Seppo Kuikka

A batch process management framework

Domain-specific, design pattern and software component based approach

TECHNICAL RESEARCH CENTRE OF FINLAND ESPOO 1999

VTT PUBLICATIONS 398

A batch process management framework

Domain-specific, design pattern and software component based approach

Seppo Kuikka

VTT Automation

Dissertation for the degree of Doctor of Technology to be presented with due permission for public examination and debate at Helsinki University of Technology (Espoo, Finland) in Auditorium T2 (Konemiehentie 2, Espoo) on the 10th of December, 1999 at 12 o´clock noon.

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 1999

ISBN 951–38–5541–4 (soft back edition)
ISSN 1235–0621 (soft back edition)

ISBN 951–38–5542–2 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

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 Automaatio, Teollisuusautomaatio, Tekniikantie 12, PL 1301, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 6752

VTT Automation, Industriautomation, Teknikvägen 12, PB 1301, 02044 VTT
tel. växel (09) 4561, fax (09) 456 6752

VTT Automation, Industrial Automation, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 6752

Technical editing Maini Manninen

Libella Painopalvelu Oy, Espoo 1999

Kuikka, Seppo. A batch process management framework. Domain-specific, design pattern and software component based approach. Espoo 1999. Technical Research Centre of Finland, VTT Publications 398. 215 p.

Keywords batch control, object-oriented software, batch process management, framework, design pattern, software component, software agent

Abstract

Requirements for product and production quality and variability, as well as the needs for the efficient use of production equipment, have emphasised the benefits of *batch production* in the process industries. The resulting complexity of batch control has, however, been a challenge to control engineers. Emerging batch standards and software component technologies have now made it possible to design *flexible*, *distributed*, and *integrated* batch automation concepts to satisfy the requirements.

The batch control domain was studied in this thesis in terms of domain standardisation, existing batch control systems, and related research approaches. The applicable information technology, object-oriented software component frameworks and multi-agency, was surveyed and evaluated. Guidelines for deploying generic software *design patterns* in designing *domain-specific frameworks*, were adapted.

An experimental *batch process management framework*, was developed to fulfil the aforementioned *flexibility*, *distribution* and *integration* needs for batch automation. It also demonstrates *reusability* by the so-called *calling framework* architectural style as well as internal and external component interfaces. Framework *components* may be easily parametrized and *replaced* by customised versions. Additionally, the framework can be integrated with other systems by using component technology.

For some problem specific needs of local decision-making and interaction, enhancement of component frameworks may be needed. No applicable design patterns were found for this kind of design issue. Since the design problem is

recurrent, a generic design pattern, *Agentified Component*, was developed and experimented with within the framework of this thesis. The approach retains the deterministic nature of the framework, important in the automation domain, but simultaneously introduces the possibility of solving specific problems using a knowledge-based approach.

Preface

I have been involved in several ways with software development for automation applications during my professional life. From the point of view of this thesis, an important impetus was the 1995 Summer School on Reusable Architectures in Object-Oriented Software Development in Tampere (supported by the Finnish Program of Doctoral Studies in Computer Science). Within that excellent event my - until then - preliminary ideas of deploying design patterns for developing a domain-specific automation framework began to take shape.

I studied the themes of this study within several postgraduate seminars in Helsinki University of Technology (HUT) in the years 1995 - 1998. Another important impetus for this thesis took place during one of these seminars. I was given an opportunity to participate (starting from spring 1996) in a VTT Automation research programme CAIP (Control Architectures for Intelligent Production). A CAIP project (and a continuation project of it) made it possible to develop the experimental Batch process management framework, described in this thesis. It thus helped to verify the domain-specific, design pattern, and software component based approach to automation software development. It also made it possible to study the integration of multi-agent technologies into a component framework.

I am especially thankful to my supervisor, Professor Kari Koskinen from the Laboratory of Information Systems in Automation of HUT, who previously also acted as the leader of the CAIP research programme in VTT. I also want to express my deepest gratitude to my instructor and group leader, Dr. Olli Ventä from VTT Automation. A third person, to whom I am also indebted, is the automation systems team leader and my co-worker at VTT Automation, Mr. Teemu Tommila. These colleagues have provided for the facilities and prerequisites for this research and development work and, more importantly, they have always been willing to give professional support and warm encouragement.

Other people, discussions with whom have been important for developing the ideas and/or writing the thesis are: Professor Aarne Halme, Mr. Alexander Ran, Mr. Thomas Fischer, Mr. Kevlin Henney, and Dr. Seppo Haltsonen. I am also

grateful to Professors Pentti Lautala and Ilkka Haikala from Tampere University of Technology for providing expert criticism and valuable suggestions for this thesis.

I also want to thank all my colleagues at VTT Automation, several post graduate seminar instructors and students in HUT and my colleagues and students in Espoo-Vantaa Institute of Technology (EVITech) for relaxed but inspiring working environment. For the financial support, I express my gratitude to VTT, Tekes, and EVITech. Finally, my warmest thanks are reserved for my family; my wife Tarja and daughter Katri for their love, patience, and support during the studies and research work.

Contents

Part III. The Problem Statement

Part IV. The Proposed Solution

Abbreviations

ACL	Agent Communication Language
BDI	Belief, Desire, Intention (agent model)
CBD	Component-Based Development
CIM	Computer Integrated Manufacturing
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf (component)
DCOM	Distributed Component Model
DCS	Digital Control System
EJB	Enterprise JavaBeans
ERM	Enterprise Resource Management
ERP	Enterprise Resource Planning
FIPA	Foundation for Intelligent Physical Agents
IDL	Interface Definition Language
ISA	International Society for Measurement and Control
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
MAS	Multi-agent System

MES	Manufacturing Execution System
OPC	OLE (Object Linking and Embedding) for Process Control
PFC	Procedure Function Chart
SCM	Supply Chain Management
SFC	Sequential Function Chart
SQL	Standard Query Language
UML	Unified Modelling Language

Glossary

Agent - see Software agent

Agent intention
An agent's commitment to act while in a certain mental state.

Agent interaction
An agent's capability to exchange information and knowledge with other agents and the environment. Uses a transport protocol (e.g. TCP/IP), an agent communication language (ACL, e.g. KQML or FIPA ACL), and a negotiation protocol (e.g. contract net).

Allocation
A form of co-ordination control that assigns a resource to a batch or a unit.

Arbitration
A form of co-ordination control that determines how a resource should be allocated when there are more requests for the resource than can be accommodated at one time.

Architectural design pattern
A fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (Buschmann *et al*., 1996).

Basic control
Control that is dedicated to establishing and maintaining a specific state of equipment or process condition.

Batch
1. The material that is being produced or that has been produced by a single execution of a batch process.
2. An Entity that represents the production of a material at any point in the process.

Batch control
Control activities and control functions that provide a means to process finite quantities of input materials by subjecting them to an ordered set of processing activities over a finite period of time using one or more pieces of equipment.

Batch process
A process that leads to the production of finite quantities of material by subjecting quantities of input materials to an ordered set of processing activities over a finite period of time using one or more pieces of equipment.

Batch process management
The control activity that includes the control functions needed to manage batch production within a process cell.

Batch schedule
A list of batches to be produced in a specific process cell.

CAIP – SWF project
A research and development project Software Factory (SWF), belonging to a VTT Research Programme Control Architectures for Intelligent Production (CAIP). The project was carried out in VTT Automation in the years 1996 and 1997. The basic framework of this thesis was developed in this project.

Component - see *Software component*

Component framework
A software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances.

Component interoperation
The capability of a software component to request a service and respond to a request using well-defined application level interfaces. The parameter delivery between the client, requesting the service and the server, providing it, is automated.

Component system
A set of related components that collectively accomplish a function larger than that accomplished by a single software component.

Control module
The lowest level grouping of equipment in the physical model that can carry out basic control.

Control recipe
A type of recipe which, through its execution, defines the manufacture of a single batch of a specific product.

Co-ordination control
A type of control that directs, initiates, and/or modifies the execution of procedural control and the utilisation of equipment entities.

Design pattern
A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems (Gamma *et al*., 1995).

Equipment control
The equipment-specific functionality that provides the actual control capability for an equipment entity, including procedural, basic, and co-ordination control, and that is not part of the recipe.

Equipment entity
A collection of physical processing and control equipment and equipment control grouped together to perform a certain control function or a set of control functions.

Equipment module
A functional group of equipment that can carry out a finite number of specific minor processing activities.

Equipment procedure
A procedure that is part of equipment control.

Equipment unit procedure
A unit procedure that is part of equipment control.

Exception handling
Those functions that deal with plant or process contingencies and other events which occur outside the normal or desired behaviour of control.

Formula
A category of recipe information that includes process inputs, process parameters, and process outputs.

Framework - see *Software framework*

General recipe
A type of recipe that expresses equipment and site independent processing requirements.

Header
Information about the purpose, source and version of the recipe such as recipe and product identification, creator, and issue date.

Master recipe
A type of recipe that accounts for equipment capabilities and may include process cell-specific information.

Path
The order of equipment within a process cell that is used, or is expected to be used, in the production of a specific batch.

Procedural control
Control that directs equipment-oriented actions to take place in an ordered sequence in order to carry out some process-oriented task.

Procedural element
A building block for procedural control.

Process cell
A logical grouping of equipment that includes the equipment required for production of one or more batches. It defines the span of logical control of one set of process equipment within an area.

Recipe
The necessary set of information that uniquely defines the production requirements for a specific product.

Recipe procedure
The part of a recipe that defines the strategy for producing a batch

Scenario
A single path through a use case, one that shows a particular combination of conditions within that use case.

Site recipe
A type of recipe that is site specific.

Software agent
A computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives.

Software component
A reusable, executable, self-contained piece of software, which is accessible only through well-defined interfaces.

Software framework
A partially complete software (sub)system that is intended to be instantiated. It defines the architecture for a family of (sub-)systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment, a framework consists of abstract and concrete classes.

SWFBatch project
A continuation R&D project to the CAIP – SWF project, financed by Tekes, VTT and Honeywell-Measurex Inc. The project was carried out in VTT Automation in the year 1998. The framework of this thesis was integrated with a DCS and enhanced with multi-agency in this project.

Unit
A collection of associated control modules and/or equipment modules and other process equipment in which one or more major processing activities can be conducted.

Unit recipe
The part of a control recipe that uniquely defines the contiguous production requirements for a unit.

Use case
A typical interaction between a user and a computer system.

1. Introduction

1.1 Motivation and background

Digital process control and information management systems and programmable logic and PC-based systems are often distributed. *Traditional distribution* of control and management systems is ruled partly by the necessary distribution of the controlled process equipment and by the requirement for high availability of automation. Also the benefits of decomposing functionality and layering it into an automation hierarchy of production planning, production control, coordination, and basic control have been recognized for a long time (Williams, 1989).

The above kind of hierarchical distribution of control activities is necessary but not always sufficient. *Functionally originated distribution* is also needed from the design perspective. This means that control decisions shall be made as autonomously as possible, partly near process equipment, partly on various levels of the automation hierarchy, and partly near operators responsible for production. Additionally, for business reasons, specific control algorithms and other proprietary expertise, shall often be enclosed within autonomous information processing entities near the controlled process units. These requirements are implicit, for example, in the new standardization efforts for batch control (ISA, 1995; ISA, 1999a).

New *integration needs* are also emerging. Process control and management systems must be able to interoperate with other information systems on a high semantic level, not only convey data. Interoperation, especially with Manufacturing Execution Systems (MES) and Enterprise Resource Planning (ERP) systems, becomes more important for optimal production. The implementation is expected to be *open*, i.e. systems, subsystems, or even individual control entities, may be acquired from various vendors and still be able to interoperate. These needs are evident, for example, when considering the interfaces between the so-called level 2 (including process management), level 3 (including MES), and level 4 (including ERP) functionalities in the abovementioned automation hierarchy (ISA, 1999b).

Within bulk chemical process industries it is often possible to achieve cost advantage by continuous production on high production levels. This is especially true, if the products are undifferentiated, sold on the basis of their chemical composition and purity. For these

products there are normally several, globally operating manufacturers, which compete mostly on the basis of price.

The situation is different when *smaller quantities* of *differentiated* high-technology products (for example pharmaceuticals, speciality chemicals, or biotechnology products) are produced. The products are purchased because of their effect rather than their composition. An extensive skill-base is needed in both chemical development and production. The prices for these products are also normally higher than those of undifferentiated products (Sharratt, 1997).

In the case of differentiated products and relatively small production quantities, modern, flexible *batch production* is a necessity (Rosenof & Ghosh, 1987; Fisher, 1990; ISA, 1996; Sharratt, 1997). Also individual customer service and product quality requirements highlight the benefits of batch processing and, accordingly, *batch control*, as compared to continuous processing and control (Tommila, 1993). Small batch-oriented plants can be re-sized economically according to changing market demands. Multipurpose process units allow further *flexibility* to production of various products with the same processing equipment.

In batch control, *recipes* provide a means to describe products and the manner in which to produce them. A recipe contains production-related information on a sufficiently detailed level for a specific product (master recipe) or for its batch-wise production (control recipe). Batch-process equipment, on the other hand, is formed of *process cells* and *units*. Major batch-processing activities can be conducted in consequent units along so-called *paths* within a process cell (ISA, 1995).

The challenge to batch control, which results from the requirements of flexible production, is the variability and *complexity* of control tasks. Equipment controls need to be designed for several alternative products, production configurations, and operating conditions. Product batches must be optimally scheduled and process equipment must be allocated to them dynamically. The operations of the batches must also be effectively co-ordinated. Moreover, this new functionality needs to be *included* into the *architecture* of the batch control and management system, in order to be *reused* from one application to another.

Batch Process Management control activity consists of control functions Batch Management and Process Cell Management. Batch Management, based on recipes, is focused on batch products and production. Process Cell Management, based on equipment descriptions and procedures, is focused on equipment management and control

(ISA, 1995). The *separation of concerns* of product- and production-oriented Batch Management, from equipment- and control-oriented Process Cell Management, emphasises the need for *distribution, integration and flexibility* within batch process management software architecture.

The *research* work within batch process management has lately been concentrated on Petri Net based, mostly analytic approaches (Tittus *et al*., 1995; Tittus & Egardt, 1996; Åkesson & Tittus, 1998; Johnsson, 1996a,b; Årzen & Johnsson, 1996; Johnsson & Årzen, 1998). Some researchers (Fleming & Schreiber, 1998) have focused on batch control design problems, arisen from the need to integrate production-oriented procedural control and process-equipment-oriented equipment control. In spite of the great software architectural challenges of batch process management, few researchers have concentrated especially on them, most notably Simensen and his colleagues, (Simensen *et al*., 1997).

1.2 Hypotheses and goals of the study

The above-mentioned *distribution* needs require that software modules, which are used for the control tasks, must be well encapsulated, containing both the information and operations needed to implement the services requested. This leads to the use of *distributed object technology*. For the system to be internally coherent and well *integrated* with other systems, the interoperations of relevant information entities shall take place via defined interfaces only. This leads to the deployment of *software component technology*.

The interoperations within batch control are often complex and thus the simple approach of composing components by using scripts is not sufficient. Instead, a proper way to create larger, functional entities is to develop *frameworks based on distributed component models*. One efficient way to design frameworks is to make systematic use of generic expertise on the object-oriented and component-oriented research and development, so-called *design patterns.*

The main *hypothesis* of this study is that domain-specific frameworks for batch control are to be developed on the basis of sufficient and judiciously applied *domain knowledge* while exploiting *generic design patterns*. Domain knowledge is needed not only *to define* the requirements for the framework and to specify it, but also *to make design decisions* concerning the structural and behavioural aspects of the architecture.

Traditionally, the value of domain knowledge has been acknowledged when defining requirements but design decisions have been considered domain independent. This study

indicates that design issues, or so-called *forces*, cannot be explicated without domain-specific considerations. Thus domain knowledge is needed in deploying generic design patterns to the design of the framework.

The definition of requirements is based on *analytic requirement models*, and the design is based on *synthetic design pattern models*. In this study, batch domain dependency is explicit for both of these. In order to be valuable in practice, the domain-specific frameworks should also be *designed* to be *reusable* for several different application projects in the batch domain. This emphasises the need to correctly abstract *commonalities of the domain* into the design of the framework.

In certain applications and *specific problems of the batch automation domain* (for example, in dynamic unit allocation within batch process management), both local decision making and collaboration between decision-making entities are needed. One way to achieve this, is the so-called *multi-agent* technology, which can be applied within several system architectural settings. A further *hypothesis* of this thesis is that *multi-agency* can also be applied in industrial domains like batch control, if it is seamlessly *embedded* within domain-specific component frameworks. This new approach is here called *agentifying* the component framework.

The main *goals* of this study are both to *refine* the *distribution*, *integration*, and *flexibility needs* in batch-control domain, as well as to *adapt* the above *technologies* for the development of batch-domain specific frameworks. Moreover, the validity of the hypotheses is to be *tested experimentally* by verifying that the domain-specific requirements for a new batch process management system can be fulfilled with the help of the new information technologies.

1.3 Research approaches and methods

The *batch control domain* is studied and evaluated with a uniform approach to assess the new batch control standard (ISA, 1995; ISA, 1999a), the foremost commercial batch control systems (InBatch, 1998; VisualBatch, 1998; OpenBatch, 1999), and the research in the domain, referred to in the previous section. In all areas, the *literature study* proceeds from process, equipment and control aspects to recipe-oriented product and production aspects and then to architectural issues. With this approach it is possible to *compare* the batch standard, the batch control products and the new development concepts *systematically*. It makes it possible to evaluate how well the commercial products conform to the standard. It also indicates *opportunities for research and*

development, especially in areas discussed in the standard but not adequately covered by the systems or research.

The approach in presenting and discussing *object-oriented software component frameworks* is different. The background of the study is industrial software development and training experience in automation software, which is traditionally developed with structured methodologies, if any. A *concise survey on objects, components* (OMG, 1995; Microsoft, 1996; Rogerson, 1997; Arnold & Gosling, 1998; OPC, 1998; D'Souza & Wills, 1999), *and component composition* (Bosch, 1997; Buchi & Weck, 1997; Kopetz, 1998; Voas, 1998; Thompson, 1998; Szyperski 1998) sums up the need for and benefits of these new techniques which complement in a vital manner the older, structured techniques. The new techniques are the *basic ingredients* in the development approach of this thesis.

When *reviewing* and discussing *design patterns* (Alexander, *et al.*, 1977; Gamma, *et al*. 1995; Buschmann, *et al*. 1996; Vlissides, 1998), the models for the design work, the approach is both informative and *critical*. The ways in which design patterns, valuable knowledge for developers, are also unnecessarily 'invented' (Aarsten, *et al.*, 1995; Buschmann, *et al*., 1996) are pointed out. The approach in representing frameworks on the other hand, strives to extract the basic *characteristics* of good *frameworks* from practical examples (Dagermo & Knutsson, 1996; Doscher, *et al.* 1998; Hodges, 1998). Also *guidelines* on how to design domain-specific frameworks with the help of design patterns are developed.

Agent technologies are often offered as a generic solution for the needs of functional distribution, integration and flexibility. Accordingly, the original approach of the *survey of agencies* was wide. It encompassed agent architectures (Hayes-Roth, 1995; Musliner, 1995; Ferguson, 1995; Mueller, 1996; Bradshaw, *et al.*, 1997; Genesereth, 1997), various forms of knowledge representation (Genesereth, 1995; Finin, *et al.*, 1997; FIPA, 1998) and the reactive (Brooks, 1991) and deliberative (Fikes, 1993) as well as intentional (Cohen & Levesque, 1990) functionality of the agents. Later, it became clear that the contribution in fulfilling the batch control needs would come rather from *local, problem-specific decision-making* and *multi-agent interaction* (Kuroda & Ishida, 1993; Jennings, *et al.*, 1995; Barbuceanu & Fox, 1995; Chauhan & Baker, 1998). The focus in studying agent technologies was subsequently shifted to these issues.

The *research problem* is *formulated* and the development *approach* is *justified* in detail both from the domain, i.e. *batch process management*, and the *information technology* points of view. This is needed to focus the limited resources on new, experimental design

issues, leaving more conventional tasks, like the refinement of user interface and databases, aside. Quoting one of the foremost researchers of software engineering, Victor Basili, (Basili, 1996):

> *Software engineering is a laboratory science.* It involves an experimental component to test or disprove theories, to explore new domains. We *must experiment* with techniques to see how and when they really work, to understand their limits, and to understand how to improve them. … Our goal is to build *improved products*. However, unlike manufacturing, software engineering is *development*, not production. We do not re-produce the same object, each product is different from the last. Thus the *mechanisms for model building* are different; we do not have lots of data points to provide us with reasonable accurate models for statistical control. … There is a *lack of useful models* that allow us to reason about the software process, the software product and the relationship between them.

In this thesis both *analytic, semiformal* requirements models and *synthetic, design pattern models,* are used as a basis for the *experimental development work*. In an analogous but not similar manner to, for example, physical theories and models explaining the laws of nature, generic design patterns explain recurrent design issues in software engineering. As an experimental physicist observes and measures natural phenomena, so a software researcher should observe and measure the results of software products and processes when testing them and experimenting with them.

In order to test and experiment, the software must be *designed*, here with the help of design patterns, and *implemented*, in this case using software component technology. From the development point of view, design patterns are seen as models with the help of which the framework is to be synthesised. The selection and judicious deployment of the patterns is thus a major concern. Guidelines for this are developed in the thesis. The selection of a given design pattern can even be seen as a minor hypothesis in its own right. It is tested by experimentally verifying its usability in the framework.

While the development of the framework is mainly *synthetic* work, its testing and use or reuse is predominantly an *experimental* task. However, as previously mentioned, there is also a specific need to *enhance the framework by agentifying*, preferably by reusing the basic framework as such. The task was pursued, in concert with the design-pattern based approach, by *developing a generic design pattern* for this domain-independent and recurrent design purpose. This part of the work is *model building*, creating a design model, which is then *experimentally verified* by applying it to a domain-specific problem.

The study is complemented with practical *considerations* of the new information technologies applied and the results achieved in this thesis. This is accomplished by considering first the general implications of components and frameworks to software markets and development processes (Jacobson, *et al.*, 1997; Szyperski, 1998; Brown, *et al.*, 1999; Vayda, 1999). Subsequently, the experience gained in this study is used to *focus on* automation markets, software-design practices, training, and further research and development.

1.4 Contributions

The main contributions of this thesis are: the *refinement of requirements* for new batch process management systems, the *adaptation of new information technologies* to fulfil these needs and, based on the above, the *development and agentifying of an experimental, reusable component framework* for batch process management.

The batch-control domain has been studied in terms of the domain standardisation, existing control systems, and related research approaches. It can be concluded that *domain knowledge* is needed both *in defining requirements* for the framework and *in making design decisions* for it. The *separation of concerns* of product-related and production-related functionality from the equipment-related and control-related functionality and the vital *interactions* between them have been found to form a good basis for new architectural designs.

The applied technology, object-oriented software component frameworks, has been both surveyed and evaluated. *Object orientation and software component technology*, which are the constituents of the development, have been described in a concise manner. An approach to *deploy generic software design patterns to design domain-specific frameworks*, has been developed. It concentrates on the commonalities of all applications in a given domain and strives to make the domain-specific software design comprehensible to both domain and information technology experts.

An experimental batch process management framework, Figure 1.1, has been developed in this thesis, to verify the hypotheses of Chapter 1.2.

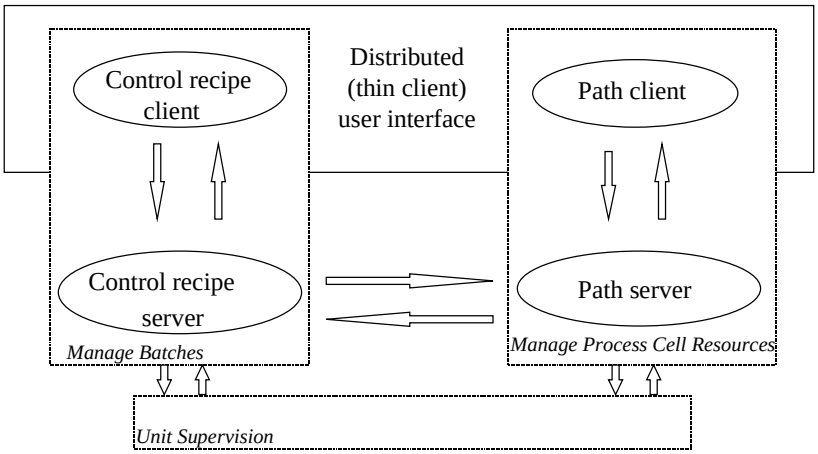

Figure 1.1. An overview of the batch process management framework.

The framework demonstrates the realisation of the *distribution, integration* and *flexibility* needs for batch automation. It also demonstrates *reusability* by the so-called *calling framework* architectural style and external and internal component interfaces in such a manner that *components* can be easily *replaced* by customised versions.

The *architectural, behavioural and distribution aspects* of the framework have been designed with the help of chosen design pattern models in order to solve the batch specific design issues and to achieve the main goals of the thesis. As an experimental part of the work, the framework has also been *reused* by adding application specific components. It has also been *integrated* via standard component interfaces *to a distributed digital control system.*

For some problem-specific needs of local decision-making and interaction (in this thesis, dynamic unit allocation), the framework may have to be enhanced, preferably by reusing the original framework. For this kind of design task no applicable design patterns were found. Since the design problem is recurrent, *a generic design pattern, Agentified Component* was developed and experimented with. This approach retains the deterministic nature of the frameworks, which is especially important in the automation domain but it simultaneously introduces the possibility of solvíng specific problems by a knowledge- based approach.

The research and development work of this thesis has been carried out at *VTT Automation Industrial Automation* within *CAIP – SWF* and *SWFBatch* projects, both of which have also included other tasks, not reported in this thesis. The author participated, with other

members of the project team, in the requirement-definition phase of the batch process management framework. He is responsible for the definition of *Manage Batches* and (the present version of) *Manage Process Cell* control functions. The author alone is responsible for the design and implementation of the experimental framework. In integrating the framework to a DCS, the author has developed the component-based connection concept and designed and implemented the Windows NT resident part.

1.5 Structure of the thesis

The thesis is divided into five parts, each containing one or more related chapters.

Part I. Introduction includes:

Abbreviations

Glossary

Chapter 1. Introduction

Part II. Review of the State of the Art includes:

Chapter 2. Batch Control as a Development Domain. The batch control domain is studied in terms of domain standardisation, existing control systems, and related research approaches.

Chapter 3. Object-oriented Software Component Frameworks. The applicable, fundamental information technology: object-orientation, software components, design patterns, and domain-specific frameworks are surveyed and evaluated.

Chapter 4. Multi-agency. Software agents are introduced, multi-agent interaction methods are surveyed and evaluated, and ways to develop applications having multi-agency features are discussed.

Part III. The Problem Statement includes:

Chapter 5. The Research and Development Problem. The research and development problem of this thesis is first presented in a general way. The problem is then justified in detail both from several domain-specific and information technology points of view.

Part IV. The Proposed Solution includes:

Chapter 6. The Development of the Batch Process Management Framework. The definition, design, and implementation of the experimental batch process management framework is presented.

Chapter 7. The Reuse and Enhancement of the Framework. The scheme of reuse of the framework is presented, the manner by which it was integrated to a digital control system is described, and its enhancement by agentifying is detailed.

Part V. Discussion includes:

Chapter 8. Conclusions. Here the technical results of this thesis are summarised.

Chapter 9. Considerations. The implications of the applied technologies, and the results obtained, for automation business, design and training are tentatively discussed. Outlines for future research and development projects are presented, as well.

2. Batch Control as a Development Domain

2.1 Introduction

This chapter presents the development domain for the batch process management framework. First, the new batch control standard, fundamental for development and research in the domain, is introduced. Then the three leading commercial batch automation systems are presented, which at least partially conform to the new standard. Finally, research work related to this thesis is reviewed.

The frame of reference adapted for the presentation of batch control in this chapter emphasises the need to consider production and product related issues, realised in recipes, separate from process equipment and control issues, realised in equipment entities, Figure 2.1.

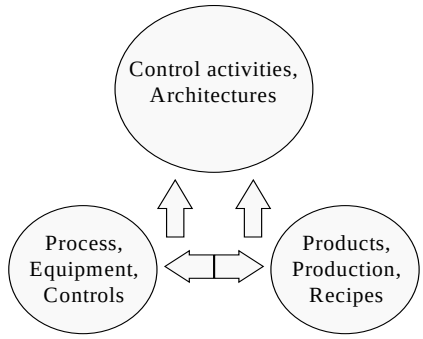

Figure 2.1. A frame of reference for batch control presentation.

It is furthermore argued that this separation of functionality and, which is more important, the interactions between the parts, form an excellent basis when evaluating control activities and commercial system architectures and research concepts for architectures, as well. The above frame of reference is explicit also in the contents of the sections of this chapter concerning batch control standard, commercial batch automation systems and research work. This general approach, is also applied when developing the new software framework for batch process management, as described in Chapter 6.

The approach is both *descriptive* and *evaluative*. The main concepts and artefacts are described and information is given on their application. While describing the commercial automation systems and the latest research developments, a general evaluation of the approaches is given. The closer assessment of the approaches in relation to this thesis will be given in Chapter 5.

Dealing with commercial batch automation systems was considered beneficial, although not absolutely necessary from the research point of view. The commercial systems have a role as *references* for the results of this thesis. They also act as potential *platforms* into which parts of the experimental software framework, developed in this thesis, can be embedded. This chapter is not, however, meant to be an all-encompassing survey of the field. Only such issues of the batch control domain that are relevant from the process management research and development point of view are covered.

2.2 Standardisation

2.2.1 Background

Industrial processes are usually classified as *continuous* or *sequential* (discrete). In continuous processes, the goal is to keep the process conditions constant for longer times whereas sequential processes are characterised by frequent and planned transitions between operational states. *Batch processes* are a subset of sequential processes. They deliver their products in discrete amounts of material called *batches*. Input materials are transformed into products in an ordered set of processing activities carried out in one or several pieces of equipment (Rosenof & Ghosh, 1987; ISA, 1995).

Currently, about 50% of the industrial processes include batch processing. The main industries are the manufacturing of pharmaceuticals, food and beverages industry, metallurgical industry and chemical industry. The increasing emphasis on specialized high-technology products, customer service, and product quality all highlight the benefits of batch processing.

Small plants can be economically re-sized according to changing market demands. Multipurpose process units allow the flexible production of many different products. When units are correctly sized and organized as networked production lines, batch plants can produce efficiently several batches of different products in parallel. The quality of each batch can be analysed, and corrected if necessary, before the delivery to the customer.

A challenge in batch processing is its *complexity*. At any instant, a large plant may contain several different batches of various products and each batch may be at a different stage of processing. Process equipment must be designed for various potential configurations and operating conditions. Batches must be scheduled and process equipment, raw materials and other resources must be allocated to them. The sequential operations of the batches must be monitored and coordinated. Shared resources, like utilities and manpower, must be managed. Finally, historical records must be collected and maintained for each batch.

Unlike traditional regulatory controls, there have been no widely accepted theories, practices or control products for batch applications. Fortunately, this situation is changing. Since the early 1980's, terminology and models have been developed, first by the German NAMUR (Interessengemeinschaft Prozessleittechnik der chemischen und pharmazeutischen Industrie) and then by the Instrument Society of America (ISA, 1995) and the International Electrotechnical Commission (IEC, 1997). Recently, World Batch Forum (WBF) and to some extent also European Batch Forum (EBF) and Japan Batch Forum (JBF) have contributed to the standardisation work of ISA and IEC.

The purpose of the standardisation efforts has been to *emphasise good practices* and to *improve communication* between collaborating parties. This helps to reduce the time to market for new products and the costs across the process and product life cycles. The basic concepts have been well accepted by many leading batch automation system manufacturers as well as many engineering companies designing and end user companies using batch control in their production. The research community within the domain has accepted – and contributed to the development of – the new standards.

The treatise in this chapter is based mainly on the standard ISA-S88.01-1995 (ISA, 1995), which is the first part (Models and Terminology) of the new ISA/IEC Batch Control standard. More detailed data models, recipe representation and, data exchange format are being worked out in the second part of the standard (ISA, 1999a), of which currently a thirteenth draft has been published and utilized also here, where appropriate.

According to the standard the main independent functions of a batch control system can be divided into *procedural control* and *equipment control*. The task of the procedural control is to accomplish the processing activities of batch production defined in the product related recipe. To enable this, it uses services provided by the equipment control. In the next section, batch process, equipment, and equipment control are described. Then the format, classification and use of recipes are presented. Last, the *control activities* in

batch control, integrating equipment control and procedural control aspects with the help of *co-ordination control*, are described.

2.2.2 Batch process, equipment, and equipment control

The term *process* refers to a sequence of chemical, physical or biological operations for the conversion, transport or storage of material or energy (ISO 10628, 1992). These activities are performed by a physical *process plant* consisting of the necessary equipment. ISA standard (ISA, 1995, pp. 18 … 24) divides a *batch process* hierarchically into *process stages*, *process operations* and, finally, into *process actions*. The physical equipment model comprises *process cells*, *units*, *equipment modules*, and *control modules*.

Process cells are logical groupings of equipment that include the equipment required for production of one or more batches. They can be categorised as single product and multi-product process cells. The process cells can also be divided according to path (or train) structure such as single path, multiple path, or networked process cells. Figure 2.2 shows an example of a single product, multiple path process cell of a fictive fine chemical factory.

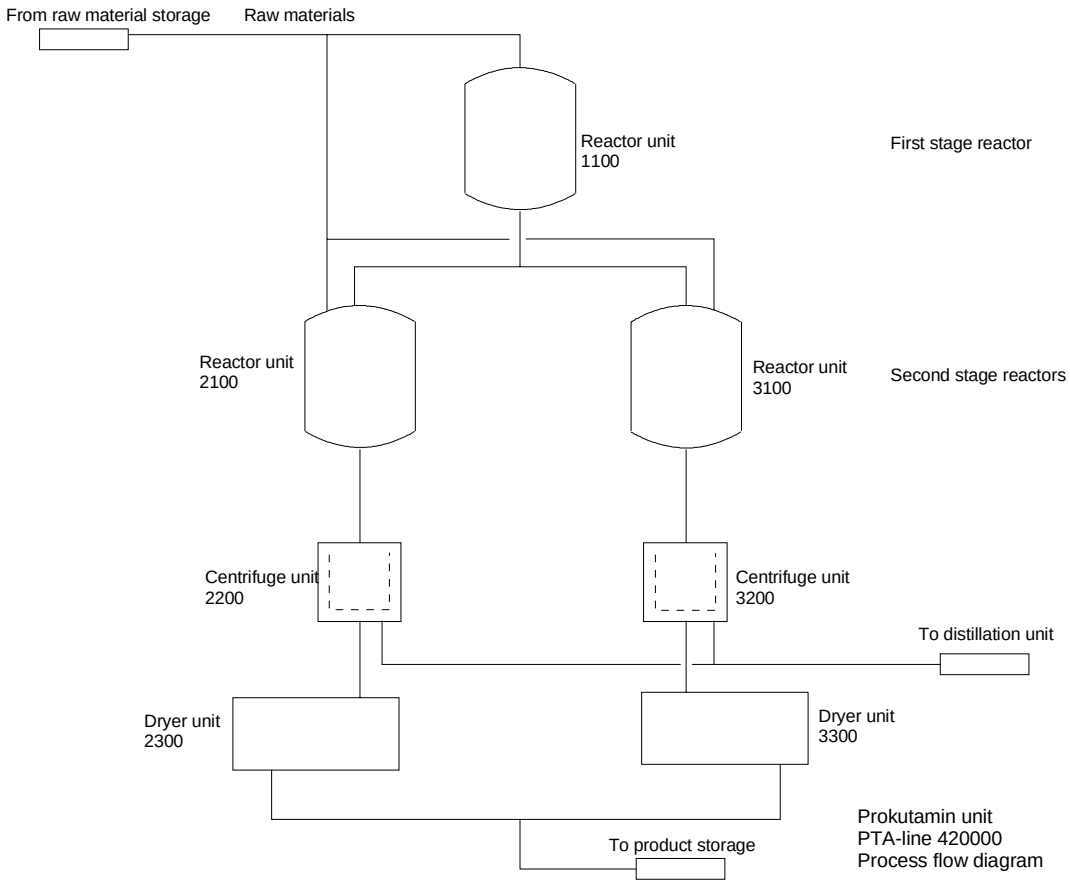

Figure 2.2. A part of a fine chemical process cell, modified from ATU, 1992.

In a batch process cell, each *unit* is usually capable of processing several products, and each product can, with certain constraints, be produced by more than one set of units. A unit is a collection of associated control modules and/or equipment modules and other process equipment in which one or more major processing activities can be conducted.

The *types of control needed* in batch processes are *basic control, procedural control, and co-ordination control*. These control types are run together to enable the production of various products using the same set of process equipment. Interoperation of the control types is implemented by exchanging command messages and the respective responses between the corresponding control activities or the (sub)systems implementing the controls.

The *basic control*, including discrete-logic control, interlock control, regulatory control, and exception handling, does not differ functionally from continuous processes. The main difference is in the requirements for flexibility and adaptability due to a greater amount of external stimuli. For example, changes in setpoint values are frequent due to the fact that they may differ on different steps of a recipe procedure. In addition, the amount of exceptions is normally greater and their nature more complex than in continuous processes. Basic control may also include equipment-phase logic, containing a set of steps to accomplish a process action, for example adding material to a vessel. It is important to note, however, that this kind of sequence in basic control is equipment-specific, not product-specific.

Product-specific *procedural control* is typical of batch process control. A recipe procedure is decomposed into unit procedures, which are further decomposed into operations and phases. There are also various possibilities to decompose a procedure. In simple cases there is no need for operations and unit procedures, in complex cases additional levels (for example non-standard super- and sub- operations and macro phases) may be needed. Sequential function chart (SFC) or preferably process function chart (PFC), developed within ISA-S88.02 (ISA-TR88, 1996; ISA, 1999a, pp. 120 … 140) can be used to represent the procedural logic in a comprehensible and easily modifiable manner.

Co-ordination control directs the recipe procedure of a batch, as well as, the way in which it uses the process equipment. It contains the allocation and arbitration of equipment and other resources to batches. It also manages the use and allocation of common equipment, the selection of a specific path through the equipment for a batch, and the co-ordination of multiple simultaneous batches in a process cell.

A fundamental concept in the batch standard is the *equipment entity* that contains both the process equipment and the equipment control needed to control it. The control is seen from outside as a service provided by the equipment, so called *equipment capability*. The use of these capabilities to accomplish processing tasks is important in batch processing. In practice, the capabilities are usually implemented as *equipment procedures*, i.e. pre-programmed sequential controls. For example, a reactor unit may have preprogrammed sequences for mixing and charging raw material.

The standard (ISA, 1995) uses – somewhat misleadingly – the same names for the equipment entities and the process equipment belonging to them. Thus a process cell, a unit, an equipment module and a control module mean both so-called intelligent equipment provided with controls and physical process equipment, as such. The physical

process equipment can be used to produce products only by controlling them manually or by automatic control in a manner described below. The intelligent equipment entity, on the other hand, can be regarded as a (set of) intelligent resource(s). The *services* of it can be combined in order to perform the different processes required in production of various products.

Process cell control may contain the basic control of several units and an execution of a control recipe procedure as well as invocations of unit procedures. On process cell level most important is, however, co-ordination control, especially resource allocation, since each process cell contains normally several process units and a process cell produces several batches simultaneously. Furthermore, decisions have to be made about potential paths for a given batch and also potential longer-term reservations may have to be co-ordinated in beforehand.

Unit control contains, for basic control, setpoint calculation for equipment modules and control modules. For procedural control, the unit equipment control either executes in an autonomous manner unit procedures or runs lower level controls with the assistance of the control recipe. The process unit may need co-ordination control as well, for example when using common resources and when interacting with other units by using unit-to-unit requests.

Equipment and control module control is mainly basic control that controls directly actuators and other control modules. Equipment and control module control does not perform product specific procedural control and the contribution to procedural control is limited to specific mode transfers and responses to service requests.

According to the standard (ISA, 1995, p. 23), *a unit shall only handle a single batch at a time*. This is somewhat controversial and some colleagues in the domain would like it to be relaxed. There is, indeed, no absolute necessity for this constraint from the end user point of view. The constraint seems to be included in the standard to make the implementation of a (standard conformant) batch automation system and projectwise designs of equipment controls clear and manageable.

2.2.3 Recipes

A *recipe* consists of the information needed to uniquely define the production requirements of a specific product (ISA, 1995, p. 35). Thus the recipe defines both the

product and the way in which it is produced. The general structure of a recipe is shown in Figure 2.3.

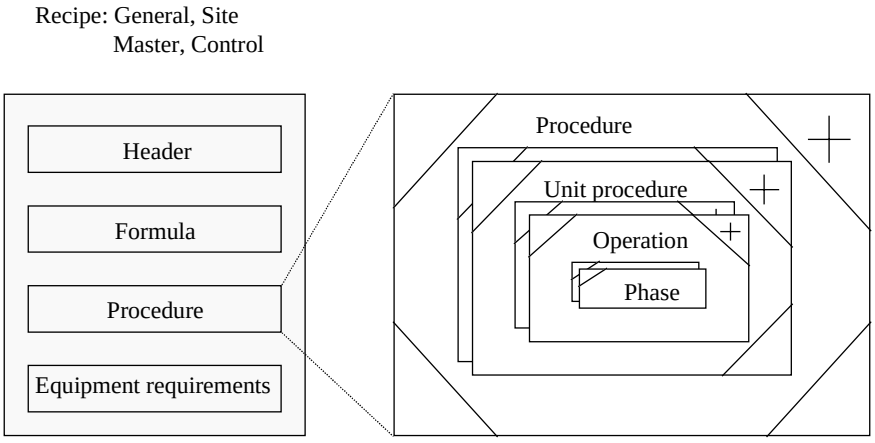

Figure 2.3. The structure of a recipe.

The *header* contains management information such as the name, version, status, date and author of the product. The *formula* specifies the process inputs (information on the materials, energy and other resources needed) and the expected outputs (information on the products, byproducts and waste products). It also defines the required process parameters that are needed for the characteristics of the products or processes (like the temperature setpoints, flowrates, mixing times and reaction times). The formula thus defines, directly or indirectly, the *parameters* to be transferred to equipment procedures during recipe execution.

The *recipe procedure* defines the sequential and parallel actions needed to produce a batch of a certain product. It can be hierarchically decomposed into *recipe unit procedures*, *recipe operations* and *recipe phases*. Recipe procedures can be created by combining pre-defined library elements with pre-programmed equipment procedures included in the equipment control. *Equipment requirements* are used to select the specific process equipment for a given batch. The selection can be based on the equipment type, equipment properties, material transfer paths, or specific unit names.

The standard recognizes four types of recipes according to their level of detail and equipment specificity. *General recipes* describe, from a chemical point of view, the process needed for transforming raw materials into end-products. General recipes are

applied enterprise-widely, indicating material and processing requirements for multiple manufacturing sites.

In *site recipes*, plant specific information, such as the available process equipment, is added. They correspond approximately to the written instructions given to plant operators in manual or semiautomatic production. Site recipes are derived from the information in a general recipe by taking into account local material availability and processing capabilities. The management of general and site recipes is an engineering design activity, not a control activity (Brandl, 1998) and these kinds of recipes can be used for enterprise-wide and site-wide production scheduling, respectively.

Master recipes are type specific definitions of the processing steps that must occur in a given process cell, using its control capabilities. They are created based on the information specified in a site recipe or in a general recipe, the specific processing capabilities of the process cell and the possible material flow connections between units. The master recipes can be stored in a *recipe library* and transformed into executable control recipes.

The creation of *control recipes* from master recipes includes, for example, the definition of a batch identifier, the allocation of process equipment, and the scaling of recipe parameters according to the batch size. The master recipe, acting as a template for control recipes, defines the possible actions in a process cell and the potential paths through the cell. The executing control recipe defines the specific actions and the actual path for a given batch.

A recipe contains production-related information for the manufacture of a specific product but it does not contain actual scheduling information or equipment control information. Thus, by keeping the *product related information as separate as possible from the equipment control* it is possible to produce the same product starting from a single recipe and using different equipment entities. It is also possible to produce various products with their respective recipes by using a single equipment entity.

What some researchers find somewhat difficult to accept, is the constraint that S88.01 standard specifies exactly *four levels* in the recipe procedure hierarchy (procedure, unit procedure, operation, phase). This makes, however, a recipe procedure understandable from the user point of view and is thus necessary in the first part of the standard (Models and Terminology) that should be comprehensible to all users of the standard.

Unambiguous levelling is also important from the design point of view, as exemplified well in (Fleming & Schreiber, 1998) and described in Chapter 2.4. The constraint of the four recipe hierarchy levels is, however, properly relaxed and the level concept generalised in the Data Models section of the (draft of) second part of the standard S88.02 (ISA, 1999a). S88.02 is intended mainly for the use of system suppliers, integrators, and researchers.

Another issue that will be clarified in the second part of the standard is *recipe representation*. A Procedure Function Chart (PFC) representation is introduced as a standard and well-defined way to represent a recipe procedure for both master and control recipes. It is based on a proven representation, Sequential Function Chart (SFC), which is defined in the standard IEC 60848. The S88.02 working group takes the stand that SFC should be the preferred language in displaying and implementing equipment phases, whereas PFC should be used on other levels of the hierarchy.

The greatest difference between PFC and SFC is that in PFC, each step of the procedure can independently take care of its own completion, which need not be specified with an external transition with respective transition conditions, like in SFC. The new PFC representation also makes a graphic notational difference between a procedure, unit procedure, operation and phase and indicates with notation (with a + sign) if there are lower level procedural elements in a given procedure, unit procedure or operation, as can be seen in Figure 2.3.

2.2.4 Batch control activities

Both equipment entities and control recipe procedural elements may, at any time, be in various *modes* and *states*, which together define the status of an equipment entity or a procedural element, respectively. The modes specified in the standard (ISA, 1995, p. 57) are automatic, semi-automatic and manual for procedural elements, and automatic and manual for the basic control functions of equipment entities. The procedural elements and equipment entities may change their mode due to internal or external stimuli and these changes may propagate mode changes in other procedural elements or equipment entities.

Twelve *states for recipe procedural elements* have been defined (idle, running, complete, pausing, paused, holding, held, restarting, stopping, stopped, aborting, aborted). The states and transitions between them, Table 2.1, are vital for the batch control functionality.

Table 2.1. State transition matrix, recipe procedural elements, modified from ISA, 1995.

Command	**No Command**	**Start**	**Stop**	**Hold**	**Restart**	**Abort**	**Reset**	**Pause**	**Resume**
Initial state									
Idle		Running							
Running	Complete		Stopping	Holding		Aborting		Pausing	
Complete							Idle		
Pausing	Paused		Stopping	Holding		Aborting			
Paused			Stopping	Holding		Aborting			Running
Holding	Held		Stopping			Aborting			
Held			Stopping		Restarting	Aborting			
Restarting	Running		Stopping	Holding		Aborting			
Stopping	Stopped					Aborting			
Stopped	Aborted					Aborting	Idle		
Aborting									
Aborted							Idle		

Procedural elements and equipment entities typically react differently to external commands and other stimuli, for example failures or other notifications from other entities, while being in different states. State transitions are additionally often propagated

in complex ways between various procedural elements and equipment entities. State transition matrices like the one above, are helpful in analysing and designing state-dependent batch control functionality. They can be used, as indicated in Chapter 6.4 of this thesis, in deciding which states, if any, can be aggregated as a single entity based on similar or analogous behaviour.

Possible states for equipment entities are for example: on, off, closed, open, failed and available. Both the modes and the states of the equipment entities have been presented in the standard as examples. All the presented modes and states are not needed in every application and, on the other hand, some applications need additional states. By using the modes and states presented – when they are applicable to the application in question – one can, however, ease the communication on these issues.

Batch control is decomposed in the standard into *control activities* according to Figure 2.4, below. Compared to the original figure in the standard, the control activities *Process Management*, *Unit Supervision*, and *Process Control* are here expanded with respective *control functions*. The control functions *Manage Batches* and *Manage Process Cell Resources* are especially interesting from the framework development point of view, Chapter 6. The control activities Unit Supervision and Process Control are referenced in Chapter 7 from the framework use point of view.

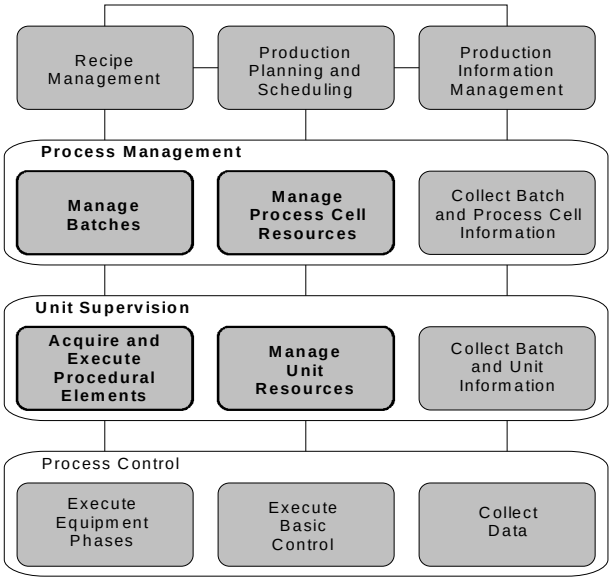

Figure 2.4. Batch control activities, modified from ISA, 1995.

Recipe Management contains all activities involved in the creation, storage and maintenance of general, site and master recipes. The end product of the control activity is the master recipe, which will be used by Process Management. Creating a master recipe requires knowledge of the processing and control capabilities of the process cell. Recipe management also includes the co-ordination of the recipe information with the scheduling information.

Of the tasks of *Production Planning and Scheduling*, the standard encompasses planning of a batch schedule for each batch to be produced. This activity uses higher level (enterprise and plant-wide) production plans, recipes from Recipe Management and resource availability information from Process Cell Management, to design a batch schedule using a chosen scheduling algorithm. The feasibility of the planned schedule is also ascertained.

Production Information Management collects, refines, stores, and reports the information originated in batch production. Of the resulting reports, Production Planning and Scheduling needs actual production summary information and information on resource usage. Recipe Management, on the other hand, needs comparisons between the planned and the actual production. Batch Management needs information on the process history and also planned – actual comparisons. Users and external information systems may need, additionally, information concerning product and production quality and potential failures.

Process Management contains two major control functions, in addition to batch and process cell information collection. *Manage Batches* is a control function in which a control recipe is created from a copy of a master recipe, a batch is initiated based on the scheduling information and operator input, and the execution of the batch is supervised. *Manage Process Cell Resources* is a control function in which process cell resources are managed by allocating units and other equipment, by arbitrating multiple requests for the same equipment, and by providing a mechanism to control unallocated equipment. The process management activities are mostly focused on co-ordination control, since there may simultaneously be several batches in production within a process cell.

Unit Supervision is the control activity that ties the control recipe execution to the equipment control via *Process Control.* Unit Supervision executes unit level recipe procedures and commands of Process Management by initiating and parametrizing equipment procedures. To accomplish the required tasks, Unit Supervision manages its own dedicated resources and may also request services from other units. On the other hand, it delivers information about the production of units to Process Management.

The lowest level, *Process Control* encompasses sequential, regulatory and discrete control distributed among the units, equipment modules and control modules. Regarding the execution of recipe procedures, the main function of the process control is to execute *equipment phases* according to the commands received from Unit Supervision. Phases do not interact with physical process devices directly, but through *basic controls*, e.g. PID-controllers included in control modules.

Batch control activities in a specific plant can be implemented by different batch automation (sub)systems from various vendors. That is why the draft of the second part of the standard has defined a means for Data Exchange, primarily for the interfaces between the control activities:

- Recipe Management to/from Process Management
- Production Planning and Scheduling to Process Management
- Production Information Management from Process Management

The data exchanges (form and format) are defined by database schemata, which are described by standard query language (SQL) as *exchange tables* (ISA, 1999a, p. 44 … 64). The draft standard does not specify how and by which tools the exchange tables are implemented or how they are utilised.

It is, however, questionable to specify data exchanges with conventional SQL-tables, when the future batch control systems will probably use method invocations of *software component interfaces* and object databases. The implementation-oriented convention, brought about by the SQL-tables into the draft standard, is an unnecessary constraint that may later prove to be obsolete.

The standard does not constrain the data structures or data exchange within the control activities, which is reasonable. Each vendor can freely design the internal structure of a control activity. Also the communication within the control activity can be implemented in various ways; e.g. using file transfers, SQL-queries, and method invocations of software component interfaces. This offers vendors many opportunities to supply customers with solutions designed and implemented even with the most advanced software technologies.

2.3 Batch control systems

2.3.1 Background

It is evident that the new batch standard (ISA, 1995), will become a reference to all information and automation systems in the domain, see e.g. (Haxthausen, 1998). Various software products will converge to deal with the same batch concepts and to use the same terminology. This eventually increases the *openness and competition* in the market. Various products can, on the one hand be integrated to each other, and on the other hand, be compared with each other according to the functionality they provide, by using the standard as a reference model.

The first part of the standard SP88.01 is particularly aimed at being a reference model but also the standard draft dSP88.02 is useful as it completes and refines the first part with the *data models*. If the vendors will make their data available to other applications complying with these models, it is possible that one system can generate data that can then be used in another system. Also the data exchange formats applied are important for the interoperability between the systems.

However, only when the batch control systems will be architecturally based on software *component technology*, discussed in the next chapter of this thesis, it will be possible to successfully mix and match individual components, developed by various vendors. These commercial-off-the-shelf (COTS) components may then be combined with self-made components using batch automation frameworks to best achieve the specific business goals of a plant.

It is useful and instructional to compare the three leading commercial batch control systems (InBatch by Wonderware Corp., VisualBatch by Intellution Inc. and OpenBatch by Sequiencia Inc.) using the reference model provided by the standard. In addition to providing an example on how to use the reference model of the standard (ISA, 1995) in system comparisons, the comparison also gives information on how to incorporate ‘foreign’ software components into these systems.

The comparison is made so that first batch equipment related support is discussed in the section Batch process, equipment, and equipment control and then recipe related support is discussed in the section Recipes. Finally, the architectural structure, the high level implementation of batch control activities, provided by the automation systems, is discussed, first in general, then considering the systems as component platforms. This

approach is compatible with the approach of the previous chapter, although not the one that would be conventionally used by batch control system vendors.

2.3.2 Batch process, equipment, and equipment control

In *InBatch*, there are three ways to develop a model for a batch plant (InBatch, 1998). Either a so-called comprehensive model or a connectionless model, or a hybrid of them can be used to configure the process model. A description of the approaches and the respective benefits and liabilities are presented below.

In the *comprehensive model*, a physical process is defined using units and the connections between them. The units are defined as in the standard; the connections are comprised of material transfer equipment, i.e. pumps, valves, etc., necessary to transfer a batch from one unit to another. Plants may have units that are connected to more than one unit and some plants have also multiple connections between two same units. The connections which are included in the model can be further divided into segments.

All units that have the same processing capabilities are grouped in the same *process class* (named somewhat misleadingly). All connections between the same two process classes are grouped in a *transfer class*. In addition to process classes, specific instances of each of the process classes *may* be defined. These *process instances* for the units *must* be defined always when more than one unit of the same class is used in a given recipe procedure. When process instances are defined, the resulting transfer instances must also be defined. The batch management system is responsible for co-ordinating unit to unit material transfers when it executes a recipe, which refers to process classes only.

In the *connectionless* model of InBatch, a physical process is defined with units only. All units that have the same processing capabilities or perform the same function are again grouped into the same process class. *Connections are not defined* when using this approach. The movement of material between units is accomplished using so called complementary process phases, which are added to normal phases, needed to implement the actual processing operations.

In the *hybrid* model, a physical process is defined with units and connections. However, only the static, non-flexible material paths are defined as connections. Flexible paths, i.e. the paths that involve many possible destinations are not defined as connections. Process classes and respective units, as well as transfer classes and connections are defined as in

the comprehensive model. The flexible paths will use the complementary process phase approach described in the connectionless model. Thus, this approach minimises the number of connections that exist in the model, while still preserving the connections for the paths that are constant.

In InBatch the batch management system is responsible for executing batches. The batches consist of a recipe and a train assignment. The recipe is typically equipment independent, referring to process and transfer classes only. The *train* is used to provide a list of potential equipment to the batch management system for dynamic selection while the batch is executing. If a unit is not in the assigned train, it is not available to be used for the production of the batch. A train can contain one or more units, and a unit can be a part of multiple trains. Trains provide thus in InBatch a standard compliant way to represent various paths through the process.

In *VisualBatch*, a batch plant is modelled by first defining an *area* to represent the extent or scope of the plant model. In the area, one or more *process cells* are defined to contain the equipment needed to produce a batch of a given product. Then the *units* that reside in the process cell are defined. For each unit, the *equipment phases* that execute on the unit are defined (VisualBatch, 1998).

If there are several *identical units* in the plant, a *unit class* for each type of unit in the plant can be defined. For example, mixers, heaters, and reactors could be unit classes in VisualBatch. For each unit class, the class properties are defined. Then a unit instance for each physical unit in the process cell is defined in an object-oriented manner by giving values to the unit properties. The use of unit classes lets recipe authors build class-based recipes, similarly to the procedure in InBatch.

Once the units have been defined, they can be linked together. The *linking of the units* enables phases to communicate across units. When the phases must communicate across the units, VisualBatch verifies that the units are linked at runtime. Linking units defines also the path (or the train) between units similarly to InBatch.

VisualBatch handles multiple requests for the same resource by allowing the operator of the system to configure *equipment arbitration*. Equipment arbitration co-ordinates the allocation of resources when there are more requests for a given resource than can be served at a specific instance of time. Configuring arbitration consists of defining for each piece of equipment in the plant: its equipment identifier, maximum owners, and other equipment needed for execution.

In *OpenBatch* (OpenBatch, 1999) a batch plant is modelled as areas, process cells, units, equipment modules and control modules *strictly according to the batch standard* (ISA, 1995). In addition to unit classes (like in InBatch and VisualBatch), also *process cell classes* and *equipment and control module classes* are introduced. The concept of equipment entity (or equipment class) as a collection of equipment that has essentially the same capabilities, is well defined in OpenBatch. It simplifies both the configuration of similar equipment and allows recipe procedures to be configured to work with a class of equipment instead of a specific piece of equipment.

When starting the batch equipment definition in OpenBatch, the first step is to define at least one process cell class on a blank template representing the specific area. In addition to the name and icon of the process cell class, also specific process cell instances are defined. Then so-called *arbitration* information for each process cell instance is defined. It is a list of required resources that must be available prior to starting any procedure on this specific process cell. In this way, one process cell class can describe several instances, each having, for example, various raw material supplies.

The selection of specific *unit instances* from unit classes takes place in OpenBatch in a similar manner than the configuration of process cell instances. In addition to unit name and arbitration information, also associated data tags (corresponding to control system data points) are defined. At the same time as units are added, also possible inter-unit flow paths can be defined graphically.

InBatch, VisualBatch, and OpenBatch comply fairly well to the standard as to the modelling and terminology of the batch plant. VisualBatch and especially OpenBatch describe the physical equipment model in a more thorough and object-oriented manner (properly named classes and instances) than InBatch. The terminology in VisualBatch and OpenBatch is strictly compliant with the standard. InBatch also introduces non-standard vocabulary, presumably due to compatibility to older versions of the product.

2.3.3 Recipes

InBatch contains a recipe management system (InBatch, 1998) that enables master recipes to be constructed and edited. A master recipe becomes a control recipe when it is assigned to a train and initialised. The control recipe is process cell specific. InBatch provides table driven and graphical editors to construct and edit recipes. Recipes can be

saved, retrieved, and printed. A revision history capability enables the users to enter, save, and review the change history for each recipe.

The recipe parts provided by the InBatch recipe editor, are the same as in the standard (ISA, 1995): the Header, the Equipment Requirements, the Formula, and the Procedure. In InBatch, *Equipment Requirements* specifies the process classes (i.e. unit classes) and their attributes. When defining the attributes, the user must specify the minimum and maximum values for each attribute. When a specific characteristic is required, the minimum and the maximum are assigned the same value.

When the trains were defined in equipment definition, it was possible to have multiple destination units available for a given transfer. The *operator* may be allowed to *select* the desired destination *unit*, or the selection is to be done *automatically*. A so-called *unit selection mode* is used to define or change the selection method when defining the equipment requirements for a recipe. The InBatch Recipe Editor automatically inherits all process and transfer phases associated with the process classes defined in the equipment requirements. Only these phases can be used to build the recipe procedure.

Defining multiple instances in equipment definition allows the recipe builder to process in or transfer to multiple units of the same process class. The process class instances can also be assigned their own specific attribute ranges, or a specific unit can be assigned. The *formulas* in InBatch are standard-compliant, consisting of input, output and process parameters. The procedures define the sequence of process actions needed to execute one batch of a recipe.

The *VisualBatch* Recipe Editor displays recipes graphically like the InBatch Recipe Editor. The VisualBatch Editor complies fully with the Batch Control standard (ISA, 1995) and supports IEC 1131-3 symbols and sequential function charts (SFC). In addition to *recipe procedures, operations and phases* like in InBatch, also *unit procedures* are identified as a part of a procedural recipe hierarchy, as also specified in the standard.

Typically, when creating an operation in VisualBatch, a specific unit is assigned to it. This is adequate if the operation runs in one unit only. However, when an operation needs to run in several units, so called *class-based recipes* can be used. A class-based recipe is a recipe that defines the equipment in terms of a unit class. The recipe can be assigned to any unit in the unit class at run-time.

Recipe formulas can be created in VisualBatch on any recipe level. Once created, formula values (or equivalently formula parameter values) are set from the higher recipe level.

Thus, when creating formulas for an operation, a unit procedure or a procedure, their values are set respectively from the unit procedure, the recipe procedure or by the operator when the batch is started.

Also the *OpenBatch* Recipe Editor *complies* fully *with the Batch Control standard* (ISA, 1995) and supports IEC 1131-3 symbols and sequential function charts (SFC). Recipe procedures, operations and phases, are identified as a part of a procedural recipe, like in InBatch. A one-to-many relationship exists between recipe phases and equipment phases. For example, a recipe procedure can require any reactor that can heat and agitate to be used. In various batch runs this phase may be implemented by different reactors (OpenBatch, 1999).

The OpenBatch recipe hierarchy (procedures, unit procedures and operations) *uses SFC* to represent all levels of procedural functionality. On the highest level, only the connections between unit procedures are described within the recipe. Each unit procedure can be opened to reveal its operations, which in turn can be opened to reveal their embedded phases. Looping and parallelism are likewise allowed on every procedural level.

InBatch, VisualBatch and OpenBatch comply fairly well with the standard (ISA, 1995) as to the format and terminology of recipes. The emphasis in recipe management in all systems is on Master recipes and Control recipes. General recipes and Site recipes are considered simply as generalisations of Master recipes. VisualBatch and Openbatch describe the procedural recipe hierarchy in a somewhat more thorough and object oriented manner (using recipe classes and instances) than InBatch. Unit procedures are missing from InBatch. The recipe terminology in InBatch, VisualBatch, and OpenBatch is compliant with the standard.

2.3.4 System architectures

Wonderware's *InBatch* (InBatch, 1998) supports scheduling batches, initialising batches, co-ordinating the execution of batches with the control system, interfacing with operators, and storing all batch activity. The Batch Manager, the Batch Scheduler, and the Batch Display programs implement this functionality.

The *Batch Scheduler* dispatches batches, which are ready to run, to plant floor operators. Scheduling involves a manual entry of the batch identification, the master recipe, the

batch size, and the train into the Batch Scheduler. After the parameters have been entered, each batch is initialised by validating the recipe and checking that the recipe's equipment requirements are satisfied.

The *Batch Manager* directs the execution of each batch. It co-ordinates the usage of process units for each batch. Based on the procedure of the recipe, the so-called blocks of control software are signalled to execute. Phase block control logic, located in the control system, is responsible for controlling the process. If the phase block is ready to be executed, the phase parameter values are downloaded to the block, and the block is started.

The Batch Manager also interfaces with *Batch Display* programs. The Batch Display provides operators with information on all batches initialised and/or executing in the system. Through these displays, operators can put a batch or phase in hold, as well as restart and abort batches or phases.

InBatch uses a relational database as its historical repository to provide access to all batch history data. The Batch Management System writes all information related to the production of a batch to the history database. The retrieval of historical data in the form of reports is provided by the runtime Reporting System. Reports can be automatically triggered during the execution of a batch or at the end of a batch.

There are two ActiveX objects that are provided with InBatch. The SFC ActiveX Object provides a standard SFC visual representation of a batch from within any ActiveX container application. The Batch ActiveX Object provides access to all scheduling and batch management functions from any ActiveX container application. This control provides the functionality for the Batch Scheduler and Batch Display applications.

Intellution's *VisualBatch* (VisualBatch, 1998) uses the Intellution WorkSpace for recipe development from a single location, integrates with Man Machine Interface (MMI) packages, and integrates batch data and recipes into Enterprise Resource Planning (ERP) systems. Both VisualBatch and InBatch use a client-server architecture. A typical VisualBatch system consists of the following computers: a VisualBatch Server, one or more Clients, and a Development Workstation. In addition, if a MMI system is included, the batch system architecture can include one or several MMI Clients.

The *VisualBatch Server* is the batch management system that co-ordinates the functionality of recipes, the equipment database, and each *VisualBatch Client* during the execution. The VisualBatch Server also generates batch event data and communicates

with the relational database, and OPC-aware process hardware. The VisualBatch Workstation is used to develop recipes and equipment database. During the development, the automated batch process can be modelled and tested using a simulator.

When configured to do so, VisualBatch also stores *batch data and recipes in a relational database*. This provides an access to batch data and integrates data into a potential Enterprise Resource Planning system. Integrating data with manufacturing systems can be accomplished in several ways; the data may even be in the same relational database.

The functions provided in VisualBatch Integration Services (VBIS) can also be used to integrate VisualBatch data with manufacturing enterprise data. VBIS is a collection of Application Programming Interface (API) services that allows external programs to monitor and control VisualBatch. When the data is stored in separate relational databases, the data can be integrated by writing a SQL program to query the VisualBatch and ERP data and make joins of them.

The *OpenBatch* batch control system (OpenBatch, 1999) is, unlike InBatch and VisualBatch, commonly integrated into other automation systems as an OEM-product, providing the needed batch control functionality. Thus its software architecture, consisting of seven major subsystems, Figure 2.5, is most interesting from the point of view of system integration.

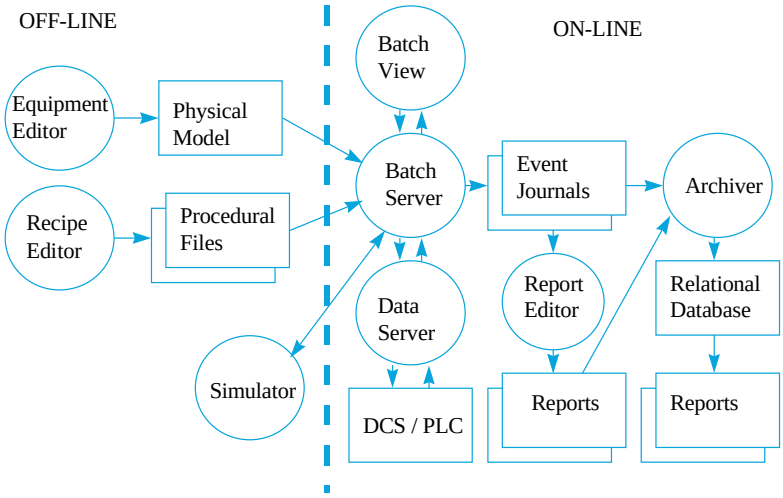

Figure 2.5. OpenBatch architectural overview.

Of the OpenBatch subsystems, the Equipment Editor and the Recipe Editor have been discussed previously. The OpenBatch *View* provides an operator interface to communicate with the OpenBatch Server. Many components of the View are implemented as ActiveX controls, and they can thus be inserted into container applications, notably Web browsers. From the functional point of view, it is interesting that using the View the user can choose one of three allocation scenarios:

- Specify a unit to be allocated before the execution of the recipe.

- Prompt an operator for instantiating a unit from a unit class during the recipe execution.

- Choose automatically the first available unit during the recipe execution.

The OpenBatch *Server* is a centralised execution engine that executes the recipes and co-ordinates communications between the View, the process connected devices, other OpenBatch subsystems and external software packages. The server also provides automatic restart control based on journaling all actions taken, so that a full recovery can be achieved in the event of a control system failure.

The *Simulator* is used to simulate and test recipes against equipment specifications without a connection to the actual process. The *Archiver* is used to transfer electronic batch record data created during the recipe execution to ODBC-compatible databases. The Report Editor is used to create and customise graphical batch reports from batch record data. Phase execution information is available via these reports.

OpenBatch has been licensed by several automation system vendors (VisualBatch has been developed on the basis of a former version of OpenBatch) as a part of their own product, which is a sign of its good interconnectivity and interoperability with other systems. Also ERP and MES systems can be integrated with OpenBatch, as was the case with InBatch and VisualBatch, to provide for example inventory management and tracking as well as automated scheduling.

The architectures of InBatch, VisualBatch and OpenBatch are distributed in a traditional manner, based on the computer hardware distribution. The main functionality lies in servers, which are situated in database oriented computing nodes, in InBatch to the extent that a relational database is an integral part of it. The use of a relational database is optional, but well integrated also in the architectures of VisualBatch and OpenBatch.

All architectures claim to be open. Support for database integration through SQL, application programming interfaces for other applications and most notably OPC-standard interfaces for hardware are important assets of all three systems. Genuine software component based interoperability, in which components originated from various vendors could be composed together, to be discussed in chapters five and six, is still lacking in these commercial systems. Steps to this direction have, however been taken, most notably with InBatch's ActiveX controls, VisualBatch's component base kernel, so called ICore architecture, and OpenBatch's COM-based integration characteristics.

2.4 Related batch research

2.4.1 Background

In addition to industrial research and development, also vendor independent research has profited from the ISA SP88 standardisation. Especially research on process equipment partitioning, recipe structures and architectural concepts have benefited. Also other standards, technologies, and development approaches have been exploited and integrated within these research efforts.

Much of the research in the domain has lately been concentrated on *Petri Net based approaches*. This is because the so-called *safe* nets are easily related to Sequential Function Charts, (Åkesson & Tittus, 1998). Nets are useful, in particular, for analysing concurrency and reachability aspects of dynamic systems. The approaches of various researchers are not coherent, however.

The batch control approach by Tittus and his collaborators from Chalmers University (Tittus *et al*., 1995; Tittus & Egardt, 1996) concentrates mainly on the batch process and equipment design, as well as, on a static and dynamic synthesis of batch operations based on the so-called *Labeled Petri Nets*. In a labeled net, each transition is associated with an event. Recently, the researchers in Chalmers University have also studied deadlock avoidance in the execution of batch recipes (Åkesson & Tittus, 1998).

In Lund Institute of Technology, Årzen and Johnsson have developed the so-called *High Level Grafchart* (Johnsson & Årzen, 1996a,b; Årzen & Johnsson, 1996; Johnsson & Årzen, 1998), a batch control concept based on Sequential Function Charts and *High Level Petri Nets*. A High Level Petri Net allows a compact graphical description of

systems with similar parts. The researchers in Lund have also developed an implementation by using G2 – development platform and tool of Gensym.

Besides Petri Nets, some researchers, most notably Fleming and Schreiber (Fleming & Schreiber, 1998) have concentrated on integration of batch equipment control and procedural control into conceptually sound design practices. Some others have concentrated more into software architectural issues, most notably Simensen, (Simensen *et al.*, 1997). The approach of this thesis is another example on this.

The above-mentioned approaches, which are related to the batch process management functionality, are discussed and evaluated in the next sections. There is, of course, a lot of research concentrating on other batch subdomains, for example on batch scheduling, exception handling, validation, and supply chain management, which are not covered in this chapter.

2.4.2 Batch process, equipment, and equipment control

According to the batch standard (ISA, 1995, p. 34), partitioning the process equipment is a challenging task:

> *Effective subdivision of the process cell into well-defined equipment entities is a complex activity, highly dependent on the individual requirements of the specific environment in which the batch process exists. Inconsistent or inappropriate equipment subdivisions can compromise the effectiveness of the modular approach to recipes.*

Fleming and Schreiber (Fleming & Schreiber, 1998) have taken this challenge in earnest. They have developed a methodology to modularise the batch process starting from P&I diagrams into Process cells, Units, Equipment modules and Control modules so that it supports the development of reusable recipes in an optimal manner. The approach is based on a top down decomposing of process equipment and a respective bottom up composing of recipe procedures, Figure 2.6.

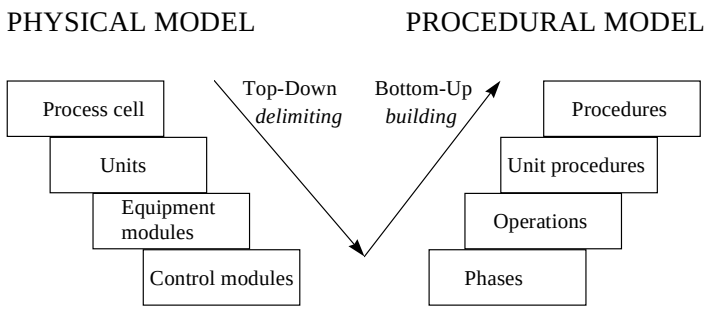

Figure 2.6. Batch process equipment decomposition and recipe composition.

The approach gives guidelines for identifying and *delimiting Units, Equipment modules and Control modules* and identifying and *building the respective Unit procedures, Operations and Phases*. Recipe development takes into account the needed basic, procedural and co-ordination control. Specifically, transfers are seen as important parts of co-ordination control. The approach emphasises the need to either include transfers within communicating units (transfers in and out respectively) that control their own resources or to have equipment modules containing specific phases to control transfers.

Batch plant equipment is modelled in Tittus's approach (Tittus *et al*., 1995; Tittus & Egardt, 1996) with two kinds of resources: *processors* and transfer or *transport devices*. Processors are partly autonomous, container-type units equipped with control devices needed to manipulate a batch. Transport devices, on the other hand, open and close connections between processors, enabling and preventing material flow. The processing capabilities of the plant are thus determined by the capabilities of the various processors and the connections by the transport devices between these processors.

Åkesson and Tittus also introduce a concept of a (*connection*) *line* for deadlock prevention and avoidance purposes as the *set of required transport devices* in order to connect source processors with target processors (Åkesson & Tittus, 1998). For a recipe to move material between two processors, the recipe needs to book the source and the target processor and the corresponding line.

The line is responsible for booking all necessary transport devices. Unfortunately, this kind of an active line *does not conform* to the standard. The need for a set of active devices in between units has also been recognised elsewhere. For example, the transfer

class instances in InBatch's comprehensive model are an implementation of active devices between units.

Another approach, explicit in the batch standard as well as in VisualBatch, OpenBatch, and in Chapter 6 of this thesis, is to avoid using sets of active devices in between process units. Instead, by *including* suitable equipment and control modules in units, it is possible to develop process units *autonomous* enough to take care of material transfer between them. This approach, contrary to that of Åkesson and Tittus, gives possibilities to develop the autonomy of process units further, for example by increasing their communication and even negotiation capabilities as demonstrated in Chapter 7 of this thesis.

In a *mapping model*, in Åkesson and Tittus approach, each processor is modelled as a Petri net with exactly one place. All physically possible connections are modelled by means of transitions between the places. A *synchronisation model* shows the mode of operation of the resource while executing batches. In the synchronisation model, all processors and transport devices are modelled as finite state automata. The states are, however, restricted to only three (unbooked, wait and operate). A *behavioural model* describes the detailed behaviour of each processor. So-called local hybrid controllers denote the termination of an operation, started by the recipe.

2.4.3 Recipes

The emphasis on the High Level Grafchart (Johnsson & Årzen, 1996a,b; Årzen & Johnsson, 1996; Johnsson & Årzen, 1998) is on modelling and controlling the procedural sequence of recipe execution. High Level Grafchart (HLG) differs from the well known Sequential Function Graph in the possibility to include *extended procedural elements* (steps) in the procedural sequence.

The actions contained in the steps of HLG resemble conventional programming language statements, implemented in this case by the action types of G2 tool. An application programmer specifies the transitions between the steps using events and conditions, which tell when the transitions should fire. The events and conditions are translated during compilation into G2 rules.

So-called *macro steps* can be used in HLG to represent procedural elements having a hierarchical structure of (sub)steps, transitions and other macro steps. Sequences, which are executed in several places, can be represented as re-entrant Grafchart procedures,

which can be methods of general G2 objects. If a procedure step is to be started in parallel with other steps, a so-called *process step* is used. The transitions after a process step become fireable immediately after the execution of the process step has been started.

From an object-oriented point of view, an interesting extension in HLG, compared to SFC, are the so-called *Object Tokens*. In SFC, a token simply indicates whether or not the step is active. In HLG, an object token, instantiation of an object token class has attributes, which may be altered from inside step actions. The Object tokens are inspired by High-Level Petri Nets.

Johnsson and Årzen have presented (Johnsson & Årzen, 1998) *four different recipe structures* with their advantages and drawbacks: Control recipes as function charts, Control recipes as object tokens, Multidimensional recipes, and Process cell structured recipes. This indicates the expressiveness of their High Level Grafchart approach. On the other hand, the use of this expressiveness instead of the batch standard's approach of separating recipes from process equipment arouses questions of standard conformance.

In Tittus's approach so called *mapping recipes* are generated from product specific general recipes by transforming them into a Petri Net form. The amount of variations of recipes is then minimised by *synchronising the result with the Petri Net model of the plant*, thus creating in effect a product and plant equipment specific set of recipes. To obtain a suitable representation for control, the set of recipes is then translated into another representation, a so-called synchronisable master recipe, consisting of *building blocks*, each controlling an operation or a material transfer.

The *synchronisable recipes* describe alternative specifications for the so-called event sequences through the plant. Collectively, they form a joint specification of the system behaviour, which can be utilised to *generate a supervisory control algorithm* for the plant. The generated algorithm is theoretically complete (following all uncontrollable events) and trim (completing all specified batches) but it assumes a fully deterministic plant.

Both of the research approaches described above, have their main *emphasis on analytical modelling* and only to some extent on a synthesis of procedural control, i.e. sequential functioning of batch processes. Petri Net based methods are, indeed, appropriate for analysis purposes and the approaches above continue the research tradition started already when developing the SFC standard.

2.4.4 Architectural concepts

The approaches described in the previous sections have also succeeded in exploiting some of the benefits of (non-distributed) object orientation in the implementation of the batch control architecture. In Årzen/Johnsson approach this has taken place mainly in terms of object oriented G2 – methods from within recipes and encapsulation of synchronisation functionality by object tokens.

In Tittus's approach implementation inheritance has been utilised when modelling process and control equipment (equipment classes). The distributed computation aspect has not been considered in the approaches, in addition to the inherent modelling capability of parallel activities with Petri Nets.

A more extensive object-oriented modelling approach is reported by Simensen (Simensen *et al.*, 1997) which resembles the generic OMT modelling (Rumbaugh *et al*., 1991). In Simensen's modelling approach, an overall Information Model for Batch Control is divided into a Functional Model, a Domain Model and several Dynamic Models. The Functional Model consists of ISA S88.01 Control Activity Model, described in Chapter 2.2, extended by the so-called Organisation information and Engineering control activities, the contents of which are not, however, further detailed.

The Domain Model is an object-oriented representation of static structures of a batch plant. It consists of *batch objects, resource objects* (i.e. process equipment) and *control objects* (i.e. recipes). Some objects are so-called *hierarchical or composite objects*, objects aggregated of ordinary objects. For example, a process unit is regarded in this context as a hierarchical object, made of basic equipment objects and having, for example, temperature control capabilities. The composite object may have an icon of its own and it may be connected to other objects. A *system*, an entity implicated to a flow of material, energy or information, can also be represented as a composite object.

The Dynamic Models are divided into two categories: *normative or control models* and *descriptive models*. Control models express how a batch process is supposed to operate by design. Examples of these models are recipes, State Task Networks, Petri Nets and High-Level Grafcharts. The term descriptive model is used for what actually happened, i.e. batch histories and equipment logs.

In the dynamics of batch production, modelled with the dynamic models above, it is considered problematic, how to represent *a batch as a totality* during its lifecycle in its relevant so-called *views* (as an order, as a control recipe, as a batch history). Dynamic

inheritance, where a batch as an entity belongs to various classes over time, is proposed to solve this problem. It may be argued that this approach is more object-oriented than the standard since it tries to encapsulate recipe and equipment control into one entity.

The *ISA S88.01 standard* (ISA, 1995), however, *takes another stance* in the integration aspect of the batch. It maps the procedural control represented by recipes with capabilities provided by individual equipment entities, even with several alternative mappings (levels of collapsibility), but keeps these entities *separate*. In this sense it emphasises the independence of both recipe and equipment control as entities of their own. The standard further recognises state and mode changes separately for both recipes and equipment entities.

3. Object-oriented Software Component Frameworks

3.1 Introduction

This chapter presents the main technological background for the batch process management framework of this thesis. First, object orientation and software component technology, the basic constituents, are described in a concise manner. Then, the design pattern approach for object-oriented software design is presented as an important means to develop frameworks. Finally, the domain-specific software frameworks are introduced, both by definition and by two representative examples, relevant in the domain of the thesis.

A frame of reference in this chapter, adapted to present the relevant software technologies emphasises the need to consider both the *inheritance* based techniques of object orientation and the newer *composition* based software component technology, Figure 3.1.

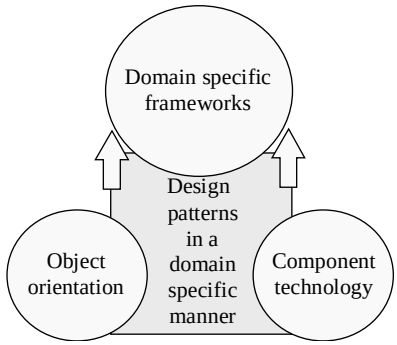

Figure 3.1. A frame of reference for technological background presentation.

It is, however, argued that instead of pure inheritance or composition, a better way to build frameworks is based on a *domain-specific* deployment of generic design pattern methodology. Batch control as the application domain has been presented in Chapter 2. The frame of reference is seen in the contents of this chapter's sections concerning objects and components, design patterns and domain-specific frameworks. The approach adapted is used when developing a new architectural framework for batch process management, described in Chapter 6.

This chapter does not deal with object modelling techniques, employed in software requirement definition and analysis. Those techniques, important as they are in all software development, are applied similarly from the technical modelling point of view both when developing frameworks and when developing ordinary applications. However, using commercial-off-the-shelf (COTS) *components* in the development process brings about the need to consider their granularity already in the analysis model of the framework or application.

Furthermore, when developing a domain-specific component framework, the *focus* of the analysis is on a *family of applications* in a domain rather than on a particular application. Therefore, concentrating on *commonalities* in the domain (in this case in batch process management) is needed. Generic object modelling techniques are employed using *domain analysis* point of view in developing the batch process management framework. The use of the object models is reported in Chapter 6.2, Logical Models.

The approach in this chapter is both *descriptive and evaluative*. The main technologies are presented both by defining the concepts and by giving descriptions on their application. When discussing the latest research developments, related to the theme of this thesis, also a general evaluation of the approaches is given. The justification of the approach of this thesis in relation to the presented technologies is postponed to Chapter 5.

The entire software development process or organisation is not a research topic of this thesis. They are, however, affected by the reuse oriented, component based development, and are thus discussed tentatively in Chapter 9. The *design guidelines for domain-specific component frameworks*, which are important for achieving the goals of this thesis, have been developed and are presented in Chapter 3.4.

3.2 Objects and components

3.2.1 Object orientation in short

Traditionally, software in automation and control domain was developed by using structured analysis and design. The main characteristics were separation of functionality and data, as well as decomposition as the principal means for advancing in the development process, see for example (Kuikka, 1985).

Object orientation has brought about the possibility to encapsulate functionality and information together into entities, objects, belonging to classes which define both the

properties (attributes, or instance variables) of the objects and their operations (services or methods). The classes normally make up class hierarchies, in which lower level classes, the so-called subclasses, inherit both attributes and operations from upper level classes, or the super-classes.

Inheritance has become (in addition to structured analysis based decomposition) the principal means for managing complexity in software development. Object orientation has reached a certain level of maturity and is a standard approach in developing new software, partly due to the possibility to *reuse* implementation – and to some extent also analysis and design.

In object orientation, inheritance is the basic mechanism of code reuse. For example, a class may be added into a class hierarchy by inheriting most of its properties and operations from an existing class and adding some new properties and operations. The methods of object classes on various levels of class hierarchies may have same names; i.e. the names may be *overloaded*. In the case of overloading a compiler may be able to resolve the to-be-invoked function from the functions having the same name. This is called *early binding.* In object oriented languages, it is also possible to resolve the called function at runtime, which is called *dynamic* or *late binding*.

Polymorphism is especially important for the reusability of class libraries. Dynamic binding is one example of polymorphism since the methods having the same names actually have many (*poly)* functionalities or forms (*morphs*), (Haikala & Märijärvi, 1997, p. 299). Some operations in the subclasses may be redefined by overloading the method name and changing or *overriding* the respective functionality but maintaining the signature (the method call with parameters) of the super-class.

Due to same signatures, the operations of the super-class may be invoked without a change in calling code also for the objects of the new subclass. This is possible, because the super-class operation is not bound to a function of a specific class until at runtime. Nevertheless, the subclass must normally (if no so-called dynamic link libraries, loaded during the execution, are used) be linked to the application before the execution.

It is also important that information content and functionality may well be added to the object classes of an application *gradually*, thus exposing the objects not until it is absolutely necessary. Therefore, it is possible to develop new properties and operations into an application in phases during which new, derived object classes are introduced. As Bertrand Mayer puts it in an article (Mayer, 1998) the *open-closed principle* is one of the

central innovations of object technology: the ability to use an object as it is, while retaining the possibility of adding to it later through inheritance.

Although the *implementation inheritance*, described above, is a powerful mechanism for design and implementation reuse, it has its restrictions. In some cases the changes made to the upper level classes in class hierarchies make it exceptionally difficult to correctly anticipate all the changes taking place in the derived classes. The problem is called a fragile base class problem, when it is so severe that it threatens the integrity of the whole system. For the design guidelines necessary in this case, see for example (Mikhajlov & Sekerinski, 1998).

What is perhaps even more constraining in inheritance-based object orientation is that object orientation as such does not provide actual *runtime reuse* possibilities. It should be possible to add new functionality or modify existing functionality in applications while they are in use. As described above, even when using implementation inheritance with dynamic binding, the new, inherited classes often have to be developed, compiled and linked to the application before use.

Whereas ordinary objects are constrained to a single process or thread in one computing node, *distributed objects* can reside in any thread or process in any computing node on a network which is accessible remotely by other objects. Robust distributed object systems allow objects, written in various programming languages, to communicate using standardised messaging protocols (for example TCP/IP). Distributed objects allow applications to be split into subsystems that can be executed in several computers and, the benefits of distributed computing are reached.

Distributed objects as such do not, however, give any help in finding or *locating objects or services* that are capable of performing operations which a given object needs. Objects must thus know the methods of other objects in detail. While standardised, but low level messaging protocols are used in inter-object communication, even normal message passing functionality, let alone higher level interoperation between objects is tedious to implement with conventional distributed object techniques.

3.2.2 Software component technology

In order to enable also runtime reuse and to alleviate the problems of implementation inheritance, as well as difficulties in implementing distributed applications; *software*

component technologies have been developed. Software components are *reusable, self-contained pieces of software*, which are accessible only through well-defined *interfaces*. A component has to be self contained, autonomous enough to be able to perform a well-defined task, which has its place within various applications. The services provided by the component must be explicit to the potential users, as well. That emphasises the importance of interfaces.

Instead of implementation inheritance, *composition* of existing components, discussed in the next section, is the main mechanism of reuse within software components. Components use inheritance, too, but now at the *interfaces* in which only signatures are inherited, whereas each inheriting component class defines its own implementation. This is conceptually comparable to object inheritance from an abstract superclass.

Some authors adopt more general definitions for components. For example, Ivar Jacobson defines a component (Jacobson, *et al.*, 1997) as "a type, class or any other work product that has been specifically engineered to be reusable". The definition adopted in this thesis, above, is closer to those of OMG's CORBA (OMG, 1995) and Microsoft DCOM (Microsoft, 1996) which consider a component to mean an encapsulated module of code which is transformed into run-time objects providing services to their users. Desmond D'Souza uses the term a *component in code* which becomes a *component instance* (D'Souza & Wills, 1999, p. 390) for this kind of a component.

The strict definition in this thesis, as well as the ones by OMG and Microsoft, emphasise the role of components as operational run-time entities whereas Jacobson uses the term 'component' for all kinds of reusable software artefacts. The benefit of Jacobson's definition is that it makes explicit the need to reuse all artefacts of software engineering and to manage them in a coherent manner. This thesis, however, considers components from an operational point of view and in such a level of detail that analysis and design artefacts cannot be included in the definition of components.

An ideal component (in the strict sense) should be an autonomous unit that maintains its encapsulation when considered from various points of view. In automation, important for the autonomy of components is (Kopetz, 1998) that they can be considered as independent units of *service provision, error containment, reuse, design and maintenance* and, *validation*. Well-designed error containment is needed in critical applications: a component should either operate correctly, remain silent, or produce a detectably incorrect result, without disturbing the other components in the system. It is also important to be able to validate the proper operation of a component separately, both in the value domain and in the temporal domain.

Components can at best, be used by any developer, with mixed languages or platforms. They can be *arbitrarily distributed* (considering reliability, availability and security aspects) within a local network, intranet, extranet, Internet, or in some applications, even in a mobile network. In most cases, as also in this thesis, components are implemented with object-oriented technology, enabling also the exploitation of the significant benefits of object orientation in development work. Components also differ from other types of reusable software modules in that they can be selected and invoked at runtime as binary executables.

Interfaces define the way (methods with input and output parameters) in which the components are accessible. From the interface definitions, so-called proxies and stubs are automatically generated (Microsoft, 1995, Chapter 7, p.5). Proxies are representatives of the server's (component provider's) components in the client's (component user's) address space. Stubs are representatives of the client's objects in the server's address space, Figure 3.2.

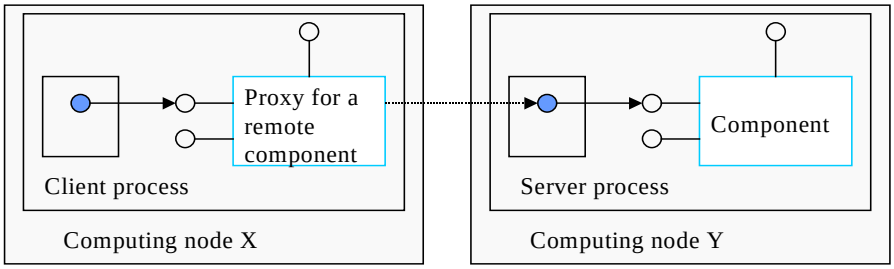

Figure 3.2. Accessing a component in a remote server.

The generic *standard interfaces* define functionality for various common purposes, for example for file systems, compound documents and user interface controls. These interfaces may be inherited as such, or overridden by the application developers. *Custom interfaces* have to be defined and the corresponding functionality implemented when domain-specific or application-specific functionality is developed for software components. Components, both existing ones and those developed during the application project, may then be used in composing the needed functionality.

Component *lifetime management* is an important aspect in component technology. Components are instantiated or constructed dynamically, when their capabilities are needed and they have to be removed or destructed respectively when they are no longer needed. There exist mechanisms based on both explicit reference counting and so-called garbage collection (Rogerson, 1997, pp. 63 … 83; Arnold & Gosling, 1998, p. 47).

With reference counting, the application has a better control of the lifetimes of the components. Reference counting has, however, to be explicitly implemented. Garbage collection is automatic and helps to avoid the so-called dangling references but may cause non-deterministic overhead, which may affect adversely time critical automation applications. Effective incremental garbage collection algorithms, introduced recently, alleviate this problem, however. Contrary to a common belief, memory leaks may also occur when using garbage collection, as noted by Java's developer Gosling (Arnold & Gosling, 1998, p. 48). Accordingly, component lifetime issues shall be taken care of in component-based system development irrespective of the chosen implementation language.

The need to *distribute components* and supply them with well-defined application programming interfaces has brought about object or component distribution models. The most important of them are CORBA (Common Object Request Broker Architecture), developed by the standardisation body Object Management Group, (OMG, 1995) and DCOM (Distributed Component Model), developed by Microsoft Inc., (Microsoft, 1996). The third distributed object model, EJB (Enterprise JavaBeans), has been developed by Sun Microsystems (Sun Microsystems, 1998). This object model is currently being integrated with CORBA technology.

The above-mentioned component models, as well as application-domain bound *component model specifications* are important enabling technologies when developing domain-specific frameworks. The most important component model specification for process-control domain at present is OLE for Process Control (OPC, 1998). OPC is designed to allow control room client applications to access plant floor data and control devices in a consistent manner.

An OPC server comprises several component instances from three main component classes: server, group, and item. The OPC server component acts as a container for OPC group components. The OPC group component provides mechanisms for containing and logically organising OPC items, which are references to factory floor data. Both generic and domain-specific distributed component models have been discussed more thoroughly elsewhere, for example in (Kuikka & Karhela, 1998).

The use of components brings about risks, which may be especially great in critical automation applications. The risks can be categorised into areas of *component design, procurement, use* and *maintenance,* (Lindqvist & Jonsson, 1998). The design of individual components may be inadvertently or even intentionally flawed, they may contain too many features to be usable and sometimes they may be poorly documented.

When buying COTS components, it is often difficult to ascertain that the components are properly validated, especially if the marketing channel is unknown or insecure. The deployment of, for example, user interface components by the end user may also be such as was not intended by the designers, due to lacking skills and/or poor user documentation.

If the design is not documented well enough, it is also difficult for application developers to analyse the effects involved in component use on the system's performance and potential side effects on other functions. Updating policies of COTS components may also be insufficient, especially if a small vendor has provided the components. Perhaps the greatest risks lie, however, in integrating components into applications, or composing them, which is described in more detail in the next section.

3.2.3 Component composition

Application specific (sub)systems and domain-specific frameworks are constructed from the nearly autonomous components, described in the previous section. The general term *component system* (Jacobson *et. al.*, 1997) is used for these "sets of related components that collectively accomplish a function larger than that accomplished by a single component". The development of component systems should be such that characteristics (for example real time properties and testability) that have been established on the component level also hold on the system level. If that is the case, the *principle of composability* is fulfilled (Kopetz, 1998).

Application and framework development consists in essence of *selection, adaptation* and *composition* or *assembly* of components. When selecting commercial components, developers must also consider, in addition to the requirements, the quality factors and risks involved, described in the previous section. From the composition point of view, the most important factor is, however, what impact the selected component will have on the developed system as a whole. This, so called *composition compatibility* of the components can be improved, for instance by component wrapping (Voas, 1998), by putting a software layer around the component to *limit* what it can do.

Very seldom, however, can domain-specific components be used straightforwardly as such. Some kind of *adaptation* is necessary. A component can be adapted using *white-box* approach and *black-box* approach. The white-box approach means implementation inheritance of component classes. Thus potential changes in the design and

implementation of the reused super class functionality have to be considered, too. The black-box approach in adapting components does not change the original component. It can be carried out, for example, by complementing a known component with a *wrapper*. In this case new functionality is *added* to the component, (Bosch, 1997).

One important special case of adaptation is to wrap a CORBA object, JavaBean or DCOM component so that it can appear in one or two other component models. Such adaptivity was especially appreciated in a Workshop on Compositional Architectures (Thompson, 1998). In order to make this kind of wrapping possible; the developed components should not be strongly dependent on the underlying component model. On the other hand, all components need not necessarily be fully interoperable with other components. Several levels of (required) interoperability have been discussed, as well as solutions toward component model interoperability.

Composition of components relies theoretically on component interfaces only, and can thus be called *black-box reuse* of components, (Szyperski, 1998, p. 137). According to Figure 3.3, individual components, acting in the role of clients use the services of other components via their interfaces. When composing a single component, also the other components, needed by the component to-be-composed, have to be available.

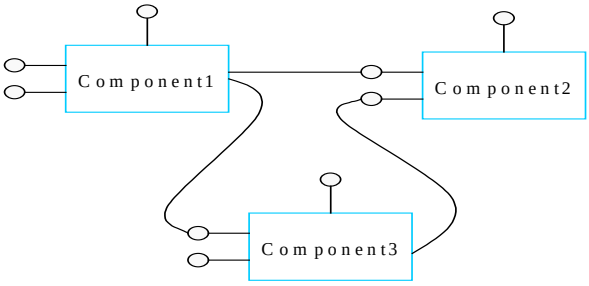

Figure 3.3. Black-box component composition.

White-box reuse, on the other hand, means that the client component is fully aware of the implementation of other components' operations, which is, of course, in contradiction to the idea of encapsulated, independent components. Buchi and Weck (Buchi & Weck, 1997) introduce a *grey-box component*, which reveals a part of the component's internals, not just the relations between input and output. The internals are specified in their approach by abstract statements of imperative languages, enriched by data types like sets and sequences.

Another approach to specify the semantics of component composition is to use *contracts* between components. Contracts are descriptions of interaction between components. Reuse contracts (De Hondt *et al.*, 1997) specify both dependencies on other components and conflicts between them. There are different types of component composition contracts: pre- and post-conditions for interactions is a common approach, abstract statements and formal specifications can also be used. Pre- and post-conditions can be defined, for example with UML's OCL (Object Constraint Language), (UML, 1997). Also a diagrammatic notation has been developed which can be used as an alternative or complement of textually written constraints, (Kent, 1997).

Metadata, machine-readable, descriptive information associated with components, is also regarded as an important *composition enabler* (Thompson, 1998). A client should be able to *describe what* it wants to use (from components and their interfaces) instead of *refer to* which specific component or interface it needs. Simple examples of this kind of information are the interface repository in CORBA (OMG, 1995) and the type library information in COM (Microsoft, 1995). In complex systems, metadata may be organised by using views, e.g. a security view and a reliability view.

As well as composition enablers, there are also *inhibitors of composition* (Thompson, 1998). The most important of them are the relative immaturity of the field, the missing definitions of some key terms and the lack of analysis information. Also computational complexity, which is inherent, whenever semantically rich metadata has to be processed, may inhibit composition. In this respect, a key strategy proposed by Thompson is to keep systems so simple that they can be shown to work well in a given application domain.

When composing systems from components off the shelf, one integrative approach is to apply a mix of formal and informal approaches in order *to build trust* on the composition. This approach has been adopted by Bertrand Mayer and his colleagues (Mayer *et al.*, 1998) and it is based on six principal techniques: design by contract, formal validation, reuse library techniques, global public scrutiny, extensive testing and metrics efforts. By the global public scrutiny, the authors mean making components freely available in source code and seeking contributions and criticism in the worldwide Internet community. Contracts between components were described above; other constituents of the approach are familiar practices in producing quality software.

Ivar Jacobson and his colleagues (Jacobson *et al.*, 1997) put emphasis on various *variability mechanisms* when developing component systems: inheritance, extensions, parametrization and configuration. They also give recommendations about the kinds of contexts for which various variability mechanisms are most appropriate. From the point

of view of process management system framework, parametrization and configuration are especially interesting variability mechanisms in component composition, because they have been proved to be useful in many traditional automation systems, as well.

Another interesting approach to component composition is to use *automatic software generators*. GenVoca generators, (Batory & Geraci, 1997) synthesise software systems by composing ready-made components from reuse libraries. GenVoca library components are originally designed to export and import standardised interfaces (components may contain multiple classes and their methods). They should thus be interchangeable, and interoperable with other components. GenVoca also provides techniques to decompose existing applications into reusable and composable components.

The GenVoca developers correctly note, however, that all syntactically correct compositions of components are not semantically correct and express the need to thoroughly validate the generated compositions of library components. They also show that components that export and import immutable interfaces are often too restrictive for software system synthesis. Components need to enlarge upon instantiation in a manner analogous to the previously discussed approach of Jan Bosch, using wrappers.

One claim of this thesis, justified in more detail in Chapter 5, is that individual components and their generic, *domain independent composition* as such, even with the help of various formal and informal techniques and generators, is *not* a sufficient methodology of software development in domain-specific contexts. In addition to generic compositional techniques, described above, also design patterns are needed when *domain-specific component frameworks* are developed.

3.3 Design patterns

3.3.1 Background and definitions

Object and component oriented development is normally begun with *analysis* in which the requirements of the software system to be developed are found out and a model – or more often a set of models – of the system is developed. The software requirement specification or definition document consists of the models and additional textual requirements and descriptions. This document, developed originally in the analysis phase of a software-engineering project, will then be the basis both for the design and implementation and for quality assurance of the various baselines of the software product.

Specification and modelling is usually iterative, it may consist of several refinement cycles to the original requirements specification, it may cover separately various behaviours and features of the system, and thus continue in time to design and even implementation phase. The idea is, anyhow, that the requirements specification deals with requirements, *what* is to be designed and implemented.

For the *synthesis* in object-oriented development, especially for the architectural design, there has not existed too much methodological support. In object orientation, design modelling notations and techniques are largely the same as those employed in analysis. As such, they do not give developers any implementation oriented support to proceed in design modelling in the way the informal requirements by a customer, vague as they may be, do in analysis modelling.

The lack of methodological support for architectural design is considered in this thesis as more *severe in object oriented* than in structured approaches. This is due to the fact that the fairly straightforward decomposition approach of structured design has in object orientation been replaced with a more complicated approach. The information content and behaviour is designed for collaborating, but independent and encapsulated entities, objects and software components.

Because of the above-mentioned needs for a synthesis methodology, a substantial and serious part of the object oriented software community has begun to research, develop, and deploy *design patterns*:

> *A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems (Gamma et al., 1995).*

When developing software frameworks, the so-called *architectural design patterns* are an especially important subclass of design patterns:

> *An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them (Buschmann et al., 1996).*

The definitions above, although rigorously phrased, are somewhat mechanistic and do not necessarily convey the richness of the design pattern concept and the *significance of the approach* for development. The origins of design patterns lie in architecture, more specifically in the works of Christopher Alexander, who has written much on patterns in architecture for buildings and urban communities.

In a seminal work on patterns, (Alexander *et al.,* 1977), Alexander does not even try to formally define the term pattern, but gives instead a large collection of pattern entries. Each entry links *a set of forces*, a configuration of *artefacts*, and a *process* for constructing a particular realisation. The entries thus *intertwine* the problem space ('what'), solution space, and construction space ('how') issues, so that they may evolve concurrently when patterns are used in development.

The first published work about *patterns in software development*, was Erich Gamma's doctoral thesis in 1991[1]. Since then – and also largely independently of each other – several leading designers and writers in object-oriented community have contributed with theses, books and articles on pattern deployment. Patterns are becoming important in object-oriented software development, because developers, striving towards quality software, take the time to study them, use them in various application domains, discuss them, and develop and write them.

It is interesting that in several object-oriented design patterns of the vast pattern literature, the *interface inheritance and composition of objects* are important, the same themes that differentiate component based software development from traditional object-oriented development. This is an excellent basis for implementing the design-pattern based framework design with component technology, the approach chosen in this thesis.

Lieberman (Lieberman, 1986) noted already before design patterns had been introduced to software engineering that *delegation of operations to other objects* makes composition as efficient in reuse than implementation inheritance. Delegation of operations to collaborating objects is a mechanism often used in design patterns, some patterns (for example state pattern (Gamma *et al.,* 1995)) rely heavily on it. This conformance is another issue that makes design patterns and composition of components close to each other.

To make design patterns easier to study and apply in object-oriented design, *presentation schemes* have been developed for the catalogued patterns. The scheme by Erich Gamma (Gamma *et al.,* 1995) is a good example. It contains, in addition to graphical and textual design description, also information on the decisions, alternatives, and compromises that led to the documented design, i.e. the 'whys' of design:

[1] Although already in 1987, Ward Cunningham and Kent Beck used Alexander's ideas to develop a small, five pattern language, (Beck & Cunningham, 1987).

Name, the pattern name followed by its classification.

Intent, a short statement of what the pattern does, or which particular design issue or problem it addresses.

Also Known As, a list of alternative names (if any) for the pattern.

Motivation, a scenario that illustrates a generic design problem, along with the solution offered by the pattern.

Applicability, when to use the pattern, and when not to use it.

Structure, a graphical representation of the classes and/or objects in the pattern.

Participants, the classes and/or objects participating in the pattern and their responsibilities.

Collaborations, how the participants collaborate to carry out their responsibilities.

Consequences, the tradeoffs and results of using the pattern.

Implementation, hints or techniques for implementing the pattern.

Sample Code, code fragments that illustrate the implementation in a particular programming language.

Known Uses, references to real systems where the patterns are used.

Related Patterns, other patterns which are either similar or are often used with this pattern.

When *documenting the use of patterns* employed in the design, the properties of the catalogued patterns need not be repeated; *reference* to them can be made with the help of the pattern name. Pattern names are a common vocabulary, which helps designers and developers to communicate better. Patterns are in this sense, in effect "a common asset which expand people's communication bandwidth" (Vlissides, 1998).

In pattern based design, class notation with proper naming is used; see the last section of this chapter for the naming in developing frameworks. The application or the framework documentation reader is also provided with textual information about the *context* of the design problem to be solved, as well as with a description of the *problem* and its *solution*. The problem to be solved is often given in terms of the *forces*, or the design issues of the problem that shall be considered when solving it, such as requirements, constraints and desirable properties. The solution then balances the given forces.

3.3.2 Pattern collections and languages

In order to make the design patterns reachable for software developers, *collections* of them have been assembled and *pattern languages* have been developed.

The patterns described in the seminal pattern source, (Gamma *et al.*, 1995), are categorised into three groups:

- *Creational patterns*: Abstract Factory, Builder, Factory Method, Prototype, Singleton

- *Structural patterns*: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

- *Behavioural patterns*: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Creational patterns deal with the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioural patterns describe the ways in which classes or objects interact and distribute responsibility. The above-mentioned patterns have interesting relationships to each other, which are useful, when deploying patterns in design work, Figure 3.4.

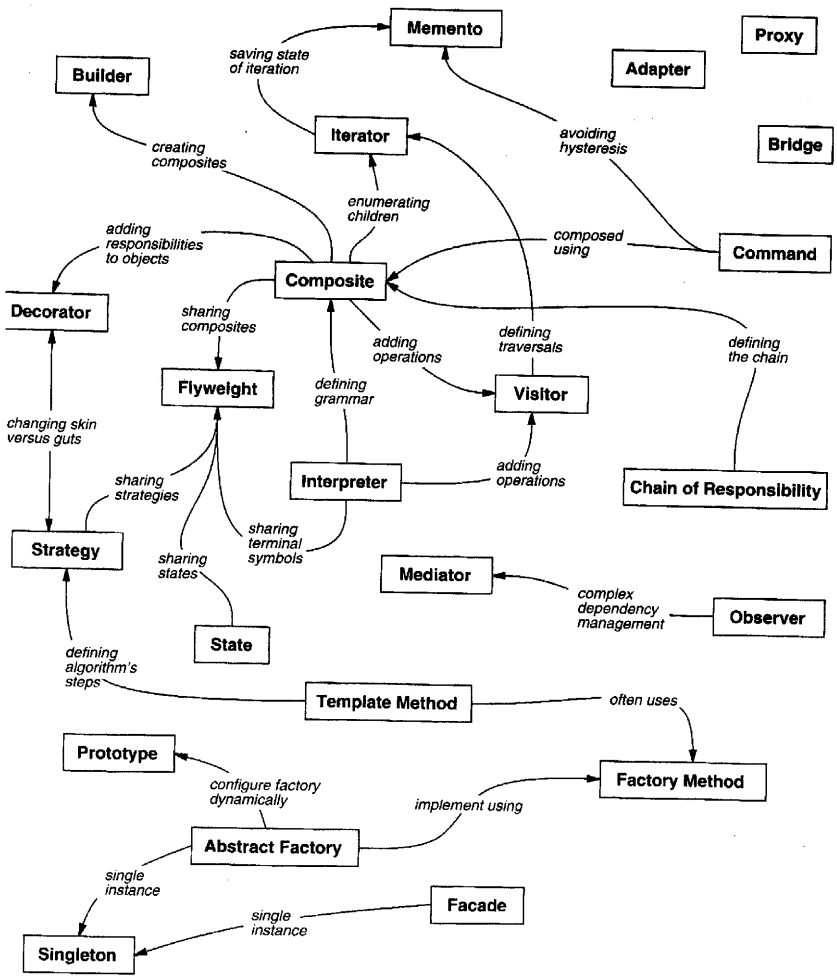

Figure 3.4. Design Pattern relationships, from (Gamma et al., 1995).

Another important pattern collection concentrates mainly on architectural patterns. It has been published by Frank Buschmann and his colleagues (Buschmann *et al.*, 1996). They divide their patterns into:

- *Architectural patterns*: Layers, Pipes and Filters, Blackboard, Broker, Model-View-Controller, Presentation-Abstraction-Control, Reflection, Microkernel

- *Design patterns*: Whole-Part, Master-Slave, Proxy, Command Processor, View Handler, Forwarder-Receiver, Client-Dispatcher-Server, Publisher-Subscriber

- *Idioms*.

The definitions of architectural patterns and design patterns in the preceding section are also valid for the categorisation above. Idioms are low-level patterns, specific to a programming language. As stated by Buschmann: "An idiom describes how to implement particular aspects of components or the relationships between them using the features of a given language".

There are patterns that have especially strong relationships to each other. Some patterns are, for example, needed to implement others, and some patterns may work best in close collaboration with others. Some authors speak in this context about *Pattern Systems*. For example, Buschmann defines a Pattern System for software architecture as "a collection of patterns, together with guidelines for their implementation, combination and practical use in software development".

Often writers, however, claim that it is more appropriate to talk about *Pattern Languages*, when discussing collectively patterns that are developed and/or deployed and adapted for a specific purpose. Conferences on Pattern Languages of Programs (PLoP, ChiliPLoP, and EuroPLoP) have acquired a notable role in Pattern development. Selected papers from those conferences have been edited into a Book series: Pattern Languages of Program Design (Coplien & Schmidt, 1995; Vlissides *et al*., 1996; Martin *et al*., 1997).

The term 'language' in Pattern Language refers more to a natural language than to a design or programming language. Indeed, as noted in (Coplien & Schmidt, 1995), PLoP was founded to create *a new literature* to "capture and refine a broad range of software development expertise in a systematic and highly accessible manner". The textual parts of pattern descriptions are needed, because they document the design decisions, the reasons for a particular pattern based solution. This is important when teaching and training patterns, when assessing the quality of the software and when making decisions on changing the solution (and respective pattern(s)) or merely refining it during the maintenance phase of a software product.

3.3.3 On the domain independence and uniqueness of design patterns

Most of the design patterns included in collections, pattern systems and even pattern languages are *independent* of a particular (vertical) *domain*. Some of the collections or languages have, however, been developed for different (horizontal) areas of information technology[2], for example for data management, user interfaces and real-time computing.

The characteristics of an application domain have been seen mainly in the decisions of *what patterns to use* and *how to adapt or vary them* when developing applications or domain-specific frameworks. This is reasonable, in view of this thesis. For example, telecommunications as an application domain and several of its subdomains are very close to automation technology and its subdomains. The similarities are explicit in required functionality, quality criteria and applicable implementation technology base. It is thus beneficial to develop and exploit *common (design pattern) knowledge base* for the applications and architectures in these application domains.

The above is exemplified by the patterns *Acceptor*, *Connector*, *Reactor*, *Router*, and *Active Object*, developed by Douglas C. Schmidt and his research group for concurrent, parallel, and distributed systems (Schmidt, 1995). These patterns have been originally developed for, and first applied in a domain-specific Adaptive Communication Environment (ACE) framework. The ACE has been subsequently applied in several telecommunication systems (Schmidt, 1998).

However, the patterns are also applicable to other distributed systems, for example new control systems. The design issues of the Schmidt's design patterns include event demultiplexing and event handler dispatching, signal handling, service initialisation, interprocess communication, shared memory management, message routing, dynamic (re)configuration, concurrent execution and synchronisation. All these are important aspects of real-time control architectures.

When used, for instance, in batch control, the above-mentioned patterns are best applicable to *Unit Supervision* and *Process Control* levels of functionality, due to their orientation to concurrent, time critical tasks. In this thesis, when developing *Process Management* framework for batch control, the *Active Object* pattern (Lavender &

[2] The term 'domain' means in this thesis, an application domain of information technology, for example batch control or telecommunications. The term 'area' of information technology is used for distinct functionalities of information systems irrespective of their (application) domains.

Schmidt, 1996) has been found useful, as well. In designing multithreading issues of both recipe elements and process units, a variation of the Active Object has been deployed as described in Chapter 6.

While most of the design patterns are naturally 'common heritage' for various application domains and often for various areas of information technology, there may be some patterns which are genuinely domain-specific. Unfortunately, however, on various application domains, there are published patterns that are only superficially different from others, published earlier. They thus harm the development work of the application and framework developers by unnecessarily adding to the pattern vocabulary. They make the common language harder to master and widen the search space from which to select patterns for the development of a domain-specific framework.

For example G++, A Pattern Language for Computer-Integrated Manufacturing (Aarsten *et al.*, 1995), contains both patterns (*Hierarchy of Control Layers*, *Visibility and Communication between Control Modules*) that are variations of previously published, well-known patterns (*Layers, Observer*), and original, domain-specific patterns (for example *Implementation of Control Modules*). The former ones are hardly ever referred to in literature, but the latter ones have been found valuable in domain-specific development. Also some of the patterns (*Whole-Part, Proxy, Publisher-Subscriber*) in (Buschmann *et al.*, 1996), are unnecessarily published variations of the respective original patterns (*Composite, Proxy, Observer*) in (Gamma *et al.*, 1995).

While *frameworks*, to be presented in the next section, are normally *domain-specific*, the *patterns* used in developing them, can – and in the view of this thesis should – be viewed as *domain independent*, generic constituents of a framework. It is argued that architectural design can thus be made understandable for both experts in a domain (due to domain-specificity of the framework) and for software engineers (due to generic patterns used). The need for *general readability*, and *possibility to understand* and *review* the framework with the help of common, generic design patterns, is seen as important for the way to develop a domain-specific framework.

Generic design patterns are needed both as a means of *developing the design* and a way to *document* the domain-specific or information technology area oriented framework (Johnson, 1992; Beck & Johnson, 1994). Good documents make the widespread reuse of software frameworks possible also in other application domains and areas of technology. On the other hand, while frameworks contain domain-specific behaviour, designed with the help of patterns, they are good examples of *concrete realisations of abstract design patterns, comprehensible also to domain experts*.

In many cases *domain independent but technology area specific frameworks*, (or *application frameworks* and *support frameworks*, as they are also called (IBM, 1995)) are useful. In addition to conventional examples in graphical user interface design, also functionally new ones have emerged. A good example is a framework that abstracts generic collaboration properties of groups of classes (D'Souza, 1998).

In some cases also component sets, which allow developers to assemble business applications from existing parts, are called frameworks, most notably SanFrancisco by IBM (IBM, 1997b). This is reasonable if the component sets are complemented with standardised Core Business Processes, like in SanFrancisco. Most often design patterns are deployed, however, in designing applications or domain-specific frameworks, described in the next section.

3.4 Domain frameworks

3.4.1 Background and definitions

A framework is a set of co-operating classes that makes up a reusable design for a specific class of software (Gamma *et al.*, 1995, p. 26). The framework is meant to *synthesise the architecture* and capture the architectural design decisions common to the application domain. The domain framework will be customised to a particular application, normally by parametrizing framework components, by incorporating new component classes and/or by creating application-specific subclasses of the abstract classes of the framework.

Another definition, stricter but less descriptive than the previous one, for a framework is given by (Buschmann *et al.*, 1996, p. 435):

> *A framework is a partially complete software (sub)system that is intended to be instantiated. It defines the architecture for a family of (sub)systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment, a framework consists of abstract and concrete classes.*

In addition to domain-specificity issues, there is one vital difference between design patterns, described in the previous chapter, and frameworks, described here. A pattern is an abstract design artefact, described mostly in a language-independent manner, whereas a framework is an implementation of (or a part of) a software architecture and normally

implemented in a particular programming language or languages. However, patterns and frameworks are synergistic concepts, neither subordinate to other (Schmidt *et al.*, 1996). As noted earlier, object-oriented and software component frameworks explicitly implement many patterns and patterns, on the other hand, are used to document the form and contents of the frameworks.

If the basic constituents of a framework are software components, instead of object classes, the inserting, or 'plugging in' of component instances into the framework is emphasised, as defined by Szyperski (Szyperski, 1998, p. 280):

> *A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances.*

The documentation of a domain framework resembles in many ways normal software documentation. There has to be a *domain oriented requirement definition* or specification ('what') consisting of domain business classes, which can be modelled using standard object analysis methodologies. Also the implementation and testing documentation is ordinary. Compared to application documentation, the quality requirements are greater, since the framework will be designed for *reuse* in several applications.

The main difference in documentation of normal applications and frameworks lies in *design documentation*. The design documentation of the framework has three purposes; patterns can help in each of them (Johnson, 1992). The documentation must describe the *purpose* ('why') of the framework, based on which the application developer may decide whether or not the framework really is suitable to the application problem. It must also describe *how to use* the framework when building the application. The third purpose of the framework documentation is to *describe the design*, including not only the different classes in the framework but also the way the instances of these classes, individual objects or component instances, collaborate.

When *using – and reusing – a framework*, there are two basic approaches of which various combinations exist, too. *Called frameworks* are much like conventional class libraries in the sense that application code calls the framework when some framework service is needed. In *calling frameworks*, on the other hand, the main body of the framework is used as such, and only the application specific small amount of code is written, which will be called by the framework, Figure 3.5.

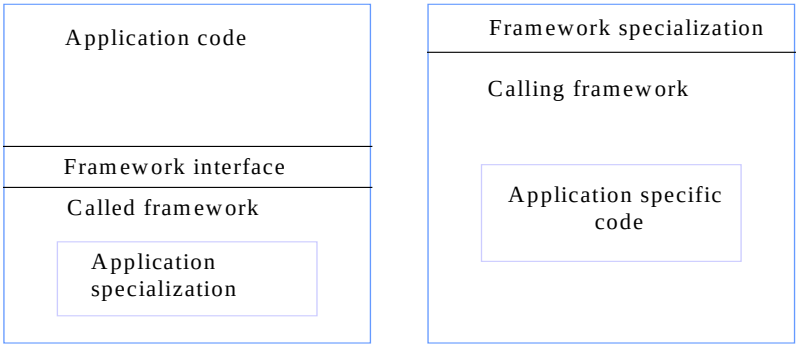

Figure 3.5. (Re)use of a called and calling framework.

The above-mentioned *inversion of control* between the application and the framework on which it is based is typical to calling frameworks. It leads to potentially higher rates of reuse and increased productivity, compared to the called framework approach or the library based software reuse. This is due to the fact that in the case of a calling framework more ready and higher quality code will normally be (re)used. The application designer's degrees of freedom in design are, of course smaller, when using calling frameworks.

Effective reuse presupposes the capability of the framework designer to include the stable, non-variable domain issues (sometimes called *frozen spots*) in the framework and yield control of application-specific structure and behaviour to the application designer. The application specific parts in the final application architecture are represented by so called *hot spots* (Pree, 1995) of the framework, parts that are designed to be easily adaptable to application specific needs.

3.4.2 Designing domain frameworks with patterns

Architectural models

As indicated in the previous section, domain frameworks are partial *implementation*s of architectures, sometimes also called microarchitectures, designed to be reusable for a family of application systems. Within software engineering, architecture may, more generally, be defined as (Garlan & Perry, 1995):

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

The above definition is focused on architecture as a *model* or frame of reference, which is needed when developing frameworks. A disciplined development of architectural frameworks can have a positive impact on various aspects of software development. Garlan and Perry, for example, note the importance of architecture in *furthering understanding, reuse and evolution* of the developed systems, as well as in providing new opportunities for analysis and management of the development process itself.

It is also beneficial to look at a software *framework from various points of view*. What is important in an architecture, varies according to the stage of development and purpose for which the architectural document is developed. Thus one architectural model is usually not sufficient, but several models, each having their own point of view and purpose, are needed.

Soni (Soni *et al.,* 1995) has identified four distinct architectures:

- The *conceptual architecture*, which describes the system in terms of its major design elements and the relationships among them.

- The *module interconnection architecture*, which consists of two orthogonal structures: functional decomposition and layers.

- The *execution architecture*, which describes the dynamic structure of the system.

- The *code architecture*, which describes how the source code, the binaries, and the libraries are organised in the development environment.

The conceptual architecture is best developed on the basis of the object-class model of the framework resulting from domain analysis. *Design patterns*, on the other hand are, according to this thesis, most helpful in developing the *module interconnection architecture* and the *execution architecture*. The code architecture is mostly influenced by the distributed component model and the development environment.

Another categorisation of software architectures is given by Kruchten (Kruchten, 1995). The classification consists of five concurrent views about the system to be developed, Figure 3.6:

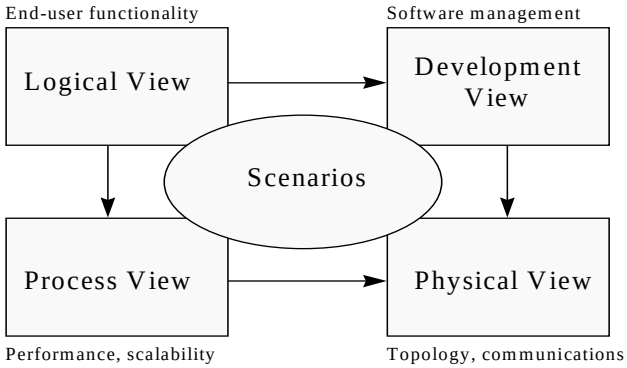

Figure 3.6. Architectural views, modified from (Kruchten, 1995).

- The *logical view* describes the class-object model of the system.
- The *process view* describes the design's concurrency and synchronisation aspects.
- The *physical view* describes the mapping of the software into the hardware and reflects its distributed aspect.
- The *development view* describes the software organisation in its development environment.
- The *scenarios view* illustrates the architectural decisions made according to the previous four views with a few selected use cases or scenarios.

In the categorisation above, *design patterns* are best employed in developing the *process view*, *the physical view* and *the development view* after the domain-specific logical view has been developed with the help of scenarios. Scenario techniques, which are first used from end-user perspective when developing the logical view, have proved to be useful also in later framework development supporting the design pattern based design, especially when designing the needed collaborations of components.

After having an idea of in *what* architectural aspects or views of framework design to exploit design patterns, it is necessary to ponder *how* they could be best used. The *approach adapted for this thesis* has been to:

- *Find feasible* design patterns for a given domain-specific design issue and *select the most suitable.*
- *Study* the pattern(s) in detail.
- Make relevant *design* and implementation decisions, *vary* the original pattern, if needed (but do *not* make a new pattern), and *name* properly.
- *Implement* and combine the pattern with the other patterns and components of the framework.

Iteration is, of course, needed in the process above; it cannot proceed without refinements in previous phases. Due to the object-oriented and component-based approach and the use of design patterns, the gradually changing framework can, however, be managed even in evolutionary software development processes. Component framework can also be well managed during the maintenance and potential enhancement phases. A precondition for this is, however, that the evolving architecture *corresponds* correctly *the domain requirements* and that the applied development procedures are *systematic*.

The approaches for *finding* design patterns, potentially suitable for the problem domain design issue, are vital in the design process. Erich Gamma has given some good advice about how to find suitable object-oriented patterns in their pattern catalogue (Gamma *et al.*, 1995, p. 28 … 29). The advice has been complemented in this thesis with domain-specific and component based issues, resulting in the following procedure:

- Consider how the domain-specific design problem can be solved with generic design patterns.
- Find out how the patterns interrelate, utilise relationship categories.
- Compare patterns of similar purpose and consider what should be variable in your design.

- When making the selection, consider the coherence of the whole architecture.

When developing a domain-specific framework, it is important to *first* clarify with the help of the requirement definition (conceptual architecture or logical view, above), the domain-specific design *problem*, and *after that* look for *the patterns*. Gamma's original text is in this respect totally domain independent, stating generally "design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems".

When finding how patterns interrelate, it is beneficial to proceed beyond the simple but important relationships of Figure 3.4. The *relationships between patterns* have been classified (Noble, 1998) more thoroughly into primary relationships (uses, refines, and conflicts), and secondary relationships (similarity, combination, tiling). Categorisation is helpful, because the search space is large, consisting of several sources: books, articles, and web pages, not only Gamma's catalogue (Gamma *et al.*, 1995).

Several design patterns, having similar purpose, make it possible to *vary different design issues*, by letting the developer change these issues without the need to redesign the framework. For example, if the design problem is the creation of domain-specific components, several patterns are available, for example Factory Method, Abstract Factory, Builder and Prototype (Gamma's catalogue), each allowing different aspects to be changed.

The component object model employed (DCOM, CORBA, or Enterprise Java Beans (EJB)) imposes, however, some restrictions to the choices available but, on the other hand, provides implementations, as well. The component platform shall therefore be considered when selecting patterns for frameworks. For example, as discussed in (Plasil & Stal, 1998) DCOM and CORBA Lifecycle Service deploy the Abstract Factory pattern in component creation. EJB, on the other hand, includes the factory method in the so-called home interface of an enterprise bean (Sun Microsystems, 1998, p. 20).

Additionally, the *composition compatibility* aspects, discussed in Chapter 3.2, should be taken care of. It is even more important when selecting the design patterns for the architecture than when selecting the components to be 'plugged' into the framework, that the new patterns do not make the architecture incoherent. A candidate pattern should thus also be considered from the framework point of view (overall architecture) and from the viewpoint of existing components (implementation restrictions), in addition to considering the way in which it solves the actual design problem.

When an appropriate pattern – or a few of them - has been found and selected from several sources, it has to be used in a way that extracts all important details to the domain-specific framework. This means a thorough *study* of the pattern, first on an *abstract* level (Intent, Motivation, Applicability, Structure, Participants, and Collaborations) then going into the *concrete* example of the pattern (Implementation, Sample Code). After that, the *implications* (Consequences, Known Uses, Related Patterns) of the pattern and its possible variations shall be considered.

After having enough information on the pattern, a *design* decision has to be made on either using it as such or developing a *variation* of the pattern and also on the *manner* in which it will be used in the framework. Sometimes the introduction of a new pattern causes a need to *restructure* the existing framework. This should be accomplished by first carefully refining the original architecture to be suitable for the new pattern and only after that integrating the new pattern to the framework.

The names for the object classes, acting as pattern participants, shall be chosen next. As described in Chapter 3.3, it is considered important to use *the original names for the generic patterns* in order to increase the general comprehensibility of the framework. It has been found optimal, however, to use such *names* for participating component classes, as incorporate both a *domain-specific part* and the *original pattern-derived part*, for example BatchMediator in Chapter 6.4. The domain part of the name is important from the point of view of an application domain expert, the pattern part from the point of view of a software engineer. It is possible for both of them to understand the concept and the relevant design issues and contribute to the design for example, when working in a joint development team.

After the needed patterns and their variations have been properly named, the classes involved in the patterns shall be identified, their *interfaces designed*, relationships established, and the attributes and operations adapted to the patterns. When developing a component framework, interface definition is especially critical, since the component interfaces are *immutable*. The defining of the interfaces is, on the other hand, supported by the interface definition languages and tools, which partially automate their use. The names of the operations used, should be partly domain-specific, partly pattern specific, as the names of the participating classes.

When *implementing* the patterns in designing and programming the framework components, some hints and even code may be acquired from the implementation sections of the pattern catalogues. The catalogue implementation examples are important for *familiarising* oneself with the essentials of the pattern. At best they are, however,

quite simple (for pedagogical purposes) and thus of restricted use in framework development as such, especially if the pattern in question has to be modified. The use of COTS components brings about the need to iterate with design, implement *wrappers* around COTS components, and optionally to utilise *contracts* when integrating them into the framework, as indicated in Chapter 3.2.

Non-functional aspects

A good *example of designing an object-oriented framework*, or actually iteratively evolving one, is given in (Roberts & Johnson, 1996). In the paper, the *process* of framework development is described with the help of so-called process patterns. The patterns include: Three Examples, White-box Framework, Component Library, Separate Changeable from Stable Code, Add Parameters to Eliminate Subclasses, Fine-grained Objects, Black-box Framework, Visual Builder, Language Tools. The long names of the patterns give a good suggestion of their contents. The patterns are applied otherwise in the sequence given, but 'separating code', 'adding parameters' and 'refining objects' are used in parallel.

Many design patterns incorporate into object-oriented framework development classic design principles, or *enabling techniques*, as called by (Buschmann *et al.*, 1996, p. 397 … 404). Some of these principles are: abstraction, encapsulation, information hiding, modularization, separation of concerns, coupling and cohesion, sufficiency, completeness, separation of policy and implementation, separation of interface and implementation, and divide-and-conquer. The significance of the design principles has increased with the emerging needs of software architectures and with the possibility to *include them into design patterns*, making applicable techniques out of these outstanding design principles.

When developing a domain-specific architectural framework, also *non-functional requirements* for the software architecture have to be considered (Buschmann *et al.*, 1996, p. 405 … 410). They are often omitted from the requirement specification, which naturally tends to be oriented towards application functionality. These requirements are usually not achievable within components. The architectural framework has to be considered as a whole. The non-functional properties of the architecture have a great impact on the development and maintenance of the framework as well as its general operability and its use of computer resources. Architectural design patterns often enhance

these non-functional aspects or - *ilities* (as they are sometimes collectively called): changeability, interoperability, efficiency, reliability, testability, and *reusability*.

3.4.3 On example frameworks

An interesting example of distributed component frameworks in automation domain is the CIM Framework Specification, (Doscher *et al.*, 1998) of the Semiconductor Manufacturing Technology consortium (Sematech). It is a large specification document, which defines an overall application framework for computer-integrated manufacturing (CIM), within semiconductor industries, Figure 3.7.

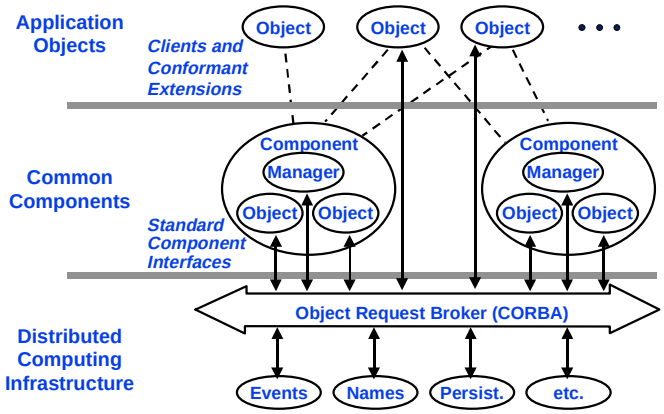

Figure 3.7. Sematech's CIM framework.

The purpose of the CIM Framework Specification is to assist potential system vendors in building integrated, flexible frameworks according to the model. This would lead the vendors to an open, multi-supplier CIM system development. An industrial end user could integrate the functionality of frameworks, representing various subdomains of the specified CIM model, to a customised, plant specific application system.

It is important, from the point of view of this thesis, that the specification of Sematech's CIM is clearly and rigorously *domain-specific*. The starting point is an object class model

of the domain, i.e., semiconductor manufacturing. Moreover, the large manufacturing domain is divided into nine subdomains corresponding the groups of common components in Figure 3.7 (factory services, factory management, factory labour, machine control, material management, material movement, advanced process control, process specification management, and schedule management).

(Sub)domains are then modelled with five *models*: component relationship model, component information model, component interaction diagram, object dynamic model, and object state tables. In addition to the models, also textual interface specifications for software components are included in the framework specification. These *interfaces* are written in the CORBA Interface Definition Language of the Object Management Group (OMG, 1995).

Sematech has also previously published an *architectural guide* (Doscher *et al.*, 1997) that provides some architectural principles and guidelines as a background of the specification. It does not consider the actual design decisions needed to develop a framework, decisions that in this thesis are suggested to be made with the help of design patterns. The Specification includes, however, interface specifications of components, corresponding the domain-specific, or so-called business objects. Other, design and implementation oriented interfaces, cannot of course, be specified at this stage.

The Sematech approach is interesting, because it implies that the developers of the specification consider it possible - and even recommendable - to develop appropriately individual components at least up to defining strictly their interfaces *before* making design decisions concerning the software architecture. In a workshop position paper of the project (Hodges, 1998) it is, however, acknowledged that “We still have a long way to go to achieve the level of rigor required to specify independent, but composable components”.

It is considered especially difficult to specify the client side of the dependency in the interaction between a client and a component. Sematech is, indeed, in the process of considering, instead of a strict conformance on component level, various levels of required conformance to the specification from the potential vendors:

- Architectural conformance (required interoperability technology)
- Syntactic conformance (required interfaces)
- Semantic conformance (required behaviour in test scenarios)

- Substitutability conformance (proven composition and decomposition)

The result of the Sematech effort, described above, is a large specification of a component framework, expecting vendors to develop compatible implementations to be used by application developers.

The DOVER project (Dagermo & Knutsson, 1996) on the other hand, is an example of a *small-scale implementation of an object-oriented framework.* The DOVER framework, which has been developed as an ESPRIT III/ESSI project, is targeted to ship control domain. The framework includes:

- A set of design objects used to develop a new ship control system
- An execution model
- Distribution mechanisms
- Configuration files
- Documentation

In DOVER, the task of a control system is seen essentially as that of a continuous re-computation of a number of output values as a response to changes in a number of input values. The concept of *Value* is thus important and it has been chosen as a class of its own, having subclasses *Computed Value, Input Value, Input Event Value, Network Source Value,* and *Pulse Value.* Dependency graphs are used to show the dependencies between different values in the system and event traces to describe the dynamic behaviour when objects from classes InputValue and ComputedValue interact.

The *execution model* of the framework is based on a real-time operating system, which is augmented by the so-called *time triggered approach.* The approach means that a control system observes the state of the environment at specific points in time, after which it decides what actions must be taken. An *Observer* pattern (Gamma *et al.*, 1995) is used in the framework in notifications of the input values. Communication with other systems is handled via interrupts and queues. Objects of the PulseValue class are used for generating start, stop, and clutch in/out signals. Objects of the InputEventValue class react on external events, set their values and notify dependent values and then reset themselves.

The distribution of functionality in DOVER is designed so that a set of values on each node is declared to be globally visible over the network. The *Proxy* pattern (Gamma *et al.*, 1995) is used to make it transparent that value objects are located on different nodes. On each node, there is a NetworkManager object, which knows about the globally visible NetworkSourceValue objects. The NetworkManager class, which is designed according to a *Singleton* pattern (Gamma *et al.*, 1995), has a DDENetworkManager and TCPNetworkManager as its subclasses.

The way to start *using the framework* is to define a Value instance for each sensor and actuator. The sensors will be instances of InputValues and the actuators instances of some subclass of a ComputedValue. Nearly all other objects in the program are dependent on one or more InputValues, according to the Observer pattern. DOVER has reportedly been used for developing two prototype implementations in the Microsoft Windows platform. The developers have *estimated* substantial benefits to be gained with the framework in terms of the amount of code to be needed, of the reusability achievable and of the quality improvable.

DOVER framework is *based on a proper use of design patterns*. An object class model of the ship control domain has been developed and a domain-specific framework developed. The framework is comparatively small and it is based on a somewhat simplified paradigm of a control system as a continuous computing engine, reading input values and acting upon them. Nevertheless, the ideas of design pattern use in making decisions on architectural design are well presented.

The DOVER approach *does not consider explicit interfaces* of individual objects; that is, technological aspects of individual components and component composition are lacking. Moreover, while the approach also includes low-level distributed communication aspects within the framework; the potential use of available middle-ware technologies CORBA or DCOM is missing.

4. Multi-agency

4.1 Introduction

This chapter presents the technological background for enhancing the batch process management framework with multi-agency features. First, agent technology, which is the basis of intelligence of individual software components in this thesis, is shortly described. Then agent interaction techniques are presented as a means to achieve co-operative interaction. Multi-agent systems are introduced both by describing their architectures and by some examples of their application. Finally, the approach adapted in this thesis, the so-called agentifying of software component frameworks, is introduced.

Agency and multi-agency are seen in this thesis, *not* as an *overall architectural paradigm*, but as a possibility to enhance a software component framework. The approach is *applicable to specific problems*, which need local decision-making and collaborative interaction. An architecture that would be solely based on autonomous agents is not considered viable within an automation system. The requirements concerning determinism, timeliness, robustness, reliability, and comprehensibility of automation functionality are too stringent. It is, however, advantageous to be able to deploy agent technology in automation, by *embedding* agency in systems and utilising it *on a local scope*, where and when considered appropriate from domain-specific premises.

The approach in this chapter is both *descriptive* and *evaluative*. The main technologies are presented both by defining the concepts and by providing information on their application. When describing the research developments, also evaluation of the approaches is given from the point of view of this thesis.

4.2 Agents

4.2.1 Background and definitions

The definition of an agent has been heavily debated during the whole period of agent oriented research, see for example (Franklin & Gaesser, 1997) for discussion. In the first issue of an international journal, devoted entirely to autonomous agents and multi-agent systems, the editors (Jennings *et al.*, 1998) give a compact definition of an agent as:

> A computer system, *situated* in some environment, that is capable of *flexible autonomous* action in order to meet its design objectives

This definition has been affected by the critical notes in (Wooldridge, 1997) following the discussion of (Franklin & Gaesser, 1997). In some other definitions, agents are ascribed additional attributes, for example on mobility, learning, and benevolence. These attributes are not considered necessary for agents in the context of this thesis but may well be seen as optional agent attributes.

Situatedness means that an agent receives input from its environment and that it can perform actions which change its environment in some way. The environment may be the whole physical world, a local area network or the Internet, for example. From the point of view of engineering, smaller environments, for example a batch process cell, are more interesting. *Autonomy* means that the agent is able to act without direct intervention of humans (or other agents) in its environment, and that it has control of its own actions and internal state. *Flexibility* means in this context, that the system is:

- *Responsive*: agents should perceive their environment and respond in a timely fashion to changes that occur in the environment.

- *Proactive*: agents should be able to exhibit goal-directed behaviour and even take initiative, when appropriate.

- *Co-operative*: agents should be able to interact, when appropriate, with other artificial agents and humans in order to solve their own problems and to help others with their activities.

Intelligent agents perform continuously three functions: *perception* of conditions in the environment, *action* to affect the conditions in the environment and *planning* or *reasoning* to interpret perceptions, solve problems, draw inferences and determine actions. The (co-operative) interactions of the agents with other agents and humans can be separated from their interactions with other computer systems and the physical environment.

The traditional Artificial Intelligence (AI) approach in agency has been knowledge representation and reasoning oriented. The traditional agents make a plan for their actions with the help of a static world model before the execution of the plan. From that premise the knowledge and tasks of the agent are well decomposable and it is possible to plan the actions in detail in advance. These systems did not, however, succeed often in real world

environments. The domain of applicability of successful systems was typically static and narrow.

However, the traditional AI techniques can be used - and have been used and further developed - as planning parts of newer, more *reactive agent architectures*. A seminal classic planner is the STanford Research Institute Problem Solver (STRIPS; Fikes, 1993). STRIPS conforms to the classic AI planning problem in which the world is regarded as being in a static state and being transformable to another static state only by a single agent. Extensions and further developments to STRIPS (Russel & Norvig, 1995) take into consideration the dynamic nature of the environment and the fact that there exist multiple agents in the system.

Another classic, although more recent, approach is *Brooks' subsumption architecture*, (Brooks, 1991), which relies heavily on individual agents' reactive interaction with the environment through perception and action. Additionally, Brooks claims that knowledge representation, in terms of models of the environment (world models), is not needed at all. It is better to use the real world (with humans and other agents) as a model of its own. He admits, however, that there is a need to consider perception - agent-action – interaction sequence, in terms of the agent's layers of intelligence operating in parallel. This approach has succeeded quite well in the automation domain when developing incrementally more advanced robot societies (Halme *et al*., 1996).

4.2.2 Agent architectures in short

The architecture of an individual agent can be defined to be a set of structural entities in which perception, action and reasoning (or a subset of them) occur, and the interconnections of these entities. Many practical agents (like the ones developed in this thesis) have *simple internal architectures*, because they are sufficient to support the local behaviour required in the application. In multi-agent systems, the individual agents are often specialised and heterogeneous. The properties of the system as a whole are achieved by the co-operation of the individual agents.

However, if and when *adaptation, learning, and evolution* are required of an agent, the requirements for the architecture of the individual agent become more demanding. A comparison between the requirements for an adaptive intelligent system (AIS) and a typical AI agent is given on Table 4.1, (Hayes-Roth, 1995).

Table 4.1. Behavioural adaptations of an AIS (modified from Hayes-Roth, 1995).

	Required AIS adaptations	Typical agent behaviours
Perception strategy	Adapt to information requirements and resource limitations	Fixed
Control mode	Adapt to goal-based constraints and environmental uncertainty	Fixed
Reasoning tasks	Adapt to perceived and inferred conditions	Single task
Reasoning methods	Adapt to available information and current performance criteria	Single reasoning method
Meta-control strategy	Adapt to dynamic configurations of demands and opportunities	Unnecessary

The Guardian agent architecture, (Hayes-Roth, 1990; Hayes-Roth, 1995), is an example of an adaptive intelligent system having *subsystems for perception, action and reasoning*. In Guardian, perception processes acquire, abstract and filter sensed data before sending it to other subsystems. The action subsystem controls the execution of external actions. The reasoning subsystem interprets perceptions, solves problems, makes plans, and guides both perceptual strategies and external actions. The main application area of the Guardian architecture has been patient monitoring, but applications also exist in the areas of power plant and materials processing monitoring. An application study in the area of adaptive intelligent robots has also been started.

Another example of an agent architecture, which puts more emphasis on the real time properties of the agent and on the separation of concerns between real-time and artificial intelligence features, is the Co-operative Intelligent Real-time Control Architecture (CIRCA), (Musliner *et al.*, 1995), Figure 4.1.

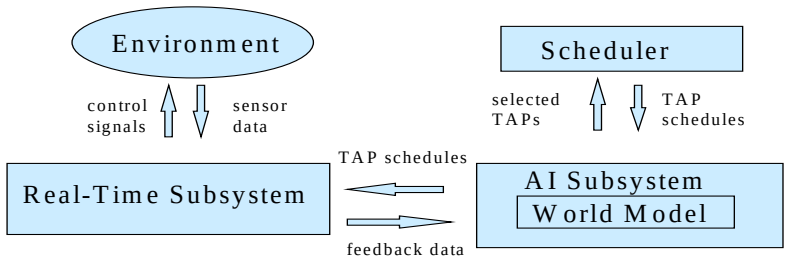

Figure 4.1. The CIRCA agent architecture.

The AI subsystem (AIS) performs reasoning about tasks and, in co-operation with the Scheduler, develops low-level control plans (test-action-pairs, TAPs). These plans are executed in a predictable real-time fashion by the real-time subsystem (RTS). CIRCA is interesting in respect to this thesis, because the *agent subsystem* (AIS) is *closely co-operating with a non-agent subsystem* (RTS), a premise that is important when incorporating agent features into an existing information system or framework.

4.2.3 Knowledge representation

Knowledge representation is needed to express the agent's state, goals, world model, or parts of it, i.e. facts and propositions concerning the agent and its environment. Knowledge should be represented in a form, which is interpretable at least by all collaborating agents. Agents need knowledge representation both to store their internal knowledge and to exchange the stored knowledge with other agents. *Rules* and *predicate logic expressions* are the most common formats for knowledge representation.

The Knowledge Interchange Format (KIF), (Genesereth & Fikes, 1992; Genesereth, 1995) is a formal, predicate logic based, but also human readable language for the *interchange of knowledge* between computer programs, for example, software agents. KIF can also be used as an internal knowledge representation format of the agents.

The main characteristics of KIF are:

- It has declarative semantics. The meaning of expressions can be understood without an agent interpreting them, unlike the meaning in logic programming languages, for example, in Prolog.

- It can be used to express arbitrary sentences in first-order predicate logic.

- Also metaknowledge (knowledge about knowledge) is expressible with it.

A knowledge base consists of a finite set of definitions and sentences. The knowledge base is structurally a set, not a sequence; the order of the definitions and sentences is unimportant. The users of a declarative knowledge representation language, like KIF, usually have their own application domain or, in AI terms, *universe of discourse*. KIF is not restricted to any specific domain.

KIF is most often used in connection with KQML (Knowledge Query and Manipulation Language, described in the next section), as the most frequently used content representation format of its messages. KQML essentially wrappers KIF as its internal packet structure. KIF is used the internal knowledge representation in, for example, an agent development environment by IBM, Agent Building Environment Developer's Toolkit ABE, (IBM, 1997a). While KIF is used as the standard knowledge format for rules in ABE, the KIF syntax is considered to be too far from the natural language of a common end-user. A mapping from KIF rules to a rule representation in a natural language is thus defined in ABE's user interface.

For many agent architectures, a knowledge base consisting of facts and simple *rules*, integrated with effectors or sensors (for perception) and actors or actuators (for actions) is sufficient, no deliberating or reasoning functionality within the agents is necessarily needed. A most notable example of these, pure reactive agent execution architectures is the subsumption architecture by Brooks, (Brooks, 1991). If agent architectures, however, contain reasoning or single agent planning functionality, they normally also need more advanced knowledge representation formats. In this thesis, a KIF-like representation is used in agent communication, whereas the local knowledge of the agents consists of simple rules.

4.2.4 Planning and execution of plans

Artificial Intelligence (AI) *plans* and traditional *control algorithms*, for example those within batch control, try to solve *the same general problem* in an industrial context, i.e. how to choose actions to influence an environment or a physical process in a desired, goal driven way. The levels of abstraction and the time scales for plans and controls normally

differ, AI plans are more abstract and have a longer time scale and a wider scope than controls (Dean & Wellman, 1991, p. 177).

In AI, plans are often generated and evaluated off-line, using a *world model* of the domain-specific environment for which the plans are made. The idea is that the AI plans are in effect sequences of steps, needed to achieve the chosen goal, a given state of the world model. Normally, the role of a feedback from the environment is smaller (or non-existent) within plans than within controls, where corrective action provided by on-line feedback is essential.

The plans of individual agents may, alternatively, consist of local rules and, additionally, of so-called *agent interactions* containing *interaction rules*. The interaction rules describe what an individual agent is supposed to do, when it gets a message from another agent, constraining thus its operation. In this way the plan of an individual agent consists of its own goals (local rules) and the goals imposed upon it by the need to interact with other agents in a coherent way.

The above kind of interaction (or, more specifically, *conversation* or *negotiation*, see the next section) approach is originated by Terry Winograd and Fernando Flores (Winograd, 1987). Mark Fox and Mihai Barbuceanu have concretised it in their agent co-ordination language, COOL, and also applied it into the manufacturing domain (Barbuceanu & Fox, 1995). In COOL design and implementation, the conversations are separate entities from the agents, which is a good design decision from the point of view of reusability.

The conversation approach seems promising from the point of view of automation applications, because in it, the autonomous, local decision-making of individual agents is integrated with conversations, which can be modelled with the help of conventional finite state machines. Several ‘pure’ agent frameworks are also available, implemented with Java, based on the above-mentioned conversation theory, and COOL implementation, see for example (Chauhan & Baker, 1998).

In some agent architectures individual agents are able to *change their plans while executing them*. Integrating the making and the execution of plans in control applications can be considered both in terms of *planning for execution* and (*re)planning while executing* the plan (Kuikka, 1997). A starting point for the first approach is the separation of the time consuming planning functionality from the execution of the plans. A premise for the second approach is that the planning and the execution of the plans shall be integrated into the same system. This is normally possible only by modifying or *adapting*

existing plans. The second approach is a more natural path for further development in the chosen conversation theory based approach of this thesis.

4.3 Multi-agent interaction

4.3.1 Background

According to (Finin *et al*., 1997) three fundamental facilities are required to enable agents to interact effectively: a common *language*, a common *understanding* of the knowledge representation exchanged, and the ability to *exchange* the previous two items. The common language can be seen as a common vocabulary of intention carrying messages between agents. The common understanding of the knowledge can be further divided into two parts. The first is the problem of translating from one knowledge representation to another, from KIF to rules, for example. The second part is the problem of defining the ontology, i.e. the semantic content of the knowledge. Knowledge representation has been discussed in the previous section concerning individual agents.

This section discusses multi-agent interaction, i.e. agent communication *languages* and *negotiation* or *conversation* protocols, Figure 4.2. The ability to *transport* common agent language and messages relies in this thesis on distributed software component model, on which also the agent functionality is designed.

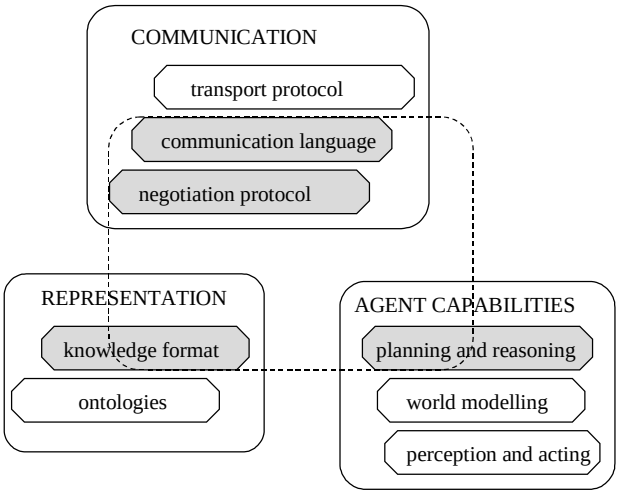

Figure 4.2. Multi-agent interaction.

In multi-agent interaction, also the *knowledge format* and the agent *planning* issues, described in the previous section, have to be considered. Knowledge representation is also needed for the agents' local knowledge base and planning for its local decision-making. Those issues belong to multi-agent interaction only to the extent that has to do with message contents, on one hand, and plan execution by conversations, on the other. The whole area of interest in multi-agent interaction is thus indicated within the dotted line in Figure 4.2.

4.3.2 Agent communication

There are several requirements for an agent communication language. Finin et. al. classify the *general requirements* into seven categories: form, content, semantics, implementation, networking, environment, and reliability (Finin *et al*., 1997). The form of the languages should be declarative, syntactically simple, and human readable. The content part of the language (expressing domain knowledge) should be separate from the proceeding of conversations. The semantics of the language should support shared understanding but, simultaneously, be formal enough for agent interaction. The implementation of the language should fit well with the software platform, especially with the underlying transport protocol and distributed network environment. Reliability is an issue, too, especially if the agents communicate in the Internet.

From the point of view of this thesis and in addition to the previous requirements, it is beneficial if the communication language is *extensible* in a domain-specific fashion. In addition to generic message contents, also contents specific to domain, in this case batch control, will be needed, as described in Chapter 7.3. It is, furthermore, necessary that the communication language should be as common and well *standardised* as possible, as it is not to be on the focus of this research and development work. When considering all of the preceding issues, there remain two language candidates: KQML and FIPA ACL.

KQML is a query language for the communication between software agents. As the knowledge representation language KIF, it has been produced by Knowledge-Sharing Effort consortium (KSE). Although there are several variations of KQML (Labrou & Finin, 1997), it can be regarded as the *de facto standard* agent communication language. One might even claim that the new, FIPA standard agent communication language FIPA ACL (FIPA, 1998) is actually a semantically well-defined variation of KQML.

KQML is both a *message format* and a *message handling protocol*. The message types convey an attitude about the content they carry (e.g. request, query), but KQML is opaque about the exact content, giving no restrictions or interpretations. The structure of the KQML message is simple (Labrou & Finin, 1997). The intention carrying message, or *performative*[3], is expressed as an ASCII string using a LISP type of list notation. Unlike in LISP, the parameters in KQML performatives are indexed by keywords and are therefore order independent.

The KQML performatives can be classified into three categories according to their usage. The *discourse* performatives convey typically a query of some kind or an attitude about the content. The purpose of these performatives is usually to make the recipient edit its knowledge base in some way. The second category holds performatives that are used to *handle errors or mechanics of the conversation*. The third category lists *special performatives*, which may be domain-specific extensions. A communication link is needed to carry KQML messages. KQML does not specify the transport protocol but it assumes that for the agents the communication appears to be point-to-point message passing.

The Foundation for Intelligent Physical Agents (FIPA; FIPA 1998) is another collaboration effort targeting to specify agent communication. The FIPA is a non-profit

[3] Performatives are originated from the so-called speech act theory. The speech act theory is derived from the linguistic analysis of human communication. The key idea is that a speaker *performs* actions when he uses language (I hereby declare …, I hereby request …).

association whose purpose is to promote the success of emerging agent-based applications, services and equipment. As indicated before, the overall structure of FIPA standardised Agent Communication Language (FIPA ACL) resembles very much KQML. The messages in FIPA ACL are called *communicative acts*, but they have the same purpose as the performatives in KQML. The communicative acts convey other mental attitudes than intention as well, namely belief and uncertainty.

The syntax of a FIPA ACL message is similar to the KQML message syntax, except that the defined communicative act set differs from the KQML performative set, as is the case with the predefined parameters, as well. There are altogether 20 message types and the set resembles the KQML category of discourse performatives. The semantics of the FIPA ACL message contents is formally defined with a Semantic Language SL. The FIPA ACL semantics is more oriented to *specifying actions of a negotiation protocol* than making a query to the receiver's knowledge base, which is typical of KQML. Well-defined semantics and orientation to agent interaction, instead of knowledge queries, were the main reasons for *choosing FIPA ACL* as the communication language for this thesis.

4.3.3 Co-operative agent negotiation

A set of agents in a distributed environment, equipped with agent communication capabilities, described in the previous section, may consider the interaction needed as either *co-operation* or *competition*. If the agents, for example, belong to one control or process management system or represent a robotics society owned by one company, they are most likely to co-operate to achieve common goals. If the interacting agents, however, represent different organisations, they will most likely have at least somewhat different goals from each other.

The negotiation problem can in general be categorised into domains according to the type of the goal (Rosenschein & Zlotkin, 1994):

- *Task oriented domain*. The activity of an agent can be defined as a set of tasks it has to achieve.

- *State oriented domain*. The mission of an agent is to move the world from an initial state to a goal state.

- *Worth oriented domain*. The agents have worth for each potential state.

To make the negotiation between agents possible, some mechanisms are needed to co-ordinate the process. A specified negotiation process or protocol includes *rules* about what the allowed actions are and how the negotiation comes to an end (or how to recognise that a negotiation result has been achieved). In a seminal work on negotiation (Rosenschein & Zlotkin, 1994), negotiation protocols are studied from a game theoretic point of view and evaluated e.g. on the basis of the properties of efficiency, stability, simplicity, distribution and symmetry. Each individual agent will, in addition, need a negotiation strategy of its own, which is a predefined (pre-programmed) set of decisions that the agent will make in certain situations.

When the nature of the required agent interaction is co-operative, co-ordination is needed to manage the dependencies between the agent activities (Kuikka & Valtari, 1998). Co-ordination problems arise, because there are alternative actions that an individual agent can choose from its local plan and, furthermore, the order and the execution time of the chosen actions affect other agents and the whole environment. There is thus a need to explicitly *represent the interactions* taking place between the agents. One important model for including these aspects of co-operation is the *Joint responsibility model* (Jennings *et al*., 1995), described in the next section in connection with the GRATE* multi-agent system.

Another model, adapted for the main approach in this thesis, is based on *conversations*, specified for individual agents' interaction with other agents, as described in the previous section (Barbuceanu & Fox, 1995). The use of conversations as units of *modelling interaction between agents* has proven to be more appropriate than the plain performatives in several industrial applications (Bradshaw *et al.*, 1997). The conversation-based interaction model does not presuppose any intellectual capabilities of the agents participating in the interaction, which makes it possible to include many kinds of agents, also simple ones, in the interaction.

The chosen approach for co-ordination is supported by both the analysis possibilities to verify coherence (reaching a common goal) and the fact that it is possible to further develop the conversations with decision theoretic planning as indicated in (Barbuceanu & Fox, 1997). It is also possible to have the agents refine their conversation rules when new knowledge is achieved from other agents, which is in effect one form of (re)planning while executing the plans (Kuikka, 1997).

4.4 Multi-agent systems

4.4.1 Background

In multi-agent systems (MAS), agents are autonomous and typically heterogeneous entities. In addition to interaction, they are capable of sensing the environment (consisting of the physical world, non-agent applications and other agents) and act accordingly in a purposeful manner. Research on MAS is mainly concerned with communication and negotiation mechanisms between agents, described in the previous chapter, as well as *organising* the agents structurally *into multi-agent systems*, and *applying them* in various application domains.

Previously multi-agent applications were classified into three areas: distributed situation assessment, distributed scheduling, planning and resource allocation as well as distributed (mainly rule-based) expert systems (Lesser, 1995). In the near future, *relevant domain-specific problem solving techniques* from the above-mentioned technology area based categories can be *integrated into existing information systems and frameworks*. Additionally, *evolutionary and learning aspects* can be incorporated. Agent functionality can be integrated (for example, in the manner developed in Chapter 7 of this thesis) into distributed, software component frameworks. The agent functionality can provide autonomous solutions for e.g. resource usage, configuration, and security problems.

The agents, negotiating with each other, as described in the previous section, can be viewed from an organisational point of view, as a *society*. The agents work to achieve their own goals and also common or *joint goals* of the society. When considering co-operating multi-agent applications, having a well defined *joint goal*, it is reasonable to *combine the traditional deliberative and the newer reactive approach* for agency. Striving for a goal which requires co-ordination from the system as a whole - while maintaining autonomy of the individual agents - implies the deliberative approach. The inevitability of changes and unknown events in the environment and the requirements for flexibility and robustness imply the reactive approach.

It is also important to take care of the *coherence* of the multi-agent system architecture. It was discussed in terms of agent interaction in the previous section. It may also be defined more generally, in terms of solution quality, efficiency, conceptual clarity of system behaviour and the possibility of graceful degradation (Aitken *et al*., 1994). These needs, as well as both the deliberative and the reactive approaches mentioned above, have been pursued in the multi-agent architectures described next.

4.4.2 Domain independent multi-agent systems

The TouringMachine architecture and BDI agents

In the TouringMachine Architecture, Figure 4.3, (Ferguson, 1995) individual agents comprise three concurrently operating, task-achieving control *layers*.

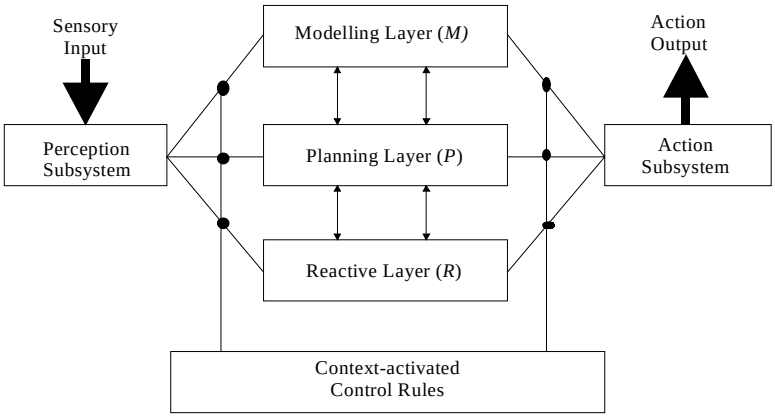

Figure 4.3. A TouringMachine (modified from Ferguson, 1995).

The Reactive layer (*R*) provides the agent with reactive, not planned capabilities. The Planning layer (*P*) allows the agent to generate, execute and dynamically repair hierarchical partial plans. The Modelling layer (*M*) allows the agent to construct behavioural models of world entities that can be used to explain observed behaviours and to predict possible future behaviours. Each layer is a so-called approximate machine, having incomplete world models. The layers are mediated (by modifying layer's data) by interlayer message-passing control with domain-specific control rules.

The structures used by an agent to model behaviour are 4-tuples of the form **<C, B, D, I>**. **C** is the agent's configuration (for example the location, speed, acceleration, and orientation of a robot). **B** is the set of the agent's *beliefs* (about the environment and other agents). **D** is its list of prioritised goals or *desires* and **I** its individual *intention* structure. The concept of *intention* (Cohen & Levesque, 1990) is used to define an agent's individual behaviour in a social context. It can be defined, in short, as “a commitment to act whilst in a certain mental state”. Intentions should be internally consistent and

consistent with the beliefs of the other agents in the multi-agent society. BDI-models[4] within the TouringMachines are implemented by templates, which the agent obtains from an internal model library.

Reasoning with behavioural models involves looking for the "interaction of observation and prediction" in a somewhat analogous manner as in *model predictive control*. First any discrepancies between the actual behaviour of an agent and that desired by it, are observed. Once the discrepancies have been identified, predictions are formed by projecting temporally the agent's configuration vector C in the context of the current world situation and the agent's intentions. Should any conflicts exist, the agent should have enough knowledge (priorities of goals, space-time urgency information, constraints) to resolve them. The conflicts can be either intra-agent (between the agent's own predicted and desired actions) or inter-agent (between the agent's predicted actions and other agents' predicted actions).

TouringMachines have been implemented as mobile agents in a simulated multi-agent traffic navigation domain, the TouringWorld. Tasks, carried out by the agents, are prioritised in advance and include goals like moving from an initial location to a target destination within certain time bounds and/or spatial constraints, avoiding collisions (with other agents and obstacles) and obeying a set of traffic rules. The if-then rules act effectively as filters between the agent's sensors and its internal layers (censors) and between its layers and action effectors (suppressors). This approach is analogous to that of suppression and inhibition in Brook’s subsumption architecture, (Brooks, 1991).

Although complete BDI-modelling, exemplified by the TouringMachines, is theoretically interesting in agency, need for it is highly application specific. The process unit allocation problem - to be solved using multi-agency in this thesis - does not require modelling of beliefs or desires, for example. The information that an agent gets in Process Cell Management from other agents and software components can be considered trustworthy and the goals of individual agents do not call for complex intention structures. The conversation approach adapted in Chapter 7.3, for agent interaction does, however, yield the future use of complete BDI-models for individual agents, if/when needed.

[4] If the agents are *considered* to have beliefs, intentions, and desires or similar mental attributes, they are called strong agents, if they do not have this kind of intentional stance, they are weak agents.

The TouringMachine Architecture described previously, as well as Brooks' subsumption architecture and many of its reactive successors, are *horizontally layered multi-agent architectures*. In these architectures, all layers of an agent have access both to the perception and to the action entities. This necessitates the use of some kind of centralised control authority (context-activated control rules in TouringMachines) and brings about complexity as to the concurrent handling of perceptions and actions by various layers. The observation above has led to the development of *vertical multi-agent architectures*, of which InterRAP, (Mueller *et al.*, 1994; Mueller, 1996) is a good example, Figure 4.4.

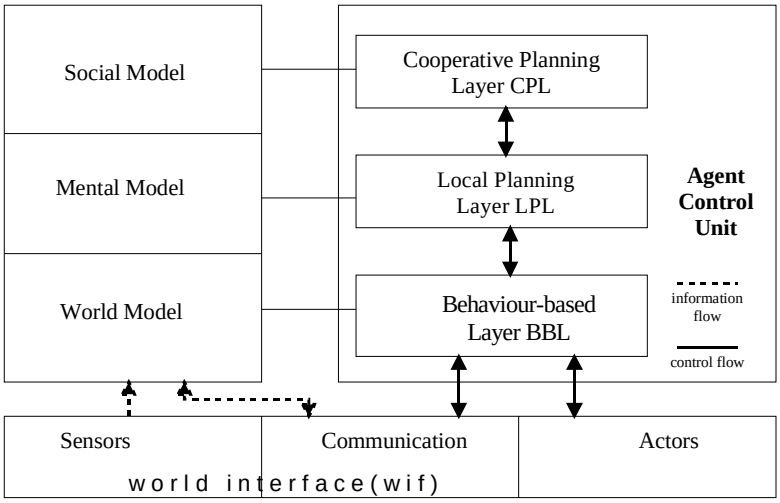

Figure 4.4. The InterRAP Architecture (modified from Mueller, 1996).

An InterRAP agent consists of a set of *functional layers*, linked by an activation-based control structure and a shared hierarchical knowledge base. The world interface (wif) contains the agent's facilities for perception, action and communication. The Behaviour-Based Layer (BBL) implements and controls the basic reactive behaviour of the agent as well as its procedural knowledge (abstract actions). The Local Planning Layer (LPL) contains a planning mechanism, which is able to activate subplans or primitive actions to achieve goals. Planning is also needed to co-ordinate the actions of the agents, for example to devise joint plans. The Co-operative Planning Layer (CPL) takes care of the co-ordination.

The agent *knowledge base* is a hierarchical blackboard system, split into three layers corresponding the functional layers above. The agent's World Model contains its object level beliefs about the world; the Mental Model holds information on goals, plans and intentions. The Social Model represents what the agent believes about other agents, and contains information about joint goals, plans and intentions.

The overall *control behaviour* of an InterRAP agent emerges from the communication among the layers. Based on events in the world, recognised by the agent, control is shifted upward until the appropriate layer deals with the situation. There are three execution paths: the reactive path, the local planning path and the co-operative planning path, each indicating the highest layer needed.

There are some interesting similarities and differences between the pure agent architecture of InterRAP above, and the overall architectural structure of agentifying of the framework developed in Chapter 7.3 of this thesis. The BBL layer is replaced in the framework by the (reused) host software component. The LPL layer is designed in both architectures using local if-then rules, which in the design solution of the thesis have access to the attributes of the host software component itself. The CPL layer corresponds to the conversation rules of the individual framework agents, which are connected with other agents with the common agent interaction interface, used only for agent negotiation.

The approach of the thesis, unlike InterRAP, separates clearly normal component communication from agent negotiation. Agents need to communicate, not only with normal components, but also with other agents in a nonagentified manner. For example, when a PathAgent in *Process Cell Management* wants to read a state variable, containing the previous BatchId, from the UnitAgent, there is no need to use an agent performative, a normal software component interface is sufficient. In the view of this thesis, only actual agent negotiation shall take place through agent interfaces. This approach, detailed in Chapter 7.3, separates the concerns of agent interaction and normal component communication and also helps to keep the number of the conversation rules in the interaction small and their structure concise.

The GRATE* architecture and Joint Responsibility

Most multi-agent architectures provide possibilities for agent interaction by signals, traditional data message based communication or speech-act-based interaction. The (generic) multi-agent architectures do not as such consider the way in which the

negotiation is carried out - leaving it to the application. The GRATE* architecture (Jennings *et al.*, 1995), together with the conversation approaches discussed previously, are notable exceptions in this respect.

The GRATE* architecture implements the so-called *Joint Responsibility Model* of agent collaboration, based on so-called *joint intentions* of the agents. Joint responsibility is needed when the system has global constraints (total cost or timing, for example), when activities by individual agents are interdependent (either positively or negatively), and when no one agent has sufficient competence, resources or information to perform the entire task alone. The model consists of two constructs: defining *individual behaviour in a social context* and defining *co-operative behaviour*.

The intentions of individual agents, which define their individual behaviour, should be stable enough and follow general policies or conventions, which tell when they should be re-examined. For the intentions to function, the commitment must operate in a way that ensures that agents behave rationally. Specifically, an agent should *reconsider its commitments* to a goal (G) if:

- G is already satisfied.
- G will never be satisfied.
- Motivation for G is no longer present.

Joint intention can be defined as a joint commitment to perform a collective action while being in a certain shared mental state. According to Jennings (Jennings *et al*., 1995), it is necessary but not sufficient to have the agents commit themselves to a common goal, the agents must also want to *achieve their goal in a co-operative manner*. There must thus be a commonly agreed *plan*, which the agents are working under.

At any instant of time, the plan is likely to be *partial*, either informationally (lacking parameter values), temporally (lacking exact ordering) or structurally (lacking detailed actions). Refining the partial plan is a complex activity for which various kinds of (often domain-specific) planning paradigms are needed.

Joint responsibility provides an explicit model of co-operation. All the agents have a joint goal and they execute *a common* plan to which they have commitment and, additionally, all the agents know mutually what the others are doing. The GRATE* *implementation* of

the Joint Responsibility model adopts a rule based approach, having so-called situation assessment rules and co-operative rules. The *situation assessment rules* are needed to:

- Decide when co-operation for a joint goal is appropriate.
- Develop an agreement on a common plan.
- Ensure that the existing commitments are honoured and the new ones are consistent with the old ones.
- Monitor the problem solving state.
- Decide what to do, if a joint commitment is dropped.

The *rules, which control co-operative interactions,* ensure that all team members are informed, if the local agent gives up its joint commitment, and they also propose remedial actions. Before the *co-operation* in a system with GRATE* architecture can proceed, a *joint action* must be established. After the agent has selected an appropriate plan to achieve the desired goal, it has to determine, whether the activity should be completed locally or whether assistance should be sought from the team members. If a joint action with several agents is needed, the agent that detects the need, becomes a so-called organiser.

After the need for a joint action has been identified, the process of establishing a co-operating group can proceed:

Phase 1

Organiser detects need for joint action to achieve goal G and determines the plan R

Organiser contacts all acquaintances, capable of contributing to R to determine if they will participate in the joint action using Joint responsibility co-operation model

W = set of willing acquaintances

Once all the team members who were identified as potential participants have replied, the *second phase* of the protocol begins and the team specifies the exact details of the common plan:

Phase 2

FORALL actions in R

select agent A from W to carry out action f belonging to R

(criteria: minimises number of group members)

calculate time t for f to be performed based on temporal orderings of R and the anticipated communication delay

send (f, t_f) proposal to A

A evaluates proposal against existing commitments (C's):

IF no-conflict(f, t_f)

THEN create commitment C_f for A to (f, t_f)

IF conflicts((f, t_f), C) AND priority(f) > priority(C)

THEN create commitment C_f for A to (f, t_f) and re-schedule C

IF conflicts((f, t_f), C) AND priority(f) < priority(C)

THEN find free time (t_f + delta t_f), note commitment C_f and

return updated time to leader

return acceptance or modified time to team organiser

IF time proposal modified

THEN update remaining actions times by delta t_f

END-FORALL

The process of agreeing on a time for each action continues until all of the actions have been dealt with. At this point the common plan is agreed upon and the organiser informs all team members about the final solution. The joint action is now operational and the agent monitors its execution - to find out when it should reconsider its commitments according to the Joint responsibility model and thus start a new distributed planning cycle for a new joint action.

Although the joint responsibility model and the conversation based planning approaches may seem totally different from the outset; they do have significant similarities. The if-then rules of an individual agent above correspond to the conversation rules in the conversation approach. In the Joint responsibility model there is, however, a common,

global plan[5], whereas in the conversation approach a common plan is not explicit, but consists of the conversations of the individual agents.

The approach of this thesis is, as described earlier, to use the conversation theory as the basic mechanism of interaction (Barbuceanu & Fox, 1995). The concept of a central 'organiser' has, however, been adopted from the Joint responsibility model.[6] Among the batch process cell management, the ProcessCellAgent is the organiser, which initiates the negotiation process with feasible UnitAgents and PathAgents after it has received the unit allocation request from the Manage Batches control activity. Also the use of local planning rules is analogous in the Joint responsibility model and the approach of this thesis, presented in detail in Chapter 7.3.

4.4.3 Multi-agent systems in automation domain

All the aforementioned multi-agent systems are generic in the meaning that they do not make a difference between various categories of agents. Agents may be internally heterogeneous, but their specific tasks cannot be seen in the architecture as a whole. *Specialisation of agent categories* can, however, be incorporated into the agent architecture, as well.

In one of many proposed *multi-agent architectures for computer integrated manufacturing*, (Rabelo & Camarinha-Matos, 1994) four categories of agents are identified: the scheduling supervisor agent, the local spreading centre agent, the enterprise activity agent and the consortium agent. The scheduling supervisor is created to control the execution of schedules. The local spreading centre agents are needed for groups of production facilities (one agent for milling operations, one for turning operations, one for transporting etc.). The enterprise activity agent represents a local controller of the production resource, having also a production maximising function. The consortium agent represents a group of enterprise activity agents, specially created to execute a certain job and destroy itself after that.

---

[5] Or 'Recipe' (R), as Jennings calls it – not to be mixed with batch control recipes, which is the reason, for not using the term here.

[6] The need for this kind of organiser, or facilitator, as it is also called, has been noticed also in several other multi-agent systems, see, for example (Finin *et al*., 1997).

A somewhat similar, *domain-specific approach*, has also been suggested by Kuroda and Ishida (Kuroda & Ishida, 1993) *for pipeless chemical batch processes*. All functional units in the control architecture for batch production plant are categorised into Process Managers, Reactors and Stations, Figure 4.5. The pipeless physical process consists of respective, mobile reactor vessels, which are moved from one static processing station to another during the production of a batch.

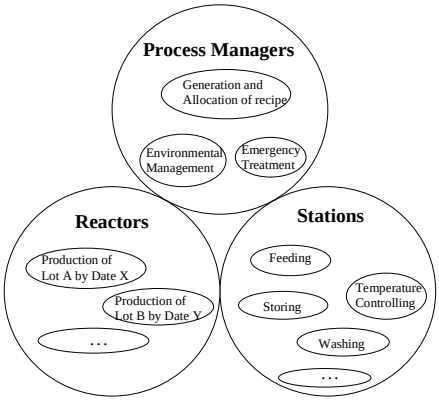

Figure 4.5. Task sharing of a batch production process (modified from Kuroda & Ishida, 1993).

The assignment of subtasks to functional units and the schedule of subtask-execution are completed by using negotiation message exchanges according to the *contract net model* (Smith, 1980). When a unit needs another unit's support in collaboration (for example, a reactor requires material from a feeder station) a contract is made, which assigns a unit and plans a work-starting time by local message exchange. Making a *contract* proceeds with advertising (making a tender) by the master, submitting bids (offering) by the potential subcontracting units and evaluating the bids and selecting (ordering) an appropriate subcontractor by the master.

The total decision-making in the system takes place in *two phases*. The *static evaluation* by the Process Manager selects and orders suitable station-candidates and orders them for each operation at the beginning of a production lot. The candidates and the evaluation results are recorded in a recipe. In *dynamic evaluation***,** as described above, a Reactor starts the lot based on the recipe and selects the most suitable Station(s) among the multiple candidates taking the static evaluation results and the dynamic situations of stations into consideration.

By using a contract net, decomposable tasks can be distributed among a group of agents in a flexible manner. Global coherence should be achieved through negotiation as a mechanism for interaction, through task decomposition and the common language shared by all agents. However, if the tasks cannot be easily divided, then the synthesis of the results may be problematic, (Aitken *et al.*, 1994). It has also been noticed that a simple bidding process does not lead to a satisfactory result because longer sequences of resource exchanges are not explored. Better solutions have been obtained by gathering 'market statistics' and adding a new inference rule that uses these statistics in the selection process.

The conversation approach of this thesis has *similarities and differences* with the Kuroda–Ishida approach. Both approaches have two phases, the first, the static phase finds the feasible process Units or Stations, respectively. Unlike the Kuroda–Ishida approach, the framework of this thesis does not need negotiation for the first phase. Starting with the actual, dynamic negotiation by an 'organiser' (the Process Manager in the Kuroda–Ishida approach and the ProcessCellAgent in the framework) is essentially similar in both approaches. However, while in the dynamic negotiation of this thesis the singleton ProcessCellAgent is participating, in the Kuroda–Ishida approach the dynamic negotiation phase is carried out by the mobile Reactors and static Stations alone.

In the framework of this thesis, multi-agent negotiation is only used in a specific, dynamic unit allocation problem, concerning all types of process Units in a similar manner, detailed in Chapter 7.3. In the Kuroda–Ishida approach, the Process Manager negotiates first with the Stations and finds out production candidates among them, using a specific set of criteria. This static station information is then inserted in (non-standard) 'Recipes', which are used by Reactors when they negotiate with the Stations using another set of decision criteria. In the framework, unlike the Kuroda–Ishida approach, there is no need for several types of negotiations between the interacting agents. The conversation approach applying the modified contract net protocol of Chapter 7.3, was thus sufficient in comparison to the two-phased, more complex approach used by Kuroda–Ishida.

4.4.4 Developing multi-agent applications

Customising multi-agent systems

When either a generic or domain-specific multi-agent system, capable of working together in a co-ordinated, collaborative manner has been developed - or acquired - the *application specific goal* has to be imposed upon it. The goal in the automation area of applications may be as varied as, for example, a production optimisation task, a motion planning task or an allocation or a scheduling problem. That is why the ways in which the common goal is presented to the multi-agent system also vary considerably. The representations are also dependent on the degree of specialisation of the multi-agent architecture itself.

Additionally, the real environments for multi-agent systems are dynamic and populated by multiple co-operative agents and possibly also by other software modules. This offers continuously new possibilities (if only for a limited time interval) for the system to achieve the original goal, making, however, the goal setting more difficult. Explicit *meta-level reasoning* in the multi-agent system implementation can be utilised and thus goal setting in the form of so-called *societal rules* or norms can be employed when expressing the needed behaviour of the multi-agent system.

However, it is often more effective to *incorporate* meta reasoning *in the multi-agent architecture* (or agentified framework) itself like, for example, in the Joint Responsibility Model of GRATE* and in the conversation rule scheme of the co-ordination language COOL. In GRATE*, the application specific rules have to be coded into the situation assessment module. In addition to local action, also the other team members have to be informed when an individual agent's commitment is given up. This is realised by the co-operation module of GRATE*, based on the information provided by the situation assessment module.

In the domain-specific CIM-scheduling approach (Rabelo & Camarinha-Matos, 1994), a common goal for the multi-agent system is given in terms of a hierarchy of business processes (BP). The deepest level in the hierarchy represents an elementary task description that can be implemented by a basic functionality, the so-called enterprise activity (EA). For the specification of the interrelationships between BPs and/or EAs, procedure rule sets (PRS) are utilised. To use more conventional manufacturing control terminology, BP corresponds to a job, an EA to a process plan operation and a PRS to precedence relations between the operations. The CIM-scheduling architecture's

specialised agents then negotiate until one enterprise activity agent or a group of them is selected to execute all the BP's EAs.

Batch production, in the Kuroda and Ishida's domain-specific architecture (Kuroda & Ishida, 1993), is started by a user presenting a recipe (containing minimum information about the process sequence) to the system's Process Manager. It will then be further refined and sent to a Reactor by the Process Manager. The Reactor interprets the recipe with its own domain-specific knowledge and starts the production process in co-operation with functional Stations having also domain-specific information contents.

Agentifying information systems and frameworks

The possibilities of *multi-agency* for distributed, autonomous, and flexible decision-making, presented in this chapter, are often needed *locally in the domain*, for *specific problems* in the application, and possibly even *intermittently* in systems having other kinds of software architectures. There is thus a definite need to *agentify* existing information systems and software frameworks.

There are far fewer *examples of enhancement* than visions and expressions of a *need to enhance* the operational functionality of automation systems by adding agent features to them. Within user interfaces and information retrieval, on the other hand, so-called interface agency is gradually becoming commonplace (Guilfoyle, 1998; Lieberman, 1998). It seems to be difficult to integrate within automation systems, both conventional information representation and communication, and knowledge representation and agent interaction.

The co-operative Intelligent Real-time Control Architecture (Musliner *et al.*, 1995), described previously, is one example of agency in the automation domain. The CIRCA architecture as a whole was, however, designed using the agent paradigm, and thus the division of work and interaction between the agentified and nonagentified parts of the system was comparatively straightforward. The KaOS agent architecture puts emphasis on the integration of objects and agents, as well, but the goal in its development is explicitly An Open *Agent* Architecture (Bradshaw *et al.*, 1997).

Multi-agent features, described in this chapter, can however, be *embedded* in new, distributed, domain-specific component frameworks. This can be achieved *locally*, in those parts of the framework, where a genuine, domain-specific need for agency is. It can

even be accomplished in such a manner that the underlying software component *framework* remains intact and is thus *reused* as such.

In the approach developed in this thesis, the starting point for agency enhancement are the component classes (so-called host components) of the framework itself, or alternatively, of an application, based on the framework. The host components have been derived from the design pattern based class model of the framework and have the domain-specific component interfaces. Those specific *components that need multi-agency* capabilities, can be *agentified* by incorporating into the so-called *agentified components*:

- The host component by component containment
- The knowledge base and inference mechanism (consisting of interactions, interaction rules and associated methods) by object implementation inheritance from a generic agent class
- The interaction capability for agent interactions (consisting of component interface for FIPA ACL communicative acts) by interface inheritance from a generic interface

All domain-specific components of the framework can be agentified – if/when needed - in this manner by designing and implementing the problem specific agency details only and *reusing the host components of the framework as such*.

There is not - according to the knowledge of the author – any other software component framework that has been *agentified*, or enhanced by agency, in the manner described above. Thus an original design pattern has also been developed to support this generic design issue. The enhancement of the domain-specific batch process management framework of this thesis by agentifying, as well as the design pattern, *Agentified Component* are presented in detail in Chapter 7.3. The specific process management problem, which requires agentifying, process unit allocation, is also described in detail in Chapter 7.3.

5. The Research and Development Problem

5.1 Introduction

Batch process management, described in Chapter 2, needs a new kind of approach to system architecture development, outlined in Chapter 1. Information technologies, described in Chapters 3 and 4, offer significant possibilities for fulfilling these needs. *The main research and development task* of this thesis is thus to develop an experimental *batch-process management framework* which*:*

- *Conforms to the requirements of the new application-domain standards and considers additional, advanced industrial needs.*

- *Relies for its design and implementation on distributed object and software component technology and on design pattern methodology.*

- *Facilitates local autonomous decision-making through multi-agent technology, embedded in the framework.*

A frame of reference for the chosen approach, Figure 5.1, emphasises the need to consider *domain-specific requirements*, realised in batch domain standards and industrial experience, as a starting point for framework development with the help of new information technology.

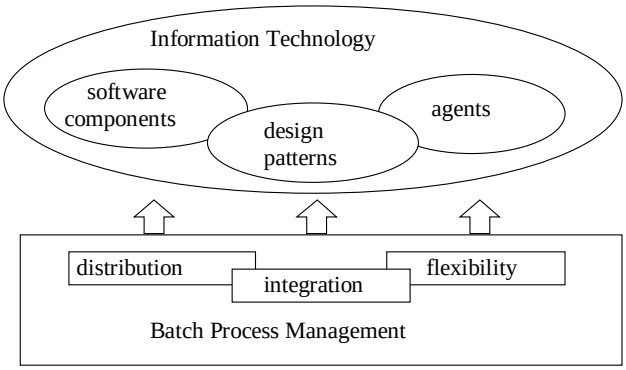

Figure 5.1. A frame of reference for the approach of the thesis.

The most important application-domain standards in batch control are ISA-S88.01: Batch control - Part 1: Models and Terminology (ISA, 1995) and ISA-dS88.02: Batch control -

Part 2: Data structures and Guidelines for Programming languages (ISA, 1999a). They are described in Chapter 2.2 and will be referred to as *the standard* hereafter. Commercial batch automation systems, which are at least partly compliant with the standard, are discussed in Chapter 2.3. Chapter 2 thus gives a concise view on the present state of the art in industry and, simultaneously, acts as *a reference* to which the concepts developed in Chapter 6 of this thesis can be compared.

Application-oriented research work is discussed in Chapter 2.4. The research in batch-process management has largely concentrated on development of *analysis* methodologies. Lately there has also been some research on architecture development and automation-application design, as indicated in Chapter 2.4. Because the research has focused on analysis problems, it has put less weight on the issues of functional distribution, integration, and flexibility, needed for variable and exceptional processing conditions and the exploitation of the autonomy of new processing equipment. This thesis thus strives to *complement* the existing batch-control research in the above-mentioned respects.

Chapter 3 gives a short overview of distributed object and software component technology (Chapter 3.2), as well as a more detailed description on how to exploit design pattern methodology (Chapter 3.3) in developing domain-specific frameworks (Chapter 3.4). Two examples of frameworks for related approaches and analogous domains have also been described as information technology *references* to the framework of this thesis.

Chapter 4 describes multi-agent technology. The issues concerning individual agents, their knowledge representation and planning capabilities are described briefly in Chapter 4.2. Multi-agent interaction is discussed, concentrating particularly on co-operative negotiation (Chapter 4.3). Multi-agent architectures are also reviewed, and finally an *alternative to pure multi-agent systems* is introduced, incorporating multi-agency within component frameworks through *agentifying* in Chapter 4.4.

The selected information technologies are considered in this thesis to be vitally *important in developing domain-specific frameworks*. In many cases these technologies are, however, deployed in isolation from each other, without realizing the value of *integrating* them. Object orientation, software components and design patterns are considered as fundamental technologies, whereas multi-agency is seen as a means to enhance the (reused) framework in a problem-specific manner. The following sections justify the chosen approach. Both the batch-process management and the information technology points of view are considered.

5.2 Justification of the approach from the domain point of view

5.2.1 In reference to the standard and advanced industrial needs

The standard is an excellent *basis for framework development*. It does not, however, address explicitly the issues of *varying* and *exceptional processing conditions*, considered important for *flexible production* in advanced industrial applications. Furthermore, the standard does not consider the possibility of *adding value* in the production by enabling *autonomous decision-making* within process units. These issues have been considered necessary requirements for the new experimental framework.

There is a great need for *co-ordination control* in batch-process management. The *separation of concerns* of the products (explicated in various types of recipes of the standard) on the one hand and the equipment (explicated in process and control equipment modularity) on the other hand, require *independence* of the corresponding information entities. At the same time they must, however, be able to *interoperate* closely in varying processing conditions in order to achieve the overall goals of *efficient production*.

At best the co-ordination should maintain independent operation also in *situations when connections* between the entities *will be temporarily lost*. This implies the need of *autonomous functionality* (a generalization of set point based control) in the control activities. The economic need to run the plant with a minimum number of operators brings about requirements to distribute the user interfaces of the framework flexibly among the control rooms. Changes of *user interface locations* from one control room to another, as well as dynamic *extensions* of the user interfaces are needed.

Also the *exceptional processing conditions* in batch production have to be provided for. When a batch-production sequence is stopped due to an equipment failure or a material jam, for example, it must react immediately in a purposeful manner. In these cases, it often has to be able to dynamically change the normal order of processing steps (recipe procedural elements).

The *requirements imposed upon the control-system architecture* by the needs above are interesting. In order to be able to react optimally to these kinds of malfunctions, both recipe procedural elements and equipment entities (for example process units) have to be independent *active entities*, and their functioning must be *explicitly state dependent*. Moreover, the larger *containing entities*, namely control recipes for products and paths (or trains) of units, have to be *flexibly configurable*.

A need for *agentifying* in batch process management may arise if a processing unit component, for example, has enough local information to decide whether or not to participate in producing a given recipe. The facts and the knowledge may be available, for instance, in terms of the services and their costs as well as the operational state and allocation status of the unit. The unit may then find out whether or not it is *locally optimal* to respond affirmatively to a service request and if so, for what price the service can be provided.

5.2.2 In reference to existing batch automation systems

The commercial batch automation systems described in Chapter 2.3 comply fairly well with the standard as to the modelling and terminology of the batch plant and the recipes. Some non-standard concepts also exist, due to the need for compatibility with older versions of the systems. The implementation of the system architectures is generally *conventional*. The distribution of functionality is mainly based on hardware distribution to client and server computers only.

The present design practices in both automation system and application development are, for the most part, based on *structured analysis and design*. The commercial control system architectures have also been developed in a structured approach, considering the required functionality and the data separately. Interaction and collaboration possibilities in the architectures are limited, which also makes the three referred systems, which represent the foremost in the domain, unnecessarily *closed*. Object orientation is exploited in commercial batch automation systems only partially, the structure of all the referred systems is explicitly execution engine and relational database oriented.

The situation is, however, *improving* due to the fact that some vendors have developed the system functionality partly with the help of (ActiveX) components. These components can be embedded into *component containers* developed by other vendors. The containers can be both automation systems and so-called back-office products. Also the fact that a part of the functionality consists of components with well-defined interfaces makes it in principle possible for other vendors to implement alternative functionality, replacing the original implementation but having the same interfaces.

It is argued in this thesis that a *distributed software component model* is a *necessary but not sufficient* condition for the achievement of co-ordination and interoperation for closely related batch-control functionality. As discussed in Chapter 3, even component

composition is not enough, the collaborating components have to be designed into architecturally larger reusable entities, *frameworks*. Their structure and behaviour shall be based on batch domain-specific semantics.

Moreover, several of the components included in the frameworks shall be *active components*, executing in their own threads. As such, they can be used as individual components, executing in parallel within commercial batch automation systems. Especially interesting in this respect are the *agentified components*, not found in commercial batch automation systems. They are best applied for achieving value-added goals (for example cost reduction in process unit allocation) which need autonomous local decision-making and co-operative negotiation.

5.2.3 In reference to the research work in batch control

The research approaches reported in Chapter 2.4 do not explicitly solve application-level distribution issues. Their emphasis is on *analysis*, and, to some extent, recipe execution as well as on *modelling* the batch equipment. The approaches have, however, exploited some object orientation in batch control implementation. In Johnsson and Årzen's approach (Johnsson & Årzen, 1998), object-oriented G2 - methods may be invoked from within recipes, and synchronisation is encapsulated by object tokens. In Tittus's approach (Tittus *et al.*, 1995), inheritance has been used in process and control-equipment models.

Distributed computation aspects have not been considered in the approaches, in addition to the modelling and analysis of concurrent activities. In the view of this thesis, the *Petri Net functionality* could be *generalised*, but it should *not* be *presented* to batch control system users as Petri Nets - or the like. For better understanding and communication, recipes should be represented as described in the *Procedural Function Chart* (PFC) of the developing ISA88.02 standard (ISA, 1999a).

Simensen's approach (Simensen *et al.*, 1997), reported in Chapter 2.4, proposes the modelling of batch-control architecture by object- modelling techniques. An interesting original, but non-standard, aspect to batch control is the concept of the batch *life cycle*, integrating both procedural control and equipment control into one entity having time dependent properties. This thesis prefers the standard's approach of separating the procedural control and the equipment control - considering it one of the strengths of the standard - but values the above-mentioned time-dependent 'change of class'. An analogous idea is *refined and further developed* in the thesis by making a control recipe

change its class when changing its state - by using a State design pattern, detailed in Chapter 6.4.

This thesis thus *complements* the existing research work. It introduces a batch-standard-based experimental framework, which is implemented on a distributed software component platform. Various kinds of batch control research problems can be experimented and explored using the framework which has active components executing in parallel in threads of their own. The fact that the components can be agentified increases its research potential substantially.

5.3 Justification of the approach from the technology point of view

5.3.1 In reference to component technology

The design and implementation of the new batch-process management framework is not restricted to any present architectures. The architectural style of the prototype framework is a *functionally distributed, multi-tiered component architecture*. Clients and servers, for example, are functional *roles* in relation to software components, adopted according to the batch-control activities of the standard. They are *not predominantly physical processes, let alone computers*, as in traditional architectures.

The framework provides software *component interfaces to other systems* which are willing to act in the role of its clients. A relational database is not used in the framework, but the components, some of which are active, are made persistent by object *serialisation*. The deployment of serialisation services provided by the Microsoft Foundation Classes class library is here considered as a proper implementation technique due to the similarity to the approach of another component technology, JavaBeans.

Domain-independent composition of individual components, even with the help of various formal and informal techniques and generators, is *not* the most effective way to reuse software in domain-specific contexts. The composition techniques, although needed, are not sufficient in domain-specific application development. Instead, *domain-specific* software *frameworks*, consisting of components, shall be designed. It is further argued that one suitable approach for developing these kinds of frameworks is based on *design patterns*.

5.3.2 In reference to design patterns

When considering *the way in which to deploy design patterns*, it is argued that it is best to use *the published, original patterns* with comprehensive guidelines. By relying on the knowledge and documentation of familiar design patterns, it is easier for other people to understand - and also criticise - design decisions made when developing the framework, as well as to learn from them. This approach also facilitates a transfer of design knowledge from one application domain to another.

As indicated in Chapter 3.3, *too many 'new' patterns* unfortunately exist. They resemble closely the original, already catalogued and publicly available patterns. Accordingly, they tend to confuse the application and framework developers rather than contribute to the common design knowledge. Another frequent pitfall in using design patterns in development work is to try to apply them always and everywhere. Design patterns often complicate the design and may also cause performance degradation, which are good reasons *not* to apply a specific design pattern in some contexts.

Design patterns should thus be *applied judiciously*, only when they provide domain-specific flexibility and variability needed in the framework. This thesis adheres to a restrained application of well-known generic design patterns. It deploys them in a manner which makes the domain-specific requirements explicit. This approach is developed in Chapter 3.4, and tested in Chapter 6 within the development of an experimental batch-process management framework. Only when enhancing the developed framework with agentifying (Chapter 7.3) has a totally new generic design pattern *Agentified Component* been needed and developed.

5.3.3 In reference to domain-specific frameworks

Without architectural frameworks, the semantic level interoperability of the components does not succeed, and the productivity potential when developing applications will not be reached. An architecture should thus consist of component *frameworks as the main reusable entities for application developers* in a given domain. The framework provides a domain ontology based skeleton for an application or for a part of it.

With reference to the example frameworks in Chapter 3.4, in this thesis it is considered best to *start from domain (standard) based requirements* models and proceed to a domain-specific framework using generic design patterns. Only after the architectural aspects have been incorporated into the framework, will there be enough information to specify the component interfaces in detail. This approach is different from the one adopted in the Sematech project, in which interfaces of individual components have been specified - by the Sematech consortium - before architectural design decisions have been made - by individual system and software vendors.

The approach of the Sematech project is clearly *component-oriented*. It is primarily concerned with standardising external properties of individual components. The approach of the Dover project, on the other hand, is system- or *architecture-oriented*. The focus is on specifying how the objects are organised and how they communicate with each other. It is argued here that both of the above aspects, component orientation and architecture orientation, are needed.

This thesis *integrates the two approaches* in framework development by *designing* the framework architecture with an architectural design technology, design patterns, and by *deploying* existing component middle-ware technology in implementation. It also provides the framework with *outer component interfaces*, which comply semantically with the batch standard. The thesis does not, however, argue for the compliance of all of the individual component interfaces within the framework.

5.3.4 In reference to multi-agent technology

The approach to agency in this thesis is distinct from several existing experimental multi-agent applications and architectures described in Chapter 4. Multi-agency is *not considered suitable as the principal, let alone only, technique* for achieving co-operative behaviour in batch-process management. Instead, agent properties should be *embedded* in a few judiciously selected software components of the framework.

The components selected for *agentifying* in the experimental framework of this thesis, are the participants in the dynamic unit-allocation decision-making in the control function, Manage Process Cell Resources. This selection is an example of a more general tendency in the automation applications of agency, namely that the use of agent technology is normally *co-operative*, striving towards a common goal, not competitive in nature, as in some other multi-agency application domains.

The contribution of this thesis with reference to multi-agency is in designing the manner in which software *components* can be *agentified* if there is a domain-specific need for that. Furthermore, the thesis demonstrates how an agent *interaction* mechanism, needed by the agentified components to reach a common goal, can be built upon existing communication mechanisms of a distributed software component model. It also shows how this kind of relatively large functional enhancement can be embedded into a reusable, calling framework, without changing the framework.

6. The Development of the Batch Process Management Framework

6.1 Introduction

This chapter presents the development of the domain-specific framework for *Batch Process Management* control activity, see Chapter 2.2. On a general architectural level, the whole *Process Management* is covered. Detailed designs and prototypical implementations are, however, developed only for the *Manage Batches* and *Manage Process Cell Resources* control functions. From the research point of view, a less important data-collection function, Collect Batch and Process Cell Information, is not covered in this thesis.

The development work is based on:

- Domain *(batch process management*) specific functionality, discussed in Chapter 2
- Object-oriented framework development with design patterns and distributed software components, discussed in Chapter 3

Detailed justification for the development approach adopted, has been given in Chapter 5, seen from both the domain and the technology points of view.

In the *requirements* definition, a class model for the *Process Management* has been developed. It has been complemented with use cases and use case based scenarios, describing the behavioural aspects of the system, seen from the point of view of the user. The class model and the scenarios as such are not, however, a sufficient basis for framework implementation.

The domain objects and components of the class diagrams co-operate together in various roles to carry out their tasks within the batch process management framework. To *design* the architecture, design *patterns* are used, especially the so-called architectural patterns, see Chapter 3.3. The creational and structural design patterns are also important, mostly when considering the architectures of individual software components. In batch control, it is necessary to consider the states and modes of recipe procedural elements and pieces of equipment. The behaviour of the components in their various roles, states and modes has to be designed. Thus many behavioural design patterns are needed.

Considering the *implementation* of the process management framework, the distribution and integration needs, described in Chapter 1, imply that the distributed subsystems must be able to *interoperate at the application level in a standard way*. This is where a distributed component model is needed. The resulting implementation, *an experimental software component framework for batch process management*, is a set of interoperating, mainly domain-specific components. It forms a *reusable design and implementation* for the developers of batch control applications.

The approach to developing the batch process management framework has been to first develop the domain-oriented class model, use cases and scenarios. This work is described in Chapter 6.2 Logical Models. Then the architecture has been developed and documented from the points of view of the functional decomposition and module interconnection aspects. The results are described in Chapter 6.3 Subsystems. Also, a user interface has been designed with the help of design patterns. The above issues cover the entire Process Management control activity.

The behavioural aspects of the architecture, covering the control functions Manage Batches and Manage Process Cell Resources, are described in Chapter 6.4 Dynamics. Framework-based applications can be distributed to several nodes of a local area network and their functionality is inherently concurrent. The necessary distributed and concurrent nature of the application has a profound effect on some design decisions. This development work is described in Chapter 6.5 Distribution.

The development has proceeded along the guidelines of Chapter 3.4. The role of the design patterns has been emphasised in this chapter by naming the relevant section *subheadings* with the names of the most important applied *design patterns*. Another way to emhasise the role of design patterns in the framework would have been to depict the framework as a package and the design patterns simply as collaborations within it (Booch, 1999, pp. 381 … 392). The explicit partition of the deployed main patterns, based on architectural, behavioural, and distribution related design issues was, however, preferred in this thesis.

As noted in the guidelines of Chapter 3.4, the relationships of the design patterns to other patterns are also important. In addition to the patterns named in the headings, other, related patterns, are used and reported in the text in the context of the main patterns. The batch process management framework of this thesis has been reported in the following publications: (Kuikka & Ventä, 1997), (Heikkilä *et al*., 1997), (Kuikka *et al*., 1999), (Kuikka, 1999).

6.2 Logical models - requirements definition

In Soni's (Soni *et al*., 1995) architectural categorisation (conceptual, interconnection, execution, code), described in Chapter 3.4, the conceptual architecture consists of domain-specific object classes and the relationships between them. It is *affected* both by the *application domain* and by the *abstract software paradigms and methods*. This architectural model is the closest to the class model of the requirements definition; other models are more design and implementation oriented. As Soni's conceptual architecture *is close* to the class model, according to Kruchten's (Kruchten, 1995) classification (logical, process, physical, development, and scenarios views), described also in Chapter 3.4, the logical view *is exactly* the class model itself.

Logical models have been developed for the Process Management control activity of the standard (ISA, 1995; IEC, 1997). The standard defines Process Management textually, giving functional descriptions of the tasks that the control activity is responsible for. The standard makes a clear distinction between the product view (explicit in the recipes of the Manage Batches control function) and the process equipment view of production (explicit in the equipment descriptions of the Manage Process Cell Resources control function).

Based on the textual descriptions of Manage Batches and Manage Process Cell Resources (see Chapter 2.2) use cases, class-object models and scenarios have been developed and documented with UML-notation, (Fowler & Scott, 1997; Booch *et al*., 1999). The development of the models has been partly helped, partly hindered (due to frequent changes and some inconsistencies) by the ongoing standardisation work of the ISA Working group WG5, (ISA 1999a).

6.2.1 Use cases

Use cases are typical interactions between a user and the system to be developed. A conceptual level use case (*a user goal use case*) achieves a specific goal for the user. In Manage Batches, the most important use cases are the Create, Modify, and Execute control recipes. In Manage Process Cell Resources, the most important use cases are Create Path, Modify Path, Modify Unit, and Maintain Process Cell.

All of these use cases are relevant for a person playing the role of an actor when interacting with the system. In Batch Process Management, only *two actors* have been

identified: *Operator* and *Configurator*. The same person may, of course, play both these roles. The actors and their respective use cases are described in Figure 6.1, below.

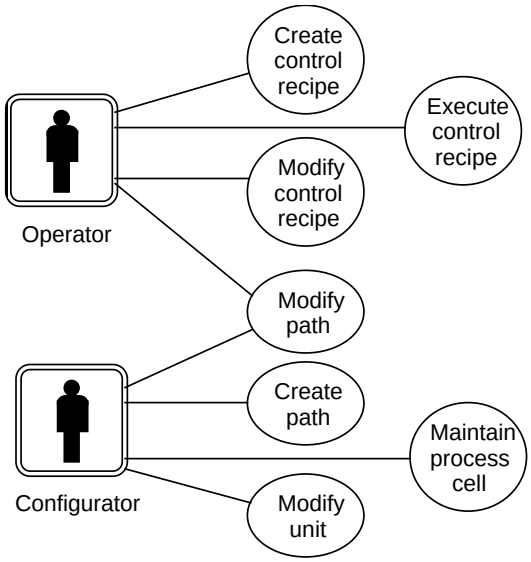

Figure 6.1. Domain actors and use cases in Batch Process Management.

The operator is responsible for executing control recipes. He also creates control recipes from master recipes, and optionally modifies the recipes and the respective paths, needed in the recipe execution.

The (process management system) configurator, on the other hand, is responsible for configuring paths. He can also make changes in the process cell configuration by adding and removing units (and services provided by them) from a configuration file. He can also update individual process unit information.

Only the high-level use cases, which are relevant to the actors when they perform their tasks, are shown in Figure 6.1. These user goal use cases can be further decomposed into so-called *system interaction use cases* (Fowler & Scott, 1997, p. 44 … 45).

6.2.2 Object classes

The most important object classes in the class model are *control recipe* and *control recipe procedural element* in Manage Batches and *process cell*, *path* and *unit* in Manage Process Cell Resources. The instances belonging to these classes, as well as the user-interface objects with batch control significance are called *domain objects*[7]. Below is a (somewhat simplified) example of an UML-diagram of the domain object classes with their main relationships, Figure 6.2.

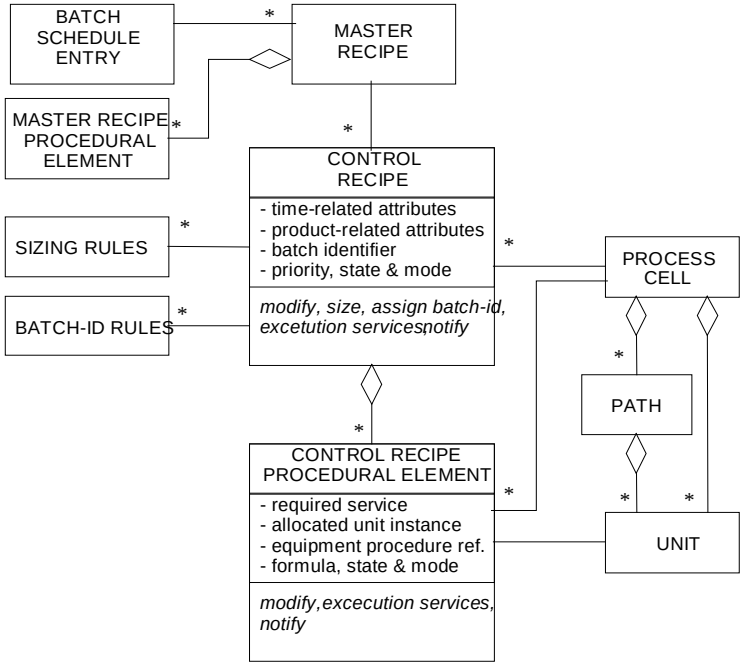

Figure 6.2. Main domain object classes in Batch Process Management.

In general, control recipes are derived from master recipes within Manage Batches but many of their attribute values depend on entities in other batch-control activities, as indicated by the relationships in Figure 6.2. Control recipe procedural elements are

[7] If they are *implemented* as software components (having IDL-interfaces) they can also be called domain components.

aggregated by control recipes but they also have relationships with Process Cell Resource Management components, which are relevant when executing a control recipe.

More specifically, a control recipe is initially created by copying a master recipe that is specified in the batch schedule entry. It is assigned a unique batch identifier (or the uniqueness of the identifier is verified if is already assigned). The control recipe is then sized according to product-related attributes, batch schedule entry and sizing rules. Various kinds of (application-specific, optionally multi-step) verifications may also be needed before the execution of a control recipe can be started. The control recipe and the aggregated control recipe procedural elements can also be modified before execution.

In Manage Process Cell Resources, on the other hand, various instances of an object class Path are created by the configurator and possibly modified by the operator. The paths consist of all the possible (from a physical-process point of view) connected collections of unit components in the process cell for producing a given product from raw materials. In the framework, a control recipe may *change a path dynamically during the batch production.* This makes it possible to balance the use of the process equipment as well as to flexibly react to faulty or otherwise unavailable process units.

The unit allocation within the Manage Process Cell Resources is vital when considering the flexibility of the framework. More specifically, quoting the standard (ISA, 1995, p. 54) about unit allocation:

> *Even though a (batch) schedule may have been planned to totally optimise the processing sequence from the standpoint of equipment utilisation, it is often desirable to allow alternate equipment to be used if the units planned for a batch are not available when planned. In this case the allocation of units to the batch – the routing or path of the batch – is a decision which must be made every time there is more than one path the batch can take through the available equipment.*

The above has been the starting point when developing the *dynamic unit allocation* scheme of the framework. The implications for the design of the unit allocation are:

- The sequences of units needed for a given batch, are originally (*pre-)planned* when planning the batch schedule within Production Planning and Scheduling control activity.

- Several alternative units should be available, to be *used* for the execution of a given control recipe within Process Management.

Arbitration of common (to various units) resources within process units (ISA, 1995, p. 55) is closely related to unit allocation:

> *If there are multiple requesters for a resource, arbitration is required so that proper allocations can be made. Arbitration resolves contention for a resource according to some predetermined algorithm and provides definitive routing or allocation direction.*

The implications of the above for the design are:

- Arbitration is needed when there are exclusive-use common resources. When planning and/or executing unit recipes, arbitration should be implemented at the Unit Supervision level.

- Only the arbitration of equipment modules and control modules is explicitly mentioned in the standard, not the arbitration of units. There can, however, be several batches (and associated control recipes) which need to use the same unit simultaneously.

The unit for the control recipe is selected in the framework immediately before the execution of the respective control recipe procedural element thus making the use of the process cell equipment dynamic and flexible. By deploying the concept of path, various kinds of unit configurations may be planned beforehand, based on configuration information of the Unit classes and the respective unit instances (corresponding to physical process units). The Unit component class has, additionally, as one of its attributes, an indication of which batch (and the respective control recipe) is using it or has been using it last.

The *selection* of a given unit instance for allocation depends, therefore on three factors:

- A service or capability, defined by the control recipe, based originally on the chemical engineer's knowledge of what kind of treatment is needed for a product in a given production phase.

- The paths available for producing a product based on the production or control engineer's knowledge of the various physical connections of the process units needed to produce a specific end product.

- The attributes of the Unit component class indicating the latest batch which has used a given unit (the unit which will be a predecessor of the new unit to be allocated), the allocation state and the operational state of the unit.

Because the above-mentioned pre-planning or reservation of paths and units is the responsibility of the Production Planning and Scheduling control activity, the batch process management framework does not give support to it, other than giving the operator the possibility of modifying the paths. It is, however, recognised that when starting to produce a critical batch, the execution of which must not be interrupted, it is important to ascertain that feasible units and paths will be available in due time.

Thus the *pre-planning* and *reservation* issues are seen as important themes for research and development when designing a framework for Production Planning and Scheduling. The pre-planning and reservation functionality of Production Planning and Scheduling is easily integrated with the dynamic unit allocation of the Process Management. The former produces a set of feasible units and paths, which the latter can then choose from during the execution of a batch.

6.2.3 Scenarios

The detailed class-object model of the framework has been complemented by some use case based scenarios, or interaction diagrams, describing the behavioural aspects of the system. Below, in Figure 6.3, is an example of a scenario capturing the normal behaviour of the Execute control recipe use case.

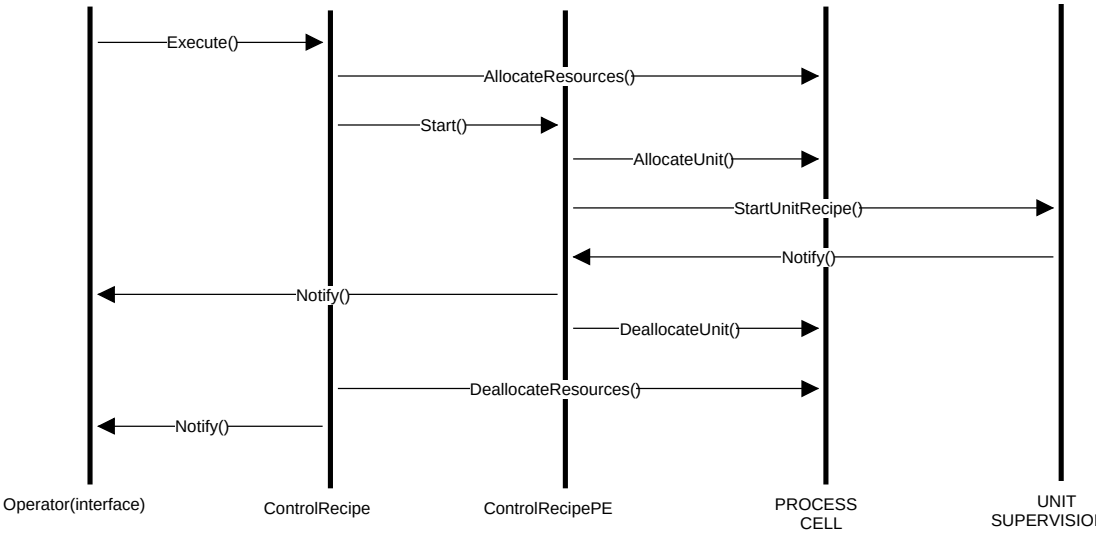

Figure 6.3. Execute control recipe scenario (or interaction diagram).

The *execution* of a control recipe begins when all the starting conditions (the time-related attributes of the control recipe) have been satisfied. First the control recipe requests the allocation of common resources (recipe or product-related attributes, i.e. raw materials and the like) from the process cell. Then the first control recipe procedural element allocates a unit, which provides the required service from the process cell, and gets, as a response, reference to an allocated unit instance. The execution of the control recipe procedural elements takes place in parallel, if needed.

In the framework, a control recipe procedural element for mixing, for example, allocates a unit which is capable of mixing. It may be a specific mixer, a reactor or some other kind of mixing vessel. The process cell component, which is unique (a singleton) for a given physical process cell, has knowledge of all its units and the services or capabilities they provide. Units are considered to be *autonomous* components, which are capable of executing their respective unit recipes, see Chapter 7.2 for details.

A control recipe procedural element is started by calling the respective unit recipe reference (in Unit Supervision) with proper parameters. The parameters are the allocated unit instance, which was received as a response from the process cell, and all the formula values needed. After the execution of a given control recipe procedural element is completed, its allocated unit will be deallocated in the process cell by the control recipe procedural element. After the whole control recipe is completed, all the resources in its use (product related attributes) will be de-allocated, as well.

Scenario techniques are descriptive for developing and refining use cases by presenting single behaviours (or traces) of sets of components. However, they cannot as such provide a complete dynamic representation of the whole system. As pointed out by Krutchen, (Krutchen, 1995) concerning his scenario view, several of the scenarios can be designed and utilised in later phases of development. Accordingly, *scenarios* have been used as working documents in investigating system interactions when designing the framework, especially the state-dependent behaviour of the components. The importance of the scenarios is seen not so much as a part of the documentation but more in aiding the development by helping to make other models complete and consistent.

6.3 Subsystems - architecture and structures

The *development view* of software architecture by (Kruchten, 1995), introduced in Chapter 3.4, is analogous to traditional software module architecture, for example functional decomposition and layering are included in the development view. There are, however, some additional issues: requirements related to the ease and evolving nature of the developed system, reuse and commonality as well as the constraints imposed by the development environment. These may well be captured into the design using design patterns.

From the framework point of view, the so-called architectural patterns, see Chapter 3.3, are most important. The layered aspects are covered by the *Layers* pattern, the component interconnection problems are assisted by the *Broker* pattern. The separation of user interface and application logic is helped by a variation of the *Model-View-Controller* pattern. Neither of these patterns will, however, suffice as such but they will have to be adapted and their use restricted when developing the domain-specific framework.

In the design it is also important to consider the *relationships* between the patterns as noted in the guidelines of Chapter 3.4. The pattern relationships, together with the requirements definition, suggest the *order* of applying the patterns. It is reasonable to begin with the pattern that addresses the *most important design aspect* and then proceed with other design issues, *one at a time,* considering the pattern-relationship information available. In this way a coherent architecture for the framework can be achieved. There are more design options than in conventional development involving decomposition, but at the same time the design issues are more difficult.

6.3.1 Layers pattern

The ISA-standard (ISA, 1995) based requirements definition gives a good reason to start with the *Layers* – an architectural pattern for the batch control domain as a whole, (Buschmann *et al*., 1996, p. 31 ... 51). A complete batch-control application is certainly large enough to require decomposition. The functionality can naturally be partitioned into layers on various functionality levels according to the control activity grouping of the standard, see Figure 2.4 in Chapter 2.2.

Additionally, the importance of *interface stability* - one of the design issues of the Layers pattern - is very important in batch control. Several teams can be independently

developing partial frameworks for the whole batch control system of a company. These teams may belong to various vendor organisations, and they may work at different times.

On the other hand, the component framework boundaries with parameter-marshalling techniques may decrease the performance of the system if large amounts of data must be transferred over several boundaries. This tends to decrease the optimal degree of decomposition and thus increase the size of individual components and frameworks. The *granularity* of the control activities and control functions of the ISA-standard is here considered an appropriate solution also for framework implementation. This design decision is due to the standard-compliant cohesive functionality, to small enough development tasks and to large enough interfacing entities.

According to a pure *Layers* – based architectural solution, the application is decomposed into groups of sub-tasks in which each group is on a particular level of abstraction. Most of the services provided by an upper level layer (n) are composed of services provided by a lower level layer (n-1). Additionally, some of the layer n services may depend on other services in layer n. Thus the layer n provides services used by the upper layer n+1 and delegates sub-tasks to the lower layer (n-1). Perhaps the best examples of utilising pure layered architectures can be found in telecommunications.

In batch process control, the Layers - architecture can be applied to the overall hierarchical control activity decomposition. The interfaces between the subsystems are reasonably well defined in the standard and the services are mainly requested from above and provided from below. For example, the tasks of the Process Management control activity request mainly the services of Unit Supervision. Also the amount of data transferred between the control activities can be kept small if this design issue is considered when defining the individual interfaces. The details of interfaces regarding the contents of the messages (or parameters of method invocations) are naturally not specified in the batch standard.

More specifically, the information contents of the interfaces between the Process Management control activity and Recipe Management, Production Planning and Scheduling as well as Production Information Management on the upper layer and Unit Supervision on the lower layer, respectively, are described in Figure 6.4. The dashed line in Figure 6.4 includes the control functions Manage Batches and Manage Process Cell Resources, which belong to the framework, as well as their respective interfaces.

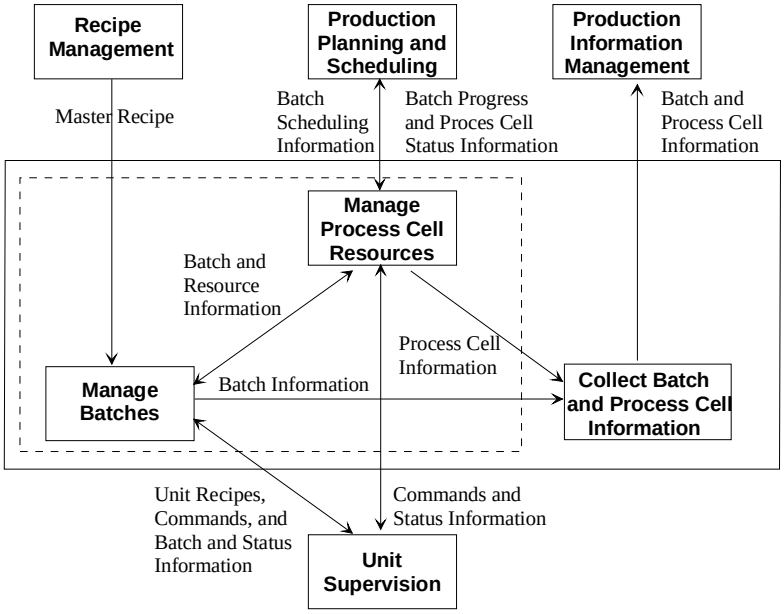

Figure 6.4. Process Management (modified from ISA, 1995).

The main tasks of the Process Management are to manage batches and to manage process cell resources. Based on master recipes, the required control recipes are created. They are executed using process cell equipment according to the batch schedule supplied by Production Planning and Scheduling, on a higher layer. Unit Supervision, on the lower layer, is the control activity that connects the control recipe to the equipment control. The Unit Supervision executes unit level recipe procedures invoked by the Manage Batches by initiating and parametrizing equipment procedures of devices. It also receives allocation and de-allocation commands from the Manage Process Cell Resources and, additionally, sends respective status information to these control activities.

There is a need for *horizontal requests*, too. For example, in Figure 6.4, there is a bi-directional interface between the control functions Manage Batches and Manage Process Cell Resources as well as unidirectional interfaces from them to Collect Batch and Process Cell Information. More importantly, however, the layered architecture scheme as such is far too general to be applied as a sole basis for architectural design and implementation. Other related patterns are needed.

6.3.2 Broker pattern

By distributing the batch-process-control functionality both horizontally inside and between the layered control activities, a flexible, maintainable and changeable framework can be achieved. The *Broker* pattern (Buschmann *et al.*, 1996, p. 99 ... 122) is, accordingly, used within the batch process management framework to structure the system into uncoupled software components that interoperate by remote service calls. There are important *design issues or forces* in the batch control domain which are balanced by using the broker pattern:

- The software components within various control activities must be able to access services provided by others through remote, location-transparent service invocations.

- It should be possible to modify, add and remove (even at runtime) components developed by various control-software vendors.

- The implementation-platform and language-specific details, especially concerning distributed communication protocols, should be hidden from batch control developers, i.e. the users of the components.

The broker itself is responsible for co-ordinating communication between the objects and components in *clients* and *servers* of the system, which are seen in this context primarily as *logical roles of software component groups*, and only secondarily as structural entities. The servers make the services (provided by their software components) available to clients through component interfaces. The clients access the functionality of the servers by sending requests via the broker. Thus the same software process can act both as a client and a server.

The decisions having effect on the sequence of the process stages and operations are made both in the context of control recipe management in Manage Batches and equipment management in Manage Process Cell Resources. This division to batch-control functions gives rise to the primary distribution of the Process Management functionality. The interdependency of the two control functions is, however, tight, which can be seen in the many interdependencies between single components in these control functions.

The user interfaces for the control functions, Manage Batches and Manage Process Cell Resources, should convey to the user a clear *distinction* between, on the one hand, product and recipe-related issues, and on the other hand, resource and equipment-related

issues. The client processes designed to implement user-interface functionality for these control activities should thus be separate from each other. These views of the process should also be easily *integrated* and *customised* according to the *tasks* of operators. Due to the need for functional distribution, the user interfaces should be independent of the respective server processes, which contain the application logic. The resulting general architecture of the framework is presented in Figure 1.1 of Chapter 1.4.

The above-mentioned client and server processes can be situated on various computational nodes of a local network or intranet. The control recipe server and the path server processes act as servers in relation to their respective user interface clients. They are simultaneously clients for the unit server and additionally, the control recipe server acts as a client for the path server, see Figure 1.1. The architectural style of the application is thus so-called *multi-tier* architecture.

Each server process contains several domain components and other objects. The servers also include a mechanism (a class-object factory) for instantiating components from their classes according to requests from the clients. The class-object factory of the framework implements an *Abstract Factory* design pattern, see Chapter 3.3, which is, in this context, an associated design pattern used by the main Broker pattern. The framework components, which are distributed according to the Broker pattern, are described in Chapter 6.2. The detailed design of their distribution among the clients and the servers is presented in Chapter 6.5.

The Broker's tasks in co-ordinating communication include *locating* an appropriate server for a requesting client, forwarding the request of the client's object to the server's software component and transmitting results back to the appropriate client. It also allows change, addition, deletion and relocation of software components. Software component interfaces are made available to service users through an interface definition language (IDL).

The Broker also offers *application-programming interfaces* to clients and servers. They include operations for registering servers and components on various computing nodes of the network. Also other *services,* for example name services and object persistence services, may be integrated into the Broker. There are several candidates for Brokers, both commercial products and research prototypes; notably the ones provided by the CORBA (Common Object Request Broker Architecture; OMG, 1995) and DCOM (Distributed Component Object Model; Microsoft, 1995) component models. The platform of the framework of this thesis is DCOM, due to its good tool support and its

significance in industrial automation, especially through OPC standardisation (OPC, 1998).

6.3.3 Model -View-Controller pattern

In Manage Batches as well as in Process Cell Management it is evident in the basis of the functional distribution needs that the user-interface (client) part of the system should be separate from the logic functionality (server). It was also important to design the implementation of the user interface to be easily *changeable*, separately from the more permanent application logic. Both the Windows desktop and Internet browsers are considered important as user-interface platforms in the batch-control domain.

This has lead to the application of a variation of the well-known *Model-Controller-View* pattern (Buschmann *et al*., 1996, p. 125 ... 143). From the domain point of view the important *design issues or forces* to be balanced by the pattern are:

- The display of process events and responses of the application logic must reflect the user's control commands immediately. This is traditionally well achieved with proprietary distributed control systems and is thus expected by the users of the batch process management system.

- The changes in the user interface should be easy, and possible to make at runtime. The inherently high requirements for control system availability must not, however, be endangered by changes in the user interfaces.

Supporting different 'look-and-feel' standards should not affect the application logic. Continuing increases in the level of batch automation also increase the need to make user interfaces similar in all information systems of a plant, since they are used by the same operators. This often makes it necessary for new systems to be adapted to company standards, or various other existing 'look-and-feels'.

In the *Model-View-Controller* pattern, the model component encapsulates core data and functionality, being independent of input and output. The view components display information to the user after obtaining the data from the model. There may be multiple views corresponding to one model. Each view has an associated controller component, which receives user input from user-interface devices (normally mouse, buttons and keyboard) as events. If the user changes the model, via the controller of one view, all the other views dependent on the same data, should reflect the changes.

What makes the design of the framework different from the Model-View-Controller pattern is the fact that the responsibilities of the view and controller have been combined and thus the design incorporates a *variation* of the original Model-View-Controller, the so-called *Document-View* pattern (Kruglinski, 1996, p. 332). The document component corresponds to the original model, and the view includes both the original view and the controller.

From the user interface point of view control recipes and control recipe procedural elements in Manage Batches and paths and units in Process Cell Management are examples of documents. The former are located in a control recipe server process, the latter in a path server process. The main views of a control recipe client are the control recipe view, representing all the control recipes manipulated by a given user and the batch view, representing all the control recipe procedural elements of a given running control recipe. In a path client process the main view is the path view which represents all the paths created or modified by a given user.

The *separation of user-interface* objects *from application-logic* components is further emphasised by the fact that clients and servers can be distributed to different computational nodes and, whether local or remote, they communicate via component interfaces. According to the functional distribution requirements of Chapter 1, the clients have also been designed to be thin, having only the functionality needed by a given user, but at the same time self-sufficient and independent of other clients.

Thus a given Manage Batches user has in his/her control recipe view a list of the most important information on the control recipes that the user is manipulating. This information makes it possible for him/her to modify and execute the actual control recipes in the control recipe server at will. In a similar manner, a given Process Cell Management user has in his/her path view a list of the path names that the user has created and/or modified.

The user interface of the batch process management framework has been implemented in the Windows NT environment, using the Microsoft Foundation Classes class library. The thin client functionality above could, however, have also been implemented with web browser technology, for example along the lines of a simulator user interface, presented in (Karhela *et al*., 1998).

6.4 Dynamics - behaviour and interoperation

Behavioural design patterns have been used when classifying the batch-process management components of Chapter 6.2 into groups according to their *roles* in the framework. The patterns have also been used to design their *behaviour* and *interoperations*, and to incorporate some new object classes to fulfil the behavioural requirements. The most important patterns that have been needed are *State pattern*, *Observer* and *Mediator*.

6.4.1 State pattern

Within the batch process management framework it is advantageous to define the behaviour of the system in various *states* (specifying the current condition) and the *modes* (of operation) of its components and to consider their behaviour both in *normal operation* and in various kinds of *exceptional situations*. Both procedural elements and equipment entities have states and modes. A control recipe procedural element, for example, can be in 12 distinct states (idle, running, complete, pausing, paused, holding, held, restarting, stopping, stopped, aborting, aborted) at any given instance of time (ISA, 1995, p. 59). Likewise, it may have three modes (automatic, semi-automatic and manual).[8]

When considering the execution requirements of the control recipe procedural elements the decision to use the *state pattern* (Gamma *et al*., 1995, p. 305 ... 313) was made. *Design issues or forces* important to batch control, balanced by the state pattern are:

- The behaviour of a control recipe procedural element depends on its state, and it must be able to change its behaviour at run-time depending on that state.

- Operations performed by the control recipe procedural element would have large, multipart conditional statements that depend on the object's state, if the state pattern were not used.

The state pattern allows an object to alter its behaviour when its internal state changes. By switching the state object associated with the control recipe procedural element

[8] In the semi-automatic mode, the procedure requires manual approval to proceed after the transition conditions are fulfilled.

(ControlRecipePE) component, the behaviour of the component can be modified. The important subset of the ISA-standard states in this context are: idle, running, paused, held and complete, Figure 6.5. While the behaviour of a control recipe procedural element changes according to its state, its interface, used by its clients, remains the same, defined by the component class ControlRecipePE.

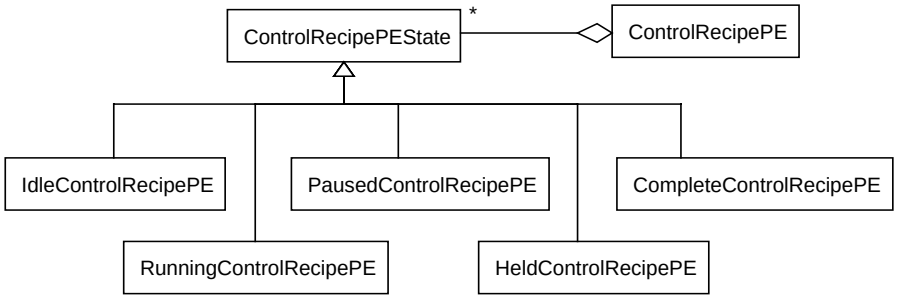

Figure 6.5. Control recipe procedural element State.

The implementation of control recipe procedural elements by state-dependent object classes, Figure 6.5, also makes the *modifications* (for example adding new states) in the software easy. For example, in the framework the need of the state 'held' was not considered necessary at the beginning. When it became necessary (as a combination of ISA-states held and stopped), its inclusion was straightforward; the code in the other classes remained intact. It is probable that additional states will also be needed in further development. However, which of the remaining 7(6) states of the ISA-standard (pausing, holding, restarting, stopping, (stopped), aborting, aborted) shall be added, depends on the needs of the new features to be developed.

In connection with the state pattern, it was also fairly straightforward to fulfil the obvious (batch) control domain need that a given control recipe procedural element should be at every instance of time in one and only one state. Each ControlRecipePE component maintains as a member variable a reference to an instance of the concrete state subclass that defines its current state. The state pattern as such does not, however, specify the class of an object that initiates the *state transitions*.

In the batch process management framework, transitions can be initiated both by a ControlRecipePE component and concrete state subclass objects. The former is needed, for example, in failure situations or when the user wants to control the operation of

ControlRecipePEs manually or semi-automatically. The latter is mostly needed in normal automatic operational state transitions, for example from ‘idle’ to ‘running’ and from ‘running’ to ‘complete’.

The degree of *decentralisation* in the transition logic decisions is thus mostly a matter of domain-specific requirements. In general, however, a high degree of decentralisation increases the expandability of the system by making it easy to incorporate new concrete state subclasses. Since these classes have to know their successor relationships it, however, also incorporates unwanted dependencies between the subclasses.

The inclusion of the state pattern in the design of the framework is, unfortunately, not free. Compared to a more conventional approach, an additional abstract class, (ControlRecipePEState) has to be implemented and its concrete subclasses constructed. The proper method delegation between these new classes must also be taken care of. This increases the complexity of the design and decreases somewhat its clarity.

In Manage Batches the benefits achievable with the state pattern were considered greater than the liabilities for the highly state-dependent control recipe procedural elements, but not for the control recipes having more state-independent functionality. In Manage Process Cell Resources the state pattern is not needed, since the allocation states of the units are few, ‘allocated’ and ‘free’. Moreover, the respective functionality, as well as operational state and mode-dependent functionality, is well managed without the use of the state pattern.

6.4.2 Observer pattern

In order to be able to propagate the information on the changes of states and modes of the batch-control components to other objects, an *Observer* pattern, (Gamma *et al*., 1995 p. 293 ... 303), was found suitable. The Observer pattern defines a one to many (subject to observers) dependency between objects so that when one object (the subject) changes state, all its dependants (the observers) are notified. The subject knows its observers. Any number of observers can observe a subject. The subject also provides an interface for attaching and detaching observer objects.

The most important *design issues or forces* to be balanced by the observer pattern in the batch control domain are:

- A subject should be able to notify other objects without making assumptions about what these objects are. The control recipes and control recipe procedural elements in Manage Batches as well as the units in Manage Process Cell Resources should be able to notify other objects.

- All the observers should be notified whenever the subject undergoes a specified change in state or mode. In batch control the number of possible state and mode changes of the subjects on the one hand, and the number of potential observers per subject on the other is large. Thus it is important to restrict the changes to be notified in order to avoid exploding the communication.

In Manage Batches, the control recipe server's ControlRecipe acts as a subject and the control recipe client's ControlRecipeView as an observer. More importantly, the ControlRecipePE also acts as a subject, and the server's BatchMediator, as well as the client's user interface object BatchView act as observers, Figure 6.6.

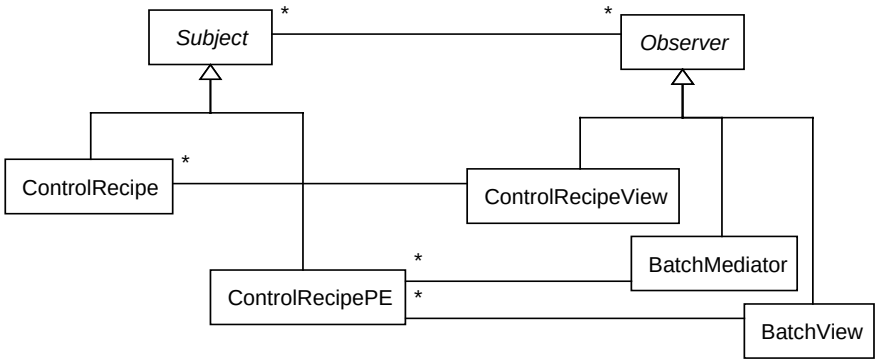

Figure 6.6. Control recipe observer.

According to the Figure 6.6 above, one ControlRecipeView observes all the ControlRecipes created and run by a given user during a given session and one BatchView as well as one BatchMediator observe all the ControlRecipePEs belonging to an active batch.

When using the observer pattern, a *push model* (with suitable parameters) is applied when the ControlRecipe and the ControlRecipePE notify their observers, respectively. This means that the subject sends the observers all the needed information about the state or mode change. This way the communication overhead is minimised in comparison to the alternative *pull model*, according to which the subject would only send minimal

notification and the observers would ask for details with separate requests. The push model assumes that the subjects know something about the needs of their observers, which is reasonable in the batch-control domain and further clarified when discussing the application of the mediator pattern.

The observer pattern presumes that 'the subject broadcasts its notifications to all interested objects that subscribe to it'. Thus there has to be a mechanism for a subject to keep track of the observers it should notify. The simplest solution for this, explained in (Gamma *et al*., 1995, p. 293 ... 303) would be to store references to observers explicitly in the subject. In the batch process management framework, however, a more versatile mechanism, the so-called *Connection points*, was chosen. It is specified in the DCOM standard interfaces (Distributed Component Model; Microsoft, 1995, Chapter 9 p. 1 … 10), but could also be implemented when using the CORBA object model.

In the Connection point mechanism the subject has an enumerated set of connection point objects, one connection point for each observer of the subject. The observer uses the connection-point interfaces (amongst other requests) to subscribe to the subject. When subscribing, it also tells the subject the identification of an object that is to get the notifications from the subject (a so-called sink object, not necessarily the observer itself). The connection points within the subjects and the sinks within the observers thus provide additional *flexibility* in the implementation of the observer pattern.

In the Batch process management architectural framework, the *concern of connectedness* has, additionally, been *separated from* the *domain functionality* by providing a class of its own for connection related issues other than the connection points themselves (which belong to their own class). From this class, the domain object classes acting as subjects (here ControlRecipe and ControlRecipePE) inherit the functionality needed for the connections to their respective observers (here ControlRecipeView, BatchView and BatchMediator).

What is more complicated is the fact that the state changes of the control recipes and control recipe procedural elements take place - seen from the point of view of the client objects - in an arbitrary order. Thus, for example, the BatchMediator and the BatchMediatorView must be able to accept notifications at any time from any ControlRecipePE of a given batch. This implies the utilisation of a proper message-queuing practice by the observers, discussed in Chapter 6.5.

The observer pattern could also be used for notifications from one ControlRecipePE to other ControlRecipePEs belonging to a given batch, in order to synchronise the execution

of the ControlRecipe as a whole. The use of the observer pattern for this kind of synchronisation would, however, give rise to a substantial number of observer-subject pairs and respective notifications per batch.

6.4.3 Mediator pattern

The above-mentioned potential use of the observer pattern alone was, however, discarded and a related *Mediator pattern* (see Figure 3.4 in Chapter 3.3) was considered to be more appropriate. A suggestion of this design decision is seen in the Observer pattern of Figure 6.5 where one of the observers is named *BatchMediator. The Mediator pattern*, (Gamma *et al*., 1995, p. 297 ... 282), defines in general an object that encapsulates how a set of other, collaborating objects interact. The mediator promotes loose coupling by keeping the objects from referring to each other explicitly. The interacting objects have knowledge only of the mediator, not of each other.

An abstract mediator defines an interface for communication with so-called colleague objects. A concrete mediator implements the co-operative behaviour by co-ordinating the colleague objects, and knows and maintains information about them. Each colleague knows its mediator and communicates with it, instead of communicating directly with other colleagues.

The most important *design issues or forces* to be balanced by the mediator pattern in the batch-control domain are:

- A set of components communicate in complex ways so that the resulting interdependencies are difficult to understand. In Manage Batches the most important interdependency is the predecessor - successor - relation between the (execution of) individual control recipe procedural elements.

- A behaviour that is distributed between several classes should be customizable without a lot of changes. Specifically, in a domain-specific component framework like the batch process management framework, it is important to minimize the number of changes in domain components. By using the Mediator as an interoperation centrepoint, it is possible to change the behaviour of the interoperation by changing only the Mediator, leaving the batch-domain components as they are. The batch components can then be reused without interoperational concerns by the application developer, who uses the framework according to application-specific needs.

- There is a need to simplify the protocols and to abstract how components and objects co-operate. In an architecture with a lot of interoperations, it is generally a good idea to replace many-to-many interactions with one-to-many interactions between the Mediator and the domain objects.

The BatchMediators mediate between the components (ControlRecipePEs) needed in a given batch, Figure 6.7.

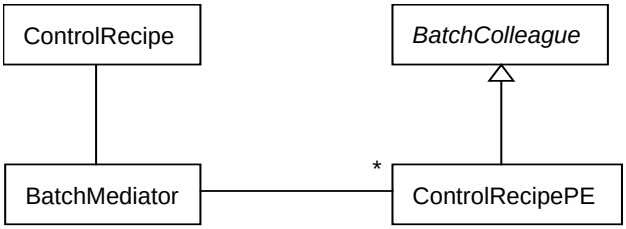

Figure 6.7. Batch Mediator.

The BatchMediator is implemented partly as an Observer in relation to ControlRecipePEs as described earlier in the text and in Figure 6.6. It gets the completion notifications from the ControlRecipePEs in the role of an Observer and then synchronises their execution in a more complex role of a Mediator.

In the framework, the BatchMediator co-ordinates the execution of the ControlRecipePEs of a given batch using *preconditions*. The parallel/sequential execution requirements are internally represented by preconditions imposed upon all individual procedural elements. The values for preconditions are given, when a ControlRecipe is originally instantiated from a Master recipe or a ControlRecipe, or when its procedural elements are modified.

The preconditions indicate which other ControlRecipePEs have to be completed in order to start a given procedural element. This approach provides good opportunities for specialisation (Mayer, 1997) and it is thus a suitable implementation basis for a *more advanced mediation*, as well. The new proposal for recipe representation (ISA, 1999a, pp. 120 … 140) indicates that the synchronization interdependencies within recipe procedural elements, executing in parallel, are to be taken into account in implementation. This makes the future tasks of a BatchMediator more challenging, but clearly better manageable than would be the case without the use of a Mediator pattern.

It is argued here that it is also true in general that the way in which objects or software components use their interfaces when interoperating is a highly *domain-specific* issue. It is fundamental to the domain-specific frameworks, described here for the batch process

management domain. The object classes participating in interoperation make so-called bi-directional contracts (Box, 1998, pp. 356 … 369), the contents of which reveal a lot of the specified domain functionality. The domain-specific deployment of both the Observer pattern and the Mediator pattern described above, are examples of this.

6.5 Distribution - distributed components and multithreading

The requirements for the distribution of functionality in the batch-control context were stated generally in Chapter 1 and the domain-specific architectural aspects in designing distribution were covered in Chapter 2. It is also important to consider the necessary *interfaces* (based on component interoperations) through which the domain components provide their services. The semantics of the application domain - batch process management - has a substantial effect on this. Additionally, it is important to consider the distribution of the components in the processes, and the threads within the processes. Also in these design decisions the application domain requirements have an important role.

6.5.1 Proxy pattern

Interfaces are of primary importance when developing software components and frameworks based on components. *Interfaces* define the way (methods with input and output parameters) in which the components residing in the servers are accessible for the clients. CORBA and DCOM have their own *Interface Definition Languages (IDL)*, to be used when defining the interfaces of components. From the interface definitions, proxies and stubs are automatically generated. They are representatives of server components in a client's address space and client objects in a server's address space, respectively, see Chapter 3.2.

Proxies and stubs are implementations of the *Proxy* pattern (Gamma *et al.*, 1995, p. 207 ... 217; Buschmann *et al.*, 1996, p. 263 ... 276). However, since they are generated automatically by CORBA and DCOM IDL-compilers, their application does not involve any design decisions per se. Thus the emphasis here is on describing the interfaces from which the proxies and the stubs are generated. The generic *standard interfaces* of the component model define common functionality for various purposes, for example for file

system, for compound documents, and for user-interface controls. These interfaces are inherited as such in the batch process management framework.

In designing the framework, the DCOM-dependent standard interfaces have been utilised sparingly, in such a manner that compatibility with the CORBA object model has been preserved as much as possible. Thus many special OLE-dependent interfaces (OLE, 1995; Brockschmidt, 1995) have *not* been used but the architectural framework has been designed and implemented with the basic DCOM-interfaces. A notable exception in this respect is, however, the utilisation of Connection point interfaces, which is described and justified in connection with the Observer pattern.

From the batch-control point of view, the *custom interfaces* are more important than the standard interfaces. They define the batch-centric services provided by the components, which represent the semantics of the developed domain-specific framework. They also require more definition and implementation effort than the *standard interfaces* because there are no interface classes from which they can be derived from.

In the architecture of the batch process management framework of this thesis, the custom interfaces are divided into *static and dynamic interfaces*. The former take care of persistence (saving and loading) of the software components and modification of the state variables of the components as static entities. The latter influence the execution of the components as dynamic, active objects. *The serialisation* services provided by the Microsoft Foundation Classes class library are considered here a proper implementation technique for persistence, due to the similarity of the approach to that of another component model, Enterprise JavaBeans. The dynamic aspects are described in connection with the Active Object pattern below.

Interfaces form *elementary contracts* between a client and the components of a server. A client, for instance, queries the interfaces for its particular use from the components, and it is also possible for a client to find out runtime type information on the interfaces and behave accordingly. The interfaces, defined with the help of an interface language, are also a good basis for designing more complex interactions when developing multi-agent properties for the framework, see Chapter 7.3.

The distribution of the domain components within the client and server processes is summarised in a tabular form. Table 6.1 presents the most important domain objects and components contained in the clients and servers of Manage Batches and Manage Process Cell Resources. Also the client and server *roles* played by the software processes are indicated.

Table 6.1. Client and server processes and domain object classes.

Control function	**Software process**	**Client/Server-role of the process**	**Domain object/component within the process**
Manage Batches			
	CRServer	S + C	ControlRecipe
			ControlRecipePE
	CRClient	C	ControlRecipeView
			BatchView
Manage Process Cell Resources			
	PathServer	S + C	ProcessCell
			Path
			Unit
	PathClient	C	PathView

As described previously, client and server processes can be distributed anywhere on the local net, or (with security related restrictions) on intranet. The domain components, as well as other objects needed in implementation, are *contained* within servers, which thus act as component containers. The server processes also act as clients for a lower level, the Unit Supervision functionality, as indicated in Table 6.1.

While implementing the framework (most importantly the domain objects) as software components having well defined interfaces, it would also have been possible to distribute functionality physically even further. For example, control recipes might reside on a different computational node to their respective control recipe procedural elements. That would, however, increase the communication overhead without bringing any obvious benefits. As an option, the possibility of distributing further is, however, important. It is

needed for example if/when the PathServer process is distributed near the physically distributed process equipment.

The design decision of client and server processes containing several domain-specific components brings about the need to utilise *multi-threading* within processes, since there is a requirement for parallel activity. For example, the control recipe procedural elements that belong to the same control recipe have to be able to run in parallel. In the design of multi-threading implementation, a variation of the *Active Object* pattern, presented next, was utilised.

6.5.2 Active Object pattern

The *Active Object* pattern (Lavender & Schmidt, 1996) uncouples method execution from method invocation in order to simplify synchronised access to a shared resource by methods, invoked in different threads of control. A variation of the Active Object pattern, making the execution of control recipe procedural elements (ControlRecipePE) independent of each other, is used in the batch process management framework. Control recipe procedural elements are active objects, which are invoked by the control recipe and synchronised by the batch mediator, but which execute in threads of their own.

The most important *design issues or forces* to be balanced by the active object pattern in the batch process management domain are:

- It simplifies concurrent programming. The message queue used by a given ControlRecipePE takes care of the messages coming to the ControlRecipePE in its various states. The messages can request the services of the ControlRecipePE from the user interface CRClient, from Process Cell Management, and from Unit Supervision subsystems. They will all be handled in order and without losing any.

- It takes advantage of parallelism. As noted before, both the sequential and the parallel execution of ControlRecipePEs is a batch control domain requirement implied by the needs of procedural control of the standard (ISA, 1995). Additionally, it is important that the responses to the user are prompt enough.

Although ControlRecipe procedural elements are invoked by the control recipe, they get a permit to execute from the BatchMediator. After being allowed to run, a ControlRecipePE allocates a process unit, executes the respective unit recipe, and de-

allocates the unit. This is performed in its own thread, separately from the main thread of the server process. After finishing, the ControlRecipePE notifies the BatchMediator, which keeps track of the completeness status of all ControlRecipePEs of a given batch. The BatchMediator is thus capable of allowing the execution of the ControlRecipePEs according to the preconditions discussed in the previous section.

When comparing the variation of the batch process management framework to the original *Active Object* pattern (Lavender & Schmidt, 1996), the following correspondences can be noted. The custom interfaces of the ControlRecipePEs (used, for example by the ControlRecipe in invocation) correspond to the *Client Interface* proxy, the ControlRecipePE's attributes represent the *Resource Representation,* and the state-wise methods represent the *Method Objects.* A class that utilises the DCOM *Apartment threading model* (CRPEApartment), corresponds to the *Activation Queue* and *Scheduler* in the original pattern. The interface pointer of a ControlRecipePE, which is returned to the caller when invoking a method of the ControlRecipePE component, represents the *Result Handle* of the (Lavender & Schmidt, 1996) - pattern.

The apartment object class (CRPEApartment) takes care of message queuing and initialisation of the COM library at the beginning of the thread execution. It also uninitializes the COM library at the end of the execution, as well as providing information about the state of the thread during the execution. All this is needed, since applying multi-threading in DCOM with the apartment threading model means that the software components residing in various threads have to be queried and message parameters *marshalled* as if they were in another process (or equivalently, in another computational node; Rogerson, 1997, p. 320 … 324).

Separating the concern of *threading functionality* from *the concern of domain functionality* within a threading class helps to keep the domain objects (here the ControlRecipePE and its state-dependent subclasses) comprehensible. The use of the DCOM apartment-threading model in the implementation simplifies the synchronisation. However, at least if deploying this Active Object pattern variation in time-critical applications, it may be beneficial to make the message-queuing priority based, instead of FIFO based as it is by default.

6.5.3 Deployment

The distribution of the batch process management framework functionality into control-function based subsystems, client and server processes and several threads within the processes, gives rise to many interactions between the distributed software components. The required *complex interactions* are, however, *comprehensible to the application developer* through the use of the Observer and Mediator design patterns. The same applies to general architectural issues through the use of Layers, Broker, and Model-View-Controller patterns and to the internal issues of the components through the use of State, Proxy and Active Object patterns.

Since the batch process management framework has been designed *as a calling framework*, see Chapter 3.4, the application developer does not have to worry about the architectural aspects, the interoperations, the state behaviour or the threading behaviour of the components, which are all encapsulated within the framework. He/she perceives the framework as a set of interconnected components, which he/she can *parametrize* and *reuse* when developing the custom application.

Control recipe procedural element (ControlRecipePE) components, which are logically aggregated by a control recipe component, are concurrently executing domain components in Manage Batches, Figure 6.8. They execute in threads (depicted with curved lines) of their own.

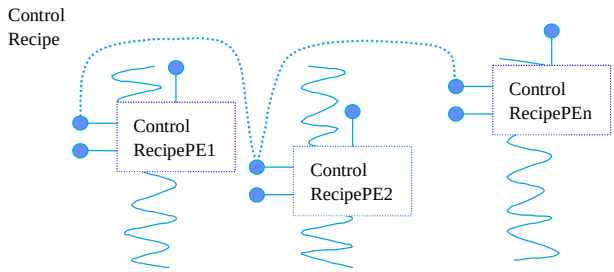

Figure 6.8. ControlRecipePEs within a ControlRecipe.

Within the path server in *Manage Process Cell Resources* control activity, unit components are logically aggregated by a process cell component in a similar manner as ControlRecipePE components by a ControlRecipe component above.

In Table 6.2, there is a list of all the *message types* of the messages sent between the client and the server processes. This list also indicates the custom interfaces, which have been defined in the framework for the respective software components. The fact that a given client component uses (or *imports*) a respective server component interface can be notationally described with a so-called dependency relationship from the using component to the interface (Booch *et al*., 1999, p. 348). A component can be used in a context if and only if all its import interfaces are provided by the interfaces (in this context also called *export* interfaces) of other components.

Table 6.2. Messages between framework clients and servers.

Direction of the messages	**Message contents**
CRClient -> CRServer	Requests for creating, modifying and running ControlRecipes and ControlRecipePEs
CRServer -> CRClient	Responses and notifications of the state of ControlRecipes and ControlRecipePEs
CRServer ->PathServer	Requests for allocation and deallocation of Units for batch production with ControlRecipes
PathServer -> CRServer	Responses to the allocation/deallocation requests
PathClient -> PathServer	Requests for displaying and coordinating Paths and Units of the Process Cell
PathServer -> PathClient	Responses to the display and coordination requests
PathServer -> *Unit Supervision*	Process unit allocations
Unit Supervision -> PathServer	Responses to process unit allocations
CRServer -> *Unit Supervision*	unit recipe invocations
Unit Supervision -> CRServer	Responses to unit recipe invocations

In the application development, the developer has to (re)define only the few *application-specific* custom interfaces that are needed in addition to the interfaces of the messages in Table 6.2, or as their replacements. The new interfaces, potentially defined by the application developer, can be inserted into the framework components in a plug-in manner, described in Chapter 7.2.

Since the batch process management framework is based on a distributed component model, the messages between components are defined in terms of the interface definition language of the component model. The *interface definition language* is *a solution provider* for low level communication issues. The developer of an application does not need to bother about the communication protocol issues.

On the other hand, the *interface definition language* is in general *a constraint* for the implementation of higher level co-operation and interaction techniques. However, also agent interactions, based on communicative acts, discussed in Chapter 4.3, can be embedded within the component interfaces. This is possible by using the *Agentified Component* design pattern presented in Chapter 7.3.

7. The Reuse and Enhancement of the Framework

7.1 Introduction

This chapter presents the *use and reuse* of the batch process management framework in applications. The framework is a so-called *calling framework*. The application developer considers it as a set of interconnected, domain-specific components, which can be inherited and/or parametrized. According to the *inversion of control* principle, the structure of the framework, presented in Chapter 6.3, as well as the dynamic and distribution properties, presented in Chapters 6.4 and 6.5, stay the same in all applications. They are thus considered to be the stable, non-variable domain-specific characteristics (or *frozen spots,* see Chapter 3.4) of the framework.

There are also changing parts (or *hot spots*). They have been designed to be replaceable for each appropriate component type. The framework is thus *customised for a specific application* by:

- Reusing the framework architecture and characteristic properties as such.
- Reusing the domain-specific components by plugging in type-specific interfaces and implementing the respective functionality.
- Giving parameter values via a user interface.

The *plug-in reuse* is possible because the framework is composed of software components, reusable in binary form. The plug-in reuse is utilised by deploying the COM–reuse techniques, *containment* and *aggregation* (not to be mixed with containment and aggregation in object class modelling).

This kind of reuse of the domain components has not been provided by the commercial systems in which various types of modules are considered as independent entities, having no common super-class. Details on how to exploit reusability are presented in Chapter 7.2. A unit recipe for mixing, MixingPE, which contains a more general procedural element, ControlRecipePE component, is used as an example.

Chapter 7.2 also presents how the framework has been *interfaced to a Unit Supervision* subsystem and *to a commercial Digital Control System (DCS)* by reusable components. These, so-called *batch automation components* consist of both Unit Supervision and DCS functionality. The DCS implements equipment procedures that can control the actuators of the process unit and collect measurement information from it. For the Unit Supervision, a unit recipe invocation interface and the respective functionality have been designed and implemented.

A method of an interface (IMixer, for example) is invoked to start the execution of the unit recipe within the process unit. The Unit Supervision part of a batch automation component contains the functionality that is needed to communicate with the DCS. Integration with the DCS has been developed using OLE for Process Control (OPC, 1998) interfaces. A process unit, Mixer, is used as an example in Chapter 7.2.

The enhancement of the framework by *agentifying*, presented in Chapter 7.3, is based on multi-agency, discussed in Chapter 4. Multi-agency is embedded into the batch process management framework so that the *framework* is *reused as such*. In this sense the enhancement of the framework is also an example of its reuse. *Multi-agent technology* is demonstrated as an enabling technology for bringing in local intelligence to those problem-specific parts of a framework having a genuine need for it.

The dynamic process unit allocation is an important batch domain-specific problem, which benefits from multi-agency. The problem is analysed in the first section of Chapter 7.3. A design pattern, the *Agentified Component*, for agentifying software components and component frameworks, is developed and presented in the second section using the *design pattern presentation scheme*, introduced in Chapter 3.3. The pattern is applied to solve the unit allocation problem and the results are compared with the non-agentified, algorithmic solution of the basic framework, in the final section of the Chapter 7.3.

The generic design problem of embedding multi-agency within a component framework was solved by *developing a specialised pattern*, there being no suitable pattern available. This is seen as an indication of the strength of the proposed approach of using generic design patterns for developing domain-specific frameworks. Using insights gained by *pattern reading and deploying*, it is possible to apply them to *pattern designing and writing* and to develop generic (usable in other application domains) design patterns. The Agentified Component design pattern of this thesis has been accepted for EuroPLoP ‘99 Pattern Writing Workshop.

7.2 Use and reuse

7.2.1 Customising the framework components

The methods of the domain-specific components of the framework, e.g. those of the control recipe procedural elements (ControlRecipePE), are accessed by two types of interfaces, a static interface (IControlRecipePE) and a dynamic interface (IStep). The methods of these interfaces, described in Chapter 6, are used as such in each application. However, a customised, type-specific component must provide *an additional interface* with which the type-specific attributes of the component can, at least, be modified (set) and retrieved (get).

The contents of the generic batch-domain component, control recipe procedural element, can be seen in Figure 6.2 of Chapter 6.2. All attributes other than the *formula* are the same in all types of control recipe procedural elements, and the application developer can parametrize them. The formula enables *reuse* of the control recipe procedural element by being a *hot spot*, see Chapter 3.4, of the framework. It contains *type-specific* (different for mixing, feeding, and storing, for example) input, processing and output *parameters*, as described in Chapter 2.2. A type-specific component interface for the procedural element is thus needed to modify these formula values.

A type-specific (here mixing) interface has been added to a generic control recipe procedural element in Figure 7.1, by using the technique of *component containment* (Rogerson, 1997, pp. 160 … 169). Another technique for component inheritance within the COM component model - component aggregation - could also have been used, but containment was chosen, due to the greater flexibility in potential changes needed in the application-specific use of the framework interfaces.

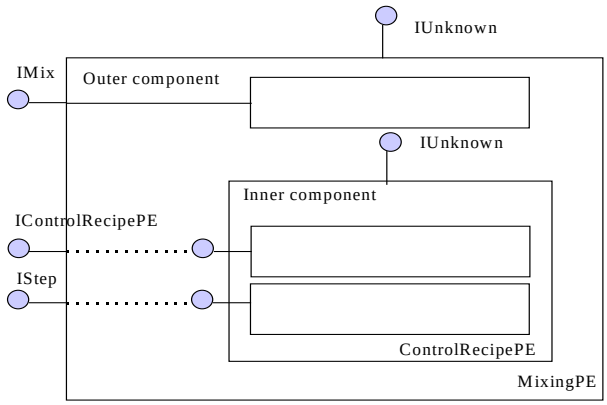

Figure 7.1. Control recipe procedural element reuse.

Logically the inner component, ControlRecipePE, is the more general one, having the outer component, MixingPE, as its subclass. If the method invocations of the outer component interfaces (IControlRecipePE and IStep) are simply delegated to the inner component, the inner component is *reused in a black-box* manner *without adaptation* (see Chapter 3.2).

If adaptation is needed by adding a filtering code, for example (dashed lines in Figure 7.1), it is *black-box adaptation*. The outer component is modified, but it still does not know anything about the implementation of the inner component, but reuses it as such. The black-box adaptation would not be possible if aggregation had been chosen as the component inheritance implementation technique. In aggregation, the interfaces of the inner component are exposed to the user of the outer component as such.

Until now, however, the customisation of the framework has succeeded with the black-box reuse approach. Black-box adaptation, which, of course, increases the application developer's work, has not been needed. This is presumably due to the fact that the component state-dependent functionality has been designed using the state pattern (see Chapter 6.4), and the multithreading functionality using the active object pattern (see Chapter 6.5). The use of these *patterns* has made it possible to encapsulate the respective functionality so that it is sufficient for various application-specific components.

It is, however, important that there is – via component containment – a possibility of employing even black-box adaptation for unanticipated application-specific needs. This is more important within component inheritance than it would be in object inheritance,

because the binary (inner) components are accessible through *immutable interfaces* only (see Chapter 6.5). On the other hand, in comparison to object (implementation) inheritance, the component inheritance has all the black-box benefits. The inner components can be, for example, binary COTS (commercial-off-the-shelf) components.

From the batch domain-specific point of view, it is important, that the inevitably type-specific part of the customised component, the *formula*, is enclosed within the outer component of Figure 7.1. When using a more traditional approach, the parameter information would be stored in a centralised database. In this approach, the parameter information, which is encapsulated within the outer component, has an application-specific interface of its own (IMix in the Figure 7.1). This design *localises* the changes needed when modifying unit recipes and also makes integration with various user interfaces and other component based systems easier.

In general, the *type-specific interfaces* include all the methods needed by a specific component type which are not provided by the generic interfaces (for example, IControlRecipePE and IStep) of the framework. Nothing but a minor loss of comprehensibility, brought about by deeper containment hierarchies, prevents the use of this reuse mechanism recursively. For example, a mixing procedural element (MixingPE) might be further reused as an inner component of a more specific mixing procedural element.

7.2.2 Connecting the framework to a DCS

As described in Chapter 2.2, control recipes are created, modified, and executed within the Process Management control activity. Control recipe procedural elements, which in this framework correspond to the unit recipes, are handled as the smallest procedural entities. On the other hand, the Unit Supervision is the control activity which, according to the batch standard (ISA, 1995), ties the control recipe execution with the equipment control of Process Control.

The Unit Supervision *executes* unit recipe procedures and commands from the Process Management, by initiating and parametrizing equipment procedures. The lowest level, the Process Control control activity encompasses sequential, regulatory and discrete control, distributed among units, equipment modules and control modules.

The Unit Supervision and the Process Control control activities do not belong to the batch process management framework developed in this thesis, see Figure 1.1. However, in order to demonstrate the use of the framework, an interoperation concept to a Digital Control System (DCS), so-called *batch automation components*[9], has been developed (Kuikka, 1998a).

Batch automation components are used to connect the batch process management framework to a DCS (in this case TotalPlant Alcont by Honeywell-Measurex Inc.). Batch automation components are independent entities at the Unit Supervision and the Process Control levels. The components contain both information (state, attributes) and functionality (services, methods) for performing control tasks on the process unit level.

The main *characteristics* of the batch automation components are that:

- They represent *autonomous process units* on Unit Supervision level of the batch standard. The units are controlled by Process Management clients. These initiate services, most importantly unit recipes, and allocate or de-allocate physical units.

- Their services are defined by *external, batch domain-specific interfaces.*

- They have, in addition to external interfaces, also *internal, generic process control interfaces* (OPC, 1998) to the DCS.

- The equipment procedures, needed to carry out the services, are encapsulated within the DCS.

The external interfaces are implemented as custom interfaces of the distributed component model DCOM, Figure 7.2. The *Unit Supervision* control activity is implemented as an executable unit server, into which the in-process OPC server of the DCS is loaded at run time. A batch automation component consists of a DCS implementation (Tommila, 1998) and one OPC Group in the in-process OPC server (DLL, provided by the DCS-vendor, Honeywell-Measurex Inc.), as well as an interfacing component within the unit server. Each batch automation component instance is uniquely identified with a process unit name.

---

[9] The naming of these components may, unfortunately, suggest to a more general use in batch domain.

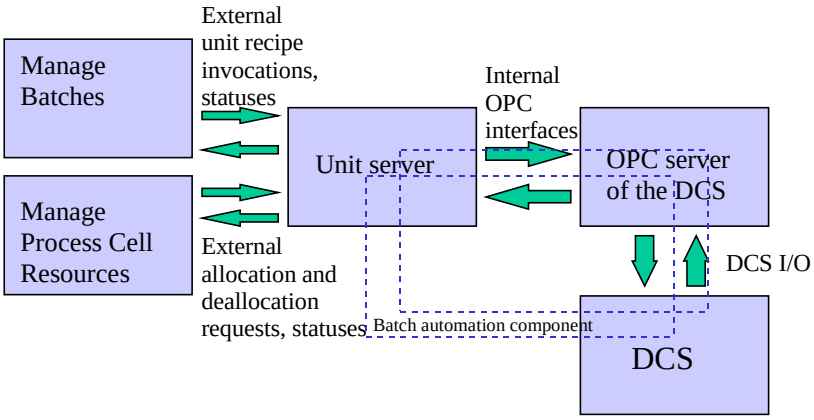

Figure 7.2. Batch automation components.

The interfacing component within the unit server process maps allocations, de-allocations and unit recipe invocations to appropriate internal OPC-interfaces that communicate with the DCS-specific OPC server. The external interfaces conform to the Process Management - Unit Supervision functional layer boundary of the batch standard and the internal interfaces conform to the OPC standard (OPC, 1998). In this way, the unit server *bridges a gap between batch domain-specific* (upper level) functionality *and generic process control* (lower level) functionality.

The *allocation* and *de-allocation* of a batch automation component has been implemented by adding and removing the respective OPC group and writing the allocation and de-allocation information via it to the DCS. OPC groups are added and removed with the AddGroup and RemoveGroup method invocations of the IOPCServer standard interface. The reads and writes needed for the allocation and de-allocation are performed with the OPCWrite method of an IOPCSynch standard interface.

In an analogous manner to control recipe procedural elements, also all the batch automation components *reuse* a common external interface (IBatchComponent) and binary code of the generic services (the methods of IBatchComponent interface) Allocate and Deallocate. The mixer component implementation provides additionally a type

specific service Mix. Thus the external interfaces[10] and their respective methods, provided by the Mixer batch automation component are:

IBatchComponent

Allocate(Unitname, BatchId)

Deallocate(Unitname)

IMixer

Mix(Unitname, BatchId, 1{Material_k, Amount_k}n, MixingSpeed, MixingTime, MixingTemperature, FinalTemperature)

The design and implementation of the generic allocation and de-allocation functionality of a batch automation component, as well as the component-type specific implementation of Mix functionality is described in detail in (Kuikka, 1998b). It uses a hand-shaking protocol for parameter download, equipment procedure activation and report upload.

Various batch automation component types can *reuse*, in addition to the generic domain-specific interface IBatchComponent above, also other interfaces. For example, a chemical reactor process unit normally has a mechanical mixing capability as well. The respective Reactor component thus contains, in a plug-in manner, the IMixer interface above. The clients of the component, for example the batch process management framework, can use a given reactor, in addition to its main tasks, for mixing operations, when the reactor is idle and simple mixers are not available.

The framework – DCS – connection has been *tested* and *experimented* in SWFBatch project (Kuikka, 1998b; Tommila, 1998) in Windows NT – TotalPlant Alcont – environment. The DCS has not been connected to a physical process unit, but the functionality has been simulated with Alcont function blocks.

The new OPC standard (OPC, 1998) also provides facilities for asynchronous alarm handling, based on the COM Connection point implementation mechanism. The versatile mechanism has already been employed within the framework, see Chapter 6.4. This

[10] The 'behaviour oriented' interfaces provided by the Batch automation components are named with nouns instead of verbs (IMixer and not IMix, for instance). The reason for this naming is that - as described in previous section - verb based interface names have already been reserved for control recipe procedural element components.

makes continuation projects concerning exceptional condition handling interesting, also from the implementation point of view.

7.3 Enhancement by agentifying

7.3.1 Problem definition

As pointed out in Chapter 2.2, the dynamic unit allocation within the Manage Process Cell Resources control function is important when considering the flexibility provided by batch process management systems. Chapter 6 presented in general the manner in which the dynamic unit allocation has been specified and implemented in the basic, non-agentified version of the batch process management framework of this thesis.

The *commercial batch control systems*, surveyed in Chapter 2.3, give some support to the dynamic unit allocation. In *InBatch*, an application developer can create and maintain several tables indicating whether or not the units in the process cell belong to paths, have appropriate attributes, and are connected to each other. The InBatch application acts in a *table-driven* way when dynamically allocating a unit. In *VisualBatch* and *OpenBatch*, the approach is *object-oriented*. For each unit class, class properties are defined and unit instances are defined by giving values to the properties used in allocation.

The dynamic *unit allocation* procedure of the framework of this thesis checks the present *feasibility of the path*. In unit allocation, a transfer connection is checked, from the process unit where the batch was previously processed to the unit to be potentially allocated. This is performed in a more convenient manner than in InBatch's table approach, resembling the configuration in VisualBatch and OpenBatch.

The non-agentified dynamic unit allocation of the framework also checks the operational state (terminology of the standard, Chapter 2.2) and the allocation state 'free' or 'allocated' of the unit. In the surveyed commercial systems, only one availability flag is used. In the framework, the operational and allocation states are separate in order to facilitate more accurate conditional allocation. However, in the non-agentified version of the framework, a process unit may be allocated only if its operational state is 'idle' and its allocation state is 'free', which makes the scheme act in practice *similarly to the commercial systems.*

However, there is *potential for substantial improvements* in situations where the units to be allocated have a large amount of local knowledge and autonomy. It is relevant to

consider that *providing a service* (running the respective Unit recipe) is *more expensive in one physical unit than in another*. For example, if a batch might be mixed both in a given reactor and in a given mixer, and both would be idle and free and properly connected, it would generally be cheaper to use a mixer than a reactor. The situation might, however, be occasionally different due to capacity usage or maintenance issues, for instance.

Actual *pricing schemes* have not been implemented in any of the commercial systems discussed. In VisualBatch, it is possible to prioritise the use of units belonging to the same unit class. However, priorities within unit classes do not consider the fact that the *same services* (mixing, for example) may be provided by units belonging to *different unit classes* (mixers and reactors, for example) and that transfer costs between the units should also be considered.

In the *agentified pricing approach* of the framework, each agent that represents a feasible unit is sent a *call for a processing proposal* from the process cell agent. Each agent that represents a feasible path is sent a *call for a transfer proposal*. Each unit/path agent then either *proposes* or *refuses* to perform its processing/transfer task. Also the price for performing the service is proposed by the agent. The price can be computed on the basis of the capacity usage, for example. The selection of a unit and a path is made by the process cell agent, based on the proposals by the unit and path agents.

In batch production there may, additionally, be several batches (and associated control recipes) needing the same process unit simultaneously. Thus, in addition to, and simultaneously with, the allocation of a process unit, *arbitration of the unit* may be needed. The problem of selecting (at a given instance) a control recipe that would get a unit for its use, can be solved by comparing the priorities of the competing control recipes, if available. Additionally, it should also be possible for an urgent, high priority control recipe to *pre-empt* the execution of a normal priority control recipe in an allocated unit.

This need for *priority-based, pre-emptive behaviour of a unit* is another *application-specific problem for agentifying*. On the one hand, local knowledge may be needed in a unit to find out the state of all the relevant equipment and control modules and to subsequently decide whether or not the unit recipe currently running can be interrupted[11]. On the other hand, the concise results (in terms of pre-emption being possible or not)

[11] These decisions are, additionally, (sub)domain-specific, different for example, when producing beverages, medicines and explosives.

from all the units have to be available at the process cell level for decision making, in order to get the urgent batches running.

The overall *approach to dynamic unit allocation* in the batch process management framework is thus based on:

- The feasibility of the units – providing the services needed by the control recipe.
- The availability of the units – having proper instantaneous allocation state and operational state.
- The feasibility of the paths – existing current connection from the preceding unit.
- The prices proposed by the unit agents for processing services.
- The prices proposed by the path agents for batch transfers.
- The pre-emption of the unit activities by high priority control recipes.

The first three issues are included in the non-agentified version of the framework, presented in Chapter 6. The last three issues are solved with the help of a developed generic design pattern. The design pattern, the *Agentified Component*, is described in the next section using the presentation scheme of Chapter 3.3. The new pattern is needed for designing a generic manner in which multi-agency properties may be embedded into a framework consisting of software components.

7.3.2 Agentified Component pattern

Name Agentified Component

Intent Integrate both local and distributed intelligence of multi-agency with host software components and frameworks reused as such.

Motivation

Software components and component frameworks are maturing and beginning to partially fulfil the everpresent goal of practical software *reusability*. Distributed component model

platforms (CORBA implementations and DCOM) enable developers to distribute applications and domain-specific frameworks on local nets and the Internet. *Component communication* is already efficient enough for application requirements in several time-critical domains, for example in telecommunications and automation.

Agent technologies, i.e. local decision making of individual agents and co-operative multi-agent interaction, are also emerging. *Multi-agency* is becoming an important enabling technology for *distributed problem-specific decision making*. The local decision making and *agent interactions* are, however, often time consuming and hard to manage. Multi-agency is not considered here viable as a primary, let alone only, approach for frameworks in time-critical application domains. It should be used instead, only where and when genuine decision making and negotiation is needed. Software components should be reused for more conventional computing and communication.

There is thus a need to *integrate*:

- *Local*, knowledge-based *decision making* (often experimental and exploratory in nature) and *distributed*, agent *interaction* (based on an agent communication language and an evolving agent interaction protocol)

with

- *Local*, component-based *computing* (including often proven and stable solutions within individual components) and *distributed*, component *interoperation* (based on a domain-specific framework on a distributed component model platform)

The Agentified Component pattern, Figure 7.3, describes how to embed multi-agency (local decision-making and agent interaction) into pre-existing software components and frameworks (local computing and component interoperation). The key classes in this pattern are *agent, host component, and agentified component*. The agent interaction requires, additionally, *interaction* and *interaction rule* object classes.

Applicability

Use the Agentified Component pattern in cases where both of the following are true:

- The software, which provides generic or domain-specific functionality, has been developed or is being developed with *software component technology*.

- A need for *intelligent decision making*, enhancing the basic functionality, has been identified, and may be attributed to a few autonomous entities, or agents[12], representing given software components. These ordinary components are called here host components, distinguishing them from the agentified components.

Do not use the Agentified Component in cases where either of the following is true:

- There is *no genuine need for local decision making and/or multi-agent interaction.* Distributed software component-based systems are mostly challenging enough, without the inherent complexity brought about by agency.

- The software to be developed is such that it would be best designed as a *pure multi-agent architecture*, i.e. it has no relevant conventional computing component communication or interoperation needs.

Structure

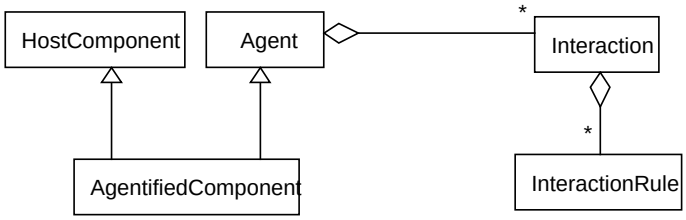

Figure 7.3. Structure of the Agentified Component pattern.

Participants

Agent

- Defines generic operations, attributes and associations for agency.

- Specifically, an agent aggregates instances of the Interaction class.

[12] Either interaction oriented, so-called weak agents, or the so-called strong agents, having an intentional stance (beliefs, desires, intentions), or hybrids of them.

Interaction

- Defines functionality needed to achieve a (sub)goal in agent interaction, from the point of view of the aggregator (parent) agent[13].

- Defines a start state and a set of final states for the interaction, the interaction ends when one of its final states has been reached.

- Specifically, an interaction aggregates instances of the Interaction Rule class.

Interaction Rule

- Defines a given state transition of an interaction if/when a message, i.e. a performative (Labrou & Finin, 1997) or a communicative act (FIPA, 1998) is received from another agent, or if internal antecedents of the transition have become true.

- Specifically defines what to do before and after the state transition of the rule, what local decisions to make and what kind of performative(s) or communicative act(s), if any, to send to other agent(s).

Host Component

- Defines local attributes and operations of the host component[14].

- Specifically encapsulates all methods to be accessed from outside (also from the agentified component), within component interfaces.

Agentified Component

- Inherits (extends) agent functionality from the agent class.

- Implements agent interactions with extended interactions and interaction rules.

- Specifically encapsulates the performative (KQML) or communicative act (FIPA ACL) – or both - within an agent component interface.

---

[13] This is a so-called pre-specified interaction protocol (FIPA, 1998) with potentially several simultaneous interactions per each agent.

[14] In a component framework, there may exist both instances of ordinary host components and agentified components, associated to them according to this pattern.

Collaborations

- In component communication or interoperation, i.e. when the agentified component acts in its *role of a component*, all messages are delegated to the contained host component, Figure 7.4, via IComponent interfaces (there may, of course, be several of them).

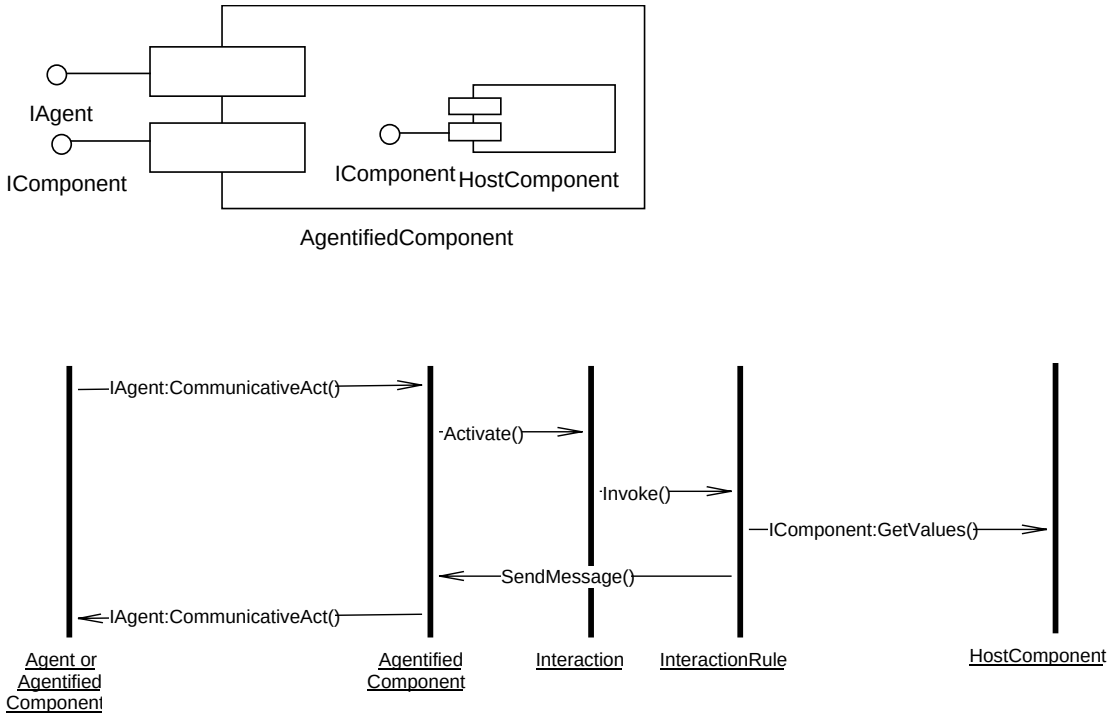

Figure 7.4. Interfaces and interactions of the Agentified Component pattern.

- In agent interaction, i.e. when the agentified component acts in its *role of an agent*, all performatives or communicative acts use the agent interface IAgent, Figure 7.4.

- After receiving a communicative act, the agentified component activates (either starts or resumes execution of) the relevant interaction. The interaction invokes a specific interaction rule according to the present state of the interaction.

- When the agentified component (or objects aggregated by it) needs services of the host component it uses the IComponent interface. The interaction rule, for example, retrieves host component attributes that are needed in local decision making. Reference (or interface pointer) to the host component interface is an attribute of the agentified component, which gets a value, when the host component is instantiated.

Consequences

Some of the benefits and liabilities of the Agentified Component pattern are:

1. Reuse and multi-agency. Components with immutable interfaces are a necessity when striving to industrial software reuse. On the other hand, when designing a solution to a domain-specific problem by using agency, both local knowledge and interaction protocols are constantly changing. The Agentified Component pattern resolves these conflicting issues by enhancement of reusable software components with (less reusable) agent properties and by leaving the original components and frameworks intact.

2. Generics and problem specifics. Generic agent functionality remains unchanged within the agent class. Problem-specific agent interaction can be specified with state machines (situated automata) and implemented by instances of the extended Interaction class. Problem-specific local knowledge can be specified in terms of inference rules, for example, and implemented within instances of the extended Interaction Rule class.

3. Optional agency. Agentified components are easily accessed also by ordinary components via their IComponent interfaces. An agent interaction or negotiation process may begin as a response to a conventional component message. When a given agentified component gets ordinary component messages, it may first decide whether or not to use agency at all in the problem to be solved. If so decided, it may start forming a team of agents and/or agentified components, which will then collaboratively (using agent interaction) work towards the common goal. However, if agency is not needed, based on information in the received message, an algorithmic solution to the problem can proceed.

4. No unnecessary agent interaction. The amount of - potentially performance degrading - agent interaction can be kept to a minimum by using component communication, also with agentified components, when they are acting in their role as components. This is unlike the situation in pure multi-agent architectures, where all communication is normally carried out with agent-interaction practices[15]. There are performatives (and

[15] Agent interaction is often layered, see for example (Finin *et al*., 1997), in three layers: transport protocol (TCP/IP, for example), agent communication (KQML or FIPA ACL, for example), and agent negotiation (conversation theory based implementation, for example).

communicative acts) for this kind of ordinary information exchange, for which the agent interaction brings about extra overheads in comparison with component communication.

5. Embedding multi-agency within a distributed component model. Where genuine agent interaction is needed, even this is encapsulated within an immutable interface (IAgent). The potential of generic agent communication languages (KQML or FIPA ACL) may thus be exploited in a generic distributed component model setting. There are no extra syntactical requirements for agent communication languages, but the interface definition language (IDL) of the component model (CORBA or DCOM) may be used for agentified components in their agent roles, as well.

6. Selection of the agent communication language. Agent communication languages are an area, where standardisation is important, since agents are, by definition, more autonomous than software components, and there is a need for intersystem interaction between 'agencies' developed by various vendors. On the other hand, standardisation is very difficult, because the problems to be solved are varied and application specific. The selection of an agent communication language (in practice either KQML or FIPA ACL), is not critical as such. It has, however, an effect on how to specify and implement interactions and interaction rules.

7. Problem specific local knowledge representation. The Agentified Component pattern does not constrain local knowledge representation within an agentified component, since it is considered to be a highly problem-specific issue. A knowledge base and an inference mechanism are to be designed and implemented within the agent or the agentified component. The Agentified Component pattern as such does not give support for this.

8. No panacea. In all of the domain-specific problems, which need enhancement by multi-agency, it is not possible to identify host components, which could be agentified in a manner described in this pattern. Some problems require either several, totally new, pure agent components or some pure agent components (so-called facilitators, for example) in addition to the agentified components. The Agentified Component pattern is thus not an all-encompassing solution to the problem of agentifying existing component-based systems and frameworks.

Implementation

Issues related to the implementation are concerned here with the application of the pattern when using the distributed component model DCOM. DCOM is important within control applications due to the NT-platform, generally applied in the branch, and the domain-specific OPC – standardisation (OPC, 1998).

1. Component containment. The logical component inheritance has been implemented with a component containment technique (Rogerson, 1997, pp. 160 … 169). The host component is instantiated within an initialisation method after the agentified component has been instantiated. When instantiating the host component, its interface pointer (or one

of them, if there are several) is saved as a member variable of the agentified component. Thus the host component is easily accessible from the agentified component.

2. Delegation of component interface method calls. The method calls of the agentified component in its component role are simply delegated to respective host component interfaces. It is, however, possible to add filtering code within the agentified component before invoking the respective host component method. The agentified component might, for example, check preconditions for invocation of the host component method[16].

3. Component garbage collection. With DCOM components, unlike with JavaBeans, for example, garbage collection is not automatic, but has to be taken care of by the reference-counting mechanism of the component[17] or individual interface. If properly implemented, the host component will take care of its own lifetime. The agentified component must also have the COM standard interface IUnknown, with methods for lifetime management.

4. Multi-agent interaction. There exist several, mostly prototypical implementations of multi-agent architectures and frameworks. In the agentified component interaction implementation, *speech-act* and *conversation theory* based approaches (Winograd, 1988) have been found applicable. Specifically, the co-ordination language COOL (Barbuceanu & Fox, 1995) and its Java based implementation (Chauhan & Baker, 1998) are good examples.

5. Agent communication. KQML has, for a long time, been the de facto standard among agent communication languages. It is, however, oriented to knowledge queries and its semantics are not unambiguously defined. FIPA ACL is more action oriented and it includes a formal Semantic Language (SL) used for defining its communicative acts. FIPA ACL is thus considered a good choice for an agent communication language implementation[18].

6. Inference rules. In small applications, a simple set of rules, which have access to host component's attributes via IComponent interface, is sufficient for knowledge representation. These local rules may be arranged in the implementation so that they are interpreted within an Interaction Rule instance. Thus there is, in effect, a single perceive – decide – act – sequence within each interaction rule invocation.

---

[16] When using another plausible component reuse method of COM, so-called *aggregation*, the host component interfaces are exposed as such and no option for filtering exists.

[17] The ability to have explicit control of component lifetimes is a substantial benefit in some control applications. If it is not needed, there are so-called smart interfaces (Box, 1998, p. 68 ... 69) for taking care of reference counting.

[18] KQML may also be used, by defining the IAgent interface to support its performatives instead of ACL's communicative acts.

Sample Code

The following code fragments and description illustrates implementation of the Agentified Component pattern in DCOM environment using C++ programming language. In this example, an experimental batch process management framework (Kuikka, 1999) is being enhanced with multi-agent properties in order to improve an existing process unit allocation scheme.

An agent class, in this application named BatchAgent, is the agent class from which agentified components ProcessCellAgent, UnitAgent, and PathAgent are inherited. On the other hand, these agentified components contain, using component containment, the respective host components ProcessCell, Path, and Unit. In COM, instantiation of the contained host component (Unit) may be done within the agentified component as follows:

```
HRESULT hr = ::CoCreateInstance(CLSID_Unit, NULL, CLSCTX_ALL, IID_IUnit, (void**)&m_pIUnit);

ASSERT_HRESULT(hr);
```

Where m_pIUnit is the interface pointer of the instantiated host component. Delegation of the methods of the agentified component to the host component, in the case where no filtering is needed, is done simply as indicated in the following code:

```
HRESULT __stdcall UnitAgent::GetValues(unsigned char** pUnitName, long* pBatchId, long* pPriority,
unsigned char** pStatus, unsigned char** pState)

{

        // Call respective host component's method
        HRESULT hr = m_pIUnit->GetValues(pUnitName, pBatchId, pPriority, pStatus, pState);
        ASSERT_HRESULT(hr);
        return S_OK ;
} ;
```

Known Uses

This pattern is a so-called proposed pattern. The author does not know of any other uses, than the one above. The pattern could also be called ‘Agent as a Component’ or ‘Componentized Agent’ when focusing on the component interfaces. However, because the premise (see Motivation) has been to enhance components with agent properties, the name ‘Agentified Component’ has been considered better.

Related Patterns

Elizabeth Kendall and her colleagues have used design patterns for developing multi-agent architectures (Kendall *et al*., 1997). She has previously also proposed a pattern language for multi-agency. The use of generic, catalogued, object-oriented patterns is documented in detail; the patterns developed for agency are described on a more general level.

Alberto Silva and Jose Delgado have developed an interesting Agent Pattern (Silva & Delgado, 1998) for distributed agent systems. Their approach has some resemblance to the Agentified Component pattern, as well as to some existing agent frameworks, with regard to the agent architectural issues. They position their Agent pattern in the middle layer, between the 'OO approach' of CORBA frameworks and the 'agent-based application frameworks'.

The approach of (Aarsten & Brugali, 1997) is perhaps closest to the one presented here. Their goal is also to enhance system functionality (in their case G++ pattern language) by agency. The focus is on weak agency issues, i.e. agent interaction and collaboration. They have also utilised their agent patterns in production scheduling in manufacturing, which is quite close to the application-specific problem in the example case of this pattern.

As to the catalogued object oriented design patterns, the Extension Object (Gamma, 1998) is perhaps closest to the Agentified Component in the interface extension sense. Also it extends existing classes with new interfaces without having an effect on those clients that don't need the added, enhancing interfaces. The pattern Role Decoupling (D'Souza & Wills, 1999) also resembles this pattern in the sense that separate interfaces (or types) are multiply inherited by a given class. The clients use only the interfaces they explicitly require when collaborating with the objects of the class in their various roles.

None of the aforementioned patterns, however, explicitly integrates *reusability* of software components (based on a distributed component model) and local and distributed intelligence through *multi-agency*, which is the intent of this Agentified Component pattern.

7.3.3 Problem solution

The design approach to agentifying is based on the Agentified Component pattern of the previous section, see especially Figures 7.3 and 7.4. The agent interaction capabilities are inherited from a common Agent class, *BatchAgent*, with its associated object classes for interaction. The *BatchAgent* class is *extended* by inheritance to make up problem specific agent classes. These agent classes: *UnitAgent*, *PathAgent*, and *ProcessCellAgent* are provided with a common IAgent interface, see Figure 7.4. The interface is here named *IBatchAgent*, which has as its only method *CommunicativeAct*. The method is needed for embedding the chosen agent communication language FIPA ACL (FIPA, 1998) within a COM interface. All batch agents use this interface for agent communication.

The collaborations of the agentified components ProcessCellAgent, UnitAgents, and PathAgents are based on both agent communication (ACL in COM) and component communication (COM) as described in Figure 7.5[19]. The negotiation behaviour of the participating agentified components in agent interaction is refined with state machines of Figures 7.6 … 7.8. The component interoperation is described in Chapter 6.5. The collaborations thus consist of agent components (structural part) and their interactions (behavioural part) as defined in UML (Booch *et al*., 1999, p. 371).

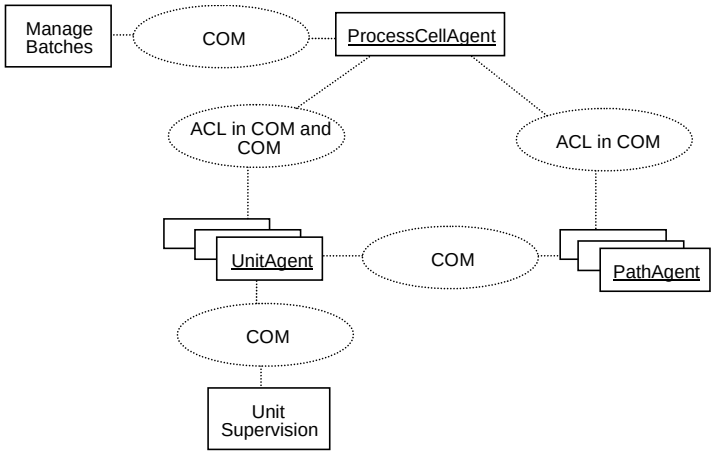

Figure 7.5. Collaborations of the agents in unit allocation.

[19] The collaborations could have been named individually as proposed in (Booch *et al*., 1999, p. 379). Here it was considered more important to make a distinction between agent interaction and component interoperation.

The ProcessCellAgent (a singleton) gets a unit allocation message (COM) from the Manage Batches control activity. The ProcessCellAgent is here instructed by the Manage Batches to use agent interaction (or not to use it, in which case an algorithm is used instead) with a parameter value in the allocation message.

The actual *negotiation process* is begun by the ProcessCellAgent, which interacts with both the UnitAgents and the PathAgents (ACL in COM) using a modified contract net protocol (FIPA, 1998). In the negotiation, the *communicative acts*: call for proposal, propose, and refuse (with appropriate parameters) are used between the ProcessCellAgent and the UnitAgents as well as the PathAgents, respectively.

In decision making, the PathAgents also need state information from the UnitAgents (COM). When an optimal solution (in terms of both processing prices of units and batch transfer prices of paths) has been reached, the respective physical unit is allocated by sending a message (COM) to the Unit Supervision control activity. After that the name of the allocated unit is sent to the Manage Batches control activity for further activities.

More specifically, the *UnitAgent* is responsible for the use of the capacity of a given physical process unit. The UnitAgent (together with its associated Unit component) knows the state and status of the unit as well as the cost of providing a service with the unit. The UnitAgent collaborates with the ProcessCellAgent in unit usage interaction, allocation, and de-allocation. In allocating and de-allocating the actual process unit, it communicates with the respective batch automation component in the Unit Supervision.

The knowledge base of the UnitAgent consists of *rules* to be applied, when the ProcessCellAgent calls for a proposal of a service by invoking the CommunicativeAct method of the IBatchAgent interface:

```
if operational state is idle and allocation state is free
        propose with serviceCost(i)

if operational state is idle and allocation state is allocated and requestor priority is higher than priority
        propose with serviceCost(i)

if operational state is idle and allocation state is allocated and requestor priority is lower than priority
        refuse

if operational state is running and allocation state is free
        refuse
```

notify error

if operational state is running and allocation state is allocated and requestor priority is higher than priority

propose with serviceCost(i)

if operational state is running and allocation state is allocated and requestor priority is lower than priority

refuse

if operational state is not available

refuse

The ProcessCellAgent also allocates and de-allocates the process unit via the UnitAgent by sending component messages. The serviceCost(i)s may change dynamically, based on the processing situation and on long-term reservations, for example. This information is local in the UnitAgent, not known by the ProcessCellAgent.

The *PathAgent* (together with its associated Path component) is responsible for the use of the transfer capacity of a specific production path (or train). A path consists of process units needed to produce a given batch and material transfers between the units. The costs of the transfers are given as input transfer costs, i.e. assigned *to* each unit in the path from the preceding unit. The PathAgent also finds out the feasibility of the path for a given batch by checking that the requesting batch (identified with a batch identifier) has used a service of the preceding unit in the path *before* using a service of the current unit.

The knowledge base of the PathAgent consists of *rules* to be applied, when the ProcessCellAgent calls for a proposal of a given unit (i) for a given batch:

if path is feasible (requesting batch has used the preceding unit(i-1) in this path)

propose with transferCost(i)

if path is not feasible (requesting batch has not used the preceding unit in the path)

refuse

The transferCost(i)s may change dynamically, based on the processing situation and the process unit as well as material availability, for example, in the Manage Process Cell Resources control activity.

The *ProcessCellAgent* (together with its associated ProcessCell component) is an agent that allocates and de-allocates process units, which provide necessary services and reside on feasible paths. The allocations and de-allocations are initiated by requests from the Manage Batches control activity. After a request for a service is received from the Manage Batches, the ProcessCellAgent first finds out from the contained ProcessCell component, all the units providing the requested service. Then it sends calls for a proposal of the requested service to the respective, feasible UnitAgents. For those units which propose the service (with a service-processing price), feasible paths are found, by sending calls for a proposal of the unit to the PathAgents.

After receiving proposals with the service costs from UnitAgents and with the input transfer costs from PathAgents, the ProcessCellAgent optimises the unit selection so that *the total cost of the unit consisting of the service and input transfer costs will be minimised*. After making the selection, it allocates the unit with a minimum total cost. Also, the de-allocation requests are initiated by the ProcessCellAgent.

The 'knowledge base' of the ProcessCellAgent consists of a sequence of calculations to be applied when it resolves the optimal units and the optimal path from the feasible ones:

```
for feasible units ( providing the requested service ) find the proposing units

if none found, return with failure code

else for the proposing units find the feasible paths ( preceding unit used by the batch )
        if none found, return with failure code

calculate totalCost and find the optimal unit
        return name of the optimal unit
```

The optimisation above is a piecewise, *short-term activity*, which takes place in connection of the unit allocation. This is plausible, since the relevant costs may change at any time and since, also the path is chosen anew within every unit allocation. The batch can, so to speak, change paths for every step in the recipe procedure. The agent structure does not preclude, however, longer-term optimisation that could also exploit proactive agent planning activities. This long-term optimisation could be embedded within the unit reservation in Production Planning and Scheduling.

As to the *agent negotiation*, the *ProcessCellAgent* initiates the unit allocation after getting an allocation request from the Manage Batches control activity. The agent aggregates a *ProcessCellConversation* class, from which a new object is instantiated each time the agent gets a new allocation request. Figure 7.6 shows a state machine representation of the ProcessCellConversation.

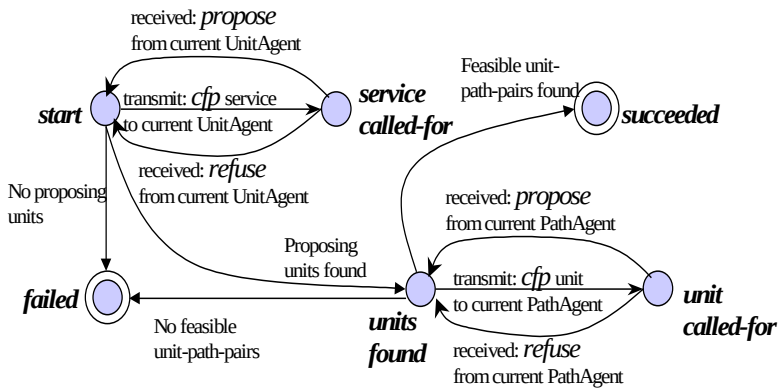

Figure 7.6. ProcessCellConversation.

The ProcessCellAgent finds feasible units (which provide the requested service) without interaction. For each feasible unit, a call for proposal (cfp) of the service is made to the respective (current) UnitAgent. If the UnitAgent proposes the service, the unit is included into a set of proposing units. For each proposing unit, the corresponding feasible paths are found by calling for proposal (cfp) of a unit to all the PathAgents in the process cell.

For all {a proposing unit on a feasible path} – pairs, the total cost is calculated. The final state *failed* indicates the case in which no proposing units or feasible unit-path-pairs have been found. The final state *succeeded* in this conversation indicates the fact that a non-empty set of feasible unit-path pairs has been found in the conversation. The minimum total cost is then obtained and the respective unit allocated.

The UnitAgent has a *UnitConversation* class from which a new object is instantiated each time the agent gets a call for proposal (cfp) of service from the ProcessCellAgent. Figure 7.7 shows its state machine representation.

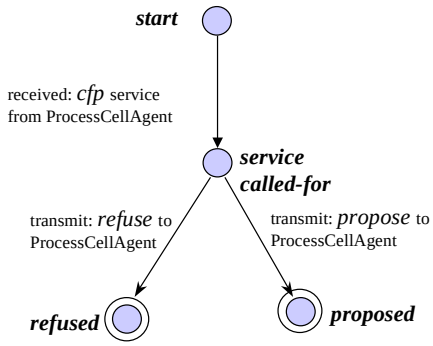

Figure 7.7. UnitConversation.

The UnitAgent evaluates the call for proposal according to its local rules and a refusal or a proposal accompanied by the processing price is transmitted to the ProcessCellAgent. From the point of view of this conversation, the refusal leads to the final state *refused* and the proposal to the final state *proposed*. If a proposition is made, the UnitAgent may or may not subsequently receive a COM message for allocation from the ProcessCellAgent, according to the optimisation performed by it.

The PathAgent has a similar *PathConversation* class from which a new object is instantiated each time the agent gets a call for proposal (cfp) of unit feasibility in the path from the ProcessCellAgent. Figure 7.8 shows its state machine representation.

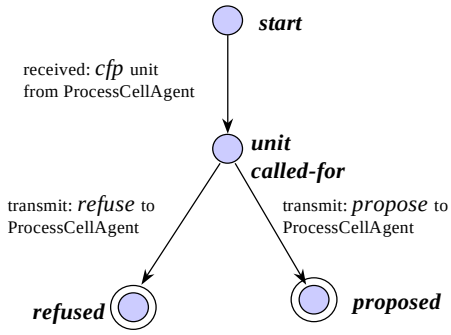

Figure 7.8. PathConversation.

The PathAgent evaluates the unit-path feasibility according to its rules and a refusal or a proposal with the transfer price is transmitted accordingly to the ProcessCellAgent. From

the point of view of this conversation, the refusal leads to the final state *refused* and the proposal to the final state *proposed*.

The above-mentioned unit allocation specific conversations and conversation rules, as well as the local inference rules have been *designed and implemented* within the agentified batch process management framework according to the generic Agentified Component design pattern of the previous section.

When *comparing the non-agentified and agentified ProcessCell functionality*, the following can be noted:

- Either non-agentified or agentified functionality for allocation can be chosen at any time with a parameter of an allocation request. Thus, if needed, a client process can compare the allocation results in a straightforward manner.

- If non-agentified functionality is chosen, the Allocate method of the ProcessCell component, contained by the ProcessCellAgent, is invoked and no multi-agency will be used. The non-agentified algorithm checks the unit and the path feasibilities, as well as the states of the units, but does not try to minimize the costs or to pre-empt the ongoing unit activity by a high priority control recipe.

- If agentified functionality is chosen, the negotiation process is started in the way described above and the total cost of the processing within the unit and of the batch transfer from the preceding unit is minimized. Additionally, if the priority of the allocating control recipe is greater than that of the running recipe within the unit, pre-emption is carried out. This is performed, however, in a simplified manner, without the state checks of the equipment and control modules used by the running control recipe.

The agentified unit allocation scheme succeeds in considering the *pricing* issues and *pre-emptive* needs of an urgent batch, described above. The negotiation procedure needs, however, more processing *time* than the non-agentified algorithm. In experiments comparing the approaches, the processing time taken by the agentified approach was on average approximately twice as long as that taken by the non-agentified approach. The average absolute time for the agentified unit allocation was, however, less than one second when the paths consisted of four consecutive process units.

The dynamic unit allocation problem, solved with agentifying, is well manageable in the sense that the conversations needed are modelled with comparatively simple state models, Figures 7.6, 7.7, and 7.8, providing together a coherent solution to the problem. In this case the benefits of agentifying were clearly greater than the liabilities.

The thesis does *not*, however, argue in favour of the developed agentifying approach in comparison with other potential optimisation techniques, whether distributed or non-distributed. The results only show the *value of agentifying a software component framework* in an applicable problem in comparison with a non-agentified approach.

The following *prerequisities* for successful agentifying were found during the experimentation:

- A specific, negotiable problem (here dynamic unit allocation)

- Sufficient problem-specific decision-making capabilities, attributable to agents associated with domain components (here software agents representing autonomous process units and paths)

- An interaction protocol, suitable for the problem (here a modified contract net)

8. Conclusions

The main contribution of this thesis is the development of an *experimental software component framework for batch process management*. As described in Chapter 5.1, characteristic of the research and development approach of this thesis is that the resulting framework:

- *Conforms to the requirements of the new application-domain standards and considers additional, advanced industrial needs.*

- *Relies for its design and implementation on distributed object and software component technology and on design pattern methodology.*

- *Facilitates local autonomous decision making through multi-agent technology, embedded in the framework.*

The batch control domain is critically studied in Chapter 2. The focus is on the domain standardisation, the existing batch control systems, and the related research. *Domain knowledge* has been found relevant in this thesis, both *in defining the requirements* for the framework (which is commonly agreed upon), and *in making architectural design decisions* for it (which is not commonly understood).

The need to consider production and product-related issues, concretised in recipes, separately from process equipment and control issues, concretised in equipment entities, is evident. This *separation of functionality* and the important *relationships* between the respective control activities form a solid basis for new architectural designs in the batch process management domain.

The applicable technology, object-oriented software component frameworks, is both surveyed and evaluated in Chapter 3. Object orientation and software component technology, which are the constituents of the development work, are described in a concise manner. The *design pattern approach* in the object-oriented software development has been critically evaluated and *chosen* as the design modelling method to develop component frameworks. Domain-specific software frameworks are also introduced, both by definition and by examples relevant to the domain of the thesis.

An approach to deploy generic software *design patterns* for designing *domain-specific component frameworks* has been developed in Chapter 3. It concentrates on the

architectural commonalities of all the applications in a given domain. The approach proceeds gradually from the domain requirements and abstract architectural points of view to more concrete design pattern issues. Guidelines for finding applicable patterns and using them have been adapted, putting special emphasis on relations of the patterns, coherence of the architecture, and domain-specific naming. The approach strives to enable joint development work by software engineers and domain professionals, which is here considered important also when designing commercial domain-specific frameworks.

In some application-specific or even domain-specific problems (like dynamic unit allocation in this framework) *decision making based on local knowledge and distributed interaction* between the local decision-making entities is needed. In Chapter 4, both the software agents, autonomous decision-making entities, and their collaborative interaction using agent communication languages and negotiation protocols are surveyed and evaluated. Several experimental multi-agent architectures have also been studied, since a considerable amount of research results about agent interaction is included in them.

From the point of view of this thesis, one of the research problems was how to *integrate multi-agent interaction in a software component framework*. The need is discussed in the more general context of *developing multi-agent applications*. The role of multi-agency is explicated as a problem-specific enhancement to the framework, not as an overall solution to the industrial needs of distribution, integration and flexibility. The main contribution in Chapter 4 is the notion of *agentifying existing information systems and frameworks*. It retains the deterministic nature of the existing systems, but simultaneously introduces the possibility of solving local problems by a knowledge-based approach.

In the development of the batch process management framework, presented in Chapter 6, the logical requirements analysis model has been developed first. It consists of use cases, object classes and scenarios based on the *batch process management* functionality. Both the *architectural and behavioural aspects* of the framework have been designed with the help of chosen design patterns in solving the batch domain-specific design problems. For the *distribution* and multithreading of the components, both design patterns and facilities of a distributed component model (in this case DCOM) have been needed.

The experimental framework of this thesis is *flexible* both from product and production as well as process equipment and control points of view. It is possible to control and co-ordinate the same processing equipment to produce various products by modifying recipes on-line. It is also possible to utilise alternative and changing equipment to produce a given product.

The framework is *functionally distributed*, because it is based on distributed batch control activities and user interface functionalities. It is also *physically distributed*, because it resides on various threads, processes and computing nodes in a local net. It is *integrated* with externally accessible (interface definition language) component interfaces for interoperation with other systems. Additionally, the internal architecture of the framework is based on components.

The developed framework is a so-called *calling framework*. Thus both the *architectural and behavioural features* of the framework are *reusable as such*. *Individual* framework *components* can be *replaced* by customised versions and *parametrized* for application specific needs. This scheme of reuse has been presented in Chapter 7. In addition to domain-specific component reuse, the framework has also been *integrated* via a standard component interface (OPC) *with a distributed digital control system* (DCS).

For the problem-specific needs (in this thesis for *dynamic unit allocation*) of local decision-making and interaction, an *enhancement* of the framework has been designed, reusing the original framework. No applicable design patterns have been, however, found for this kind of design issue. Since the design problem is at least potentially recurrent, *a generic design pattern, the Agentified Component* has been developed. The pattern, as well as its use in solving the dynamic unit allocation problem, are described in Chapter 7.

The thesis does *not* argue in favour of the developed agentifying approach in comparison with other optimisation techniques, whether distributed or non-distributed. The emphasis is on the *value added by the agentifying* approach to an existing framework, in a situation where sufficient local information exists.

The batch process management framework of this thesis is *experimental* in two meanings of the word. It has been developed *when experimenting* with design pattern and software component based approaches to domain-specific frameworks. The implemented framework is thus a prototype, not a software product. Many conventional programming issues concerning, for example, user interface and data management functionality, have not been refined.

Furthermore, the framework has been developed *to experiment* with so-called intelligent concepts, in particular multi-agency. There have been no substantial constraints of compatibility with old product versions or development tools. The framework should thus indicate the development ideas based on design patterns, software components, and multi-agency in a comprehensible form.

9. Considerations

9.1 Introduction

In Chapter 3, software components and their composition were discussed from a technical point of view. However, for a *component market*, also a need-based *demand* by customers and a *supply* of high-quality components by vendors are necessary. Once the technology is mature enough, and the demand and the supply meet in a market segment, it is probable that application systems will be composed or assembled from software components in that market segment[20].

Furthermore, the component and framework technology and market will also affect the engineering and development *process* of application systems. The main emphasis is on two considerations: *the requirements of the application domain* and *the availability of components and frameworks* on the market. Chapter 9.2 discusses the implications of the component and framework technology on the software and application system market and their influence on the software development process and on the organisation within information technology in general.

The automation domain, specifically *the batch control* domain, provides some interesting points of view of its own. New information technology is generally well accepted within the automation market, especially if its benefits, both in terms of increased product and production quality as well as reduced costs, can be shown. The increased application of fuzzy, neural and genetic algorithm techniques is an indication of this. Batch manufacturers and vendors, on the other hand, have been active in developing domain standards and deploying new computational platforms.

The tradition of software design in the automation domain, however, is based on structured techniques and, in general, on implementation standards, which have proceeded more slowly than, for example, in the domain of telecommunications. In Chapter 9.3, the potential impact of the new *domain-specific*, *software component*, and *design pattern* based approach of this thesis to the automation software market, as well as

---

[20] At least at present, there is no indication that software engineering would in the long run differ in this respect (systems composed of components) from more mature engineering disciplines.

to automation design and training, is discussed. Also some related research and development projects based on the above technologies are reviewed briefly.

Unlike the previous chapters, this chapter is *tentative* in nature. The considerations discussed in Chapters 9.2 and 9.3 are based on the insights acquired when studying the relevant information technologies and when developing the experimental batch process management framework. Chapter 9.2 is also a context for Chapter 9.3, meant as an introduction to the general component market and organisation issues, which are then refined, and focused on automation and batch control in Chapter 9.3.

9.2 Components and frameworks in general

9.2.1 Background for commercial deployment

The size or granularity and *the adaptability* of the market components and frameworks are important issues from the point of view of *deployment* and *marketing*. In a so-called *black-box framework*, only ready-made interfaces and means for parametrization are provided, and all implementation issues are hidden. The framework, as a whole, is one *unit of abstraction* when developing an analysis model for an application system, based on the framework.

It is, however, usual that at least some *customisation* is necessary, using, for example, component containment or object-class inheritance. When deploying the framework in an application system, the framework cannot be considered as one unit. At least something of the inner structure of the framework has to be included in the analysis model of a specific application system. This complicates the deployment of the framework. However, calling frameworks (e.g. the framework of this thesis) need considerably less customisation effort than so-called callable frameworks, which in extreme cases are very close to class libraries, see Chapter 3.4.

A single component either provides a completely new functionality for an application system, or extends the existing functionality. In both cases, a component, as a *unit of extension*, should match with the respective unit of analysis in the requirements definition of the application system (Szyperski, 1998, p. 124 … 128). If the match is not achieved, a component extension could be integrated into an application system with an incomplete

or overly restrictive context. This would make the overall behaviour of the application system unpredictable.

Components are by definition the smallest *units of enhancement*, e.g. the reuse of components or the development and use of an agentified component as described in Chapter 7.2. However, when changing only the implementation of a component, more than just functional properties and inter-component dependencies have to be considered. A client may, for example, be dependent on a timely response to a request, which may be compromised when enhancing the functionality of the components in a manner incompatible with the performance requirements.

Although all frameworks are not necessarily domain-specific, they are usually *concept specific* (Szyperski, 1998, p. 274). A framework encompasses both the high level *architecture* and the integrating *infrastructure*. The *degree of integration* depends, however, on a framework. A framework that is too tightly integrated may leave out potential application systems, whereas a framework that is too loosely integrated will lead to inefficiency and complexity. In the view of this thesis, domain-specific frameworks may be tight, calling frameworks. They are a feasible concept for deployment within application-system deliveries in a given, well-defined domain.

9.2.2 Market potential

Reusable components and frameworks are the main *units of delivery* for system-integrator customers. *Application systems*, on the other hand, are the main units of delivery for end-user customers. Compliance with domain-specific requirements, sufficient documentation, and the need to market and maintain components, add to the costs of both the individual components and the domain-specific frameworks. The reuse of implementation by deploying components and the *reuse of both design and implementation* by deploying frameworks, are needed to make the components and the frameworks financially viable for vendors.

Components and frameworks are commercially justified only if the investment in their development is fully returned. This is achieved by *deploying* the components and the frameworks in a sufficient number of application systems. Within *in-house projects of end customers* the benefits are achieved by a faster development time, lower development costs, a better manageability of the application project, and by the maintainability of the software. The same benefits are also achieved when a *system integrator develops an*

application system for the end user. These benefits are added to the technical advantages of distribution, integration and flexibility, detailed previously in this thesis.

Commercial benefits can also be achieved from marketing and selling components and generic frameworks. An innovative use of new distribution channels, especially based on the Internet, is an important aspect in the creation and growth of these *primary component markets*. Software vendors can, by choosing proper intermediaries, concentrate on their core competence, i.e. the development of components. Alternatively, they can take the whole distribution chain of the software in their control.

A *secondary, complementary market*, is the market of the component *tools*. Because software component development and deployment are more demanding than traditional software, there is an urgent need for supporting tools. Component design tools are being integrated into development environments. Also simple tools for testing the use of component interfaces are usually available within the development environment. Advanced tools for component testing are, currently few or non-existent. Restrictions on the composability of components into applications and frameworks make the development of an effective testing tool a challenging but important task.

There is also a market potential for tools for (visual) component assembly and composition as well as for diagnosis and maintenance of the component system. Assembly and composition tools are likely to be tightly integrated with the development environment, whereas in diagnosis and maintenance there is potential for tools which can even be composed of components from various vendors. Component-tool vendors should, indeed, begin to see the tools that they provide as genuine components. Diagnosis and maintenance tools should be remotely operable, and conform to standards and security policies. Component vendors could also deliver, together with the actual components, the associated components needed in diagnosing the component behaviour within various environments.

System integrators, on the other hand, can focus on the *third existing market* and specialise on deploying components and frameworks, and on marketing them within *component based application systems* to end users. This market is closest to the present application system market, familiar to most information and automation system vendors. As will be described in the next section, the component assembly process, which is the core business process of system integrators, differs substantially from the component and framework development process. Thus *a small vendor should* presumably *concentrate* either on the role of a developer or of an integrator.

However, *a large vendor* may operate with big revenues by acting *both as a developer* of software components and frameworks, *and as a system integrator* to industrial end users. This is shown by some successful, large enterprise resource planning (ERP) and enterprise resource management (ERM) system providers, and by some not so successful smaller integrators. It is obvious that the best application knowledge of the components and frameworks can be kept – if so wanted - within the company that has developed the components or frameworks.

According to Szyperski (1998, pp. 339 … 344), a central paradox in component marketing, which may lead the software-component approach to a financial failure in consumer markets, is how to get people to *pay* for software components that can be downloaded from the Internet. He notes that the access charges to the Internet are so minimal that software distribution has to be moved totally from physical stores to virtual stores. He questions the usefulness of an almost free Internet access if the contents are expensive. He continues that competition will, however, take care of keeping the component price low.

In order to increase the profit from the component sales, Szyperski suggests the use of *branding.* Component users can be encouraged to browse the catalogues carrying the right brands. The brands can be established, not only by component vendors, but also by wholesalers, by brokers or by other intermediate agents. Another way to get income from components, especially from user-interface components delivered via the Internet, is the placing of *advertisements* of tangible (non-software) products in the software components.

Although branding and advertisements have mainly been discussed within the consumer markets, they also have distinctive significance in the business-to-business market. A prominent vendor brand creates trust in its components, whereas company images, created and maintained largely with advertisement, also have a distinctive effect on business-to-business trade.

Another approach to collecting revenues, suggested already by Brad Cox (Cox, 1990) is the *pay-per-use* model. This model is possible for component trade due to the licensing services in both COM and CORBA component models. A single component may be too small a unit for invoicing and the fact that components frequently use the services provided by other components makes this approach tedious. There are advantages, however. The income to the component vendors can be fairly distributed and the component users only pay for the services that they need and subsequently use.

In addition to the software products above, the associated *services* will also be important commercially. It is interesting to see, to what extent the application system *architecting* tasks are kept in the end user company and to what extent they are outsourced. When domain-specific frameworks mature, it is probable that some of the business critical application systems of the end user companies are developed in-house by integrating and customising the frameworks. On the other hand, several non-critical application systems will also, in the future, be integrated by system integrators and delivered as turn-key systems to the end users. Nevertheless, external consulting services will certainly be needed by the end-user companies for architecting and for procurement.

Consulting services will be needed in component assembly and composition, at least when large application systems are developed. Also diagnosis, maintenance and component configuration management may provide interesting possibilities for service businesses. Various distribution channels and intermediaries are also needed in the service market. It is often beneficial to package the components together, not only as frameworks, but also as class libraries or other marketable sets of integrated components.

The correct timing of market entry is crucial in the emerging component business. This is one good reason for larger vendor companies to utilise smaller, flexible and swift consulting companies and software houses as subcontractors. Research institutions and universities can also provide valuable research and development services for the component and framework market.

9.2.3 Software development process and organisation

When developing components and component-based frameworks, the traditional development *methods* are no longer sufficient. Top-down methods involving decomposition will be complemented with new approaches. Structured layering and decomposition are combined with object-oriented use cases, scenarios, object class models and design patterns, as well as component-based reuse and composition methods, as shown in Chapter 6 of this thesis.

It is anticipated here, however, that the division of a software development process into incrementally applied requirements definition, design, implementation (or constructing and packaging), and testing phases will prevail. In fact, the incremental approach is well supported by the component technology, based on encapsulation and interfaces.

When considering the software development organisation, a good starting point is *a general model of reuse processes and organisation*, proposed by Ivar Jacobson. The model defines *three software engineering processes*, each with a specific responsibility (Jacobson *et al*., 1997, pp. 232 … 239):

- A *Component System Engineering* process staffed by a team for each component system. This process designs, constructs and packages components into a component system or framework. The process captures requirements from business models, domain experts, and application end users. The results are used to incrementally architect, design, implement and test components and component systems or frameworks.

- An *Application System Engineering* process staffed by a team for each application system. This process builds application systems by selecting, customising, and assembling components and frameworks from component systems or frameworks. The process starts with requirements capture from the customer and the end users of the application system. The application system is then incrementally analysed, designed, implemented, and tested – by reusing and specialising components and frameworks. Even if the application engineers try to reuse as much as possible, they often have to analyse, design, and implement system features that have little or no support from the component systems or frameworks.

- An *Application Family Engineering* process staffed by a team focusing on the definition of the division into systems and the interfaces between them. This process captures the requirements from a range of customers and transforms them into a suite of application systems. The process produces an architecture that defines the layers of the component systems and frameworks. Also make-or-buy decisions are made, concerning applicable component systems or frameworks.

Another approach to the component-based software development process is formulated in the developing pattern language (see Chapter 3.3) ComponentDesignPatterns (Brown *et al.*, 1999). The so-called *Component Developer Perspective* to the Component-Based Development (CBD) process corresponds to the Component System Engineering process above. In the same pattern language, the so-called *Component Assembler Perspective* corresponds to the Application System Engineering process above. The Component Developer Perspective has, additionally, some similarity to the Application Family Engineering process.

A developer with the Component Developer Perspective analyses the common requirements of the component users and knows how to construct reusable components. The developer focuses on understanding the underlying component object model, and on implementing a solution that can be reused many times in many different contexts. The tasks cover "the burden of multiple-component object models, multiple platforms, and components developed in different programming languages across many address spaces". Special emphasis is put on quality issues, because the developed software will be used in several application systems.

A developer with the Component Assembler Perspective, on the other hand, adapts, customises, and integrates pre-existing components into the application system. The assembler focuses on delivering a solution that adequately solves the business problem at hand with one or several frameworks and pre-existing components. The tasks are domain oriented and focused on productivity and usability. The developers of ComponentDesignPatterns (Brown *et al*., 1999) indicate that the component assemblers often only 'glue' the components together by scripting and it may thus be difficult for them to discover opportunities to create new components based on their experiences in assembling the solution.

Also Ivar Jacobson notes that when the reuse business is first introduced into the organisation, or if the architecture is simple, it may well make sense to combine an instance of Application Family Engineering and Component System Engineering (Jacobson *et al*., 1997). When integrating these two software processes, the earlier Jacobson approach co-incides notably well with the more recent Component Developer and Component Assembler perspectives above.

The transition to component-based development leads to a need for learning on both individual and organisational levels. Individuals need to learn new technologies, processes, and tools. The organisation as a whole has to adapt to the new processes and organisational structures gradually, the larger the organisation, the longer it takes to change. Proper pilot projects and management commitment are also needed in this evolutionary process. Tom Vayda has published a list of guidelines (Vayda, 1999) for the transition to component-based development. It is presented here in a somewhat shortened and modified form:

- *Start small*. Smaller pilot projects and focused business units have a higher probability of success.

- *Pick the highest potential gain*. Choose business units with high visibility and potential to produce a set of reusable components, or a simple, but mission-critical server system.

- *Focus on three time horizons*. The introduction of components should produce useful results quickly, but the medium (change in the organization) and long-term (measurable process) targets must not be forgotten.

- *Apply both top-down and bottom-up strategies*. Both enterprise models of large-scale business processes, and small sets of components are important, for example a component set that implements the security rules of an organization.

- *Measure the results*. Collect simple but powerful project and process metrics that demonstrate the return on investment. The project metrics could include the amount of reuse, or time or cost per function point. The process metrics include the time and effort spent in various life cycle phases and tasks.

9.3 The approach of this thesis

9.3.1 On potential market impact

Component technology offers several benefits for *vendors* of *process-management and process-control systems*. Components provide interoperability, portability, and location transparency, enabling the development of coherent functionality on distributed platforms. The clear separation between interface and implementation, inherent in software components, has been traditionally characteristic of automation entities, as well.

The function blocks within distributed control systems are connected to process inputs, outputs and to each other with messages, represented in the design as graphic signal wires. The compositions of function blocks make up automation modules, one for each control circuit. If the function blocks – and, more importantly, several upper level automation entities – are implemented as software components, they can be more flexibly composed together, into component systems or frameworks, which can be developed and deployed in parallel.

From the point of view of an end-user or a *manufacturing company*, component technology provides a means to capitalise on scale when customising application systems either by developing or by buying components to be used in several applications. This is

beneficial with *user-interface components* (ActiveX Controls or JavaBeans, for example) which can already be plugged into the user interface of a control or process management system. Components are also a way to make the user interfaces of automation and process management systems compatible with office automation and business information systems, based both on desktop and browser technologies.

Batch-manufacturing companies may also be interested in embedding strategic knowledge on products and production in special *automation components*. These domain or application-specific components can be developed in the manufacturing company. After testing, they can be integrated into a generic control or process management system by using the component interfaces of the system.

By composing components, manufacturing companies, system integrators and *process-unit vendors* are also able to develop *larger functional entities* than single components. They can develop *domain-specific component frameworks* and integrate them into control or process management systems. This requires, however, more development resources than component development and is normally not possible for a manufacturing company or a process unit vendor alone.

It seems thus possible that, in addition to the four component markets described in Chapter 9.2 (components and generic frameworks, tools, application systems, services), also a fifth market, that of *domain-specific component frameworks* may eventually emerge in the automation domain[21]. Specialised expertise in the control domain can be combined with software design technology, for example with design patterns, in developing and deploying reusable component frameworks.

The frameworks can then be marketed both to *system integrators*, *process equipment vendors* and *manufacturing end-users* in a *business-to-business market.* The market volume is relatively small but, on the other hand, the customers are, unlike in the consumer market discussed in Chapter 9.2, accustomed to paying well for productive solutions.

The providers of domain-specific component frameworks within automation, can also create *new market opportunities*, especially in cases in which the computational infrastructure and generic component functionality is already in use. The platform of the

[21] There are already some early examples on this in the domain of telecommunications, most notably ACE (Schmidt, 1998).

framework of this thesis, the COM component model and the NT operating system is one example. EnterpriseJavaBeans is another, emerging platform. Furthermore, the end-user companies have to have specific *needs*, which cannot be fulfilled with traditional systems without compromising the business goals of the companies.

On the above premise, there seems to be potential in the batch control domain for software vendors, who would operate in the fifth, *domain-specific framework* market:

- To co-operate with *system integrators* in developing component frameworks (for example intelligent unit allocation or exceptional condition handling) which *enhance* a commercial system (for example InBatch, VisualBatch, or OpenBatch, discussed in Chapter 2). When commercial control systems gradually develop into a more software-component-based direction it will become easier to incorporate component-based frameworks in them.

- To co-operate with *process equipment vendors* in developing process unit specific (on the *Unit Supervision* and *Process Control* levels of functionality) frameworks. These could be *marketed to end users* as embedded parts of so-called *intelligent process units*.[22]

- To co-operate with technology-oriented *manufacturing end users*, the so-called early adopters, using both of the approaches above (developing enhancing frameworks or process unit specific frameworks). This could possibly also succeed in joint operation with a system integrator or process equipment vendor.

As indicated above, the fifth market will be, in the beginning at least, a *mixed* software *product* and *service* market. The potential market *volume* and the correct *timing* for market entry has, of course, to be thoroughly investigated, and both the technical *risks*, discussed in Chapter 3.2, as well as business risks analysed and managed. The component technology provides, however, interesting *opportunities* to start in a small way, in a

[22] The development from physical process unit vendors to *equipment entity* (see Chapter 2.2) vendors has been common long before the emergence of component technology which can, however, further intensify the development.

focused manner, and to first select the gains with high potential, as discussed in Chapter 9.2.

Batch process management has been appropriate as an example domain for the new experimental framework of this thesis, due to the relative familiarity of the process management (sub-)domain via the S88.01 (ISA, 1995) standard. Within *Batch Process Management*, the market potential is mainly in enhancing the existing systems, which have strong market positions.

It is, however, probable that *process unit specific frameworks* will prove to be commercially more significant in terms of new products and services. The reasons for this are:

- Several *process unit vendors* are already *committed* to making their process units truly autonomous, sometimes even intelligent (having local decision-making capabilities). Enclosing the control knowledge of a specific physical unit into a respective equipment entity may be a decisive competing factor for a process unit vendor.

- It is often beneficial, also, for *manufacturing companies* to *encapsulate process knowledge*, for example business critical recipes. Within pulp and paper industries, the recipes used in coating kitchens, are paper-grade-specific trade secrets, not to be exposed to control system or process unit vendors. Components and component frameworks also enable this kind of encapsulation.

- *Unit Supervision* component frameworks can be built on the basis of inexpensive solutions at *Process Control level*, because several control device and PLC vendors provide their solutions with *OPC-servers* (OPC, 1998).

- Component-based frameworks are easier to *integrate* into the new component-based upper level management execution (MES) as well as into enterprise resource planning (ERP) and management (ERM) systems than the conventional, non-component systems.

Multi-agency in the automation domain is useful when fulfilling the need to distribute knowledge and provide problem-specific decision-making capability. Which architectural concepts will be competitive is more difficult to anticipate. It seems possible that *agentified component frameworks* can take their place beside pure multi-agent systems. One reason for this is that the use of multi-agency is embedded, hidden in the framework.

It may also be made invisible to the operators, not familiar with the paradigm. The market potential of multi-agency in the automation domain is also expected to be restricted to specific problems, in contrast to components and domain-specific component frameworks, which may gain wider ground.

9.3.2 On organisation and training

The main theme of this thesis is the *development* of an experimental batch control framework. It is, however, worthwhile discussing the effects of *deploying* frameworks in automation design as well as the organisation of development and deployment. This is due to the fact that the domain-specific framework design, proposed in this thesis, cannot be separated as clearly from the application-specific automation design as within conventional control systems.

Based on the one hand on Chapter 9.2, and on the other hand the tradition of automation design, *a minimum organisational structure* for successful component-based development (CBD) within automation software can be suggested. It could consist of two separate types of teams:

- *A team for component and framework development*
- *A team for application system delivery*

The development team would belong to the automation software vendor organisation whereas the delivery team might belong to the vendor organisation (turnkey deliveries), to a separate system integrator, or even to the customer organisation (in-house development). The development teams would be permanent, whereas the delivery teams may be formed flexibly, in some cases for single projects. A more permanent team organisation could be created, however, to advance and maintain, besides development knowledge, also delivery knowledge within the organisation.

When the *development team* defines the requirements for a domain-specific framework, the commonalities of the domain must be captured. This should be based on domain experience. Additionally, the development team has to have expertise on component models, which act as platforms for the frameworks, as well as efficient component-composition techniques, for example design patterns, as proposed in this thesis. A design

pattern based approach to develop domain-specific frameworks is discussed in detail in Chapter 3.4.

When the *delivery team* analyses the user requirements and constructs models based on them, also the supply of market or in-house components and frameworks has to be considered. It is necessary to be familiar with the available COTS components already in the requirements phase in order to avoid a mismatch between units of requirements analysis and the ready-made components as units of execution, as indicated in Chapter 9.1.

Architectural frameworks, which have been developed by developer teams within system vendors, will be used by parametrizing them and developing new functionality by inheritance and composition. Potentially also the use of design-pattern based, application-specific software development is needed. The emphasis in future deliveries will, however, presumably be on *component* and *framework composition* using visual composition and assembly tools.

Tommila (ATU, 1992) gives a good *overall description of* and *guide to* disciplined and organised *automation design practices*. This approach successfully separates the iterative phases of requirements analysis, functional specification, and design of the application-specific automation. It considers the automation design process from the points of view of process, control and organisation, having the *automation degree* as a central unifying concept which strives to integrate the above-mentioned points of view.

However, the approach does not give help to the automation designer when he/she would like to proceed in the analysis and design in an object-oriented or component-based manner. This is natural because the technology provided by the present automation systems largely follows the structured approach, and is based on the configuration of function blocks, as previously noted. If and when new commercial control and process management systems, based on distributed software components, emerge, the way in which to *design automation applications* will also have to change.

There is thus a need to *incorporate* the *techniques* mentioned above, of software *component and framework development and delivery* into the more general automation-design process. It is, furthermore, possible that *design patterns* could be exploited, not only for automation-software design, but also for general automation design. The essence of design patterns is the capturing and reuse of design knowledge. Patterns have been used, in addition to software design, for designing buildings and organisational

architectures, which are indeed very close to the points of view of process and organisation in the automation design guide (ATU, 1992).

Considering the *training implications* of the above, it is obvious that automation engineers will need substantially *more knowledge on information technology* and *software design capabilities* when developing new plant-wide application systems. The amount of the knowledge to be learned is vast considering the fact that there is not too much obsolete content in the present curricula. One approach is to train automation engineers who will have a broad general knowledge of control engineering and information technology and who will specialise in one or several automation sub-domains.

Fortunately, the present automation students seem to be capable of *working responsibly in teams* whose members have various backgrounds and capabilities. This is necessary to succeed in the challenging industrial automation tasks. In addition to project work, which is at present an important *learning by doing* method when training engineers, there will also be training given in working in conflicting *roles*, as leaders and team members in developer and delivery teams, for example.

9.3.3 On future research and development

The *business aspects of software components and frameworks*, discussed previously in this chapter, will be researched in more detail in a future project on technology assessment. The goals of the project are to evaluate the technological foundations, as well as market demand and supply for component software. Special emphasis will be put on the analysis of the success factors, pitfalls and risks of component development and deployment.

Component and framework technologies will also be incorporated into a research programme, which is focused on virtual factories. One of the main tasks in that programme is to develop a specification and prototypical implementation for *a Supply Chain Management (SCM) framework*. The framework will be concentrated on the needs of independent small and medium size companies, working in a close production co-operation. Especially challenging, from the component and framework technology points of view, is the fact that the goal is to reuse the same SCM framework in different types of inter-organisational, network-based settings.

In the case of *automation design*, a co-operative project with Finnish industries has been started. The goal of the project is to develop a guide for automation design and evaluation which will support specification, documentation, quality assurance, and communication between the various participants in all phases of the design process. The focus is on applications, which have stringent requirements for the quality of automation. The project also intends to incorporate object-oriented and component-based technologies in this more general automation design process.

The batch process management framework, developed in this thesis, has already been used in *training* engineers in component-based software development and in multi-agency. The research and development experience shall, however, be further exploited for didactic purposes. Important ideas have been gained for curriculum development, focusing on a more comprehensive deployment of modern information technologies in automation engineering. For example, the integration of software components and component frameworks into commercial control systems gives an opportunity to integrate automation-software development training (developer team tasks) with automation-design training (delivery team tasks).

The projects and tasks above are concrete, i.e. incipient or ongoing research and development in the area of components and frameworks. The (more recent) multi-agency approach developed in this thesis, the agentifying of components, is considered to be one feasible way to solve application or domain-specific problems within a component framework. The ideas for its further application, presented next, are still in a conceptual phase.

The dynamic unit-allocation approach of this thesis can be generalised into problems concerning *long-term resource reservation*, with the help of agentified components. This approach presupposes, in addition to the agentifying of relevant components, also further architectural modelling of process cell equipment. When developing longer term, proactive agent functionality, there are also interesting possibilities for adaptive planning, discussed in Chapter 4.2.

There is also potential for agency within control room user interfaces. Personalised and semi-autonomous *user assistance functionality* can be based on the operators' cognitive models of the controlled process. This functionality may be embedded as agentified components into a user-interface framework. In this kind of research and development the potential of human-machine psychology and new information technology could be integrated and realised into novel user-interface concepts.

References

Aarsten, A. & Brugali, D. From Object Orientation to Agent orientation. In: Proceedings of ICRA'97 Workshop on Object Oriented Methods for Distributed Control Architectures. Albuquerque, New Mexico, 1997.

Aarsten, A., Elia, G., Menga, G. G++ : A Pattern Language for Computer-Integrated Manufacturing. In: Pattern Languages of Program Design, Addison-Wesley (Software Patterns Series), 1995.

Aitken, J. *et al.* A Knowledge Level Characterisation of Multi-Agent Systems. Proceedings of The ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 1994, pp. 179 ... 190.

Alexander, C., Ishikawa, S. & Silverstein, S. A Pattern Language. Oxford University Press, 1977.

Arnold, K. & Gosling, J. The Java Programming Language, 2. Ed., Addison-Wesley, 1998.

ATU, Process control and management - task definition of process automation. Suomen Automaation Tuki Oy (ATU), 1992 (in Finnish).

Barbuceanu, M. & Fox, M. COOL: A Language for Describing Coordination in Multi-Agent Systems. In: Proceedings of the First International Conference on Multi-Agent Systems, AAA Press/The MIT Press, 1995.

Barbuceanu, M. & Fox, M. Integrating Communicative Action, Conversations and Decision Theory to Coordinate Agents. Proceedings of the ACM Agents'97 Conference, USA, 1997.

Basili, V. The Role of Experimentation in Software Engineering: Past, Current, and Future, Proceedings of ICSE-18, IEEE, 1996, pp. 442 … 449.

Batory, D. & Geraci, B. Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering (special issue on Software Reuse), February 1997, pp. 67 … 82.

Beck, K. & Cunningham, W. Using Pattern Languages for Object-Oriented Programs. Technical Report No. CR-87-43, Submitted to the OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming, http://c2.com/doc/oopsla87.html, [referenced 29.4.1999]

Beck, K. & Johnson, R. Patterns Generate Architectures. European Conference On Object Oriented Programming, ECOOP'94, 1994.

Booch, G., Rumbaugh, J. & Jacobson, I. The Unified Modeling Language User Guide. Addison-Wesley, 1999.

Bosch, J. Adapting Object-Oriented Components. Proceedings of the 2nd International Workshop on Component-Oriented Programming, 1997.

Box, D. Essential COM. Object Technology Series. Addison-Wesley, 1998.

Bradshaw, J.M. *et al.* KAoS: Toward an industrial-strength open agent architecture. In: J.M. Bradshaw (Ed.) Software Agents. AAAI/MIT Press, 1997, pp. 375 … 418.

Brandl, D. Making Batch Plants Bloom. Chemical Engineering, June 1998, pp. 72 … 79.

Brockschmidt, K. Inside OLE, Microsoft Press, 1995.

Brooks, R. Intelligence without representation. Artificial Intelligence 47 (1991), pp. 139 ... 159.

Brown, K., Eskelin, P. & Pryce, N. Component Design Patterns: A Pattern Language for Component-Based Development. http://c2.com/cgi/wiki?ComponentDesignPatterns, [referenced 23.4.1999]

Buchi, M. & Weck, W. A Plea for Grey-Box Components. TUCS Technical Report No 122, Turku Centre for Computer Science, 1997.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. A System of Patterns - Pattern-Oriented Software Architecture. Wiley, 1996.

Chauhan, D. & Baker, A. Developing Coherent Multiagent Systems using JAFMAS. In: Proceedings of the Third International Conference on Multi-Agent Systems, IEEE Computer Society, 1998.

Cohen, P. & Levesque, H. Communicative actions for artificial agents. In: Software Agents, AAI Press/The MIT Press, 1997.

Cohen, P. & Levesque, H. Intention is Choice with Commitment. Artificial Intelligence, 42 (1990), pp. 213 ... 261.

Coplien, J. & Schmidt, D. Pattern Languages of Program Design. Addison-Wesley (Software Patterns Series), 1995.

Cox, B. Planning for software industrial revolution. IEEE Software, vol. 7, 1990.

D'Souza, D. Interface Specification, Refinement, and Design with UML/Catalysis. Journal of Object Oriented Programming, June 1998.

D'Souza, D. & Wills, A. Pattern 16.15 Role Decoupling. In: Objects, Components, and Frameworks with UML – The Catalysis Approach, Addison Wesley, 1999.

Dagermo, P. & Knutsson, J. Development of an Object-Oriented Framework for Vessel Control Systems. Technical Report, ESPRIT III/ESSI Project, nr.10496, 1996.

De Hondt, K., Lucas, C. & Steyaert, P. Reuse Contracts as Component Interface Descriptions. Proceedings of the 2 nd International Workshop on Component-Oriented Programming, 1997.

Dean, T. & Wellman, M. Planning and Control. Morgan Kaufmann Publishers Inc., USA, 1991.

Doscher, D., *et al.* (editors), SEMATECH Computer Integrated Manufacturing (CIM) Framework Architecture Guide. V1.0, 1997.

Doscher, D., *et al.* (editors), SEMATECH Computer Integrated Manufacturing (CIM) Framework Specification. V2.0, 1998.

Ferguson, I. On the role of BDI modeling for integrated control and coordinated behavior in autonomous agents. Applied Artificial Intelligence, 9 (1995), pp. 421 ... 447.

Fikes, R. & Nilsson, N. STRIPS, a retrospective. Artificial Intelligence 59 (1993), pp. 227 ... 232.

Finin, T., Labrou, Y. & Mayfield, J. KQML as an Agent Communication Language. In: Software Agents, AAI Press/The MIT Press, 1997.

FIPA, Foundation For Intelligent Physical Agents. FIPA 97 Specification, Version 2.0, Part 2, Agent Communication Language, 1998.

Fisher, T. Batch control systems: Design, application, and implementation. Instrument Society of America (ISA), 1990.

Fleming, D. & Schreiber, P. Batch Processing Design Example, World Batch Forum, 1998.

Fowler, M. & Scott, K. UML Distilled - Applying the Standard Object Modeling Language. Addison-Wesley, 1997.

Franklin, S. & Gaesser, A. Is it an agent, or just a program ? In Intelligent Agents III, Springer-Verlag, 1997, pp. 21 … 36.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Gamma, E. Extension Object. In: Pattern Languages of Program Design 3, Addison-Wesley, 1998.

Garlan, D. & Perry, D. Special Issue on Software Architecture. IEEE Transactions on Software Engineering, vol. 21, no. 4, April 1995.

Genesereth, M. An Agent-Based Framework for interoperability. In: Software Agents, AAI Press/The MIT Press, 1997.

Genesereth, M. & Fikes, R. Knowledge Interchange Format Version 1. Reference Manual, Stanford University, 1992.

Genesereth, M. Knowledge Interface Format Specification. X3T2 Group on KIF, 1995.

Guilfoyle, C. Vendors of Intelligent Agent Technologies: A Market Overview. In: Jennings, N. & Wooldridge, M. (Ed.) Agent Technology. Springer-Verlag, 1998, pp. 91 … 104.

Haikala, I. & Märijärvi, J. Software Engineering, Suomen ATK-kustannus, 1997 (in Finnish).

Halme, A. *et al*. Bacterium Robot Society - A Biologically Inspired Multi-Agent Concept for Internal Monitoring and Controlling of Processes. IEEE/RSJ International Conference on Intelligent Robots and Systems, 1996.

Haxthausen, N. Bottlenecks in batch integration - can standards help remove them ? Proceedings of the World Batch Forum 1998.

Hayes-Roth, B. An architecture for adaptive intelligent systems. Artificial Intelligence 72 (1995), pp. 329 ... 365.

Hayes-Roth, B. Architectural Foundations for Real-Time Performance. In: Intelligent Agents, The Journal of Real-Time Systems, 2 (1990), pp. 99 ... 125.

Heikkilä, A., Kuikka, S., Tommila, T. & Ventä, O. Intelligent batch control - disciplined control engineering and advanced software technology. Automation Technology Review 1997, VTT Automation, pp. 38 … 47.

Hodges, B. Component Specification and Conformance: What Components must Expose to Compose. Position Paper for the OMG/DARPA Workshop on Compositional Software Architectures, Monterey, USA., January 1998.

IBM Agent Building Environment Developer's Toolkit. Level 6, IBM Intelligent Agent Center of Competence, 1997a.

IBM, IBM SanFrancisco Technical Summary, 1997b, http://www.software.ibm.com/ad/sanfrancisco/prd_summary.html, [referenced 23.4.1999].

IBM, IBM Taligent, Building Object-Oriented Frameworks, 1995, http://www.ibm.com/java/education/oobuilding/index.html, [referenced 23.4.1999].

IEC IEC 61512-1: Batch control - Part 1: Models and terminology. Final draft international standard, 1997, International Electrotechnical Commission (IEC).

InBatch, InBatch User’s Guide. Wonderware Inc., 1998.

ISA, ISA-S88.01-1995, Standard: Batch Control. Part 1: Models and Terminology. The International Society for Measurement and Control, 1995.

ISA, ISA-dS88.02-1999a, Standard draft: Batch control - Part 2: Data structures and Guidelines for programming languages. Draft 13, 1999. Instrument Society of America(ISA).

ISA, ISA-dS95.01-1999b, Standard draft: Enterprise – Control System Integration - Part 1: Models and Terminology. Draft 11, 1999. Instrument Society of America(ISA).

ISA, Practical Guides for Measurement and Control: Batch Control., 1996, Instrument Society of America(ISA).

ISA-TR88, ISA-TR88.0.03-1996, Technical Report: Possible Recipe Procedure Presentation Format. The International Society for Measurement and Control, 1996.

ISO 10628, Flow diagrams for process plants - Part 1: General rules. International Organization for Standardization (ISO), draft international standard, May 1992.

Jacobson, I., Griss, M. & Jonsson, P. Software Reuse - Architecture, Process and Organization for Business Success. ACM Press, Addison-Wesley, 1997.

Jennings, N. *et al*. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. Artificial Intelligence 75 (1995) pp. 195 … 240.

Jennings, N. *et al*. A Roadmap of Agent research and Development. Autonomous Agents and Multi-Agent Systems. Vol. 1(1998), No.1, pp. 7 … 38.

Johnson, R. Documenting Frameworks using Patterns. Proceedings of OOPSLA, Object-Oriented Programming Systems, Languages and Applications, 1992.

Johnsson, C. & Årzen, K-E. Grafchart and Batch Recipe Structures. World Batch Forum, USA, 1998.

Johnsson, C. & Årzen, K-E. Batch Recipe Structuring using High-Level Grafchart. USA, IFAC'1996a.

Johnsson, C. & Årzen, K-E. Object Tokens in High-Level Grafchart. CIMAT'1996b, France.

Karhela, T., Kuikka, S. & Paljakka, M. Application of Web Browser and Software Component Technologies in Operator User Interfaces in Process Simulation: A Case Study on dynamic Simulation of a Rotary Lime Kiln. Eurosim Congress, 1998.

Kendall, E. *et al*. Multiagent system design based on object-oriented patterns. Journal of Object Oriented Programming, June 1997.

Kent, S. Constraint Diagrams: Visualising Invariants in Object-Oriented Models. Proceedings of OOPSLA, Object-Oriented Programming Systems, Languages and Applications, 1997.

Kopetz, H. Component-based design of large distributed real-time systems. Control Engineering Practice 6, 1998.

Kruchten, P. The 4+1 View Model of Architecture. IEEE Software, November 1995, 12 (6), pp. 42 … 50.

Kruglinski, D. Inside Visual C++. Microsoft Corporation, 1996.

Kuikka, S. An Experimental Framework for Batch Process Management. World Batch Forum (WBF'99), USA, 1999.

Kuikka, S. & Karhela, T. Application of Software component and Web browser technologies in Automation. Helsinki University of Technology, Information and computer systems in automation, Report 1, June 1998.

Kuikka, S. Requirements Definition and Specification in Automation Software Production, Licentiate thesis, Tampere University of Technology, 1985 (in Finnish).

Kuikka, S. On Batch automation components. Project report of SWFBatch-project, VTT Automation Industrial Automation, 1998a.

Kuikka, S. On Batch automation component interfaces and the responsibilities of OPC client and server. Project report of SWFBatch-project, VTT Automation Industrial Automation, 1998b.

Kuikka, S. Planning and Control. Helsinki University of Technology, Seminar on Knowledge Engineering, 1997, http://www.cs.hut.fi/~sto/planning-seminaari/kuikka/sk_tutkielm1.htm, [referenced 23.4.1999]

Kuikka, S., Tommila, T. & Ventä, O. Distributed Batch Process Management Framework based on Design patterns and Software Components. World Congress of International Federation of Automatic Control (IFAC'99), 1999.

Kuikka, S. & Valtari, K. Distributed Multi-Agent Systems, an interaction oriented approach. Helsinki University of Technology, Software Agents Seminar, 1998, http://smartpush.cs.hut.fi/SoftwareAgents/Seminarpapers/Distributed_Multi-Agent_Systems/Distributed_Multi-Agent_Systems.htm, [referenced 23.4.1999]

Kuikka, S. & Ventä, O. Object oriented, distributed batch control architecture. Proceedings of Automation Days 97, 1997.

Kuroda, C. & Ishida, M. A Proposal for Decentralized Cooperative Decision-Making in Chemical Batch Operation. Engineering Applications of Artificial Intelligence, Vol. 6 (1993), No. 5, pp. 399 ... 407.

Labrou, Y. & Finin, T. A Proposal for a new KQML Specification. CSEE/UMBBC, Technical Report CS-97-03, February 1997.

Lavender, K. & Schmidt, D. Active Object - An Object Behavioral Pattern for Concurrent Programming. Proceedings of ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA, 1996.

Lesser, V. Multiagent Systems: An Emerging Subdiscipline of AI. ACM Computing Surveys, Vol. 27(1995), No. 3, pp. 340 ... 342.

Lieberman, H. Integrating User Interface Agents with Conventional Applications. Proceedings of the ACM Conference on Intelligent User Interfaces, January 1998.

Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings of First ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA, 1986.

Lindqvist, U. & Jonsson, E. A Map of Security Risks Associated with Using COTS. IEEE Computer, June 1998.

Martin, R., Riehle, D. & Buschmann, F. Pattern Languages of Program Design 3. Addison-Wesley (Software Patterns Series), 1997.

Mayer, B., Mingins, C. & Schmidt, H. Providing Trusted Component to the Industry. IEEE Computer, May 1998.

Mayer, B. Object-Oriented Software Construction. 2nd ed., Prentice-Hall 1997.

Mayer, B. Tell Less, Say More: The Power of Impliciteness. IEEE Computer, July 1998.

Microsoft, The Component Object Model Specification. Draft Version 0.9. Microsoft Corporation and Digital Equipment Corporation, 1995.

Microsoft, Microsoft Windows NT Server DCOM Technical Overview White Paper, Microsoft Corporation, 1996.

Mikhajlov, L. & Sekerinski, E. A Study of The Fragile Base Class Problem. 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, 1998.

Mueller, J. *et al*. Modeling Reactive Behaviour in Vertically Layered Agent Architectures. Proceedings of The ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 1994, pp. 261 ... 276.

Mueller, J. The design of Intelligent Agents. Springer Verlag, 1996.

Musliner, D. *et al*. World modeling for the dynamic construction of real-time control plans. Artificial Intelligence 74 (1995), pp. 83 ... 127.

Noble, J. Classifying Relationships between Object-Oriented Design Patterns. 1998, http://www.mri.mq.edu.au/~kjx/classify.ps [referenced 29.4.1999]

OLE, OLE 2 Programmer's Reference. Microsoft Corporation, 1995.

OMG, The Common Object Request Broker: Architecture and Specification. Object Management Group, 1995.

OPC, OLE for Process Control. Data Access Standard V2.0, OPC Foundation, USA, 1998.

OpenBatch, OpenBatch 3.1 Professional Edition. Technical Brief. Sequencia Corporation, USA, 1999.

Plasil, F. & Stal, M. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. Software – Concepts & Tools, No. 19, 1998, pp. 14 … 28.

Pree, W. Hot-spot-driven framework development. Summer School on Reusable Architectures in Object-Oriented software Development, Tampere, Finland, 1995.

Rabelo, R. & Camarinha-Matos, L. Negotiation in Multi-agent based Dynamic Scheduling. Robotics and Computer-Integrated Manufacturing Vol. 11 (1994), No. 4, pp. 303 ... 309.

Roberts, D. & Johnson, R. Evolve Frameworks into Domain-Specific Languages. Proceedings of the 3rd International Conference on Pattern Languages for Programming, USA, September 1996.

Rogerson, D. Inside COM. Microsoft's Component Object Model. Microsoft Press, 1997.

Rosenof, H. & Ghosh, A. Batch process automation - Theory and practice. New York, Van Nostrand Reinhold Company Inc., 1987.

Rosenschein, R. & Zlotkin, G. Rules of Encounter. The MIT Press, 1994.

Rumbaugh, J. *et al*. Object-Oriented Modelling and Design. Prentice Hall, 1991.

Russel, S. & Norvig, P. Artificial intelligence - A Modern Approach. Prentice Hall, USA, 1995.

Schmidt, D. An Architectural Overview of the ACE Framework. A Case-study of Successful Cross-platform Systems Software Reuse. USENIX login magazine, Tools special issue, November, 1998.

Schmidt, D., Johnson, R. & Fayad, M. Communications of the ACM. Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10, October 1996.

Schmidt, D. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. Communications of the ACM, Vol. 38, No. 10, October 1995.

Sharratt, P. Chemicals manufacture by batch processes. In: Handbook of Batch Process Design, Blackie Academic and Professional (Chapman & Hall), 1997.

Silva, A. & Delgado, J. The Agent Pattern: A design Pattern for Dynamic and Distributed Applications. Third European Conference on Pattern Languages of Programming and Computing, 1998.

Simensen, J., *et al*. A Multiple-View Batch Plant Information Model. PSE'1997, Norway, 1997.

Smith, R. The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers 29, 1980.

Soni, D. *et al*. Software Architecture in Industrial Applications. Proceedings of the 17th, International Conference on Software Engineering, Seattle, Washington USA, 1995.

Sun Microsystems, Enterprise JavaBeans Architecture Specification V1.0, Sun Microsystems, 1998.

Szyperski, C. Component Software - Beyond Object-Oriented Programming. Addison - Wesley, 1998.

Thompson, C. (editor), Workshop Report, Workshop on Compositional Software Architectures. Monterey, USA, 1998.

Tittus, M., Egardt, B. & Lennartson, B. Plant and Product Models for Batch Processes. 3rd European Control Conference, Italy, 1995.

Tittus, M. & Egardt, B. On the Use of Multiple Models and Formal Control Synthesis in Batch Control. 13th World Congress of IFAC, USA, 1996.

Tommila, T. Batch process automation. Espoo, Technical Research Centre of Finland (VTT), Research Notes 1487, 1993 (in Finnish).

Tommila, T. On the implementation of batch automation components. Project report of SWFBatch-project, 1998.

UML Object Constraint Language (OCL) Specification, 1997.

Vayda, T. Organizing for Components – Managing risk and maximizing reuse. Component Strategies, February 1999.

VisualBatch, VisualBatch 3.0 Electronic Books. Intellution Inc. 1998.

Vlissides, J., Coplien, J. & Kerth, N. Pattern Languages of Program Design 2. Addison-Wesley (Software Patterns Series), 1996.

Vlissides, J. Pattern Hatching - Design Patterns Applied. Addison-Wesley, 1998.

Voas, J. Certifying Off-the-Shelf Software Components. IEEE Computer, June 1998.

Williams, T. (editor) A Reference Model For Computer Integrated Manufacturing (CIM) - A Description from the Viewpoint of Industrial Automation. CIM Reference Model Committee of International Purdue Workshop on Industrial Computer Systems, Instrument Society of America, 1989.

Winograd, T. A Language/Action Perspective on the Design of Cooperative Work. Human-Computer Interaction 3:1 (1987-88), pp. 3 … 30.

Wooldridge, M. Agents as a Rorschach Test. In: Intelligent Agents III. Springer-Verlag, 1997, pp. 47 … 48.

Åkesson, K. & Tittus, M. Modular Control for Avoiding Deadlock in Batch processes. World Batch Forum, USA, 1998.

Årzen, K-E & Johnsson, C. Object-oriented SFC and SP88 recipes. World Batch Forum, Canada, 1996.

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 398
VTT–PUBS–398

Author(s)
Kuikka, Seppo

Title

A batch process management framework

Domain-specific, design pattern and software component based approach

Abstract

Requirements for product and production quality and variability, as well as the needs for the efficient use of production equipment, have emphasised the benefits of *batch production* in the process industries. The resulting complexity of batch control has, however, been a challenge to control engineers. Emerging batch standards and software component technologies have now made it possible to design *flexible*, *distributed*, and *integrated* batch automation concepts to satisfy the requirements.

The batch control domain was studied in this thesis in terms of domain standardisation, existing batch control systems, and related research approaches. The applicable information technology, object-oriented software component frameworks and multi-agency, was surveyed and evaluated. Guidelines for deploying generic software *design patterns* in designing *domain-specific frameworks*, were adapted.

An experimental *batch process management framework*, was developed to fulfil the aforementioned *flexibility*, *distribution* and *integration* needs for batch automation. It also demonstrates *reusability* by the so-called *calling framework* architectural style as well as internal and external component interfaces. Framework *components* may be easily parametrized and *replaced* by customised versions. Additionally, the framework can be integrated with other systems by using component technology.

For some problem specific needs of local decision-making and interaction, enhancement of component frameworks may be needed. No applicable design patterns were found for this kind of design issue. Since the design problem is recurrent, a generic design pattern, *Agentified Component*, was developed and experimented with within the framework of this thesis. The approach retains the deterministic nature of the framework, important in the automation domain, but simultaneously introduces the possibility of solving specific problems using a knowledge-based approach.

Keywords
batch control, object-oriented software, batch process management, framework, design pattern, software component, software agent

Activity unit
VTT Automation, Industrial Automation, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland

ISBN
951–38–5541–4 (soft back ed.)
951–38–5542–2 (URL: http://www.inf.vtt.fi/pdf/)

Project number
A9SU00171

Date	Language	Pages	Price
November 1999	English	215 p.	E

Name of project

Commissioned by
VTT Automation,
National Technology Agency (Tekes)

Series title and ISSN
VTT Publications
1235–0621 (soft back ed.)
1455–0849 (URL: http://www.inf.vtt.fi/pdf/)

Sold by
VTT Information Service
P.O.Box 2000, FIN–02044 VTT, Finland
Phone internat. +358 9 456 4404
Fax +358 9 456 4374