispatcher, is a micro architecture, i.e., a common way to solve a focused problem. Domain architecture is a focused architecture in a well-known domain without categorising market segments that are the focus of product-family architectures. A reference architecture is an informal or formal architecture model, which is accepted and reused community-wide.

#### **Architecture styles**

A style is determined by a set of component types, a topological layout of the components, a set of semantic constraints and a set of connectors (Bass et al. 1998b). A style defines a class of architectures and is an abstraction for a set of architectures that meet it. Examining existing systems Shaw and Garlan have catalogued a set of architecture styles (Shaw 1995; Shaw & Garlan 1996). Bass et al. (1998b) gives a small catalogue by defining five main classes: independent components, data flow centred, data-centred, virtual machine, and call-and-return architectures (Table 13).

Table 13. Architecture styles (Bass et al. 1998b).

<i>Main class</i>	<i>Characteristics</i>
<i>Independent components</i>	Dominated by communication patterns among independent, usually concurrent processes, e.g. event systems and communicating processes.
<i>Data flow</i>	Dominated by motion of data through the system, e.g. pipes and filters, and batch sequential.
<i>Data centred</i>	Dominated by a complex central data store, manipulated by independent computations, e.g. repository and blackboard.
<i>Virtual machine</i>	Characterised by translation of one instruction set into another, e.g. interpreters and rule-based systems.
<i>Call and return</i>	Order of computation, usually with single thread of control, e.g. objects and call-based client-server, layered architecture.

A system is seldom built from a single style, but it is a combination of heterogeneous styles. For example, a PAC agent, which is used in the subsystem framework, is based on event-based independent components, interpreters, which are sub-classes of the virtual machine class, and layers, which represent the call-and-return style. The execution topology of layers is hierarchical, as well as interpreters. On the contrary, independent components are arbitrarily executed. An agent itself is an architectural pattern that represents these three styles. Furthermore, an intelligent agent also consists of a repository and filters, which represent data-centred and data-flow centred styles. Thus, styles are design rationale that is reused through architectures and architectural patterns.

### **Architectural patterns**

Utilisation of architectural styles and patterns are typical for domain-specific architectures, product-family architectures, and reference architectures. Patterns

are widely reused and verified solutions for their specific problems. Buschmann et al. (1996) groups patterns into three categories: architectural patterns, design patterns and idioms.

Architectural patterns express fundamental structural schema for software systems, which are applied for high-level system subdivisions, distribution, interaction, and adaptation (Buschmann et al. 1996). Layered architecture is a call-and-return style, when it defines an overall style to interact. When it is strictly described and commonly available, it is a pattern (cf. Bass et al. 1998b; Buschmann et al. 1996). For example, ROOM and ObjecTime encourage to use layers, and thus, a layer is a pattern supported by the method and the tool. In our component framework, the subsystem and integration tiers used layers mainly for subdivision, distribution, and adaptation and their intention is to support systems management and evolution (Papers IV to VI).

The Broker pattern is intended to structure distributed software systems with decoupled components that interact by remote service invocations. A broker is responsible for co-ordinating communications, as well as transmitting results and exceptions. CORBA ORB, used in the FMS family, is a practical example of a broker used in COTS (Paper VI). A generic communication component and the software bus are examples of a broker, implemented as OTS (Papers IV and V).

The Presentation-Abstraction-Control pattern defines a structure for interactive software systems that form a hierarchy of co-operating agents. Every agent is responsible for a specific aspect of the application's functionality. The PAC pattern was applied in the subsystem tier of our component framework and the intention is to guarantee the independence of applications (Paper VII).

The Reflection pattern is an example of patterns used in adaptable systems that have to be open for modifications and extensions. A reflector has two levels: a meta level provides a representation of structure and behaviour in terms of meta-objects that are utilised by the application logic at the base level. The base level is independent of the aspects that are subject to change, they are responsibilities of the meta level (cf. Welch & Stroud 1998; Robben et al. 1998). In our framework, the Reflection pattern was applied in the implementation of ECA rules: the meta level defines the structure and behaviour of a trigger as base classes and application logic is described as instances of the derived classes.

A design pattern describes a recurring structure of communicating components, which solves a general design problem in a particular context (Gamma et al. 1994). Design patterns are micro architectures and do not guarantee a good overall architecture. They do not lead to direct code reuse either, but they have to be implemented each time they are applied. The Bridge pattern is an example of design patterns, which has been reused for isolating a physical part from a logical part of an interface (Papers IV and VII).

Idioms are the lowest-level patterns. They describe how to implement particular aspects of components or relationships between them using the given language. Idioms have not been used deliberately in our experiments.

### **Domain-specific architectures**

Domain-Specific Software Architecture (DSSA) defines the process model of the domain engineering that targets reusable software architectures (Tracz et al. 1993). FODA is one of its inspirers, but DSSA is focusing on a solution domain, i.e., the solutions that meet the requirements, and the constraints that limit the number of ways to solve the problem. The approach, as depicted in Paper I, is useful in re-engineering, if commonality and variability of existing systems are examined. Hayes-Roth et al. (1995) have used DSSA in the development of a domain architecture that is based on agents, reusable components, and an application configuration tool. The approach is applied in the mobile office robots domain and has similarities with our approach. However, the viewpoints of distribution and product-family are missing.

Evolutionary Domain Life Cycle model (EDLC) is the next step toward product-family architectures (Gomaa 1995). EDLC consists of three major activities: domain modelling, target system generation and target system configuration. EDLC categorises requirements into four classes: kernel, optional, prerequisite and mutually exclusive requirements. They correspond to mandatory, optional, and conditional features of PFM, respectively. The method focuses on functional features from the end-users' viewpoint and provides two alternative approaches: the kernel-first approach and view integration approach. In the former case, the kernel is a generic domain architecture and systems are developed by extensions to the kernel. In the latter case, each system is considered a view of the domain, and the integration of these views creates the domain architecture. Domain evo-

lution is considered as a variation from the previous version of the domain model. The main contribution of EDLC was its feature-object mapping and evolutionary aspects. Mapping defines a feature with its relationships to exclusive, required, optional and specialised features.

### **Product-family architectures**

Domain architectures consider the similarity and variability of an architecture in a domain. Product-family architectures focus on reuse assets that include a base architecture and a set of common components that populate it. However, the results of performance analysis, testing, project planning, development tools and processes can also be reused in a product-family (Bass et al. 1998b; Clements & Northrop 1998). On the contrary to the domain architecture, which can be reused by a single person or a group of designers, a product-family architecture is a core competitiveness of an enterprise and has to be committed by all stakeholders.

As depicted in Figure 5, reuse assets include product features, product-family architecture, used COTS and OTS, test cases, methods and tools. However, to be successful, the development of these assets set preconditions to the organisation. Dikel et al. (1997) have discovered the organisational principles that are critical to the long-term success of a software architecture:

- Focus on simplification, minimisation, and clarification.
- Adapt the architecture to future customer needs, technology, competition, and business goals.
- Establish a consistent and pervasive architectural rhythm.
- Partner and broader relations with stakeholders.
- Maintain a clear architecture vision across the enterprise.
- Manage risks and opportunities actively.

The first two principles also deal with products, and they are applied in our approach by the separation of concerns, adaptability, and product features. The last

four principles are organisational issues that have to be adapted to the development processes of each organisation. Reuse-environments that will be presented in Chapter 6.3 glance partly at these topics from the viewpoint of development environments and frameworks.

Because product-family architectures are core competitiveness of enterprises, they are not commonly available, as reference architectures are.

### **Reference architectures**

Bass et al. (1998b) divide community-wide reusable architectures to informal and formal architectures. Informal architectures are reference architectures that are widely used but not formally specified. CASE tools and application generators are typical examples that utilise reference architecture without an exact specification about it. A formal reference architecture is the development of open systems (Bass et al. 1998b). Open systems have interface specifications that are fully defined, freely available, distributed in the form of standards, and controlled by a group of vendors and users. The ISO OSI communication model is traditionally used in the telecommunications area. The reference model of Open Distributed Processing is an architecture for heterogeneous distributed systems (ODP 1995). Its standardisation is a joint effort of ISO and ITU-T. OMG's CORBA architecture, another example, is widely used in the object-orientation community.

The OSACA model (OSACA 1996) and the CIM Framework (Doscher 1997) are reference architectures in the distributed control systems domain that are proposed for standardisation. The OSACA model is based on architecture objects and a system platform. Architectural objects (AO) are modules that have the only access to the platform through a standardised API. The architecture is layered: the platform consists of the message transport and application services systems. The communication object manager layer mediates between the platform and AOs. The approach is frame-based like our COMI interface (Paper VII), but the interface frame does not support run-time configuration. The other difference is that an AO is a single process, whereas we used a couple of processes.



The CIM Framework architecture is a component-based architecture that builds on OMG's specifications including the following layers: CORBA ORB, CORBA services, and CORBA facilities (Doscher 1997). The architecture defines mechanisms for component interoperability, substitutability, and extensibility. The CIM Framework Specification provides a reusable component design consistent with the rules stated in the architecture guide. The framework aims at manufacturing execution systems that are software systems between equipment control and enterprise planning systems. It is a business domain layer on the top of the layered CORBA architecture. However, it does not emphasise product features and variability, whereas the product-family tier of our component framework does.

### **6.1.3 Language-oriented software reuse**

Architectural description languages (ADLs) are a linguistic approach of architectural descriptions and software reuse. Shaw & Garlan (1996) elaborated six classes of properties that characterise what an ideal ADL should provide: composition, abstraction, reusability, configuration, heterogeneity, and analysis. Vestal (1993) had a few additional requirements: communication integrity, hierarchical refinement support, and ability to reason about causality and time. Clements (1996) applied FODA to refine the requirements as features that ADLs should have. The features are classified into three categories: system-oriented features, language-oriented features, and process-oriented features. The result of his survey defines the following set of minimal features for ADLs:

- An ADL has to support creation, refinement, and validation of architectures. It must embody rules about what constitutes a complete or consistent architecture.
- An ADL must provide the ability to represent most of the common architectural styles.
- An ADL must have the ability to provide views of the system that express architectural information, but at the same time suppress implementation or non-architectural information.

- If an ADL can express implementation-level information, then it must contain capabilities for matching more than one implementation to the architecture-level views of the system. That is, it must support specification of families of implementations that all satisfy a common architecture.
- An ADL must support either an analytical capability, based on architecture-level information, or a capability for quickly generating prototype implementations.

### **Academic ADLs**

UniCON and WRIGHT are ADLs that are developed by Carnegie Mellon University (Shaw & Garlan 1996). UniCON is intended to use in defining software architecture in terms of abstraction. WRIGHT specifications are based on the idea that interaction relationships among components should be specified as protocols that characterise the nature of the intended interaction. According to Clements (1996) WRIGHT has better ability to represent styles. Completeness and consistency of architecture specifications are also more comprehensive. Furthermore, WRIGHT's support for variability is also better than UniCON's. On the other hand, UniCON is more powerful in real-time issues and code generation.

Rapide is intended to support specification, analysis, and verification of system architectures composed of event processing components that have an ability to generate events independently (Luckham et al. 1993; Luckham et al. 1995; Luckham & Vera 1995). Asynchronous communication is modelled by connections that react to events generated by components. However, synchronous communication is also possible.

Causality between events is modelled by reactive behaviour of components, and the execution architecture, also called an interface connection architecture, is a poset (i.e. partial ordered event set) showing dependencies between events. The interface connection architecture is a set of interfaces, a set of connection rules, and a set of constrains. Thus, a poset captures the semantics of communication and the behaviour of a component is defined in its interface. Therefore, variability can be defined as alternative posets. An event pattern is an expression of a poset. Rapide has the same goal as scenarios, time-threads and ECA rules have

in our approach. Scenarios and time-threads define interactions with variability and timing constraints, which are defined by event patterns in Rapide. An ECA rule also has the same triple as a poset: actions, rules, and conditions.

Mode-based programming, semantic classification and semantic concepts (Tairvalsaari 1993; Meslati & Ghoul 1997; Digre 1998) have similarities with Rapide and the ECA concept. The first two try to narrow the gap between design and implementation, whereas semantic components represent business object components, i.e. the components of the product-family tier.

### **Configuration and interface languages**

Although Darwin is a configuration programming language (CPL), that is a subset of an ADL, it has much in common with ADLs in describing a system as a configuration of connected component instances (Magee & Kramer 1996; Bishop & Faria 1996). Darwin is a declarative language that divides the description of structure from that of computation and interaction and does not have connectors as ADLs have. Darwin has been used in the context of Conic and Regis systems, whereas Software Architect's Assistant is a visualisation tool for the design and construction of Regis distributed programs (Magee et al. 1989; Magee et al. 1994; Ng & Kramer 1995).

An interface definition language (IDL) is like a CPL without hierarchies of component types and the reuse of patterns. However, it is a separate language for system design. OMG's CORBA IDL and Microsoft's MIDL are examples of the commercially applied interface languages. The use of IDL encourages incremental and concurrent engineering acting as a common tool between design and implementation.

### **Used architectural descriptions**

We used the commercial ObjecTime tool that is based on the ROOM method in the development of the component framework. The ROOM ADL fulfils all above-mentioned criteria (cf. Bass et al. 1998b). The tool supports incremental development with simulation capabilities and all main architectural styles, which are defined in the previous chapter, can be used. Variability is supported by abstract actors, parameterisation and inheritance of actors, data classes and proto-

cols. An abstract actor reserves “a slot” for the component variants that are instantiated dynamically. Interfaces are described by ports that are directed and typed by a protocol. Parameterisation may be loaded by instantiation or special messages. Others, except abstract actors were used in the case study of machine control systems.

In ObjecTime, timing requirements can be expressed at the architecture-level, and implementation could be generated in RPL, C++, and C. Analytical reports are available from simulations, but more important is the ability to generate a target code, which can be analysed in a target environment. However, according to our experiences, a less formal method is also needed for sketching. OMT and the Visio tool were used for this purpose.

The CORBA IDL, which was used in the FMS family, seemed to be a too low-level language for describing a product-family architecture. The architecture could not be simulated before implementation without the additional test components. The Visio drawing tool with UML stencils was used for the definitions of components and their variations. However, there was a gap between the graphical representations and CORBA IDL descriptions, and therefore, an ADL with simulation capabilities and preferably with the ability to produce CORBA IDL is required for the architecture design of large distributed systems.

## **6.2 Mechanisms for dynamic configuration**

There are two approaches to support dynamic configuration. The first is intended to be used for updating product features. The other approach includes support software for extending and replacing existing applications and components. In order to be able to reconfigure, the software systems have to provide mechanisms and target objects, which are implemented with an ability to be replaceable. In this chapter, we compare our approach to those of others.

### **6.2.1 Changing product features**

The EDLC model (Gomaa 1995) proposed that the variants of an object type are stored in a generalisation-specialisation hierarchy, which supports the *is-a* relationship. Hyes-Roth et al. (1995) applied the EDLC model for adaptive intelli-

gent systems that have a two-level reference architecture (i.e. the cognitive and physical level of an agent) and a set of components for both levels. In addition to the planning, information base and behavioural parts, both levels have a meta-controller that generates executable behaviour. An application configuration tool that uses the tasks, methods and domains, i.e. the components of each level, is used to perform the variety of individual agents. The configuration is done at the development time. However, the meta-controller that acts at the run-time, performs the same semantics. The approach is similar to our ECA concept, except the meta-controller is intended to use for achieving a system's run-time adaptation to the changing world.

Gomaa & Farrukh (1997) enriched the EDLC model by mapping object types to component types according to the Darwin CPL. A scenario driven approach, like our approach, was used to map a feature to a *design fragment* as collaboration among the components, required to support a given feature. Conceptual fragments had been used earlier in TEDIUM (Blum 1996). Both approaches are based on generated target systems, which supports software reuse at the organisational level, but does not support the flexibility and evolution of a distributed system.

In our approach, the ECA executive acts as a meta-controller that uses the ECA rules, which are the semantic components, described as behavioural patterns. We prefer the term 'a semantic component' to a fragment because a component has its own value and users in the product-family tier instead of fragments that are pieces of a set of other entities.

### **6.2.2 Replacing applications and components**

Yasmin applied the architecture-oriented approach for dynamic configuration (Deri 1997). The architecture of Yasmin defines a droplet that has similarities with our PAC agents (Paper VII). The droplet is an architectural component with a well-defined interface that can be loaded at runtime and replaced. The responsibilities of the droplet manager and collaboration services are quite similar to the services of the co-ordination layer of the integration framework. Our approach does not cover the versioning of applications, but the semantic components of the product-family tier play the same roles as versions in Yasmin. The service manager and resource manger have similarities with the co-operation

layer of the integration tier. However, the resource manager that acts as a periodically activated garbage collector to purge resources no longer needed, is not appropriate in control systems due to the required predictability.

Yasmin, which was applied in the network management domain, used the frame-based approach to achieve runtime configurable applications. The provided services also have many similarities with our approach that encourage us to believe that the presented component framework could also be applied in domains other than distributed control systems. Yasmin was developed at nearly the same time as our PAC agents with COMI interfaces (Paper VII). Therefore, the developers of Yasmin, like us, highlighted the inability of CORBA implementations to provide support for the development of independent and interoperable applications that support runtime application evolution. Also nowadays, the situation of CORBA is the same.

Port-based objects (PBO) are dynamically configurable real-time components that are categorised to five classes (Stewart et al. 1997). Generic components are hardware and application independent, such as the mechanisms in our approach. A hardware-dependent component can only be executed when the hardware is a part of the system, such as the LON driver in our context. A hardware-dependent interface-component converts hardware-specific signals into hardware independent data that is used by generic components. The same technique is used in the class hierarchy of our adapter. Hardware-dependent computation-components are generic components that have been specialised by extended functionality of better performance. This means optional features and component variants. Application dependent components implement the specific details of an application. This corresponds to PAC agents.

A PBO is independent of the granularity of functionality and is implemented as a thread that is a finite state machine with four states: *not\_created*, *off*, *on* and *error*. The PBO approach provides a systematic way to develop configurable components with the following restrictions:

- The same operating system is used in each node.
- The operating system provides a template for concurrent processes (PBOs).

- The communication can be based on shared memory that is updated at pre-determined times only.
- No explicit synchronisation is needed among processes.

Stadel (1991) proposed a language-oriented approach for the dynamic configuration by introducing a dynamic link loader that the operating system is intended to provide. Maintenance commands are entered through the user interface of an online configuration manager. The loader saves the entry addresses of loaded classes and keeps book of all loaded code with a creation counter. Using a class that clones the state of an existing object to the new one, new objects replace the old ones. The approach provides dynamic loading for the objects of the same class, but does not support reconfiguration of components. Kniesel (1998) has enhanced the approach by component connections. However, the approach has a weakness: there should be a language-independent means to change both incoming and outgoing component connections.

## **6.3 Frameworks**

Although components and frameworks are different, they are co-operating technologies and a framework is a combination of components and patterns (Johnson 1997). Component frameworks that will be presented in the next three chapters are classified according to the tiers of our component framework. The domain-specific frameworks are corresponding to the subsystem tier or a part of it. Integration frameworks attempt to hide the complexity of the distribution, whereas product-oriented frameworks focus on product-features and business aspects.

### **6.3.1 Domain-specific frameworks**

Domain-specific frameworks are focussed frameworks that simplify the development of portable and efficient system infrastructure such as operating system (Stewart et al. 1997), communication (Schmidt & Fayad 1997; Cysewski et al. 1998) and user interfaces (Poutain 1995; Szyperski 1997). Schmidt & Fayad (1997) classify frameworks by the techniques used to extend them, which range along a continuum from white-box frameworks to black-box frameworks. White-box frameworks rely heavily on object-oriented language features, like

inheritance and dynamic binding. Black-box frameworks support extensibility by defining interfaces for components that can be plugged into the framework.

Losavio & Matteo (1997) developed a white-box framework that uses the PAC pattern as a basic architecture for the user-interface development. The BlackBox component framework, formerly Oberon/F (Poutain 1995) is a component framework extended by development tools, providing e.g., compilation and debugging capabilities (Szyperski 1997). The framework consists of a core set of layered modules and an open set of subsystems. Each subsystem is itself a set of layered modules. The BlackBox architecture has similarities with our subsystem tier, but the component framework focuses on visual components, provided by OLE, JavaBeans, and ActiveX controls. The BlackBox architecture is based on a number of design patterns, for example a hierarchical version of the model view controller (MVC) pattern, which is the predecessor of the PAC pattern. As a summary, the PAC pattern has proved that it is powerful both in the white-box and black box approach in the user interface domain.

The control-domain orientation is obvious in the multi-agent approach (Lejter & Dean 1996) that does not use the PAC pattern, but focuses on co-ordination protocols among agents. The presented solution for the negotiation-based co-ordination protocol that is based on a game theory, has some similarities with our approach. The protocol includes two components, the arbiter part and planner part of an agent. The arbiter is responsible for receiving messages from other agents, as the COMI interface in our approach. The planner part has a set of responsibilities: to accept and merge requests, to submit a composite plan, to accept the evaluated plan and finally, execute it. Our model (cf. Chapter 4, Figure 20) allocates these responsibilities to several components, for example the evaluation of the plan and its execution is separated and can be modified independently. However, the co-ordination protocol is implemented as object classes and each agent that communicates in a negotiation-based manner inherits this protocol interface. The same approach was used in our COMI interface for the run-time configuration of the applications (Paper VII).

### **6.3.2 Integration frameworks**

Expecting emerging industry middleware standards, like CORBA, DCOM, or Java RMI, to eliminate distributed software complexity is very risky (Schmidt &



Fayad 1998). Although low-level services, such as ORBs, are reaching maturity, the semantics of higher-level services, such as the CORBA services and CORBA facilities, are still vague, non-interoperable, and not commercially available (Paper VI).

Distributed programming environments commonly restrict programmers to one of inter-component interaction. Event-based and message-based integration is the most prevalent approach to loose integration, but the comparison of available solutions is impossible without a consistent model. Barrett et al. (1996) described an event-based integration (EBI) model that they applied for comparing three messaging implementations, among others the CORBA 2.0 specification (OMG 1995). The EBI framework defines a registrar and a router that includes message transforming functions, and delivery constraints, i.e. ordering, timing and atomicity properties. In the comparison, CORBA ORB without Event Service had many serious weaknesses:

- Multicast *messaging* is missing.
- *Message transforming* functions are unspecified.
- Router's *message delivery model* (polling/active) is not specified.
- *Delivery constraints* are at the most once and the best effort without priorities, timing or atomicity properties.

The formal specification of the Event Service is now available, but Messaging Service is still under the development. In our integration framework, the event-driven messaging service is responsible for the message transforming functions and delivery constraints. On the contrary, locating participants, polling, as well as point-to-point and multicast message sending are the responsibilities of the router in our integration framework. The aim is to isolate changing parts to the upper layer so that the message-transferring layer could be kept the same. However, only ordering and timing constraints were used, and missing atomicity properties caused problems in the FMS case study (Paper VI).

Crane (1996) proposed a framework for distributed interaction that supports several interaction policies, for example RPC, events, signals and asynchronous

notification. The framework provides location transparency, runtime binding, and the type and role compatibility of participants. The framework integrates the same properties as ours, except the compatibility of participants is not checked by our integration framework. Although the interaction framework provides a systematic approach for the co-operation layer, the co-ordination and message transferring layers are not supported. Pryce & Crane (1996) enhanced the interaction model by adding service endpoints for clients and servers through which a programmer can emulate alternative interaction policies. Our framework does not have this property, however, it might be useful in the integration phase.

Composite objects enhance interoperability between real-time and non-real-time systems that are connected by gateways with two timing levels (Polze & Sha 1998). CORBA stubs are used for communication without QoS guarantees and RT threads for real-time messaging. The approach has similarities with our approach that uses an adapter for different communication protocols (Paper VI). Although we do not recommend the use of CORBA in real-time systems, a composite-object might be used as a design pattern for adaptation. In our approach, the required services have to be provided by the integration framework and used by application developers. The router is responsible to order the messages in such a way that quality requirements can be achieved. The aim is to make a clear distinction between the responsibilities of the subsystem and integration tiers in order to simplify the application development. However, our experiments were not heterogeneous as regards timing requirements and RPC based and periodic messaging were not used in the same system.

A wrapper is another commonly used adaptation technique, applied for RTDBMS (Ginis et al. 1996). In our experiments, the ECA executive was the only component connected to the OODB, because there was no commercial adapter available and due to a limited schedule, it could not be constructed. A trader can adapt the changing interaction policies between frameworks by acting as a centralised changing point for evolution (Graw & Mester 1998). The approach may be useful, e.g., when control systems are connected to information systems that use different frameworks. Our component framework does not have this kind of support.

The integrated network component architecture (INCA) provides an integration framework for the business components of interactive systems (Ben-Shaul et al.

1998). Components are coarse-grained as medium-grained PAC agents are in our context, but no architectural pattern is used. Instead of a co-operation protocol, each component listens to and acts on events for which it is configured. Platform independence and location transparency have been provided on the top of Java RMI and CORBA ORB, and therefore, the architecture is a means of adapting heterogeneous component models of user interfaces. We did not need this kind of integration support in our experiments, but user interfaces are the most frequently updated software, and the used technique has tended to change during upgrades. However, the model is useful for isolating user interfaces from the application logic, as we did by the presentation components of PAC agents.

### **6.3.3 Product-oriented frameworks**

Software developers tend to migrate features to the integration framework in order to reduce customisation efforts. This results in over-features, i.e. the relevant features for a particular user or a product are a part of the standard platform (Codenie et al. 1997). Therefore, it is important to draw a distinction between product variations and the integration framework. Demeyer et al. (1997) proposed two means as a solution: to package components that they can be reused in as many contexts as possible and to design software architecture that is easily adapted to target requirements. The approach is not in contradiction with our approach. However, we underline that product features form semantic components that should be reused at the requirement analysis of a product family.

Bäumer et al. (1997) define organisational concepts and the layers of the framework according to them. The technical and desktop layers are common for the rest: the application layer for workplace contexts, the business section layer with specific classes of each business section and the domain layer for the business as a whole. The architecture is quite similar to the CORBA architecture and they are both convenient for information systems. However, the variations in workplace contexts, business sections, and business domain can be applied to the context of control systems domain as follows:

- Situation action contexts: a work cell, the role and skills of a user.
- Product variants for different control purposes.

- Product variants for different market segments.

The CIM framework (Doscher & Hodges 1997) is a vertical business domain on the top of the CORBA layers and the use of the framework presumes that the underlying layers are available. Owing to the missing commercial implementations, the framework can be applied after the other layers have been implemented. Therefore, the layers could not be developed incrementally and concurrently, as the tiers of our component framework can be. However, the development of our framework has also to be balanced, so that the features used by the product-family tier have to be supported by the subsystem and integration tiers.

Brugali et al. (1997) combine white-box and black box techniques by applying black boxes for features that are common for all systems and white-box components that have variations. The framework has a layered architecture and uses patterns to describe fine-grained, medium-grained, and large-grained components. The pattern *actions triggered by events* specifies the way to model fine-grained concurrency. Actions are sequential objects and correspond to the actions of an ECA rule. The pattern *services waiting for* and the Thread class are used to specify a medium-grained component that carries out a service. The pattern corresponds with a trigger and its set of rules. Large-grained components follow the pattern *client/server/service*. The Service class extends the Thread class corresponding to a process with its internal data, state and event generation. A service belongs to a server that offers concrete public methods for asynchronous, synchronous and deferred-synchronous service requests. In our context, CORBA components, the ORB and the ECA executive correspond to the services. Binding a service to a server should be able to be made in the integration phase, not in the design phase and therefore, the presented framework supports evolution of the framework but not the incremental development of systems.

The last two exemplars are SWE environments that are used to develop distributed systems by a product-oriented reuse method. In GenSIF (Rossak et al. 1997), a domain model, a domain architecture and a domain infrastructure are the means of the development of an integrated domain specific environment. The GenSIF's reference architecture has two parts: the first captures design rationale; the second introduces the architectural elements. The design rationale

includes concepts, rules, principles, and guidelines. Concepts include required system properties and the elements available to build the system that the architecture must support. Rules specify constraints on concepts addressing structural aspects of concepts, for example the types of connectors between components and single inheritance classification hierarchy. Principles describe what is considered to be “good practice” and correspond to software design patterns in our approach. Guidelines try to anticipate design problems and help to solve them. We did not give guidelines, but the use of principles attempts to avoid problems.

Architectural elements are building-blocks, components, contracts and scenarios. A building block is an autonomous structural primitive of the architecture and corresponds to a top-level agent in our context. A component is a cluster of building blocks and/or components with defined external behaviour and corresponds to a bottom-level and medium-level agent in our architecture. A contract is a set of requirements and constraints on one or more components prescribing a collective behaviour of participating interfaces. Contracts are behavioural models of the layers in our approach. A scenario is a dynamic configuration of architecture components describing a complete design pattern or a thread of execution in a complex system. Scenarios and time-threads are also used in our approach.

Feature-Oriented Reuse Method (FORM), extended from FODA, attempts to capture commonality and variability of applications (Kang 1998). The case tool (FORM 1999) is based on the following principles of the FORM method:

- Features are used to parameterise architectures and components.
- The layered architectural framework provides subsystems, processes, and modules.
- Components are separated from the component connection mechanisms.
- Design components are synthesised by selecting features.

The last principle means code generation and is not relevant in our approach. Although the first three principles have similarities, they are applied in a different way. Features parameterise architectures and components, but clustered fea-

tures also define the properties of components and services in our approach. The features model also defines the relationships between features, i.e. between components that implement them.

We used the layered agent, client-server and rule-based system architectures and the interface layer separates components from their connection mechanisms. The strength of the FORM framework is that the implementation technique can be selected freely after the architecture and component design.

## 6.4 Summary

Several reuse approaches were reviewed in order to discover the weaknesses and strengths of existing methods for the development of a component framework. Semantic components of the product-family tier requires such a technology-independent feature-oriented domain analysis method that supports organisational and business issues, but at the same time, it has to be product-oriented and formalised method with the support for product variations. An enhanced ODM with some properties of RSEB can be the solution. However, the method should be supported by appropriate tools.

The component framework has to be based on architectural styles and patterns. The ROOM method provides the properties required for the ADL, but its support for behavioural patterns and variations is not enough. The integration of UML with the ROOM method may be the answer to the problem, but it does not provide a solution for the reconfiguration of features and applications. The fusion of Darwin and EDLC seems a promising approach for this problem. However, the aim should have to be for a product-family architecture that is supported by the services of the integration framework in a language-independent way. The similarities of our approach with Yasmin encourage us to believe that the reconfiguration support is much more widely needed than applied and the approach is useful for the software reuse. On the contrary, the ECA concept could prove its significance once the DA method with tools are provided for generating rules automatically from the descriptions of semantic components. Thus, there is a need for the integration of the generative and component-based software development.

Several frameworks support parts of the application and integration tiers. However, none of them could be applied straightforwardly. The technology-dependent parts of the integration tier could be COTS, if its configuration is supported. The subsystem framework can only be reused in the control systems domain and the product-family tier is an organisation-dependent solution, except the generic mechanisms.

## 7. Introduction to the papers

The papers included in this thesis were written over a period of five years, from 1995 to 1999. They demonstrate how ideas have progressed over time and how principles used in the later papers were first outlined in the early papers. The topics of the papers also illustrate the views of architectural components, CBSE, CSE, and SCM, and the various issues presented in papers are summarised from these views (Table 14).

*Table 14. Summary of the issues presented in the papers.*

<i>Paper</i> → <i>Issue</i> ↓	<b>I, II, III</b>	<b>IV, V, VI</b>	<b>VII</b>
<b>Domain</b>	Embedded machine control systems	Machine and process control, and manufacturing systems	Manufacturing systems
<b>Dominating tier</b>	Product family and subsystem tiers	Product family and integration tiers	Integration tier
<b>Modelling techniques</b>	RTSA, OMT, ROOM, PFM, scenarios, time-threads	PFM, OMT, UML, scenarios, MSC	OMT, MSC
<b>Architecture</b>	Layers with components	Client-server, layers, agents and rule-based systems	Layers and agents
<b>Implementation technology</b>	C, C++	C/Photon, C++, Orbix, MFC	C++, C/Photon
<b>Main contribution</b>	Methods of product-oriented software development	Distribution platforms for product families	On-line configuration support for applications



The papers discuss various aspects of component frameworks. Different views of the papers are illustrated in the context of the component framework presented in Chapter 3 of this thesis (Table 15).

*Table 15. The relationships of the papers to the component framework.*

<i>Elements →</i> <i>Tier ↓</i>	<b>Product Features</b>	<b>Architectures</b>	<b>Components</b>
<b>Subsystem</b>	Papers I & III	Papers I, IV, VII	Papers I, III, V
<b>Integration</b>	Papers V–VI	Papers IV–VII	Papers IV–VII
<b>Product Family</b>	Papers II, IV	Papers V, VI	Papers II, VI

## **7.1 Component-based software engineering**

The following three papers deal with the software engineering methods used in CBSE and discussed briefly in Chapter 4. Domain analysis is highlighted by re-engineering and feature-oriented approaches. The last paper illustrates software architectures and components as reusable assets.

### **7.1.1 Paper I: Domain analysis by re-engineering application software**

Paper I describes a practical approach for re-engineering machine control software. Shifting RTSA models to object-oriented designs is the means of increasing software reuse by defining software architecture and components for a set of machine control systems. The focus of this paper is on a traceable development process that

- defines the scope and specific concepts of the application domain,
- provides principles for re-engineering software architectures, and
- adopts component-oriented features of the structural modelling techniques into object-oriented modelling.

Section 1 presents state-of-the-practice in modelling embedded systems. RTSA methods and tools were preferred to object-oriented methods at that time. Section 2 highlights the differences between object-oriented and RTSA and problems encountered in modelling real-time embedded systems. Section 3 describes the used re-engineering process for shifting from RTSA to object-orientation.

Section 4 states criteria for identifying the software suitable for re-engineering. Time-threads were used for illustrating important behavioural patterns and timing requirements. Section 5 gives guidelines for identifying object candidates. The classification and generalisation of object classes describe the subsystem architecture for a set of actuator control systems, as depicted in Figure 3 in Section 6. The steps for designing object classes are defined in Section 7. White classes are modifiable classes, but black classes are designed as implementation classes and adapted with the configuration parameters. Section 8 summaries the experiences of reducing the risks of the re-engineering process by the combinative and adaptive approach instead of pure object-orientation.

### **7.1.2 Paper II: Product features and component-based software**

Paper II depicts a feature modelling method and a software development environment that integrates product features to object-based models of components and produces the target code and configuration data for product variants. The contribution of this paper is to

- describe a systematic component-based software development process,
- define a modelling method for structuring product features, and
- introduce basic principles for feature-component mapping.

Sections 1 and 2 introduce common modelling techniques used in the software development of embedded systems and define the five phases of the reuse-oriented development process; domain analysis, architecture analysis and design, feature modelling, component design and software production. Section 3 represents the background and extensions of the practical feature modelling method. Section 4 gives the overview of the prototyping environment and defines two

means of mapping features to components; through the component variants and parameters.

### **7.1.3 Paper III: Development of software components**

Paper III presents prototyping as a means of early-verification in component-based software development. It presents an incremental prototyping method to reuse design knowledge of several engineering domains and put it together into the product. The aim of the paper is to

- emphasise software reuse among different engineering disciplines,
- illustrate a software component as a fusion of design models, used for different purposes, and
- use prototyping as a means of validating component candidates before implementing them into the target code.

The paper gives the motivation for integrated prototyping, and describes the heterogeneity of design methods and tools used in the development of multi-technology products. The phases of the approach describe the modelling techniques used for different purposes. Time-thread diagrams define functional components and timing requirements, and a graphical programming language is used for prototyping and integration tests with a target system before developing the embedded product by object-oriented methods and languages.

## **7.2 Software engineering of distributed systems**

The next four papers depict integration platforms implemented for different product families that also represent the different real-time levels; hard real-time, soft-real time and mission critical systems.

### **7.2.1 Paper IV: An integration platform of real-time distributed systems**

Paper IV provides a communication bus as a channel of negotiation and transmission of system data between autonomous subsystems of distributed machine control systems. The main points in Paper IV are:

- to describe a method for achieving software adaptability for product families,
- to define a software architecture for flexible distributed systems, and
- to depict a mechanism for run-time adaptability of real-time systems.

The first two sections introduce the needs of software adaptability and describe the means how to achieve it. Classification of messages and interfaces, and communication rules are the discovered means of software adaptability in the design phase. Mode-based configuration, configurable communication mechanisms, and configuration management are presented as a means of run-time adaptability.

Section 3 describes the architecture and a C++ wrapper of the demonstration environment. Layers are used for isolating system and node-level operability, while the communication layer acts as a generic configurable distribution platform. A dispatcher pattern is used for the event-driven and periodic communication mechanisms, which are configured by parameters at the run-time.

Analysis, in Section 4, highlights the drawbacks that adaptability causes in execution times, an important aspect in hard real-time systems but meaningless in other embedded systems.

### **7.2.2 Paper V: An integration platform of a product family**

Paper V presents a software bus as a transparent distributed platform of an automatic repayment systems family. Product features are defined by analysing the application domain and existing products. The architecture is divided into stable platform services and customisable application components that act as

reactive agents of subsystems. Domain knowledge is stored as configuration data into the software platform and used through a tool for configuring a network and applications.

Section 1 motivates to evolutionary software development and introduces the needs of extendible systems. Characteristics of the control system domain and background are highlighted in Section 2. The following criteria for the software platform are laid down:

- The same software platform is used for all product variants.
- Hardware change may not affect on the location-independent applications.
- It should be possible to add new applications to the existing system with a minimum of work.

Section 3 presents the integration framework and mechanisms for flexible control systems. A lightweight communication interface, suitable for embedded systems, connects reactive agents to the services provided by the integration framework. Layers isolate application-dependent software from mechanisms, which provide message-routing and network connections. Commercial software and off-the-shelf software components are used in the software bus that coordinates execution of subsystems through a centralised main-memory database.

Section 4 summarises experiences and sheds light on the restrictions on the approach, i.e., the maturity of the software engineering process, known product variants, and the routing mechanism applied for a network.

### **7.2.3 Paper VI: An ORB as an integration platform**

Paper VI concentrates on integration techniques of heterogeneous distributed systems, a flexible manufacturing systems family as an exemplar. The main contribution of this paper is

- to integrate commercial components as an OODB and a CORBA compliant ORB with off-the-shelf components,

- to provide an integration framework for legacy systems and new object-oriented applications, and
- to add event-driven communication to the RPC of the broker.

The first two sections introduce the characteristics of integration-oriented development and distributed CIM systems, as well as background knowledge for CORBA. Section 3 presents the basic components of the distribution platform, i.e., scalable application services, an adapter for legacy systems, and the ECA mechanism for event-driven behaviour.

Section 4 summarises the experiences emphasising the advantages and shortcomings of the CORBA architecture and its implementations. Utilisation of the CORBA and the ECA instead of black- and white-box components is presented as an alternative means of supporting horizontal and vertical extendibility. Performance and memory requirements are measured and evaluated to be reasonable for mission critical systems, but there is an evident need of commercial CORBA services.

#### **7.2.4 Paper VII: Dynamic configuration of architectural components**

Paper VII represents a framework for the run-time configuration of distributed software components that are medium-grained architectural components, applied for logical systems and agents with generic communication interfaces. The contributions of this paper are:

- run-time configuration as a part of software architecture design,
- a framework that supports the evolution of a product family by generic interfaces and configuration management software, and
- a solution for location-independent user interface software.

The first two sections highlight the key-factors of adaptive systems from the viewpoint of software configuration:

- How can software components be added into existing systems?
- How can different kinds of components be used in the same framework?
- How can components be changed at the run-time?

Section 3 presents a layered architecture as a basic architecture and distributed PAC agents as an application architecture. PAC agents provide the basis of stable and interchangeable software. A configurable module interface (COMI) is an implementation-independent solution for interoperability. The run-time configuration management employs a COMI for achieving software flexibility and extendibility.

Section 4 presents the run-time configuration scenario and its performance tests. Although performance does not satisfy real-time requirements, the approach is suitable for mission critical and soft real-time systems.

## 8. Conclusions and further research

In this thesis, we have studied the development of a component framework of a distributed control systems family. According to the architecture-oriented software reuse, we have presented the architecture of the component framework that supports incremental development and maintenance of the distributed control systems, constructed by the concurrently developed application components.

The development of a component framework has been studied from the viewpoints of CBSE, SCM and CSE. CBSE provides the design rationale that have been adopted and applied to the dynamic configuration and concurrent application development. The viewpoints of the component framework have been presented as

- the problems, from which we derived the requirements of the component framework.
- the tiers and elements of the component framework and its relationships to the different viewpoints.
- the method, how the tiers and elements of the component framework can be developed.
- the evaluation results of the constructed prototypes and the SWE approaches related to the development and maintenance of component-based software.

The main results and prospects for further research are discussed in the following chapters.

### 8.1 Answers to the research problems

The answers to the research problems, set in Chapter 1.4.1, are formulated as follows:



Q1. What requirements does a control systems family set down for a component framework?

The requirements of the component framework have been obtained through studying the problems that appear in the practical component-based software development of distributed control systems. Although the studied domain areas, machine, process and manufacturing control systems have significant differences, we discovered the following similarities:

- The problem domains were quite stable and therefore, the component framework could be used at the product-family level.
- Each systems family had a multitude of technical features and several variations that had to be defined and implemented by the component framework.
- Application developers might be subcontractors that had to use the same component framework as the systems integrators.
- The organisational culture favoured the use of third-party components.
- The component framework had to support the roles and responsibilities of different stakeholders.
- The component framework could have to be used in the concurrent software development.
- The size of product variants varied, and therefore, extendibility and scalability were required.

Briefly, the component framework has to support simultaneously the application developers, the system integrators and the suppliers of distributed control systems.

Q2. What kind of a component framework supports the evolution of a distributed control systems family?

We applied the adaptive design for the problem solving by focusing on the changes of the control systems families. Changes in the markets, used technology and the application domain are the main reasons for the evolution of the component framework, and therefore, the product family, integration and subsystem tiers have their own scope and techniques for achieving adaptability.

At first, the product-family tier clusters the product features to the semantic components that are implemented by the properties of the architectural components within the subsystem tier and the services of the integration tier and their interaction mechanisms. Fine-grained features are mapped to the properties of the building-blocks, primary components and their configuration parameters. The subsystems use standard interfaces with strictly defined policies, demanded by the integration tier that hides the changes in the used technology providing an application-specific layer for the subsystems integration. The architecture of the product-family tier balances the architectures of the subsystem and integration tier providing a systematic way to describe, manage and change product features. The ability to configure applications dynamically is a part of the support software in the product-family tier.

The support for systems' evolution can be scaled in each tier. The reconfiguration support for applications, integration platform and product features can be implemented in a way that is suitable for the needs and used technology. The larger the systems family is, the more comprehensive support is needed for product variations.

Q3. How should the component framework of a product family have to be developed?

The development of the component framework embodies the following reuse assets: product features, product-family architecture, components, and mechanisms and policies for the use of components. Domain engineering produces the features model, scenarios and time-threads that give the overall understanding of the structural and behavioural properties of the systems and their execution con-

straints. We proposed the QFD technique for the qualification of product features and COTS components. The used techniques are technology-independent due to heterogeneous design methods and tools used in the development of control systems.

The product-family architecture combines the layered, presentation-abstraction-control, client-server and rule-based architectural styles. The layered architecture is used for:

- providing the portability of the framework,
- isolating the frequently changing parts from the more stable parts, and
- separating technology-dependent parts from application-dependent parts.

Layers are used in the subsystem and integration tiers. PAC agents have been applied for applications with the ability of the run-time configuration. The client-server architecture is obeyed between the applications and the integration tier that provides transparent communication, co-ordination, allocation and configuration services for the systems. The timing requirements define which communication service is selected and how the integration tier has to be allocated. The rule-based system architecture provides inherent software flexibility for a product family.

The amount of required adaptability depends on the used COTS, heterogeneity of communication media, operating systems, and environmental devices. Design patterns and standard interfaces for communication and configuration have been applied to achieve adaptability. An adapter, a wrapper and a filter are used for the adaptation of legacy software to the integration tier.

Due to the incremental integration and long installation phase, control systems need the configuration mechanisms that are integrated parts of the integration tier and are used for customising application and product features. In small systems families, the configuration support is a simple user-interface with the ability to change product features by data-manipulation. Reconfiguration support for fine-grained features is only needed in the complex systems that have a long installation and introduction phase.

Q4. How does a component framework of a product family change the component-based software development?

The presented component framework sets some preconditions and restrictions, but it also gives freedom to select the methods and tools applied within the tiers that are often used by different organisations. However, the systematic way to develop and manage reuse assets set the following preconditions:

- The definition and management of product features require a mature product family and development process. The whole organisation has to be committed to the systematic way to produce the reuse assets.
- The suppliers of COTS have the responsibility to describe all features of their components. The purchasing of COTS has to be guided by required features including reuse requirements.
- Stakeholders share the product-family architecture that has to be kept as stable as possible within each tier.
- Developers have to be familiar with the architectural styles and patterns.
- The interfaces and policies of components have to be defined thoroughly.
- The applications have to meet the specification of the interface layer.
- The configuration of a system is a top-down activity that can be automated.

The above-mentioned conditions require that the stakeholders of the product-family -- marketing staff, application developers, integrators, and maintenance staff -- work together more closely than they do nowadays. Owing to COTS and distributed application development the component framework needs an organisational infrastructure that provides the services needed for sharing the knowledge of reuse assets.

## 8.2 Topics of further research

We developed the integration platforms for machine and small process control systems and used the commercial ORB as a basic component for the manufacturing systems family. Our continuous work focuses on a generic integration tier of embedded systems. This provides that the features of the services and components in the integration tier have to be defined and managed in a way that the tier could be configured for the needs of the applications. Several communication services, based on multi-protocol stacks, will be needed, as well the quality and security services, especially for Internet applications.

The other topic, which is further studied, is the formalisation and enhancement of the feature modelling method and the validation of the product family architecture. A formalised feature modelling method with variation support is the precondition that the semantic components can be defined and the configuration rules can be produced automatically. This presumes that the generative approach of the features modelling has to be integrated with the services of the integration tier and the components of the subsystem tier. The feature modelling method also needs special tools for modelling and managing features of large distributed systems. Another project will be launched for this theme. The existing design tools are appropriate for small systems, in which the reuse assets need not to be shared. The developers of large systems, the kind that most networked control and embedded systems will be in the future, need the supporting infrastructure with the ability to share and manage reuse assets over organisational boundaries.

The ROOM method that was used in the case study proved to have appropriate support for architecture modelling. However, its support for defining variation points is limited. Therefore, there is a need for further studies to develop an integrated development environment with the following properties:

- The product-family tier should have to be supported by the tools to define and manage product features, create semantic components, and generate and validate the configuration rules of the target systems.
- The subsystem and integration tiers need a tool for clustering component variants according to the defined architecture styles, mechanisms, and policies.

- The evaluation of the architectural alternatives must be able to be validated as regards functional and quality requirements.
- Reuse assets have their own quality requirements, such as openness, maintainability, and portability, which have also to be validated at the architecture level.

The subsystem tier that used the PAC architecture in the control domain needs alternative architectures for different problem domains. In order to get a better understanding of the suitability of the component framework, it needs to be applied for several domains. The application areas that have mature product families and development processes are promising domains for component frameworks.

## References

- Alonso, A., Carcia-Valls, M., de la Puente, J. 1998. Assessment of timing properties of family products. In: Development and Evolution of Software Architectures for Product Families, van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429. Germany: Springer-Verlag. Pp. 161–169.
- Barrett, D. J., Clarke, L. A, Tarr, P. L., Wise, A. E. 1996. A Framework for Event-Based Software Integration. ACM Transactions on Software Engineering and Methodology. Vol. 5, No. 4, pp. 378–421.
- Barry, D. 1994. Finding critical features. Object Magazine. July/August. pp. 69–72.
- Bass, L., Chastek, G., Clements, P., Northrop, L., Smith, D., Withey, J. 1998a. Second Product Line Practice Workshop Report. CMU/SEI-98-TR-015. 32 p. Available from <http://www.sei.cmu.edu/publications/featured/technical.html>.
- Bass, L., Clements, P., Kazman, R. 1998b. Software Architecture in Practice. SEI series in software engineering. Reading, Massachusetts: Addison-Wesley. 452 p. ISBN 0-201-19930-0.
- Bellay, B., Gall, H. 1998. Reverse Engineering to Recover and Describe a System's Architecture. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429. Germany: Springer-Verlag. Pp.115–122.
- Ben-Shaul, I., Gish, J. W., Robinson, W. 1998. An Integrated Network Component Architecture. IEEE Software, September/October 1998, pp. 79–87.
- Bergmans, L., 1998. A Notation for Describing Conceptual Software Architectures. In: Bosch, J., Hedin, G., Koskimies, K., Bruun Kristensen, B. (eds.), NOSA '98, Proceedings of the First Nordic Software Architecture Workshop, Research Report 14/98 of the Department of Computer Science and Business Administration, Univ. of Karlskrona/Ronneby, Ronneby, Sweden. ISSN 1103-1581.

Bishop, J., Faria, R. 1996. Connectors in Configuration Programming Languages: are They Necessary? Proceedings of 3<sup>rd</sup> International Conference on Configurable Distributed Systems. Annapolis, MD, USA, May 6–8 1996. Los Alamitos, CA. IEEE Computer Society. Pp. 11–18.

Blum, B., 1996. Beyond Programming. To a New Era of Design. New York: Oxford University Press. 423 p. ISBN 0-19-509160-4.

Bosch, J. 1998. Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study. 1<sup>st</sup> Working IFIP Conference on Software Architecture, October 1998. 13 p. <http://bilbo.ide.hk.r.se:8080/~bosch/articles.html>.

Bratthall, L., Runeson, P. 1998. Architectural Design Recovery of a Family of Embedded Software Systems. An Experience Report. Proceedings of the 1<sup>st</sup> Nordic Workshop of Software Architecture. Höskolan Karlskrona. TR 14/98.

Brown, A. W., Wallnau, K. C. 1998. The Current State of CBSE. IEEE Software, September/October 1998, pp. 38–46.

Brugali, D., Menga, G., Aarsten, A. 1997. The Framework Life Span. Communications of the ACM, Vol. 40, No. 10, pp. 65–68.

Buhr, R. J. A., Casselman, R. S. 1993. Designing with Timethreads. SCE-93-05. Department of Systems & Computer Engineering, Ottawa, Canada. 42 p.

Burns, A., Wellings, A. 1995. HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. Real-time safety critical systems, Vol. 3. Amsterdam, The Netherlands: Elsevier Science B.V. 313 p. ISBN 0-444-82164-3.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M. 1996. Pattern-Oriented Software Architecture – A System of Patterns. Chichester: John Wiley & Sons. 457 p. ISBN 0-471-95869-7.

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., Zullighoven, H. 1997. Framework Development for Large Systems. Communications of the ACM. Vol. 40, No. 10, pp. 52–59.



Campbell, G., Burkhard, N., Facemire, J., O'Conner, J. 1992. Synthesis Guidebook, SPC-91122-MC, Software Productivity Consortium, Herndon, Virginia.

Clements, P. C. 1996. A Survey of Architecture Description Languages. 8<sup>th</sup> International Workshop on Software Specification and Design, Germany, March 1996. 10 p. From [http://www.sei.cmu.edu/publications/articles/survey\\_adl.html](http://www.sei.cmu.edu/publications/articles/survey_adl.html).

Clements, P., Northrop, L. M. 1998. A Framework for Software Product Line Practice – Version 1.0. A living document from <http://www.sei.cmu.edu/plp/framework.html>.

Coad, P., Yourdon, E., 1990. Object-Oriented Analysis. Englewood Cliffs, New Jersey: Prentice-Hall. 232 p. ISBN 0-13-629122-4.

Codenie, W., Hondt, K. D., Steyaert, P., Vercammen, A. 1997. From Custom Applications to Domain-Specific Frameworks. Communications of the ACM. Vol. 40, No. 10, pp. 71–77.

Crane, S. 1996. A Framework for Distributed Interaction. International Workshop on Development and Evolution of Software Architectures for Product Families, Madrid, Spain, November 1996. 10 p. Available from <http://outoften.doc.ic.ac.uk/~isc/research/pub.html>.

Cysewski, G., Gromadzki, T., Lyskawa, H., Piechowska, M., Szejko, S., Kozlowski, W. E., Vahamaki, O. 1998. Reusable Framework for Telecontrol Protocols. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429, Germany: Springer Verlag. Pp.6–13.

Day, R. 1993. Quality Function Deployment. Linking a Company with Its Customers. Milwaukee, Wisconsin: ASQC Quality Press. 245 p. ISBN 0-8739-202.

Demeyer, S., Meijler, T. D., Nierstrasz, O., Steyaert, P. 1997. Design Guidelines for 'Tailorable' Frameworks, Communications of the ACM, Vol. 40, No. 10, pp. 60–64.

Deri, L. 1997. Yasmin: a Component Based Architecture for Software Applications. Proceedings of 8<sup>th</sup> IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering. Los Alamitos: IEEE Computer Society. Pp. 4–12.

Digre, T. 1998. Business Object Component Architecture. IEEE Software, September/October, pp. 60–69.

Dikel, D., Kane, S., Ornburn, S., Loftus, W., Wilson, J. 1997. Applying Software Product-Line Architecture, IEEE Computer, August 1997, pp. 49–55.

Dionisi Vici, A., Argentieri, N., Mansour, A., d'Alessandro, M., Favaro, J. 1998. FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. 5<sup>th</sup> International conference on software reuse. Los Alamitos: IEEE Computer Society. Pp. 166–174.

Dittrich, K. R., Kotz, A. M., Mülle, J. A. 1986. An Event/trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. SIGMOD RECORD. Vol. 15, No. 3, pp. 22–36.

Dolan, T., Weterings, R., Wortmann, J. C. 1998. Stakeholders in Software-system Family Architectures. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429. Germany: Springer-Verlag. Pp. 172–187.

Doscher, D. 1997. CIM Framework Architecture Guide. V. 1.0. Technology Transfer 97103379A-ENG. October, 1997. 66 p. Available from <http://sematech.org/public/division/fi/cim/cimarch.html>.

Doscher, D., Hodges, R. 1997. SEMATECH's Experiences with the CIM Framework. Communications of the ACM, Vol. 40, No. 10, pp. 82–84.

Duenas, J. C., de Oliveira, W. L., de la Puente, J. A. 1998. A Software Architecture Evaluation Model. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429, pp. 148–157.

Eriksson, C., Lundbäck, K. L., Lawson, H. 1995. A Real-Time Kernel Integrated with an Off-Line Scheduler. 3<sup>rd</sup> IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control. Ostend, Belgium, May 1995. 7 p.

Fayad, M. E., Schmidt, D. C. 1997. Object-Oriented Application Frameworks. Communication of the ACM. Vol. 40, No. 10, pp. 32–38.

Ferguson, I. A., 1995. Integrated Control and Coordinated Behaviour: a Case for Agent Models. In: Intelligent Agents, Wooldridge, M., Jennings, N. R., (eds.). Lecture Notes in Artificial Intelligence, LNAI 890. Germany: Springer-Verlag. Pp. 203–218.

Fisher, M. 1995. Representing and Executing Agent-Based Systems. In: Intelligent Agents. Wooldridge, M., Jennings, N. R., (eds.). Lecture Notes in Artificial Intelligence, LNAI 890. Germany: Springer-Verlag. Pp. 307–323.

FORM 1999. FORM Case Tool, Version 1.0. Available from <http://selab.postech.ac.kr/form>.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. O. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison-Wesley Publishing Company. 383 p. ISBN 0-201-63361-2.

Garlan, D., Perry, D. E. 1995. Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 269–274.

Genesereth, M. R., Ketchpel, S. P. 1994. Software Agents. Communications of the ACM. Vol. 37, No. 7, pp. 48–53.

George, G. W., Kryal, E. 1996. The Perception and Use of Standards and Components in Embedded Software Development. – A report for the OMI Software Architecture Forum, July 1996. Draft. 28 p. Available from <http://www.osaf.org/library/market.pdf>.

Gergeleit, M., Streich, H. 1994. Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus. 1<sup>st</sup> International CAN Conference 1994, Mainz, 13–14 September. *Can in Automation e.V.*, Erlangen. 6 p.

Ginis, R., Wolfe, V. F., Prichard, J. J. 1996. The Design of an Open System with Distributed Real-Time Requirements. *Proceedings of Real-Time Technology and Applications*, June 10–12, 1996. Brookline, Massachusetts, Los Alamitos: IEEE Computer Society. Pp. 82–90.

Glass, R. L. 1995. A Structure-Based Critique of Contemporary Computing Research. *Journal of Systems Software*, Vol. 28, pp. 3–7.

Gomaa, H. 1995. Reusable Software. Requirements and Architecture for Families of Systems. *Journal of Systems Software*, Vol. 28, pp. 189–202.

Gomaa, H., Farrukh, G. A. 1997. A Software Engineering Environment for Configuring Distributed Applications from Reusable Software Architectures. *Proceedings of 8<sup>th</sup> IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*. Los Alamitos: IEEE Computer Society. Pp. 312–325.

Goullon, H., Isle, R., Löhr, K. P. 1978. Dynamic Restructuring in an Experimental Operating System. *IEEE Transaction on Software Engineering*, Vol. SE-4, No. 4, pp. 298–307.

Graw, G., Mester, A. 1998. Federated Component Frameworks. *Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98*. TUCS General Publication. Weck, W., Bosch, J., Szyperski, C. (eds.), No. 10, October 1998, pp. 93–100.

Greenwood, N. R. 1986. FMS control and communications. The problems and the potential. *Proceedings of the 5<sup>th</sup> International Conference on Flexible Manufacturing Systems*. Kempston, Bedford: IFS Publications Ltd. Pp. 1–7.

Griss, M. L., Favaro, J., d'Alessandro, M. 1998. Integrating Feature Modelling with the RSEB. *5<sup>th</sup> International Conference on Software Reuse*. Los Alamitos: IEEE Computer Society. Pp. 76–85.

Gyllenswärd, E., Eriksson, C., 1994. A Software Architecture for Complex Real-Time Systems. IEEE Euromicro Workshop on Real-Time Systems, Västerås, Sweden: IEEE Computer Society. Pp.110–115.

Hawryszkiewicz, I., Rose, T. 1995. Notification Agents for Maintaining Awareness. Proceedings of Concurrent Engineering, CE'95 Conference, August 23–25, Virginia: McLean, Pp. 305–314.

Hayes-Roth, B., Pflieger, K., Lalanda, P., Morignot, P., Balabanovic, M. 1995. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 228–301.

Hodgson, R. 1991. The X-model: a Process Model for Object-Oriented Software Development. 4<sup>th</sup> International Conference on Software Engineering and its Applications. Toulouse, France. 17 p.

Holibaugh, R. 1993. Joint Integrated Avionics Working Group (JIAWG). Object-Oriented Domain Analysis Method (JODA), version 3.1, Special Report CMU/SEI-92-SR-3. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Jacobson, I., Griss, M., Jonsson, P. 1997. Software Reuse. Architecture Process and Organization for Business Success. Harlow: Addison Wesley Longman. 497 p. ISBN 0-201-02476-5.

Johnson, R. E. 1997. Frameworks =(Components+Patterns). Communications of the ACM. Vol. 40, No. 10, pp. 39–42.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., Peterson, A. S. 1990. Feature-oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/Sei-90-TR-21. Software Engineering Institute. Pittsburgh: Carnegie Mellon University. 147 p.

Kang, K. C. 1998. Feature-Oriented Development of Applications for a Domain. 5<sup>th</sup> International Conference on Software Reuse. Los Alamitos: IEEE Computer Society. Pp. 354–355.

Kappel, G., Rausch-Schott, S., Retschitzegger, W., Vieweg, S. 1994. TriGS: Making a passive object-oriented database system active. *Journal of Object Oriented Programming*, Vol. 7, No. 4, pp. 40–51, 63.

Kappel, G., Vieweg, S. 1994. Database Requirements of CIM Applications. *Integrated Manufacturing System. An International Journal*, MBC University Press, Vol. 5, No. 4/5, pp. 48–63.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwing, J. 1997. Aspect-Oriented Programming. *Proceedings of ECOOP'97. LNCS 1241*. Germany: Springer Verlag. Pp. 220–242.

Kniesel, G. 1998. Type-safe Delegation for Dynamic Component Adaptation. *Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98*. TUCS General Publication, Weck, W., Bosch, J., Szyperski, C. (eds.), No. 10, October 1998. Pp. 9–18.

Kopetz, H., Damm, A., Koza, C., Malazzani, M., Schwabl, W., Senft, C., Zainlinger, R. 1989. Distributed fault-tolerant real-time systems: the MARS approach. *IEEE Micro*. Vol. 9, February, pp. 25–40.

Lange, D. B. 1998. Mobile Objects and Mobile Agents: The Future of Distributed Computing? *Proceedings of ECOOP'98. LNCS 1445*. Germany: Springer-Verlag. Pp. 1–12.

Lawson, H. 1992. *Parallel Processing in Industrial Real-Time Applications*. New Jersey: Prentice Hall, Englewood Cliffs. 514 p.

Lejter, M., Dean, T. 1996. A Framework for the Development of Multiagent Architectures. *IEEE Expert*, December 1996, pp. 47–59.

Lim, A. S. 1996. Abstraction and Composition Techniques for Reconfiguration of Large-Scale Complex Applications. *Proceedings of 3<sup>rd</sup> International Conference on Configurable Distributed Systems*. Annapolis, MD, USA, May 6–8 1996. Los Alamitos, CA: IEEE Computer Society. Pp. 186–193.

Ljung, S. 1986. Robot Cells and System for Small Part Assembly. Proceedings of the 5<sup>th</sup> International Conference on Flexible Manufacturing Systems. Kempston, Bedford: IFS Publications Ltd. 512 p.

Losavio, F., Matteo, A. 1997. A Method for User-Interface Development. Journal of Object-Oriented Programming, Vol. 10, No. 5, pp. 22–27, 75.

Luckham, D., Vera, J., Bryan, D., Augustin, L. 1993. Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems. Journal of Systems and Software, Vol. 21, No. 3, pp. 253–265.

Luckham, D, Kenney, J., Augustin, L. M., Vera, J., Bryan, D., Mann, W. 1995. Specification and Analysis of System Architecture Using Rapide. IEEE Transaction on Software Engineering, Vol. 21, No. 4, pp. 336–355.

Luckham, D., Vera, J. 1995. An Event-Based Architecture Definition Language, IEEE Transaction on Software Engineering, Vol. 21, No. 9, September 1995, pp. 717–734.

Lycett, M., Paul, R. J. 1998. Component-Based Development: Dealing with Non-Functional Aspects of Architecture. Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98. TUCS General Publication, Weck, W., Bosch, J., Szyperski, C. (eds.), No. 10, October 1998, pp. 83–92.

Macala, R. R., Stuckey, L. D., Gross, D. D. 1996. Managing Domain-Specific, Product-Line Development. IEEE Software, May 1996, pp. 57–67.

Magee, J., Dylay, N., Kramer, J. 1994. Regis: A constructive Development Environment for Distributed Programming. Distributed Systems Engineering Journal, Vol. 1, No. 5, pp. 304–312.

Magee, J., Kramer, J. 1996. Dynamic Structure in Software Architectures. Proceedings of SIGSOFT '96. Pp. 3–14.

Magee, J., Kramer, J., Sloman, M. 1989. Constructing Distributed Systems in Conic. IEEE Transactions on Software Engineering, Vol. 15, No. 6, pp. 663–675.

Mannion, M., Keepence, B., Harper, D. 1998. Using Viewpoints to Define Domain Requirements. *IEEE Software*, January/February 1998, pp. 95–102.

Mannion, M., Boyle, D., Keepence, B. 1999. A Survey of Domain Engineering Methods. Submission for ACM. 32 p.

McCarthy, D., Dayal, U. 1989. The architecture of an Active Data Base Management System. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data. SIGMOD RECORD*, Vol. 18, No. 2, pp. 215–224.

McGraw, G. 1998. Why COTS Software Increases Security Risks. *SES'98, Reliable Software Technologies. OTS Workshop*. 5 p. Available from <http://www.rstcorp.com/ots/>.

Mendonca, N. C., Kramer, J. 1998. An Experiment in Distributed Software Architecture Recovery. In: *Development and Evolution of Software Architectures for Product Families*. van der Linden, F. (ed.). *Lecture Notes in Computer Science*, No. 1429. Germany: Springer-Verlag. Pp.115–122.

Meslati, D., Ghoul, S. 1997. Semantic classification: A genetic approach to classification in object-oriented models. *Journal of Object-Oriented Programming*, January 1997, pp. 25–37.

Mowbray, T. 1997. *Inside CORBA*. Massachusetts: Addison-Wesley. 374 p. ISBN 0-201-89540-4.

Ng, K., Kramer, J., 1995. Automated support for Distributed Software Design. *Proceedings of 7th International Workshop on Computer-Aided Software Engineering (CASE 95)*, Toronto, Canada, July 1995. Pp. 381–390.

ODP. 1995. Secretariat: ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing, part 3: Architecture. Document ITU-T X.903 (ISO/IEC 10746-3). Geneva, Switzerland: ISO/IEC Copyright Office. 78 p.

OMG. 1995. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, Revision 2.1 (1997).



OPC, Foundation. 1996. OLE for Process Control (OPC) standard. From <http://www.opcfoundation.org/>

OSACA. 1996. Open System Architecture for Controls within Automation Systems. ESPRIT III. EP 9115, OSACA II Final Report. 73 p. Available from [http://www.osaca.org/osaca/sub\\_site/related\\_projects/osaca\\_2.html](http://www.osaca.org/osaca/sub_site/related_projects/osaca_2.html).

Parnas, D. 1976. On the Design and Development of Program Families. IEEE Transaction on Software Engineering, Vol. SE-2, No. 1, pp. 1–9.

Perry, D. E. 1998. Generic Architecture Descriptions for Product Lines. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429. Germany: Springer-Verlag. Pp. 51–56.

Perry, D., Kramer, J. 1998. Session 2: Architectural Descriptions. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429, Germany: Springer-Verlag. Pp. 49–50.

Pfleeger, S. L. 1997. Experimentation in Software Engineering. In: Advances in Computers. Vol. 44. Zelkowitz, M. V. (ed.), San Diego: Academic Press. Pp. 127–167.

Polze, A., Sha, L. 1998. Composite Objects: Real-Time Programming with CORBA. Proceedings of the 24<sup>th</sup> EUROMICRO Conference, Vol. 2. Los Alamitos, CA: IEEE Computer Society. Pp. 997–1004.

Poutain, D. 1995. The Oberon/F System: A lightweight, portable, object-oriented component framework. Byte, January 1995.

Prieto-Diaz, R. 1987. Classifying Software for Reusability. IEEE Software. Vol. 4, No. 1, pp. 6–16.

Prieto-Diaz, R., Neighbours, J. M. 1986. Module interconnection languages. Journal of Systems and Software, Vol. 6, pp. 307–334.

Pryce, N., Crane, S. 1996. A Uniform Approach to Configuration and Communication in Distributed Systems. Proceedings of 3<sup>rd</sup> International Conference on Configurable Distributed Systems. Annapolis, MD, USA May 6–8, 1996. Los Alamitos, CA: IEEE Computer Society Press. Pp. 144–151.

Ran, A., Xu, J. 1996. Structuring Interfaces of Complex Software Components. Proceedings of 2<sup>nd</sup> International Software Architecture Workshop, ISAW-2, San Francisco, October 1996. Pp. 39–43.

Robben, B., Matthijs, F., Joosen, W., Vanhaute, B. Verbaeten, P., Leuven, K. U. 1998. Components for Non-Functional Requirements. Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98. TUCS General Publication, Weck, W., Bosch, J., Szyperki, C. (eds.), No. 10, October 1998, pp. 75–82.

Robbins, J. E., Medvidovic, N., Redmiles, D. F., Rosenblum, D. S. 1997. Integrating Architecture Description Languages with a Standard Design Method. Dept. of Information and Computer Science, University of California, Irvine. ICS-TR-97-35.

Rossak, W., Kirova, V., Jololian, L., Lawson, H., Zemel, T. 1997. A Generic Model for Software Architectures. IEEE Software, July/August 1997, pp. 84–92.

Rossak, W., Zemel, T., Kirova, V., Jololian, L. 1994. A Two-Level Process Model for Integrated System Development. Tutorial and Workshop on Systems Engineering of Computer-Based Systems. Los Alamitos, CA: IEEE Computer Society. Pp. 90–96.

Santori, M. 1997. OPC: OLE for Process Control. Real-time Magazine, 4/97, pp. 78–81.

Savola, R., Pulli, P., Heikkinen, M., Seppänen, V., Kurki, M., Taramaa, J. 1995. Concurrent Development of Multi-Technology Products – Integrating Virtual and Executable Models. Proceedings of the Concurrent Engineering, CE95, Johstown, Pennsylvania, June 1995. Concurrent Technologies Corporation, Oakland, California. Pp. 645–651.

Schmid, H. A. 1996a. Creating Applications from Components: A Manufacturing Framework Design. IEEE Software, November 1996, pp. 67–75.

Schmid, H. A. 1996b. Design Patterns for Constructing the Hot Spots of a Manufacturing Framework. Journal of object oriented programming, Vol. 9, No. 3, pp. 25–37, 73.

Schmidt, D. C., Fayad, M. E. 1997. Lessons Learned. Building Reusable OO Frameworks for Distributed Software. Communications of the ACM. Vol. 40, No. 10, pp. 85–87.

Selic, B., Gullekson, G., Ward, P. 1994. Real-time object-oriented modelling. New York: John Wiley & Sons. 525 p. ISBN 047-1599 174.

Seppänen, V. 1990. Acquisition and Reuse of Knowledge to Design Embedded Software. VTT Publications 66. Espoo, Finland: Technical Research Centre of Finland. Ph.D. Thesis. 218 p. + app. 10 p.

Seppänen, V., Niemelä, E., Taramaa, J. 1995. CACE+CASE=Better Reuse of Mechatronic Software. Proceedings of CASE'95 workshop. Toronto: IEEE Computer Society. Pp. 289–295.

Shaw, M, Garlan, D. 1996. Software Architecture. New Jersey, USA: Prentice Hall. 242 p. ISBN 0-13-182957-2.

Shaw, M. 1995. Comparing Architectural Design Styles. IEEE Software, Vol. 12, No. 6, pp. 27–41.

Siegel, J. (ed.). 1996. CORBA: Fundamentals and Programming. New York: John Wiley & Sons, Inc. 693 p. ISBN 0-471-12148-7.

Simos, M. 1995. Organization Domain Modelling (ODM). Guide book, Version 1.0, STARS-VC-A023/011/00. March 1995.

Simos, M., Anthony, J. 1998. Weaving the Model Web: A Multi-Modelling Approach to Concepts and Features in Domain Engineering. 5<sup>th</sup> International conference on software reuse. Los Alamitos: IEEE Computer Society. Pp. 94–102.

Sodhi, J., Sodhi, P. 1999. Software Reuse. Domain Analysis and Design Process. New York: McGraw-Hill. 344 p. ISBN 0-07-057923-7.

Soininen, J.-P. 1997. Asiakastarvelähtöisyys elektronisen tuoteperheen suunnittelussa [Customer oriented development of electronic product family]. Licentiate Thesis. VTT Julkaisuja 822. Espoo, Finland: Technical Research Centre of Finland (VTT). 111 p. + app. 18 p. (In Finnish.) ISBN 951-38-4531-1.

Stadel, M. 1991. Object Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program. ACM SIGPLAN Notices, Vol. 26, No. 1, January 1991, pp. 99–108.

Stewart, D. B., Volpe, R. A., Khosla, P. K. 1997. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. IEEE Transaction on Software Engineering, Vol. 23, No. 12, pp. 759–776.

Szyperski, C. 1997. Component Software. Beyond Object-Oriented Programming. New York: Addison Wesley Longman Ltd. 411 p. ISBN 0-201-17888-5.

Taivalsaari, A. 1993. Object-oriented programming with modes. Journal of Object-Oriented Programming, June 1993, pp. 25–32.

Taramaa, J. 1998. Practical development of software configuration management for embedded systems. PhD Thesis. VTT Publications 366. Espoo, Finland: Technical Research Centre of Finland (VTT). 147 p. + app. 110 p. ISBN 951-38-5344-6.

Tracz, W., Coglianese, L., Young, P. 1993. A Domain-Specific Software Architecture Engineering Process Outline. ACM SIGSOFT, Software Engineering Notes, Vol. 18, No. 2, pp. 40–49.

Tran, V., Liu, D. B. 1997. Component-based Systems Development: Challenges and Lessons Learned. Proceedings of 8<sup>th</sup> IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering. Los Alamitos: IEEE Computer Society. Pp. 452–462.

Törngren, M., Backman, U. 1993. Evaluation of Real-Time Communication Systems for Machine Control. Proceedings of the Swedish National Association on Real-Time Systems Conference, SNART in co-operation with DAMEK, KTH, 25–26 August 1993. Stockholm: The Royal Institute of Technology, No. 2. 17 p.

Törngren, M., Lind, H. 1994. On Decentralization of Control Functions for Distributed Real-Time Motion Control. International Symposium on Robotics and Manufacturing, ISRAM'94, August 1994. Pp. 699–704.

Törngren, M., Wikander, J. 1992. Real-Time Control of Physically Distributed Systems. Application: Motion Control. Computer Electronic Engineering, Vol. 18, No. 1, pp. 51–72.

van den Hamer, P., van der Linden, F., Saunders, A., de Sligte, H. 1998. An integral hierarchy and diversity model for describing product family architectures. In: Development and Evolution of Software Architectures for Product Families. van der Linden, F. (ed.). Lecture Notes in Computer Science, No. 1429, Germany: Springer-Verlag. Pp. 66–75.

van der Linden, F. J., Muller, J. K. 1995. Creating Architectures with Building Blocks. IEEE Software, November 1995, pp. 51–60.

Welch, I., Stroud, E. 1998. Adaptation of Connectors in Software Architectures. Proceedings of 3<sup>rd</sup> International Workshop on Component Programming, WCOP'98. TUCS General Publication, Weck, W., Bosch, J., Szyperski, C. (eds.), No. 10, October 1998. Pp. 47–52.

Vestal, A. 1993. A Cursory Overview and Comparison of four Architecture Description Languages. 9 p.

Available from <http://www-ast.tds-gn.lmco.com/arch/arch-ref.html>.

Voas, J. 1998. Certifying Off-the-Shelf Software Components. IEEE Computer, Vol. 31, No. 6, pp. 53–59.

Wooldridge, M., Jennings, N. R. 1995. Intelligent Agents: Theory and Practice. Knowledge Engineering Review, Vol. 10, No. 2. 62 p.

*Appendices of this publication are not included in the PDF version.  
Please order the printed version to get the complete publication  
(<http://www.inf.vtt.fi/pdf/publications/1999/>)*

Published by



Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland  
Phone internat. +358 9 4561  
Fax +358 9 456 4374

Series title, number and  
report code of publication

VTT Publications 402  
VTT-PUBS-402

<b>Author(s)</b> Niemelä, Eila			
<b>Title</b> <b>A component framework of a distributed control systems family</b>			
<b>Abstract</b> <p>A component framework is based on a software architecture, a set of components and their interaction mechanisms. This thesis examines the component-based software development by reviewing the requirements for a component framework development, proposing a model of a component framework of a distributed control systems family and demonstrating results with cases drawn from the control systems families.</p> <p>The product families of the machine control systems, process control systems and manufacturing systems are studied to set the requirements for the component framework. Three main problems are discovered. A lack of appropriate modelling methods prevents describing product features and variability at the software architecture level. Interoperability and adaptability of software components that are required in the integration phase are inadequate in most cases. Furthermore, integrators and maintenance staff also need support software for extending and upgrading systems.</p> <p>The component framework of a distributed control systems family introduces two dimensions: tiers and elements. The three tiers of the component framework define the subsystems in the first tier, integration platform in the second tier, and the product family in the third tier. The tiers explain the domain, technology and business viewpoints of the framework correspondingly. The elements define the product features, software architecture, components and their interaction mechanisms. The development and utilisation of the component framework have three main tasks, described as the viewpoints of the component-based software development, concurrent software engineering and software configuration management.</p> <p>The development of the component framework is presented by the development of the reuse assets: the product features, product-family architecture and software components. The architecture styles, key-mechanisms, services and components of each tier are depicted. The framework mixes agent, layered, client-server and rule-based system architectures and their mechanisms to provide a coherent solution for software flexibility and stability required by the product families.</p> <p>The results are analysed as regards the evaluation criteria, set for the component framework as the result of the problem analysis. Variability and adaptability are examined at the architecture and component level, as well as the interoperability of tiers, services and applications and interchangeability of product features and components.</p> <p>The adaptive approach restricts the affects of the changes in the business, technology and application domain to the corresponding tiers that provide their own mechanisms for adaptability. The integration tier could be reused community-wide, whereas the subsystem tier is domain-specific and the product-family tier is always an organisation-dependent solution.</p>			
<b>Keywords</b> software engineering, component framework, component-based development, distributed control systems, software configuration management			
<b>Activity unit</b> VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN-90571 Oulu, Finland			
<b>ISBN</b> 951-38-5549-X (soft back ed.) 951-38-5550-3 (URL: <a href="http://www.inf.vtt.fi/pdf/">http://www.inf.vtt.fi/pdf/</a> )		<b>Project number</b> E6SU00081	
<b>Date</b> December 1999	<b>Language</b> English	<b>Pages</b> 188 p. + app. 68 p.	<b>Price</b> F
<b>Name of project</b> KIURU, DYNAMO, ARTTU		<b>Commissioned by</b> Technology Development Centre of Finland (Tekes), VTT Electronics, the Foundation of Emil Aaltonen, industrial companies	
<b>Series title and ISSN</b> VTT Publications 1235-0621 (soft back ed.) 1455-0849 (URL: <a href="http://www.inf.vtt.fi/pdf/">http://www.inf.vtt.fi/pdf/</a> )		<b>Sold by</b> VTT Information Service P.O.Box 2000, FIN-02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

