BETA *Institute for Business Engineering and Technology Application*

On Systems Architecting

a study in shop floor control to determine architecting concepts and principles

Arian Zwegers

Ta Prohm, one of Angkor’s temples, was built in approximately 1186 AD. After the defeat of the Khmer empire, most of the temples were left to the jungle. For centuries, the jungle threatened the structures of Ta Prohm. The temple was intertwined by trees that reduced the strength of its structure. Ta Prohm survived the encroaching trees, but would have gone lost in the jungle without human intervention. However, removing some of the trees might lead to a partial collapse of the temple.

After years of evolution, shop floor control systems frequently consist of entangled components. Modifications violate the system architecture, and the complexity of the system becomes increasingly hard to manage. Because of the entanglement of the components, changes in a component propagate to other components. Systems become so complex, that they can hardly be changed anymore.

Systems architecting tries to maintain the integrity of complex systems so that they can be adapted to future requirements. This study explores the systems architecting discipline.

On Systems Architecting

A study in shop floor control to determine architecting concepts and principles

On Systems Architecting

A study in shop floor control to determine architecting concepts and principles

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 30 juni 1998 om 16.00 uur

door

Arian Zwegers

geboren te Asten

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. E.J. Sol
en
prof.dr.ir. J.C. Wortmann

en de copromotor:
dr.ir. H.J. Pels

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Zwegers, Arian

On systems architecting : a study in shop floor control to determine architecting concepts and principles / door Arian Zwegers. - Eindhoven : Technische Universiteit Eindhoven, 1998.
Proefschrift.

ISBN 90-386-0699-4
NUGI 684

Subject headings: systems architecting / shop floor control / enterprise integration

Printed by University Press Facilities, Eindhoven
Cover photos by Jelle van Maare

BETA Central Secretary, Pav. A03, PO Box 513, 5600 MB Eindhoven, The Netherlands.
Phone: +31 40 247 3983. Fax: +31 40 245 0258. Email: BETA@tm.tue.nl

Table of Contents

1. Introduction

1.1 Introduction

A system *architecture* is the manner in which system components are organised and integrated. The architecture is important in determining the effectiveness of a system because it establishes the functionality of the system. It is also important because it establishes the limitations or possibilities for changing the system in the future; the value of a proper architecture lies in the fact that it provides ‘future flexibility’.

Architecting, the (planning and) designing of architectures, arose long time ago. It was a response to problems that were too complex to be solved by pre-established rules and procedures. Many thousands of years ago, city states began public works of such complexity that engineering alone could not solve the resultant problems. Egyptians built canals and irrigation systems that not only exploited the Nile, but largely determined the economic status of the country. Phoenicians built maritime and naval fleets. The Greeks and Romans built cities, aqueducts, fortifications, and empire-spanning road systems. From these endeavours arose the first civil, military, and naval architectures (Rechtin, 1991).

Conceiving architectures required *architects* who were able to bring together various disciplines in order to understand and resolve problems of large complexity. Building architects had to bring together structures, psychology, art, and aesthetics. Civil-works architects had to integrate structures, mechanics, politics, economics, geology, and hydrology. Naval architects combined structures, mechanics, hydrodynamics, aerodynamics, meteorology, economics, and military engineering. All had to take into account the physical and social environment in which their efforts were undertaken. Systems architects do likewise, though their interests are primarily functional rather than aesthetic (Rechtin, 1991).

Similar developments as described above are underway in aerospace, electrical, industrial, and other engineering fields. Several observations justify this statement. First, it can be seen in the literature. The word ‘architecture’ is now widely used in communications, space systems, computers, software, networks, Computer Integrated Manufacturing, and transportation systems. Second, if there are architectures, there must be architects to design them. Indeed there are, although they have gone under different names. They have been called configurators, chief engineers, chief designers, design team leaders, and advanced-systems engineers. But they are architects, nonetheless. Even more, architecting is not limited to the engineering domains anymore. From politicians to football coaches, people who outline the structures of complex undertakings, such as treaties and football teams, are called architects in everyday language (Rechtin, 1991).

1.2 Shop floor control

In this thesis, the focus is on architectures of shop floor control systems. This section places shop floor control in the larger frame of production control.

Bertrand *et al.* define production control as ‘the coordination of supply and production activities in manufacturing systems to achieve a specific delivery flexibility and delivery reliability at minimum costs’ (Bertrand *et al.*, 1990a; p. 17). Figure 1-1 shows a production control framework based on the Material Requirements Planning (MRP) and Manufacturing Resource Planning (MRP II) approaches as they are frequently applied in industry.

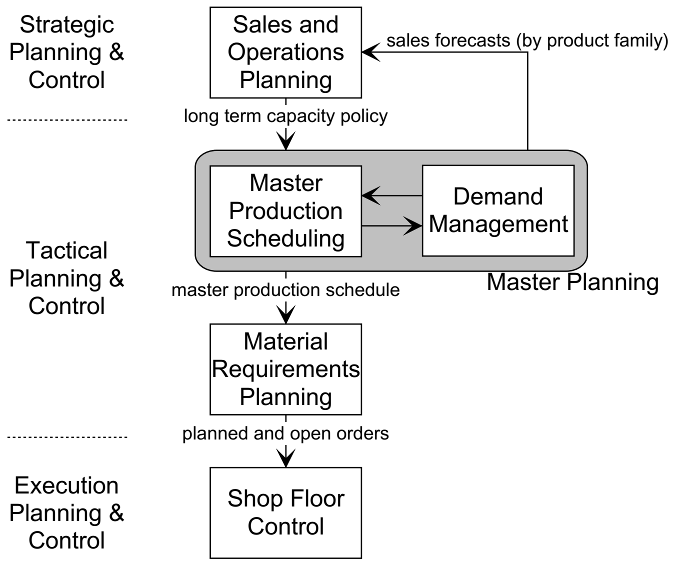

Strategic Planning & Control
Sales and Operations Planning
sales forecasts (by product family)
long term capacity policy
Master Production Scheduling
Demand Management
Tactical Planning & Control
master production schedule
Master Planning
Material Requirements Planning
planned and open orders
Execution Planning & Control
Shop Floor Control

Figure 1-1 Production control framework

Source: Higgins *et al.* (1996)

The strategic planning and control layer consists of the sales and operations planning activity. Its objective is to ensure that manufacturing capacity is adjusted to anticipated levels of sales. The real output of the sales and operations planning process towards master production scheduling should be viewed as a set of statements with respect to future enhancements or reductions of capacity (Higgins *et al.*, 1996).

The tactical planning and control layer plans and controls purchasing and manufacturing activity for all (inventory) items in response to firm and anticipated demand. Master production scheduling drives the MRP logic (Higgins *et al.*, 1996).

Execution or operational planning and control issues essentially involve taking the output from the tactical planning phase, e.g. the planned orders from an MRP system, and managing the manufacturing system in quasi-real time to meet these requirements. The execution planning and control layer supplements the MRP logic with the required logic to plan and control the manufacturing environment on a day-by-day basis. Its most important objective is to minimise lead times and work-in-process (inventory), primarily by accurate control over which orders are released on the shop floor and how. Frequently, separate work load control and work order release functions can be distinguished (Higgins *et al.*, 1996).

A framework for production control is outlined above from a functional point of view. Before focusing on shop floor control, a technical point of view is taken by reflecting upon the evolution of production control systems (see Figure 1-2). Figure 1-2 and the following discussion are based on (MESA, 1997).

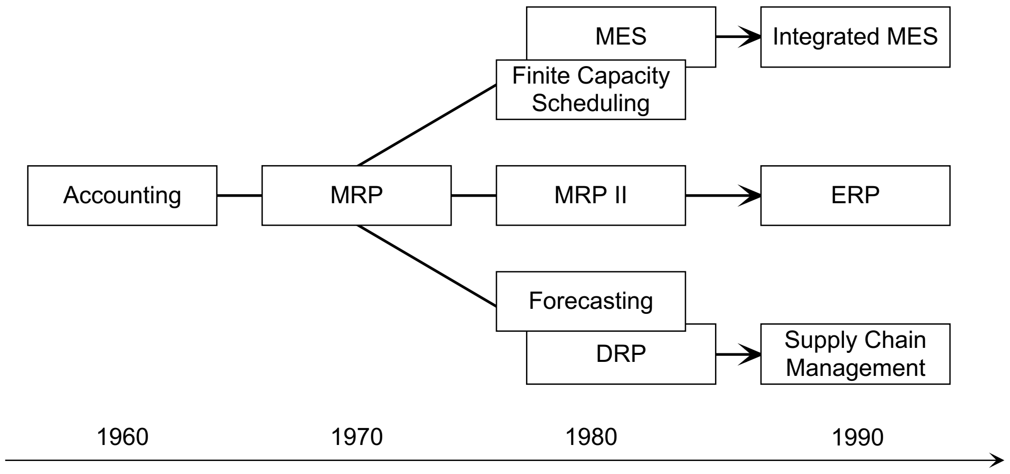


Figure 1-2 The evolution of manufacturing systems

Source: MESA (1997)

The first computerised business systems were used in accounting. By the late 1960’s or early 1970’s, from these accounting systems evolved Material Requirements Planning (MRP), which was intended to help manufacturers better plan material availability. By the late 1970’s and early 1980’s, computers were more powerful and capable of handling more data and being used interactively by more people. MRP evolved into Manufacturing Resource Planning (MRP II) as shop floor reporting systems, purchasing systems, and related functions were added.

At about the same time, many companies began to realise they needed other systems to manage other aspects of their businesses. MRP II did not address the requirements of forecasting and managing demand in distribution, nor did it adequately manage the shop floor and the many disparate activities that took place there. Forecasting and Distribution Requirements Planning (DRP) were developed to address the requirements in distribution. Likewise, Manufacturing Execution Systems (MES) and a number of unique, function-specific systems such as scheduling and quality management, evolved. While these systems helped manufacturers solve specific business problems, they were not integrated and often could not take advantage of data from or pass data to other systems.

In the late 1980’s and early 1990’s, another generation of systems became available. These systems attempted to solve the ‘islands of information’ problem by providing broad comprehensive solutions. MRP II systems became Enterprise Resource Planning systems.

DRP became Supply Chain Management and the shop floor solutions evolved into Integrated MES systems. In all cases, solution-focused systems such as quality or maintenance management, remained viable alternatives for companies which required more functionality than what was available in the integrated solutions.

Manufacturing Execution Systems emerged as a response to a deficit of both Material Requirements Planning and Manufacturing Resource Planning systems: both are not designed to respond to real-time data as it happens on the shop floor. Although planning systems could create a shop schedule weekly or even daily, planners often find at the end of the day that schedules are not realised and few tasks went as planned. Therefore, a new type of software solution appeared. These systems ensure that the plans coming from planning systems actually get done, and if they do not, they make sure that the planning systems are notified and updated (Esch, 1995). For that type of production control software, the term ‘Manufacturing Execution Systems (MES)’ was coined by the consulting firm Advanced Manufacturing Research (AMR).

According to AMR’s definition, a MES is ‘an information system that resides on the plant floor between the planning systems in offices and direct industrial controls at the process itself’. Figure 1-3 shows that AMR positions MES between MRP II or Enterprise Resource Planning (ERP) systems on the one side and supervisory control systems on the other side. Supervisory control refers to terminals, PCs, workstations, or servers linked to operators, programmable controllers, or other shop floor devices. The execution system takes plans from a corporate manufacturing planning (MRP II) system and executes them on the shop floor. The MES is supposed to transform data from shop-floor devices such as programmable logic controllers (PLCs) and distributed control systems (DCS) into information an MRP II system can use to improve overall planning (Hill, 1995).

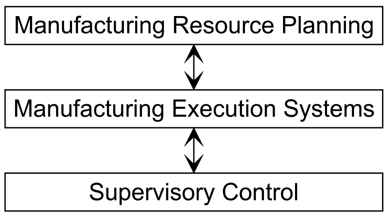

Manufacturing Resource Planning
Manufacturing Execution Systems
Supervisory Control

Figure 1-3 Production control software

A group of about 25 MES developers and vendors joined forces to promote the MES concepts, and formed MESA (Manufacturing Execution System Association) International. In 1993, their total market for integrated MES systems was $150 million. In 1995, AMR projected the total market would be $280 million before growing to $414 million and $613 million in 1996 and 1997, respectively. Growth rates were projected to continue at 30 percent or greater through the end of the decade (Hill, 1995; MESA, 1997).

Manufacturing Execution Systems are commercial, standard implementations of shop floor control software. Rather than using the term ‘MES’, this thesis uses the more general term ‘shop floor control systems’.

1.3 Problem statement

Shop floor control systems have grown in importance. However, with the rapid influx of computerised equipment and their interfaces to ERP systems and supervisory control equipment, the complexity of shop floor control systems is exploding. The result of this tendency is that the complexity of these systems becomes hard to manage, and that the systems are hardly maintainable and sensitive to failures. Apart from consequences for the effectiveness of the system, the flexibility (i.e. the modifiability, extensibility, and reconfigurability) diminishes.

The aim of this thesis is to provide a contribution to solving or avoiding these problems. Emphasis is on the architecture of shop floor control software, i.e. on the manner in which components of shop floor control systems are organised and integrated. The creations of architects have to last for many years, unlike the creations of other professionals such as authors, actors, or even engineers. After time, new business objectives and requirements will come up, and the current shop floor control system will no longer satisfy. Rather than throwing away the old shop floor control software and starting development of a new solution from scratch, a company wants to change its existing control software. It is the architect who establishes the limitations or possibilities for changing the system in the future. This thesis aims to formulate guidelines with which system architects are able to design more flexible systems.

1.4 Research objective

The primary research objective is to find architectural guidelines that lead to more flexible systems. This implies a number of research questions:

1.1. A central preposition in this thesis is that the system architecture determines the limitations and possibilities of changing the system in the future. Van Waes (1991) and other authors notice, however, that most publications assume that the reader understands what is meant by the term ‘architecture’. The consequence of such an assumption is that the term remains undefined, and that different readers interpret the term differently. Therefore, the first research question is about the meaning of the word ‘architecture’: what is (an) architecture? And, what is architecting?

1.2. The next research question is about how an architect manages complexity. Architecting emerged as a response to complex problems. Architects developed their own means to combat complexity, their own architecting concepts. What are these architecting concepts?

1.3. With only architecting concepts, an architect can not create a flexible system. The architecting concepts enable one to manage the complexity of the design problem. However, guiding principles are needed that lead to flexible systems. What architecting principles lead to flexible, evolving systems?

The secondary research question is to evaluate the suitability of several theories for the design of flexible systems. The evaluation is carried out by means of the discerned architecting concepts and principles. The application domain is the area of shop floor control. The secondary research question is broken down into three subquestions:

2.1 When the problem of legacy systems that hinder the evolution of shop floor control systems became obvious, several frameworks were constructed to remedy this problem. Some of these proposed solutions became known as enterprise reference architectures. One enterprise reference architecture is examined in particular, namely CIMOSA. The research questions with regard to CIMOSA are: to what degree does CIMOSA incorporate the architecting concepts and principles that are found as answers to research questions 1.2 and 1.3? How and to what degree does it contribute to the evolution of (shop floor control) systems?

2.2 Whereas the enterprise reference architectures try to achieve continuous evolution of enterprises, also more specific solutions were proposed, such as reference models for shop floor control. Almost the same research questions are posed for these reference models as for the reference architectures: to what degree do these reference models incorporate the architecting principles that are found at research question 1.3? How and to what degree do they contribute to the evolution of shop floor control systems?

2.3 The architecting concepts and principles can be applied during the development of a specific system. Both enterprise reference architectures and reference models for shop floor control are generic solutions; they aim to support the development of many (shop floor control) systems. A shop floor control system can just as well be developed without these reference architectures and reference models. How can the architecting concepts and principles support one in the design of a specific system?

1.5 Research approach

The occasion to start this research project was the observation that hierarchical control architectures might lead to expensive, rigid systems. In the second half of the 1980's, Philips and Digital Equipment Corporation experimented with control architectures. The two companies developed the CAM Reference Model, which was based on hierarchical control concepts. During the CIMphony project at Philips Centre For manufacturing Technologies, the CAM Reference Model was the basis for the development of a control architecture for an assembly line (Philips CFT, 1987). The project demonstrated that a thoughtless implementation of the hierarchical control form leads to very expensive systems.

The research project was started in 1993 in the section Manufacturing Technology of the department of Industrial Engineering & Management Science at Eindhoven University of Technology. At its initiation, it was envisaged that the project would examine and simulate several control forms that offered alternatives for the hierarchical form, which was the dominant control form at that time. However, it was realised that writing a thesis on architectures required 'field work' with very complex systems. This was needed in order to acquire knowledge and insight about systems architecting. Systems architecting was a young, immature discipline, that is if one could talk about a 'discipline' at all. It was a science in a 'pre-paradigm phase' (Kuhn, 1970). At the time, one could not learn it from study books. Therefore, this research project started with working in the field for a significant period to acquire knowledge and insight.

During the first two years, this research project contributed to the ESPRIT project VOICE*, which validated the Open System Architecture for Computer Integrated Manufacturing (CIMOSA). This validation was performed by the development of manufacturing control and monitoring solutions according to the CIMOSA concepts in three types of industry. The contribution to the VOICE project consisted of the application of the CIMOSA concepts to model the control, monitoring, and production processes of a Greek aluminium foundry. A second contribution was the definition of an 'engineering approach' that should describe how to use CIMOSA in practice.

A research goal during the VOICE project was to evaluate the suitability of CIMOSA for shop floor control architecting. During the VOICE project, however, it became clear that more research was needed in the concept of 'architecting' and even an 'architecture' itself. The terms were too vague and needed to be deepened before CIMOSA could be evaluated or before alternative control architectures could be compared. The primary research questions were formulated. Later, the discerned architecting concepts and principles were used to evaluate CIMOSA and reference models for shop floor control.

In the first half of 1996, an action research project was carried out at Baan Company. The Gordian project (named after the famous Gordian knot) investigated the possibilities to place functionality that was formerly spread out over several software packages in one, new package. To do so, the entanglements of the functionality with other packages had to be examined, and solutions to disentangle the packages were proposed. The Gordian project was used to sharpen the discerned architecting concepts and principles.

In 1994, the model factory that formerly belonged to Digital's Cooperative Engineering Centre was moved to Eindhoven University of Technology. This model factory was one of the research objects for this project. The model factory was used as an example in this project for designing architectural solutions.

1.6 Structure of the dissertation

Figure 1-4 provides a schematic overview of the remainder of this dissertation. In Chapter 2, architecture and related notions are defined, and the process of architecting is introduced. Chapter 3 focuses on the concepts to manage the complexity faced by a development project; three such architecting concepts are elaborated upon, while three other concepts are shortly discussed. Three architecting principles are formulated in Chapter 4 that aim to guide architects in the development of flexible, future-proof systems. Chapter 5 discusses the theory

* VOICE: Validating OSA in Industrial CIM Environments

behind enterprise reference architectures, and the practice of enterprise integration. Chapter 6 presents various reference models for shop floor control. Both the reference models in Chapter 6 and the reference architectures in Chapter 5 are confronted with the discerned architecting concepts and principles. Chapter 7 presents the application of the concepts and principles in an example. A shop floor control system is defined based on agents. Finally, Chapter 8 presents the main conclusions of this research project, and mentions a few possible directions for further research.

Chapter 2, Architecture Definition
- Definition of architecture, reference model, and reference architecture
- Definition of architecting

Chapter 3, Architecting Concepts
- Concepts to manage overall system complexity
- Introduction of domains, decomposition hierarchy, and views as main architecting concepts

Chapter 4, Architecting Principles
- Principles to obtain future system flexibility
- Introduction of modularity, structural stability, and layers as main architecting principles

Chapter 5, Reference Architectures for Enterprise Integration
- Theory and practice of enterprise integration
- CIMOSA and architecting concepts and principles

Chapter 6, Reference Models for Shop Floor Control
- Evolution of control architectures and enabling technology
- Shop Floor Control reference models and architecting concepts and principles

Chapter 7, Shop Floor Control Architecting
- Architecting of an agent based control system
- Application of architecting concepts and principles

Chapter 8, Conclusions and Suggestions
- Conclusions
- Directions for further research

Figure 1-4 Overview of this thesis

2. Architecture Definition

2.1 Introduction

The objective of this chapter is to provide an understanding of the term ‘architecture’ as used in this thesis. In Section 2.2, an initial definition of ‘architecture’ is derived. It is shown that architecture has three main meanings in everyday language, namely related to science, style, and structure.

For a good understanding of the concept, it is necessary to learn from other disciplines, where researchers and engineers have decades of experiences with architectures and architecting. Section 2.3 gives some opinions on architecture from various disciplines, starting – naturally – with the building discipline, where architects have been around from time immemorial. Digital systems engineering was the first to borrow the term. However, the original thoughts in this field about architecture (e.g. (Amdahl *et al.*, 1964; Blaauw, 1966; 1971; 1976)) appear to be quite out of date for this thesis’ purposes. Some efforts in information management have enlarged the scope of thinking about architectures. Except for building science, software engineering is probably the discipline where architectural design is best understood. Finally, various opinions on architecture stemming from the domain of this thesis, Computer Integrated Manufacturing, are reflected.

In Section 2.4, the definition as used in this thesis is presented. A distinction is made between architectures, reference architectures, and reference models. Since architectures are frequently confused with other concepts and vice versa, the differences between architectures on the one side and the other concepts on the other side are given. Subsequently, the objectives and roles of architectures are clarified.

Section 2.5 explains architecting: the process or design activities that is responsible for the system architecture and the integrity of the system. Architecting is positioned towards engineering to explain their differences and to demonstrate why both have a critical impact on the success of a development project.

An architecture encompasses more than the overall structure of a specific system. However, frequently architectural specifications only deal with static system aspects. Section 2.6 shows that architecting involves the dynamic and non-functional aspects of a system as well.

2.2 Initial definition

The term ‘architecture’ is widely used in publications. It is generally assumed that the reader knows what this term means (Zachman, 1987; Van Waes, 1991; Soni *et al.*, 1995; Garlan and Perry, 1995). In this section, an initial definition is derived by a small dictionary survey. In Section 2.4, the definition is refined.

The word ‘architect’ is derived from two Greek words:

- ‘archi’ (Gr. ‘αρχη’) = a beginning, start, principle, e.g. as in ‘archetype’ or ‘archbishop’.
- ‘tekton’ (Gr. ‘τεκτων’) = bricklayer, mason (Hionides, 1978).

The same dictionary gives a translation for the word ‘architekton’ (Gr. ‘αρχιτεκτων’), which simply means ‘architect’. In the view of the ancient Greeks, an architect is a master builder, a chief artificer.

Merriam-Webster’s collegiate dictionary (Merriam, 1993) defines architecture as:

1. the art or science of building; specifically: the art or practice of designing and building structures and especially habitable ones

2a formation or construction as or as if as the result of conscious act

2b a unifying or coherent form or structure

3. architectural product or work
4. a method or style of building
5. the manner in which the components of a computer or computer system are organised and integrated.

The fifth meaning sticks out for two reasons. Firstly, because architecture is explicitly associated with computers. In fact, the discipline that deals with the design of computer systems, digital systems engineering, was the first to adopt the term ‘architecture’ from building science. Secondly, because this definition is clearly the one that is most related to the one used in this thesis. Other dictionaries give similar definitions to Merriam-Webster’s. Combining Merriam-Webster’s second, third, and fifth definition, and comparing their definitions with those of other dictionaries and everyday usage shows basically three different main meanings:

- the art and/or *science* of planning, designing, and constructing buildings (e.g. ‘Vitruvius wrote ten books on architecture’);
- a *style* and/or method of design and construction (e.g. ‘Beijing’s Tiantan, the Temple of Heaven, is considered by many as the perfection of Ming architecture’);
- a unifying or coherent *structure* (e.g. ‘the architecture of the system consists of the following components’).

For the moment, it suffices to remember that there are three meanings of ‘architecture’, related to science, style, and structure. In later sections, this thesis focuses on the third meaning, structure, and advocates that an architecture incorporates dynamic and non-functional aspects besides static, structural aspects.

Note that all three meanings are uncountable nouns. In English, an uncountable noun has only one form; it does not have separate singular and plural forms. Only the latter meaning has a countable form as well. A countable noun has a singular form and a plural form. When it is singular, it must always have a determiner in front of it. If one speaks about ‘*an* architecture’, always the third meaning is used (Crowther, 1995).

In the next section, all three different meanings are used confusedly. It is up to the reader to pick the right understanding. In later sections, more precise definitions are given which attempt to avoid a confusion of tongues.

2.3 Architecture in various disciplines

2.3.1 Building science

The first building science theory was formulated by Marcus Vitruvius Pollio, a Roman architect and engineer in the first century BC. His ten books are the oldest and most influential works on building science in existence. The instructions he gave were followed for hundreds of years. Most building architects are familiar with Vitruvius' three targets of architecture as shown in Figure 2-1: firmness (Latin: 'firmitas'), efficacy ('utilitas'), and gracefulness or charm ('venustas') (Morgan, 1960; Germann, 1980).

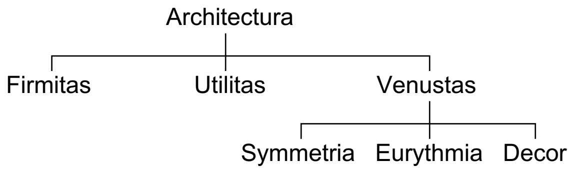


Figure 2-1 The targets of architecture according to Vitruvius

Source: Germann (1980)

A modern interpretation of architecture is given by Bax (1996). He defines 'architecture' as the quality of the coherence of all categories of technological design. Architecture is independent of a discipline. The discipline 'architecture' is the field in which this quality is achieved within a building. As such, Bax considers architecture as the science of (controlled) complexity.

2.3.2 Digital systems engineering

Probably the first to use the term 'architecture' in an engineering discipline outside the building discipline were the designers of the IBM System / 360. These pioneers adopted the name 'architecture' from building science because its original meaning refers to the first element, the archetype, in this case of a digital system design. In their view, the term 'architecture' is used to describe the attributes of a (hardware) system as seen by the programmer, i.e. the conceptual structure and functional behaviour, as distinct from the organisation of the data flow and controls, the logical design, and the physical implementation (Amdahl *et al.*, 1964).

The architecture is one of the three elements or phases in the design of digital systems as discerned by Blaauw (1966, 1971, 1976). The first, the architecture, specifies what functions the system can perform. The architecture is not the vague, general appearance of the system, but it covers all details a user might find out and if necessary will find out. These details are most apparent when a system is operating at the lines of its abilities (Blaauw, 1966).

The second is the implementation, which states how these functions can be performed by a logical structure. The third is the realisation, which concerns the physical structure which embodies the logic. In particular the components which are selected and the locations where they are placed are of concern here.

Furthermore, Blaauw claims that for defining ‘good architecture’, digital systems engineering is assisted by building science. A good digital system architecture needs to have structure, thereby clearly embodying a basic idea. It needs to be functional, i.e. it should bear the user (either man or machine) in mind. And it needs to comply with the ‘laws of beauty’, namely symmetry, balance, and independence.

In digital systems engineering, an understanding of ‘architecture’ is adopted that is very close to the building sector. The architecture should serve the user; it is seen as the feeling a user senses when dealing with a certain system. However, the following sections show that in other disciplines architecture does not refer so much to the user’s view of a system as to the system’s structure and protocols that organise and integrate the system components, and which may be quite hidden from the user. This latter view is adopted by this thesis.

2.3.3 Information management

An initial contribution to information systems architecture is the definition of a descriptive framework by Zachman (1987). He claims that as size and complexity of information systems increases, architectures are essential as logical constructs for defining and controlling the interfaces and the integration of all system components. In order to describe a complex system, a set of architectural representations is needed rather than a single architecture.

The first axis of Zachman’s framework consists of the architectural representations, used by an involved actor, i.e. the owner, the designer, or the builder. The owner has in mind a product that will serve some purpose. The architect transcribes this perception of a product into the owner’s perspective. Next, the architect translates this representation into a physical product, the designer’s perspective. The builder then applies the constraints of the laws of nature and available technology to make the product producible, which is the builder’s perspective. Each of these representations has a different nature from the others; they do not just increase the level of detail compared to a previous representation.

The framework’s second axis consists of the different types of descriptions oriented to different aspects of the object being described. Zachman points out that each of the mentioned representations can be seen from three different perspectives, depending on what has to be described. In the case of information systems, these perspectives are:

- the data to be processed,
- the processes to be controlled, and
- the networks via which the processes communicate.

Combining the two axes — a set of architectural representations as perceived by different actors, and the different perspectives upon these representations — results in Zachman's framework for information system architecture as shown in Table 2-I.

Table 2-I Framework for information systems architecture

	Data description	Process description	Network description
Scope description (ballpark view)	List of entities important to the business	List of processes the business performs	List of locations in which the business operates
Model of the business (owner's view)	E.g. entity/relationship diagram	E.g. functional flow diagram	E.g. logistic network
Model of the information system (designer's view)	E.g. data model	E.g. data flow diagram	E.g. distributed systems architecture
Technology model (builder's view)	E.g. data design	E.g. structure chart	E.g. system architecture
Detailed description (out-of-context view)	E.g. data base description	E.g. program	E.g. network architecture
Actual system	Data	Function	Communications

Source: Zachman (1987)

For Zachman, every representation short of being the final physical product is an architecture. However, he considers the out-of-context representations as less interesting "architecturally", since they do not depict the final product in total and are more oriented to the actual implementation activities.

Van Waes (1991) developed a framework for architectures in information management. In broad lines, her framework follows the same ideas as Zachman's. It combines three views, namely the function (process-oriented), object (data-oriented), and communication view, and four architectures, namely the business, information, systems, and technical architecture. The four architectures are defined for four actors, respectively the owner, user, designer, and implementer.

The most valuable contribution to architectural thinking by Zachman and Van Waes is the axes of their frameworks. They recognised that each actor in the development process needs his own (set of) representation(s). The difference between the various representations is their nature rather than the level of detail. Furthermore, they noticed that each actor might have three perspectives (or views) to look upon systems; in other words, they distinguished three possible viewpoints to describe a system. This thesis adopts the requirement that various representations are needed that differ in nature, and that various viewpoints are possible to look upon these representations. The representations and viewpoints are elaborated upon in Chapter 3, where the architecting concepts 'domains' and 'views' are introduced.

2.3.4 Software engineering

Researchers and engineers in software engineering have adopted the term 'architecture' as well. Perhaps of all disciplines, software engineering is the one where designers are most conscious about the importance of sound architectures. Nevertheless, there is no consensus about the subject; almost every paper in the field starts by saying that no standard, universally-accepted definition of the term ‘architecture’ is agreed upon. Although a standard definition is not available, there is no shortage of definitions.

Perry and Wolf (1992) consider a software architecture as a set of architectural elements that have a particular form. Similar to Zachman and Van Waes, they distinguish three different classes of architectural elements: processing, data, and connecting elements. Perry and Wolf consider an architecture as a necessary framework in which requirements are satisfied and which serves as a basis for the design.

Garlan *et al.* (1995) state that a system’s architectural design is concerned with describing its decomposition into computational elements and their interactions. Design tasks at this level include organising the system as a composition of components; developing global control structures; selecting protocols for communication, synchronisation, and data access; assigning functionality to design elements; physically distributing the components; scaling the system and estimating performance; defining the expected evolutionary paths; and selecting among design alternatives.

Soni *et al.* (1995) state that software architecture is concerned with capturing the structures of a system and the relationships among the elements both within and between structures. Software architectures describe how a system is decomposed into components, how these components are interconnected, and how they communicate and interact with each other. Based on a survey on the role of architecture in the design and development of large systems within Siemens, Soni *et al.* notice that different structures are used at different stages of the development process. Each structure describes the system from a different perspective.

Soni *et al.* argue that the four different architectures they distinguished are needed because of the growing complexity of software throughout history (see Figure 2-2). Initially, only the code architecture was required. The module and execution architecture became necessary

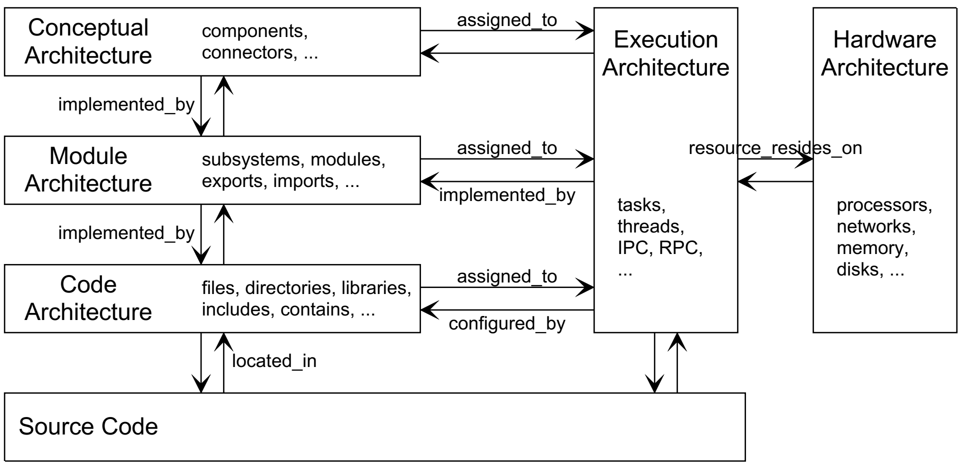


Figure 2-2 Relationships among software architectures

Source: Soni *et al.* (1995)

when systems became larger and distributed. Now, software engineers would like to use communicating objects and assemblies of reused components. Therefore, a high-level structure is described in the form of a conceptual architecture. On the other hand, Zachman and especially Van Waes reason that their various architectures are wanted as representation for each of the involved actors. Obviously, more actors became necessary to control the growing complexity of software.

Garlan and Perry (1995) found that the term 'architecture' is used in a number of ways in software engineering. Among the various uses are a) the architecture of a particular system, as in 'the architecture of this system consists of the following three components,' b) an architectural style, as in 'this system adopts a client-server architecture,' and c) the general study of architecture, as in 'the papers in that issue are about architecture.' Note that the three common uses show a remarkable resemblance with the three dictionary meanings as stated in Section 2.2, namely structure, style, and science. Garlan and Perry continue by stating that most uses of the term 'software architecture' focus on the first of these interpretations.

A discussion group at Carnegie Mellon University's Software Engineering Institute developed a typical definition: the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. The same institute has collected various definitions for software architecture (SEI, 1997). They notice that these definitions and views do not preclude each other, nor do they represent a fundamental conflict about what software architecture is. Instead, they represent a spectrum in the software architecture community about the emphasis that should be placed on architecture — its constituent parts, the whole entity, the way it behaves once built, or the process of building it. Taken together, they reflect the various aspects of software architecture.

The software engineering community realised that software architecture is not only about structures (components and interfaces), but also about system behaviour (interaction between components, protocols). Furthermore, this community introduced an architectural design phase in the system life cycle, in which requirements should be satisfied and which should serve as a basis for detailed design activities. This thesis adopts the viewpoint that a system architecture is more than the system's structure, and that system behaviour is part of architectural specifications as well.

2.3.5 Computer Integrated Manufacturing

In addition to the previous disciplines, the term 'architecture' is nowadays widely used in Computer Integrated Manufacturing. In the next section, a description and definition is given which is used throughout the remainder of this thesis. In this section, some well known views on CIM architectures are given.

Probably the first to adopt the term 'architecture' in the design of control systems for automated production systems were the designers of the Automated Manufacturing Research Facility (AMRF) of the US National Institute of Standards and Technology, formerly called the National Bureau of Standards (Simpson *et al.*, 1982; Albus *et al.*, 1981). In their description of the AMRF control architecture, they outline three hierarchies, namely an organisational, computational, and behavioural hierarchy, that together form the control system hierarchy.

Bakker (1989), referring to an internal Philips report (Philips CFT, 1987), states that the architectural description of a system describes the functionality of each of the tasks that have to be executed. Allowable inputs and outputs are defined and the relations between the tasks are specified. The architecture describes what the components do and describes the relations that exist between the components.

Jones *et al.* (1989) argue that the basis for achieving the integration of manufacturing processes with engineering and production functions into a CIM system lies in the design and implementation of a system architecture for CIM. They believe that a system architecture must include three separate but related architectures for production management, data management, and communications. This partition of concerns makes it possible, to a large extent, to optimise the design of each structure independent of the design of others.

The CIMCON ’90 conference on the design and implementation of global CIM architectures (Jones, 1990) shows that the distinct CIM architecture developments throughout the world have been abundant, but each of them is based on its own architectural definition. No consensus has been achieved on what a CIM architecture is, and there is no way of justifying an architecture proposal. Bernus *et al.* (1996) give a summary of the state-of-the-art on CIM architecture as presented during CIMCON ’90. Opinions on CIM architectures range from ‘a set of models which describe the elements and the relationships between these elements of the whole CIM system’ (Chen *et al.*, 1990a) to a ‘style of constructing or designing the information processing within a manufacturing enterprise’ (Spur *et al.*, 1990).

Böhms (1991) defines a CIM architecture as a CIM model showing manufacturing activities, material and information flow objects, information and production technology components and their interrelations on a global implementation-independent level within a specific manufacturing system. A CIM architecture does not show all the details necessary for the implementation of the architecture in reality.

Frequently, the relation between the architecture and system components is emphasised. For example, Timmermans (1993a) defines an architecture as a description of system components and their interfaces. According to Dilts *et al.* (1991), a (control) architecture makes a (control) system from (control) components.

A distinction is seen between ‘architecture’ in the ‘art or science’ meaning and ‘architecture’ in the ‘model of a structure’ meaning. During the 1990’s, the CIM community gradually accepted the dual meaning of the word. Williams *et al.* (1994b) shed some light on this distinction. They define two types of architecture which deal with the integration of manufacturing entities or enterprises. These are:

- The structural arrangement (design) of a physical system such as the computer control system part of an overall enterprise integration system (Williams' type 1). Examples are the NBS or AMRF model (Jones and McLean, 1986), the reference model for manufacturing planning and control (Biemans and Vissers, 1989; Biemans, 1989), and the Factory Automation Model (Graefe and Thomson, 1989).
- The structural arrangement (organisation) of the development and implementation of a project or program such as a manufacturing or enterprise integration or other enterprise development program (Williams' type 2). Examples are CIMOSA — Open System Architecture for Computer Integrated Manufacturing (AMICE, 1989; AMICE, 1993a), the GRAI Integrated Methodology (Doumeingts *et al.*, 1987; Chen *et al.*, 1990a), and the Purdue Enterprise Reference Architecture (Williams, 1994).

The distinction by Williams *et al.* between enterprise reference architectures (type 2) and 'the rest' (type 1) was useful in a time of emerging reference architectures. Nevertheless, Williams' type 1 architectures are badly defined. In the next section, a distinction is made between architectures, reference models, and reference architectures. The examples given above for Williams' type 1 architectures should be considered as reference models rather than architectures.

This thesis does not look upon an architecture as a *description* or *model* of a structure (or anything else). In the next section, it is explained that descriptions or models are needed to represent an architecture, and that the term 'architecture' does not refer to a model but to the entity being modelled.

Since around 1990, architectural thinking has been maturing. The introduction of enterprise reference architectures made the focus in architecting shift from the management of *complexity* to the pursuit of *future flexibility*. Furthermore, an important premise of the enterprise reference architectures is that CIM systems are not static but have to be adapted continuously. This implies that the specification of the system architecture is no longer a one-time activity, but preserving the integrity of the system as part of architecting is a continuous process.

2.4 Architecture in this thesis

2.4.1 Definition

After reviewing various opinions on architecture in the CIM domain and other fields, it is appropriate to describe what is meant in this thesis by 'architecture'. In Section 2.2, the results of a dictionary survey showing the three main meanings of the term 'architecture' are presented. In this section, the definition of the term as applied in the remainder of this dissertation is given. The aim is not to give a final definition of 'architecture' — an unachievable goal —, but to set a frame for the remainder of the dissertation, and to raise some intuitive notion for architectures.

This thesis reserves the term '*architecture*' for *the manner in which the components of a specific system are organised and integrated*. As such, a system architecture determines the *nature* or *essence* of a system. An architecture of a specific system or product is the result of a design process. It is laid down in a set of specifications that describe the system components, their interfaces, and their behaviour. Architectural specifications do not show all the details necessary for the implementation of the conceived system in reality. The abstraction level of the specifications is higher than that of the specifications that are the result of later design activities.

A *reference model* is a *generic manner to organise and integrate system components*. It serves as a point of departure for the design of a large number of systems in a specific application area. Recall that the second dictionary definition refers to architecture as a style and/or method of design and construction. The style suggests features that are characteristic of several systems. For buildings, these features are usually common throughout a certain time period or place. For automated manufacturing control systems, a similar reasoning holds; for instance, most systems developed in the 1980's follow the hierarchical features as originally outlined in the NBS model. Styles range from abstract architectural patterns and idioms (such as 'client-server' or 'layered' organisations) to concrete reference models. Unfortunately, the term 'reference *model*' seems to refer to the style's representation, rather than to the style itself. Nevertheless, this dissertation uses the term 'reference model' for a certain architectural style.

Finally, the term '*reference architecture*' is used for a *framework in which system related concepts are organised*. As such, a reference architecture can be used as a tool with which one can analyse and design systems. An *enterprise reference architecture* or a *reference architecture for enterprise integration* is a *framework in which enterprise related concepts are organised*. A reference architecture for enterprise integration should point toward purposeful organisation of enterprise concepts (Nell, 1996); it should identify and structure concepts for enterprise integration, most notably life cycle activities and enterprise modelling concepts such as views. To some extent, this interpretation resembles the first meaning from the dictionaries in Section 2.2, the art and science explanation, since both try to gather and apply knowledge. Architecture (as a discipline) should deliver reference architectures (as paradigms expressing thoughts about state-of-the-art architecting).

The relation between reference models and architectures is one of instantiation. A reference model specifies the general structure of the system and shows which tasks have to be executed. In addition, the relation between the tasks is shown (Bakker, 1989). A designer might use a certain reference model to specify an architecture. The architectural description of the system specifies the specific functionality of each of the reference model's generic tasks. Allowable inputs and outputs are defined and the relations between the components are specified. A reference model can lead to a number of different architectures. For instance, many network architectures of current data communication systems are derived from the well-known ISO-OSI reference model. The OSI model itself is not a network architecture; the exact services and protocols applicable for each layer are defined but not specified (Tanenbaum, 1988).

The relation between reference architectures and architectures is more complex. Part of the outcome of a development process supported by a reference architecture are the architectural specifications. For instance, a reference architecture such as CIMOSA contains a modelling framework and modelling methodology, with which designers can specify architectures. However, reference architectures aim to provide more support in a system development trajectory than just during architectural design activities; they attempt to cover the full development cycle. In Chapter 5, the role of reference architectures in a system development process is illustrated.

Reference architectures might be accompanied by reference models. Reference architectures offer the frameworks with which reference models can be made. The provision of reference models is one way in which reference architectures support designers.

2.4.2 Architectures, systems, models, structures, and infrastructures

Architectures are frequently confused with other, related concepts. This subsection tries to lighten the gloom a bit by confronting architectures with these related concepts.

Architectures, systems, and models

The relationships between architectures, systems, and models is as follows. The architecture is merely one of the properties of a specific system. However, it is not a concrete, physical property; it is a conceptual one. It can be seen as an abstract ‘view’ of a real (concrete) system. Just like other properties of a system, the architecture can be modelled; it is visualised by means of a symbolic model, which may be called a representational model (Van Waes, 1991). Figure 2-3 shows the relationships between systems, architectures, and models. Architecting comprises an abstraction from the real system (the CIM system, its control system, the physical means used to realise the control system, and so on) to yield an abstract system (the abstract view of the real system a designer has in mind), and formalising this abstract view to a symbolic system (the symbolic model, for instance consisting of a diagram with text).

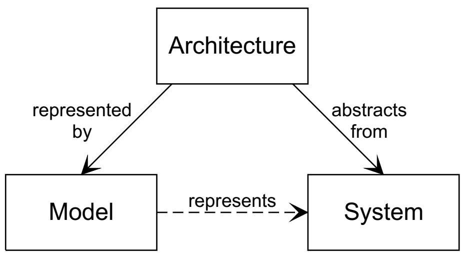


Figure 2-3 Relationships between system, architecture, and model

Adapted from Van Waes (1991)

The distinction between an architecture as an abstract view and a symbolic model as a representation of the architecture is not commonly made; models are called architectures.

After all, whereas the architecture is an abstract product of architecting, the models are concrete results. It is only natural to mistake a concrete product for an abstract one.*

Architectures and structures

A system architecture encompasses more than a system's structure. In many papers, the structure of a system is put on a par with the system architecture. However, an architecture is not identical to a structure. The specification of system components and their relations is only part of architecting, though one of the most important parts.

Besides the static structure of a system, architecting also deals with the dynamic processes of the system. The architect specifies how a particular system behaves in operation, e.g. a shop floor control architect specifies the protocols by means of which controllers coordinate their actions. For instance, the control structure as part of an architecture based on holonic control concepts enables both hierarchical and heterarchical behaviour, depending on the situation. Note that this interpretation of the dynamic aspect of architectures is a difference with the classical building science explanation. Although a building does enable some sort of dynamics, it is static in nature. The behavioural aspect is more relevant for some disciplines (shop floor control, software engineering) than for others (building science, product design).

Architectures and infrastructures

The relation between an architecture and an (information technology) infrastructure is perhaps best explained by means of an example. The architecture of a shop floor control system is the manner in which the controllers are organised and integrated. These controllers might need additional functionality to function properly, such as computing and communication functionality offered by computer and network technology. The components that provide necessary additional technology are considered as parts of an infrastructure. Clearly, an infrastructure has an architecture as well. This dissertation adopts the interpretation of Timmermans (1993a), who defines infrastructure as hardware and software (computers, networks, operating systems, applications, and so on) that is shared between different autonomous units. Other authors, such as Renkema (1996), take a broader perspective on infrastructures.

Van den Berg (1998) claims that 'an infrastructure is in the first place a provision (if positively interpreted) and a restriction (if interpreted negatively)'. Indeed, both architectures and infrastructures establish the possibilities and limitations of changing systems in the future. However, both have a different effect on the future flexibility of systems. In the example above, the system architecture determines the future flexibility of the shop floor control system by the structure and protocols that integrate the various controllers. The infrastructure determines the future flexibility of the shop floor control system by its ability to offer the enabling functionality when the system evolves. An architecture sets the flexibility from the inside of a system, an infrastructure from the outside.

* It could be argued that the term 'reference *model*' is incorrect. A better option might be to use the words 'reference architecture' and 'meta-architecture' instead of 'reference model' and 'reference architecture' respectively. However, this dissertation borrows the terms from literature.

2.4.3 Objectives and roles

Architectures and architectural models serve several objectives and/or roles along different points in a system life cycle. They are as follows.

To manage complexity

An architectural model allows one to present the essence of a complex system in a (simple) model. An architectural model supports the ability to comprehend complex systems; it presents them at a level of abstraction at which a system's high-level design can be understood. It supports the analysis of relationships as an aid to understand complexities in a design environment. In particular, an architecture is needed in complex, dynamic environments (Van Waes, 1991). Zachman states that the increased scope of design and levels of complexity of system implementations are forcing the use of architectural models for defining and controlling the interfaces and the integration of the system components (Zachman, 1987). Moreover, at its best, architectural descriptions expose the high-level constraints on system design, as well as the rationale for making specific architectural choices (Garlan and Perry, 1995).

Architectural models abstract away from details instead of from the essential complexity. Brooks claims that 'the complexity of software is an essential property, not an accidental one' (Brooks, 1995; p. 183). Descriptions of a software entity that abstract away its complexity often abstract away its essence.

To serve as a set of specifications

An architecture may be seen as a result of the design process. It is laid down in specifications, which are derived from the requirements, and from which the desired system can be built. Specifying an architecture is concerned with the specification of components, their interactions, and the constraints on these entities and their interactions. These unambiguous specifications define the scope of future development activities, and serve as a basis for further design and implementation activities. Modular components can be developed independently, provided that their interfaces are well-defined. By defining modular components, the total system design is split into a number of smaller subdesigns, making the complexity of the total system more manageable. As such, the architecture enables a top-down development approach, in which each subsystem can be implemented from the bottom up and independent of other subsystems with assurance that the various subsystems can be integrated with each other. Finally, the distinction in components may serve as the managerial basis for cost estimation and project management.

Means of communication

Furthermore, an architectural model may play the role of a means of communication during a system (re-)design process. The architect can use it to visualise various aspects of the system to be designed, thus providing the various parties concerned with a basis for discussion and decision-making. By producing order in chaos, architectural models help each party to clarify its perception of the problem. Visualisation and explanation of the relevant aspects of the problem area, and the possible relationships between them, supports the various actors to focus their attention on the essential elements, thus providing a basis for discussion of the problems.

To indicate the most vital system elements

Furthermore, the architecture determines the nature and quality of a system. As such, an architectural model indicates the invariant or most vital system elements, that must be treated carefully during system re-design. Systems evolve and are adapted to new uses, just as buildings change over time and are adapted to new uses. One frequently accompanying property of evolution is an increasing brittleness of the system, caused by violations of the architecture – for example, removing load-bearing walls often ends with disastrous results. Violations of the architecture frequently lead to an increase in problems in the system and contribute to an increasing resistance to change, or at least to changing gracefully (Perry and Wolf, 1992). By explicitly indicating the 'load-bearing walls' of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications. The architecture should mark the system's invariant load-bearing walls, mostly interfaces and constraints.

Means to reduce the impact of changes

Another role of an architecture involves its contribution to the effective re-design of a system. The architecture should reduce the impact of changes to the lower component levels, and to as few components as possible. Both for shop floor control systems and for products, it is advantageous to use as many parts of the existing system or product design as possible. In a re-engineering trajectory, an architectural model of the system allows one to pinpoint and discuss the areas requiring major change, and to integrate the new specifications into the existing model. Furthermore, architectural change is not so much determined by the system components, as well by the interfaces between these components; the ease with which components can be modified, replaced, or with which the system can be extended by new components is dependent on the extent to which the interfaces of the new components match those of the old ones. In addition, the use of standards in the architectural specifications positively affects system modifiability and extendibility.

Means to gain strategic benefits

Finally, a (product) architecture may have certain strategic importance for a company. The development of a new product brings together a wide range of technologies. Only a few of these technologies contribute to ultimate competitive advantage. Successful companies do not compete on (and even give away) the enabling technologies on which their core utility is based. By the architectural design of functions that can be filled in by cheap, standard components, companies profit from the strong competition in the markets for these components, and are free to focus on their true sources of competitive value. In addition, a company might extend the value of its product by publishing the product's interfaces to the outside world. Other enterprises might use this product as an indispensable part for their own products. By controlling the interfaces of an enabling product, a company might direct itself in a profitable monopoly position (Rappaport and Halevi, 1991; Morris and Ferguson, 1993).

2.5 Architecting

Rechtin (1991) gives a good impression of the process of architecting. This section is adapted from his standard work on systems architecting.

To understand the role of an architect, it is beneficial to compare architecting with engineering. Distinguishing between architecting and engineering is important if the architecting function is to be effective. Two distinctions have to be made: function versus form, and complexity versus specificity.

Engineering is form-based and system architecting is function-based. Many disciplines come together in the practice of design, resulting in technical, judgmental, and professional conflicts. In the past, the resolution of these conflicts became a task for the architect; he became an integrator, making all pieces fit. Good architectures were generally derived from compatible pieces, i.e. they were system engineered. Later, architects began to appreciate that (still) better architectures might be based on complete submission of the individual parts to the purpose of the whole. Architectures began to be designed from the top down, driven by the function, instead of the form, of the system.

An architect reduces complexity, uncertainty, and ambiguity to workable concepts, whereas the engineer makes feasible concepts work. Complex systems contain dozens if not hundreds of possible variables and hundreds of interactions among these variables. The architect's problem is to reduce this complexity to a manageable degree, specifically to the point where the techniques of engineering analysis can be brought to bear. The 'nice-to-have' functions have to be separated from the essential ones; the load-bearing walls have to be distinguished from the decorations. The result has to be one or more well-specified possible systems. At this point, efficient engineering can begin.

An essential part of architecting with respect to complexity is *structuring*, which can mean bringing form to function, bringing order out of apparent chaos, or converting the partially formed ideas of a client into a workable conceptual model. The key techniques are balancing the needs, fitting the interfaces, and compromising among the extremes. The challenge is the control, if not the reduction, of complexity and uncertainty (Rechtin, 1991; 1992). Figure 2-4 displays Rechtin's view on architecting as a compromise of extremes. This is represented by pairs of competing factors pulling in opposite directions, which are held together by fit, balance, and compromise. Similarly, Bemelmans argues that 'the art and science of development is to find a balance between functional and performance requirements on the one hand, and acceptable costs, effort, and duration on the other hand' (Bemelmans, 1990; p. 9)

Another way of distinguishing architecting from engineering is by the tasks typically performed. *Architecting* is working *for* a client and *with* a builder, helping determine the preferred architecture, relative requirement priorities, and acceptable performance, cost, and schedule. Toward the end of the project, architecting is also certifying completion and satisfactory operation of the system. The systems architect is not just concerned with the system architecture, but also with the process by which the system will be built. The architect

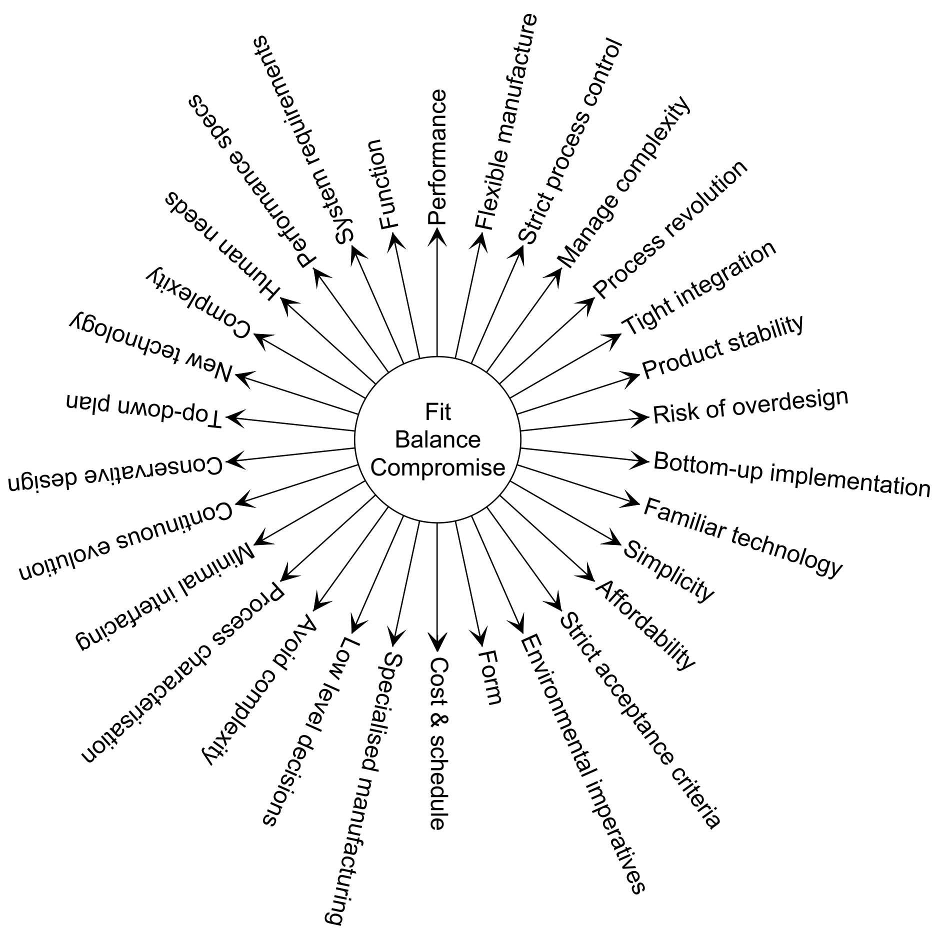


Figure 2-4 The tensions in systems architecting

Based on Rechtin (1991)

tends to concentrate on concepts, synthesis, top-level specifications, technical as well as non-technical interfaces, and mission success.

The architect's responsibility goes beyond the conceptual integrity of the systems as seen by the user, to the conceptual integrity of the system as seen by the builder and other stakeholders. The architect is responsible for both *what* the system does as well as *how* the system does it. But that responsibility extends, on both counts, only as far as is needed to develop a satisfactory and feasible system concept. After all, the sum of both is nearly the whole system, and the architect's role must be limited if an individual or small team is to carry it out. (Rechtin and Maier, 1997).

Engineering is working *with* an architect and *for* a builder, applying the best engineering practices to assure compliance at the system level with the designated architecture and with applicable specifications, standards, and contracts. Toward the end of the project, engineering is certifying such compliance. The engineer tends to concentrate on defined subsystem interfaces, analysis, and performance to specification. Because the success of a development project depends both on a realisable architecture and on successful implementation of it, the architect and the engineer necessarily share responsibility for each other's success.

Figure 2-5 shows where and to what degree architecting plays a role in a system development project. The widths of the lines indicate the strength of the connection. Figure 2-5 does not show the relation of an architect during adaptation phases.

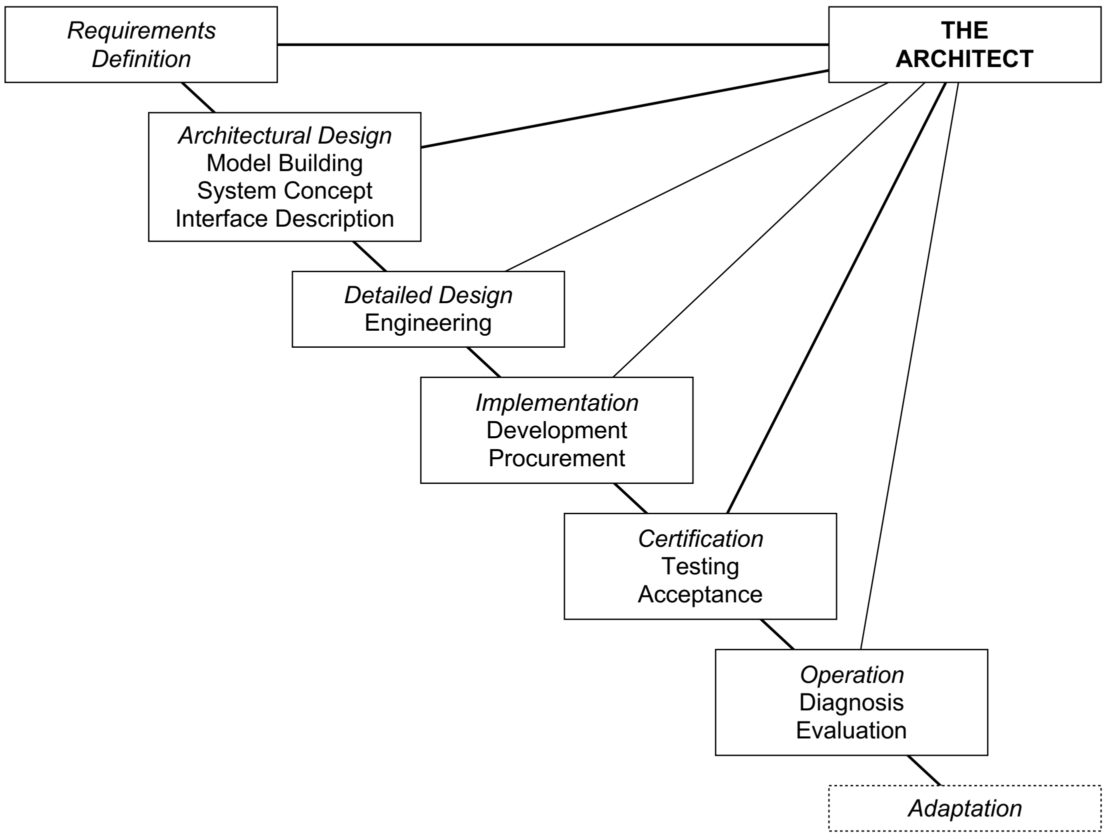


Figure 2-5 The role of the architect

Adapted from Rechtin (1991)

2.6 Structure, behaviour, and non-functional aspects

Besides the structure of a system, an architectural model should specify the system's behaviour and various non-functional aspects. In practice, emphasis is put on the structure of a system. Rechtin (1991), for example, says that 'the essence of architecting is structuring'. For information technology (IT) architectures, however, Hammer notices that 'present IT architectures concentrate too much on the structure of the system and do not adequately deal with the dynamic aspects. The emphasis is on modularisation of the system and the interfaces but not on its behaviour. Even if the various interaction channels are modelled, there are no restrictions of the actual interaction patterns that can occur at runtime' (Hammer, 1997; p. 304).

As a counter-example, in electrical engineering it is common practice to specify the static relationships and dynamic behaviour of (real-time) systems. The structure of a system is outlined by a hierarchy of transformation diagrams. The commonly used Ward/Mellor method prescribes the use of data flows, functions (i.e. data transformations) and stores to construct transformation diagrams. A dynamic interpretation can be given to the transformation diagrams by modelling the communication between parallel processes. Ward and Mellor (1985) make a distinction between three perspectives:

- *the static view*
  In the static view, the focus is on the structure of the data and control transformations, the data and control flows, and the stores. The static view gives an impression of the relations between the various parts of the model.
- *the dynamic view*
  The dynamic view adds the time element to the model. Examples are continuous and discrete data flows, with which time aspects are modelled.
- *the event view*
  In the event view, the occurrence of certain events is added to the model. Events do not carry any information and are modelled by control flows. They trigger processes and thereby indicate the synchronisation moments.

The behaviour of a system can be defined in terms of concurrent processes, including the restrictions on the possible execution paths. The latter is usually formalised in a communication protocol. Concurrent processes are executed at the same time, and have mutual interactions, for instance by means of data exchange. In order to collaborate, processes have to synchronise their mutual actions, or provisions have to be made for asynchronous communication.

During architecting, one has to be aware of the problems that connecting parallel processes and synchronisation might create, such as deadlock, individual starvation, and blocking. These problems might need a considerable effort to anticipate and prevent. A static structure which appears quite simple, might exhibit a rather complex behaviour when it is put into operation. When a subsystem is put in a larger whole, it has to cooperate with other subsystems in a transparent way. Interdependencies are created between subsystems, since the subsystems expect a certain behaviour from each other. These interdependencies stem from the application domain and the architectural choices made.

In order to verify system behaviour, some tools have become available with which the behaviour of the specified system can be simulated. Simulation tools verify whether system elements will work together as a system. Well-known classes of these tools are based on concurrent programming formalisms (see e.g. (Rooda, 1996)) and Petri nets (see e.g. (Van Hee, 1993; Van der Aalst, 1994)). In software engineering, various tools have been developed to explicitly model system behaviour according to the diagrams proposed by the Unified Modelling Language, such as sequence diagrams, collaboration diagrams, and state diagrams (Rational, 1997). With these tools, one is able to detect system consistency errors such as deadlocks before the actual system is constructed.

Aside from structure and behaviour, various non-functional aspects should be specified. Again, Hammer observes that ‘there is not enough attention for the non-functional aspects of an IT architecture like performance/timeliness, dependability (reliability, availability, safety, security, robustness) and reusability. Since we also miss adequate development methodologies that support the design of these system dimensions during all development phases, these considerations enter the construction of an IT system often only during the implementation phase' (Hammer, 1997; p. 304). An architect has to explicitly consider implementation, requirements, and long-term client needs in parallel. A requirements-centred approach assumes that a complete capture of documentable requirements can be transformed into a satisfactory design. However, requirements-modelling methods generally fail to capture performance requirements and ill-structured requirements such as modifiability, flexibility, and availability. Even where these non-functional requirements are captured, they cannot be transformed into an implementation in an even semi-automated way (Rechtin and Maier, 1997).

This dissertation recognises Hammer's observations that dynamic and non-functional aspects do not get the attention they deserve. However, it is not the intention of this thesis to explicitly remedy this shortcoming.

2.7 Summary

The system architecture is the manner in which the components of a specific system are organised and integrated. Related terms such as reference model and reference architecture are explained. The roles of architectures and architectural specifications involve the control of complexity during development projects, and the provision of future system flexibility.

Architecting is the process which initially defines the system architecture, and which preserves system integrity, i.e. compliance with the architecture, while the system is developed or adapted. The architect is responsible for what the system does as well as how the system does it. Rather than a one-time activity, architecting is often an ongoing process, since many systems evolve after initial delivery.

3. Architecting Concepts

3.1 Introduction

The objective of this chapter is to provide the architecting concepts that play a key role in designing system architectures. By means of these concepts, total system complexity is reduced or at least made more manageable. Architects use these concepts when they specify system architectures. Doing so, they aim for the guiding principles such as modularity in order to design flexible systems. These principles are presented in Chapter 4.

Both the architecting concepts and the principles are illustrated by means of the Gordian project. This action research project was carried out at Baan Company, and concerned the decoupling of warehousing functionality into a separate package. The Gordian project is introduced in the next section.

Section 3.3 presents the first architecting concept, namely domains. Development is characterised by several domains that contribute simultaneously to the creation of a system. A domain is defined as a product model together with all representations of this model. The functional, technology, and physical domain are distinguished. Within each domain, an architecture can be discerned. For architecting, the functional and technology architectures are arguably most important.

In Section 3.4, (decomposition) hierarchy is introduced as an architecting concept. Design problems can be decomposed in smaller parts that can be solved in series or in parallel. If divisions are chosen properly, the resulting separate subsystems are more manageable than the original system. By means of decomposition, hierarchical structures are formed. It is shown that the opposite technique of decomposition, namely composition, is applied in architecting as well.

Views are ‘windows’ through which some aspects of a system can be observed, and other, extraneous detail is suppressed. Views divide systems into aspect systems; hierarchy divides systems in subsystems. The set of views chosen to describe a system is variable and depends on the purposes of the user. This architecting concept is described in Section 3.5.

Domains, decomposition hierarchy, and views are the three most important architecting concepts in the context of this thesis. However, different authors recognise different concepts. Three other architecting concepts are shortly mentioned in Section 3.6, namely variants, status, and versions. It is argued that these concepts are of minor importance for shop floor control architecting.

3.2 The Gordian project

This chapter and the next chapter are illustrated by some of the results of an action research project, the Gordian project.* The project was named after the famous story in Greek mythology about Alexander the Great disentangling the Gordian knot. The project focused on controlled growth and evolution along predictable lines. This desirable but difficult characteristic stands as a challenge to many disciplines, such as information system development and manufacturing engineering.

The project took place at the Baan Company, which rapidly became one of the largest software developers in the Netherlands during the 1990's. The Baan Company is a supplier of a set of standard software packages for Enterprise Resource Planning (ERP). Previously, the set was called 'Triton'; in 1996, it was named after the company. The studied version, Baan IV, consisted of eight packages (amongst others Distribution, Manufacturing, Service, and Finance) and comprised more than 2.5 million lines of code.

During the fast growth of Triton, problems emerged concerning the integrations between its various packages. After its introduction in 1990, Triton had grown fast for several years. Pieces were built on top of each other, and sometimes relations were created in a rather haphazard way. As a consequence, customers came across more problems, application developers solved more bugs, at the same time introducing more and more problems (bad fixes). Whenever functionality was added or modified in a particular package, this change might create problems in other packages. The fact that changes propagated to other packages indicated that the interfaces between packages could have been defined better. Furthermore, whenever a new piece of functionality was introduced into Triton, congestions occurred at subsequently development, documentation, and testing. Although exceptions had occurred, all packages had to be ready and released at the same moment; spreading of work load was not possible. Integrations were needed, but it was desirable to decouple the packages, thereby decreasing the necessity to release packages at the same moment. In short, due to tight integrations between packages, new functionality was hard to add, and application development became a process that was difficult to control.

The management of Baan's development organisation was aware of the problems described above, and decided that packages had to be decoupled. The Gordian project was started as a test case. It focused on the decoupling of warehousing functionality in the Baan packages. Warehousing functionality expanded throughout the development of the Baan applications. This growth has been realised by many developers who introduce and improve functionality in a number of ways and places. The evolution has lead to redundancies and inconsistencies. In Baan IV, every package contains its own warehousing functionality, such as generate/update stock mutations, print documents, register material issues/receipts, and so on. To overcome these problems, Baan Development decided to restructure warehousing functionality in Baan V.

* Parts of the results of the Gordian project have been published in (Van den Berg and Zwegers, 1996; Van den Berg and Zwegers, 1997; Van den Berg, 1998).

The objective of the face-lift was to put the dispersed warehousing functionality into a separate package called Warehousing. Furthermore, reading and updating of warehousing data should be done by standard Warehousing software libraries. Material issue, receipt, confirmation and approval should be centralised within Baan Warehousing. The same applied to printing of documents that were previously printed in other packages, such as Goods Received Notes, Storage Lists, and Bills of Lading.

This section introduces the Gordian project, which is used as an example to illustrate architecting concepts and principles in this chapter and the next. The following section presents the first architecting concept, namely domains.

3.3 Domains

Domains are probably the most effective concept to control the complexity faced by architecting. The architecting concept of domains is best explained by Erens' dissertation on the development of product families (Erens, 1996). The applicability of the ideas developed by Erens goes beyond the area of developing product families; Erens' theories could be treated as design theory in general. In this thesis, Erens' functional, technology, and physical domains are adopted as three levels of abstraction. Note that the functional domain is sometimes called the logical, or conceptual domain, and that the technology domain is called the technical domain by some authors. Before Erens' ideas are outlined, earlier efforts by Brandts (1993) and Van den Kroonenberg (1975a, b) are presented.

Previous work on domains was carried out by Brandts (1993) in his thesis on the design of industrial systems. Brandts presents a design cube with three dimensions: attributes, (sub)systems, and design abstraction (see Figure 3-1). The first two dimensions are borrowed from system theory, which is Brandts' starting point for the formulation of his ideas on the design of industrial systems. Attributes are the means to distinguish between elements, i.e. attributes are the qualities, or characteristics of elements. Subsystems and aspect systems are defined as follows:

> 'A *subsystem* of a system S is a subset of E (the set of elements of S) with all the attributes of the elements in question. An *aspect system* of a system S is the set E with only a subset of the original attributes' (Van Aken, 1978; p. 29).

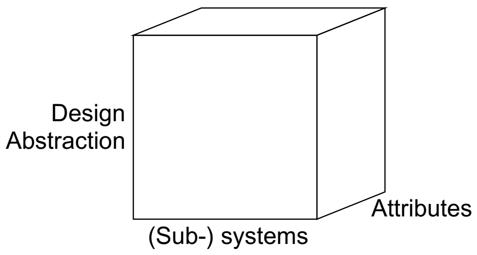


Figure 3-1 Brandts' design cube

Source: Brandts (1993)

The third axis represents the level of design abstraction of the knowledge describing the object during design. Brandts notices that the description of an object changes during the design process; it is rough and abstract in the beginning and complete at the end. The object design is transformed from abstract to concrete, and design can be seen as the process of moving from the upper plane in the design cube to the lower plane (Brandts, 1993).

In his dissertation, Brandts proposes a division of the design process for industrial systems, which is derived from Van den Kroonenberg's work on product design (Van den Kroonenberg, 1975a, b). This phasing distinguishes a *function-definition* phase, a *working-principle-definition* phase, and a *form-definition* phase. Brandts explains the three phases for product design:

> 'In the first phase, the (abstract) functions of the product are determined. Next, working principles to execute the functions designed in the first phase are designed. By doing so, the structure of the product is determined. The eventual form is determined in the third phase. There, details are designed and the object design is concretised' (Brandts, 1993; p. 84).

Similar phases as proposed for product design are used for manufacturing system design, control system design, and the design of the financial system and its control system. Figure 3-2 shows the relation between the design phases of the various industrial (sub-) systems. In the first phase, objective definition, the objective of the industrial system is defined. Brandts explains the design process of industrial systems:

> 'The industrial system design process will go roughly from top to bottom and from left to right in Figure 3-2. The top to bottom direction follows logically from the fact that the design process goes from abstract to concrete. The left to right direction follows from the fact that a subsystem A at the right of a subsystem B cannot be designed

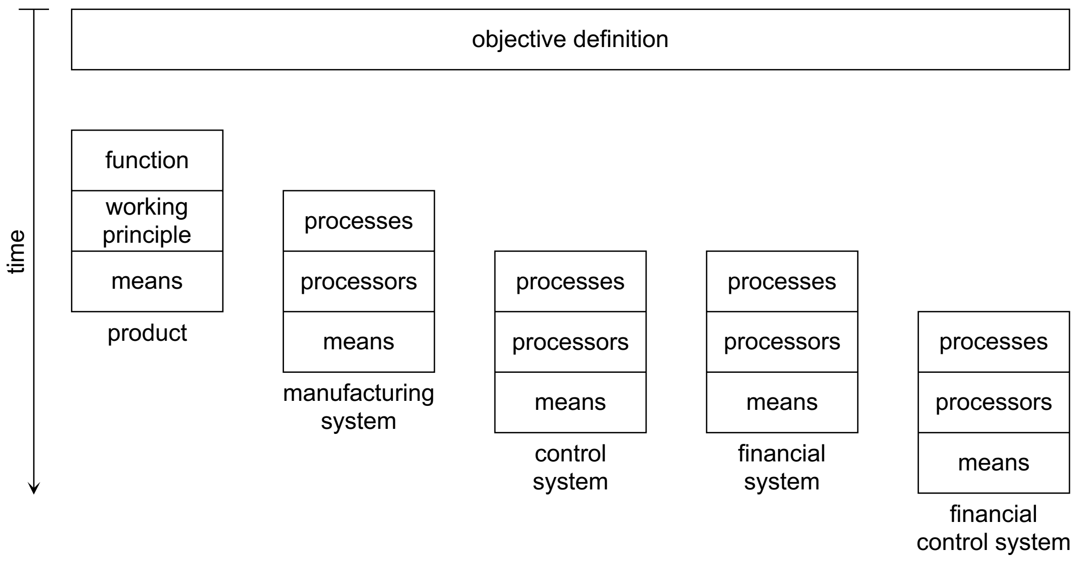


Figure 3-2 The relation between the design phases of industrial subsystems

Source: Brandts (1993)

> without information from B. A manufacturing system, for instance, cannot be designed without knowledge of the products to be produced' (Brandts, 1993; p. 97).

Almost similar to Brandts' three design phases, Erens (1996) states that development is characterised by several domains that contribute simultaneously to the creation of a system. A domain is defined as a product model together with all representations of this model. Figure 3-3 shows the three domains that suffice to capture product information in the development process:

- functional domain,
- technology domain,
- physical domain.

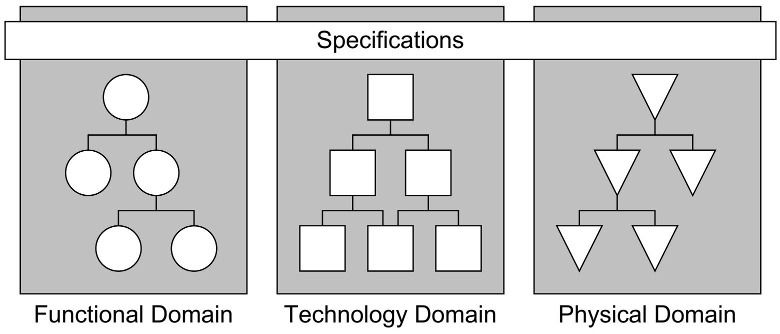


Figure 3-3 Domains and product models

Source: Erens (1996)

The *specifications* (in the top of Figure 3-3) are the starting point for the development of a system, or for changing part of the system. The initial specifications can not be attributed to one particular domain as they provide an often informal description of the required function, the technological constraints, and the physical constraints. An example of a frequently occurring constraint is the requirement to (re)use parts of the existing infrastructure.

In the *functional domain*, models are made that capture the function of a system. The models should result in a complete and unambiguous description of the system. Interdependencies among functions should be clearly expressed. These dependencies concern interface definitions that deal with interactions between functions (Erens, 1996).

In the *technology domain*, the technological solution of the design problem is specified. It consists of a set of modules or solution principles, which together cover the required function. Functions are mapped onto modules, thereby taking into account constraints on the solution. If functions are interconnected to express functional dependencies, then modules have interfaces for technological dependencies (Erens, 1996).

In the *physical domain*, the system is described in terms of physical assemblies. It is a description of the physical implementation. This domain might differ from the technology domain because of physical reasons, for example resulting from a specific assembly process.

Such decisions are often the responsibility of engineering. Nevertheless, engineers should be involved in both the functional specification and the technological realisation to guarantee the creation of a suitable physical model (Erens, 1996). For the same reason, architects – who primarily focus on the functional and technological domains – should have enough knowledge of the physical domain to prevent them from designing unrealisable functional and technological models. Note that the word 'physical' should not be taken too literally. In software engineering, for example, there are no such things as 'physical assemblies'. There, the physical domain constitutes of descriptions of the source code.

The distinction of three domains is made, since each domain represents a typical design problem. Each function can be realised in different technological solutions. The user is generally interested in the function, not in the technological solution. An engineer is in general interested most in the technological solution, just like a constructor focuses on the physical realisation. Therefore, the domains must be separated in the design process, so that the designer can discuss the function, without bothering about the solution, or so that the solution can be changed without affecting the function.

Furthermore, the distinction of the three domains avoids a classical error: confusing implementation dependence and level of detail. It is an infamous misunderstanding to relate a low level in the design process to implementation details. This would imply that functions could be decomposed in technology modules and that modules could be decomposed in physical assemblies. It is possible, however, to specify in detail what a system should do (the 'what') without using implementation details (the 'how'). In other words, details might relate to the functionality of a system, its technology, or its physical realisation. In the functional domain, it is not a matter of less detail but a matter of implementation independence and indeed abstraction from some details, namely implementation details (Ward and Mellor, 1985; Stevens *et al.*, 1994). Therefore, rather than having one domain of functions, technology modules, and physical assemblies (left-hand side of Figure 3-4), three distinct domains are distinguished (right-hand side of Figure 3-4).

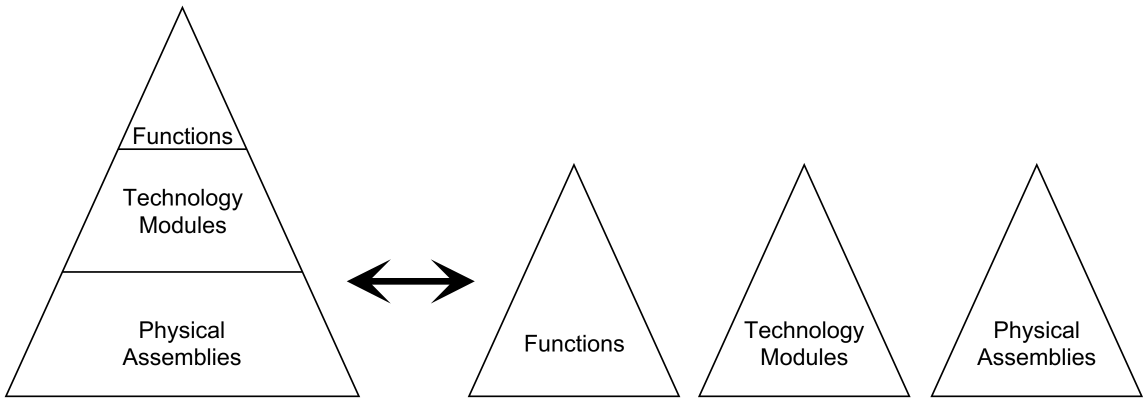


Figure 3-4 Implementation dependence and level of detail

Gordian project

Preferably, packages should be decoupled in the functional, technology, and physical domain in order to facilitate their evolution. All three domains should be taken into account. The Gordian project, however, focused on decoupling in the functional and the technology domain. Figure 3-5 shows decoupling in both domains. The left picture shows two modules in the functional domain; the right picture is a possible representation of the same two modules in the technology domain. A DLL (Dynamic Link Library) is a software library that consists of functions that can be used by other software components.

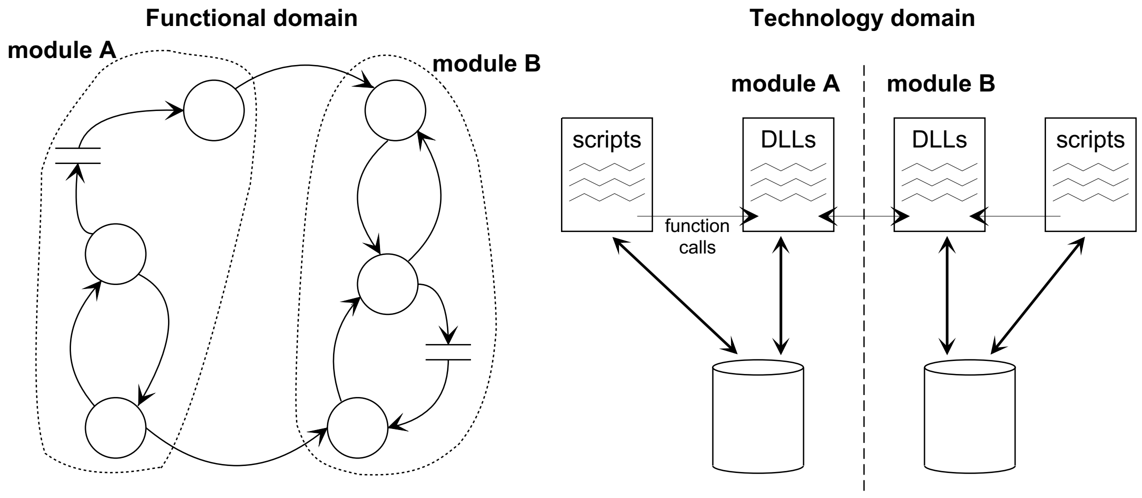


Figure 3-5 Decoupling in the functional and technology domain

Packages have to be decoupled in the technology domain in order to streamline integrations and harmonise interfaces. If packages are decoupled in the physical domain, they can be put on the market independently, and they can be shipped to customers separately. However, this does not guarantee an easy modification of a package; integrations/entanglements in the technology domain might hinder a smooth evolution. Entanglements such as packages writing in tables of other packages are major obstacles for modification of these packages in the future. Section 3.5 shows that basically three types of integrations are present in the technology domain. They can be replaced by a special type of integration, which uses libraries as shown by the DLLs in Figure 3-5. This latter type is 'technically' more flexible than the other three, and effectively decouples packages in the technology domain.

Packages have to be decoupled in the functional domain in order to achieve minimal coupling between packages. Decoupling in the technology domain can significantly improve system maintenance, since it streamlines integrations between packages. However, undesired entanglements in the functional domain might still be present. For instance, tables might be updated from various packages, which threatens the integrity of the data. In the technology domain, one determines how to update a table in a package from another package, and decoupling this type of integration eases its modification in the future. In the functional domain, the fact that a table in a package is updated from another package is determined. One establishes what functionality belongs to what package in the functional domain.

Obviously, the borders between packages have to be carefully set to avoid unnecessary interfaces between packages.

3.4 Decomposition hierarchy

Besides the use of domains, one of the most common concepts in architecting is to decompose a design problem into smaller parts that can be solved in series or in parallel. In some application areas, effective decompositions are well-known and little search at that level needs to be conducted as part of routine design activity (Erens, 1996). This process can be repeated until the activities and their deliverables have become manageable, thereby structuring design data in a hierarchical way. The triangles in the right part of Figure 3-4 show that each domain has its own hierarchical structure. In practice, however, the hierarchical structure is frequently confused with hierarchical control relations as in organisation diagrams. This is the difference between 'boxes-in-boxes' (or 'parts-within-parts') and 'bosses-above-bosses'. The right-hand side of Figure 3-6 shows the same hierarchical structure as the left-hand side, but the latter could be confused with a control structure.

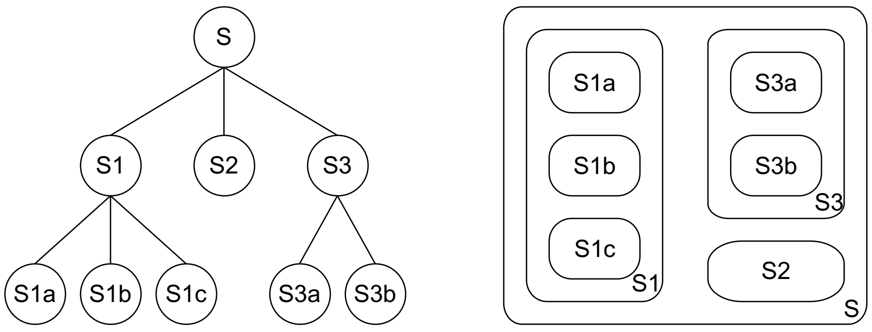


Figure 3-6 Decomposition hierarchy

Adapted from (Van den Hamer and Lepoeter, 1996)

Brandts (1993) gives three reasons for dividing systems into subsystems, i.e. for introducing a hierarchical (or nested) structure. The main reason is to avoid (or rather: manage) complexity. Industrial systems are often too complex to design as one system. Therefore, these systems are divided into separate subsystems, which are more manageable than the original system. The second reason is the wish to build systems in a modular way. Modular systems have advantages in terms of extensibility, adaptability, and reuse. If a subsystem has a well defined functionality and interface, it can be reused in different parts of the design or across multiple projects. Subsection 4.2 discusses the topic of modularity in more detail. The last reason to divide the object design into separate subsystems is to develop these subsystems in parallel and reduce the overall development time. Design processes can be completed more quickly if separate subsystems are designed in parallel. Designers work in parallel on smaller object designs, resulting in a design process evolving more rapidly. In addition to these three advantages, Brandts mentions a disadvantage to divide systems into subsystems; subsystems are all designed and optimised individually. The combination of the subsystems will not necessarily produce a global optimum. To prevent sub-optimisation, a lot of communication is necessary between designers, implying a non-divided design process.

The complementary technique of decomposition is the composition of a set of subsystems into one system. Architecting is basically a top-down activity; systems are decomposed into subsystems. However, when the functional model is specified in detail and the functions are allocated to technology modules, these modules need to be composed into a solution to the original problem. According to Erens (1996), it is not always the case that the composition of modules or assemblies meets the original function due to the complexity of the detailing and allocation process.

Composition also occurs when the specifications prescribe the use of an existing technology, for instance because of the reuse of existing technology components. Then, (the functions of) these components have to be incorporated into the functional model, without violation of the implementation-independent character of the model. Incorporation means that these functions have to be composed into higher-level functions.

Finally, composition may be part of a 'centre-out' design approach. For example, the Ward/Mellor method, which is often used in electrical engineering, prescribes to use a preliminary transformation diagram with which to construct transformation diagrams at higher and lower levels (Ward and Mellor, 1985). Transformations in the preliminary diagram are grouped so that higher level transformations are specified. The higher level diagrams are used to specify the lower level diagrams. When both higher and lower level diagrams have been constructed, the preliminary diagram has become obsolete.

Basically, architecting is a process of concurrent refinement of the functional and technology domains. Erens and Verhulst notice that 'once functions are defined at a given level of the functional hierarchy, they cannot be decomposed independently of the evolving hierarchies in the technology domain and the physical domain. Consequently, an iterative scheme of decomposition and allocation between the functional domain and technology domain must be used to incorporate new information regarding functions and solution principles. This process suggests a zigzag pattern' (Erens and Verhulst, 1997; p. 171). Architecting is in principle concurrent; both the functional and technology domains are refined simultaneously. The possibilities to realise decomposed functions by means of technology modules is constantly checked. Possibly, the structure of the technology modules can be used to make decisions about arbitrary decomposition possibilities in the functional model.

Note that this observation is in line with Suh's well-known Axiomatic Design theory (Suh, 1990). According to this theory, design is the creation of synthesised solutions in the form of products, processes, or systems that satisfy perceived needs through the mapping between functional requirements in the functional domain and design parameters of the physical domain. Design parameters have to be selected properly in order to satisfy functional requirements. Both functional requirements and design parameters have hierarchies. In order to decompose these hierarchies, a designer has to travel back and forth between the functional domain and the physical domain in developing the functional requirement and design parameter hierarchies. Note that Suh does not distinguish a technology domain. For an extensive discussion between Erens' and Suh's domains, the reader is referred to (Erens, 1996).

Gordian project

The Baan applications can be hierarchically decomposed in packages, modules, and business objects. Figure 3-7 shows the structure of the Baan applications. A Baan package (e.g. Triton Manufacturing) consists of modules (e.g. Item Control). At their turn, modules consist of business objects such as Item Data. Finally, during run-time multiple sessions can be invoked from a business object (e.g. Maintain Item Data). The Gordian project primarily focused on packages, mostly because packages had to be introduced on the market independently of each other. In other words, packages had to be decoupled from each other.

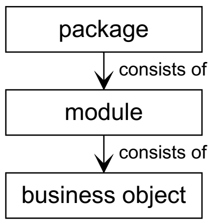


Figure 3-7 Baan application structure

3.5 Views

Views are essentially 'windows' through which selective aspects of a system can be observed. Whilst some particular aspects or attributes are emphasised, other extraneous detail is suppressed to avoid obscuring the real issues at stake. The set of views chosen to describe a system is variable. A good set of views should be complete and mostly orthogonal, i.e. the set of views should cover all aspects of a system and each view should capture different pieces of information. Not all views are equally important to system developmental success, nor will the set be constant over time (Rechtin and Maier, 1997).

Whereas hierarchy divides systems in subsystems, views divide systems into aspect systems. For example, one can take an interest in the data aspect of the functional domain and construct a conceptual data model in which the various data entities and their relationships are shown. The right-hand side of Figure 3-8 shows a view on a system (left-hand side of Figure 3-8) with only a subset of the original relations.

Dolan *et al.* (1998) state that an architecture is an artefact to support various stakeholders in managing complexity and coordinating their development activities. Stakeholders such as customers, developers, maintainers, and users have certain roles to fulfil. Each stakeholder is concerned with certain aspects, and has its own view on the system.

Regarding views, a position is taken different from Van den Hamer and Lepoeter's (1996), who consider the view concept to be inseparably linked to the development steps. They argue that many products are too complex to represent in one single type of representation, and that, therefore, multiple levels of abstraction or 'views' are needed. Design starts with high-level

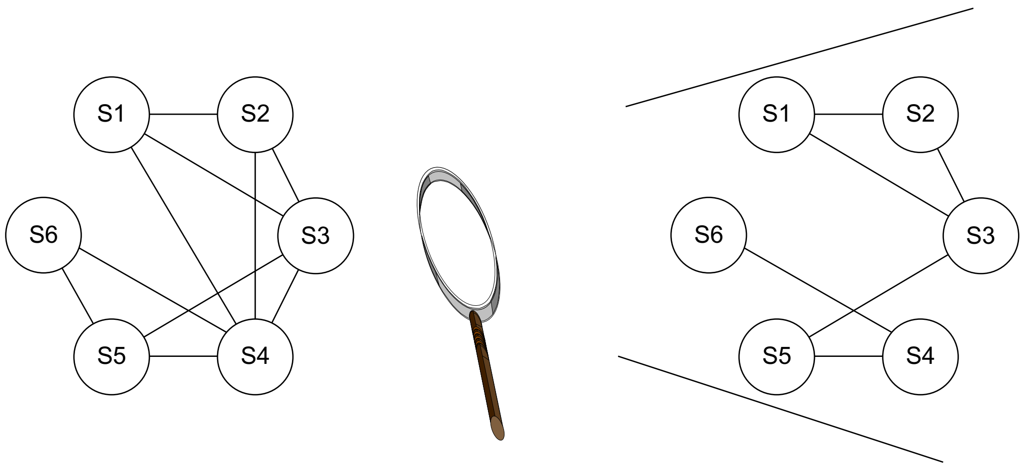


Figure 3-8 Views

descriptions of systems, after which it proceeds to a lower level of abstraction with greater detail. Complex design processes can be made more manageable by using multiple steps, i.e. by introducing multiple design views.

However, this thesis asserts that views are not linked to development process steps or levels of abstraction. A view is a window through which a particular aspect system is regarded. Linking views to development steps or levels of abstraction is wrong by definition. On the other hand, Van den Hamer and Lepoeter's 'view' is more or less similar to the concept of a 'domain' as advocated by Erens and this thesis. Also, Zachman's 'views' (the rows in Table 2.I) are linked to process steps, whereas his three perspectives (the columns in Table 2.I) are oriented to different aspects of the object being described. These perspectives – data, processes, networks – are similar to the definition of views in this dissertation.

Gordian project

Entanglements between Baan packages occur in a number of ways in the technology domain. A view is taken where only entanglements between packages are considered. Three types of integrations are distinguished: table-table integrations, software-table integrations, and software-software integrations. These types of integrations are illustrated by Figure 3-9. In the context of the Gordian project, the term 'integration' indicates a relation between subsystems in the technology domain.

The first type of integration is the *table-table integration*. References are made from a table in a certain package to a table in another package. Figure 3-9 illustrates an example of a reference to an item record (table B) from a History by Item table in the Triton Process package. This table contains an item field, 'mitm' that refers to the item table. By means of this reference, the History by Item table has access to the fields in the Item table. This way, the Database Management System (DBMS) enables the History by Item table to use the description field of the Item table. The DBMS provides the required consistency and integrity mechanisms. Normally, a user does not notice these references, but he might run into them when he tries to delete a record of e.g. the main item table. Although sometimes items are not used anymore, they can not be deleted since references are made to these items. In the example of Figure 3-9, deletion of the parent (a record of the Item table) is not allowed if any child (a record of the History by Item table) refers to the parent; the referential control delete mode is ‘restricted’. Although the item might not be produced anymore, and history data are obsolete, references still exist, and the item record has to be present.

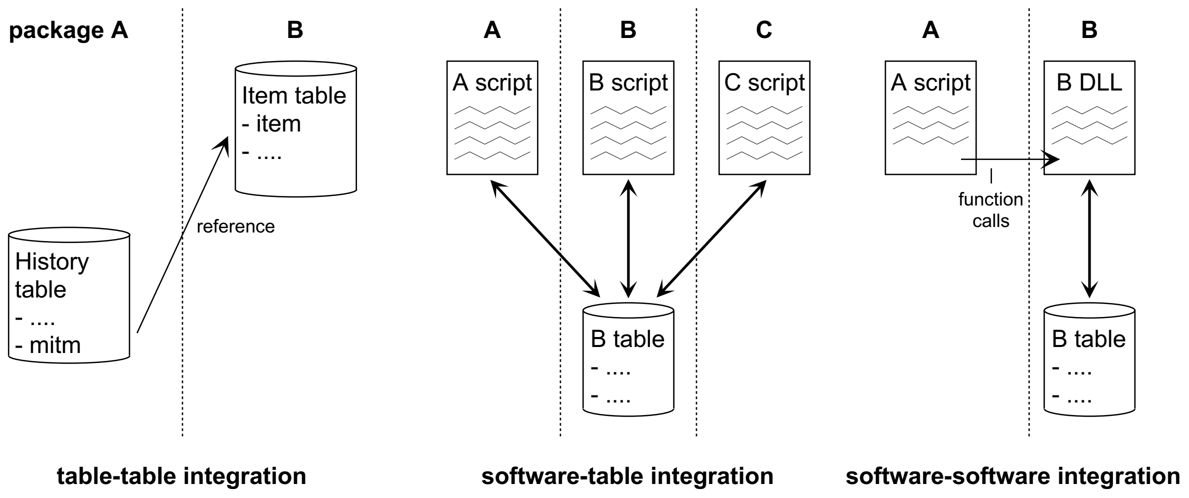


Figure 3-9 Three types of integrations between packages

References between packages imply that a customer needs to purchase the package that contains the tables that are referred to. Consider the previous example in relation to a customer whose line of business is the process industry. If this customer has bought the Baan Process package, he needs to purchase the ITM module (that contains the Item table) of the discrete manufacturing oriented Baan Manufacturing package as well, in order to have item control functionality. Clearly, this is an undesirable situation.

The second type of integration is the *software-table integration*. Scripts in a certain package write (and read) in tables of another package. Figure 3-9 gives an example, in which records of a table in package B are deleted, inserted, or updated by scripts and Dynamic Link Libraries (DLLs) from other packages. DLLs and ‘includes’ are software libraries that consist of functions that can be used by other software components. DLLs and includes are dynamically and statically ‘linked’ respectively. The benefits and drawbacks of DLLs towards includes are outside the scope of this thesis.

The third type of integration is the *software-software integration*. Scripts in a certain package call functions of a DLL in another package. An example is depicted by Figure 3-9, where a package A script calls a function of a DLL belonging to package B. Note that the particular script needs to ‘know’ the DLL mentioned. The script needs information about the package the DLL belongs to, the version of the package, and so on. In addition, the script must check whether the DLL is present at all; it is possible that the user has not bought the specific module. In this kind of structure, all information for a proper invocation of other functions needs to be present in the script.

In the Gordian project, only integrations between packages were considered. In this view, three types of integration were distinguished. Chapter 4 shows that the three types of integration are handled similarly.

3.6 Other concepts

The previous three sections present the three most important architecting concepts for evolving one-of-a-kind systems, such as shop floor control systems. Note that these three concepts can also be found in Brandts' design cube (see Figure 3-1). In addition to these three concepts, architects make use of a number of concepts that are known to handle design complexity. In this section, variants, status, and versions are discussed. This section builds on the work of various authors such as Van den Hamer and Lepoeter (1996), Erens (1996), and Brooks (1995).

3.6.1 Variants

Companies offer a large variety from which customers can choose their ideal products. Preferably, these products have many common modules to reduce the efforts in development and increase the economy of scale in manufacturing. Still, specific modules must be developed to create the necessary differentiation of products for individual customers and market segments. So, different *variants* are to a large degree similar, but not identical. This leads to the need to explicitly manage their commonality as well as their differences. Since the variant concept is mainly applicable to products and less applicable to shop floor control systems, this thesis does not cover this concept in detail. For more information about variants, product families, and variety, the reader is referred to (Erens, 1996).

3.6.2 Status

The status concept corresponds to organisational procedures that are used to maximise the likelihood that system design is satisfactory. Examples of such procedures are the introduction of verification steps, validation steps, and approval procedures. When design information passes such a step, this results in a change in *status*. The change in status does not correspond to any change in the design information itself; it only represents the fact that the organisation has decided that the information meets certain requirements. For instance, in software development it is custom that systems go through alpha and beta testing. If these tests are completed satisfactory, the product is released to the market (Van den Hamer and Lepoeter, 1996).

Brooks (1995) suggests some measures to control changes in software development. Tight control is an effective technique during debugging software systems. First, a responsible person must authorise component changes or substitution of one version for another. Then, there must be controlled copies of the system: one locked-up copy of the latest versions, used for component testing; one copy under test, with fixes being installed; and finally 'playpen copies' where each man can work on his component, doing both fixes and extensions. In such systems, the data in an engineer's private workspace is not visible or accessible to others in the same project. When this data is promoted to a higher level, it becomes available for use by other team members (Van den Hamer and Lepoeter, 1996).

3.6.3 Versions

Quantification of change is an essential technique to manage change (Brooks, 1995). Designers modify systems in multiple steps, and each step results in a new version of the design information. Systems are modified to add functionality, correct mistakes, or to optimise the design. In all cases, a new *version* is created because one wants to modify the design. Every product should have numbered versions, and each version must have its own schedule and a freeze date, after which changes go into the next version. Philips, for example, uses a two-level version identification scheme for change management of consolidated product information: a 12-digit article code is updated whenever the 'form, fit, or function' of the design changes, and a secondary identification (typically a date) is used for manufacturing changes which do not normally affect the usage of the product (Van den Hamer and Lepoeter, 1996).

Versions and variants are often confused with each other. The words 'version' and 'variant' are used respectively when references are made to design evolution and to the development of families of related products. Erens (1996) explains the essential characteristics of both:

> 'A variant is a unique configuration of modules of a product family, while a version is a semantically meaningful snapshot of a design object at a point in time. Both variants and families can have versions.' (Erens, 1996; p. 7)

3.7 Summary

Three architecting concepts are presented that enable architects to manage overall system complexity. Architecting emerged as a response to complex problems, and concepts to effectively manage complexity have been developed. Three such concepts are domains, decomposition hierarchy, and views.

Three domains are distinguished: the functional, technology, and physical domain. Each domain represents a typical design problem. Therefore, they must be separated, so that designers may focus on one domain, without dealing with design problems in the other domains. The functional and technology architectures are most important for architecting.

Hierarchical (or nested) systems are formed by means of decomposition. Systems can be decomposed in smaller parts, which are more manageable than the original system. If the decomposed subsystems are modular, they make the overall system easier to extend or modify.

Views emphasise particular aspects of a system and hide the complexity of other aspects. They allow an architect to focus on the issues that are important for his purposes. Views are not linked to development steps or levels of abstraction.

4. Architecting Principles

4.1 Introduction

The previous chapter gives the architecting concepts that play a key role in managing the complexities which are faced during system architecting. The objective of this chapter is to present the architecting principles with which architects can design flexible systems. These principles represent a vision about systems in which it is anticipated that systems will be changed in the (near) future. The principles should lead to future flexibility by encouraging an evolutionary development of systems. Just like the previous chapter, this chapter is illustrated by some of the results of the Gordian project.

In Section 4.2, the first architecting principle is presented, namely modularisation or the pursuit of modularity. Modularity is achieved when subsystems are moderately complex, minimally coupled, and their internal cohesion is maximal. Furthermore, interfaces have to be small and limited in number, they have to be precisely defined, and a clear distinction has to be made between a module’s input interface and output interface.

Section 4.3 introduces the pursuit of structural stability as the second architecting principle. Just like buildings, systems need to be structurally stable during their evolution, otherwise they might collapse. A system has to be able to perform its functions, although it is not ‘finished’ yet. Some systems are foreseen to be changed in the future. The intermediate forms have to be structurally stable, so that they do not depend on components to be added or modified.

Layers might naturally result from a modular architecture. Layers reflect design decisions where mapping tasks are decomposed in layers, and each layer performs a specific part of the mapping. As such, a layer builds upon its underlying layer. Layers are discussed in Section 4.4.

Besides facilitating change, an architect needs to express to what extent change is possible. The consequences of architectural choices have to be made explicit. Section 4.5 focuses on the indication of changeability.

A prerequisite to facilitate the change of a system is that the development organisation must be prepared to change. Architects should be aware that violations of system architectures might happen because of organisational characteristics. Section 4.6 discusses the relation between organisations and the systems they develop.

4.2 Modularity

Hierarchical systems can often be approximated as nearly-decomposable. A distinction can be made between the interactions among subsystems on the one hand, and the interactions within subsystems on the other. Decomposable systems are systems where the interactions among individual subsystems can be neglected. In most systems, however, the individual subsystems do have relations with each other, but the intercomponent linkages are generally (much) weaker than intracomponent linkages. Systems are called nearly-decomposable if they can be partitioned into subsystems with the property that the relations between the elements of each subsystem are stronger than those between elements from different subsystems (Simon, 1981; Van Aken, 1978).

Rather than the term ‘near-decomposability’, the term ‘modularity’ is used in this thesis. The concept of modules has been applied in several disciplines. In the third century BC, for example, the Chinese used modular components to produce the terracotta warriors that guarded the tomb of Emperor Qin Shi Huang. Each individual warrior was composed of mass-produced bodies, heads, hands, ears, and so on. No two of these figures was made the same, since heads and armour details were sculpted individually to give every warrior its own character (Mazzatenta, 1996). Many years later, the building architect Le Corbusier introduced the term in the context of a system of standard measures for buildings to achieve harmonious designs. Later on, building architects used the term for exchangeable standard building blocks that enabled them to construct flexible houses.

The concept of modules played an important role in the development of structured programming. One of the first proposals to apply modules to master the increasing complexity of information systems stems from Parnas (1972) who introduces ‘information hiding’ as the criterion that ‘every module is characterised by the design decisions it hides for other modules.’ More detailed criteria are proposed by Yourdon and Constantine (1979) and Pels (1988), who refine the concept of information hiding into three criteria:

- moderate complexity: the structure of a module should be understandable by its designer and its users;
- minimal coupling: the amount of specifications that is known by other modules should be minimised (information hiding);
- maximum cohesion: the complexity of a module should be large compared to the complexity of its view on its environment.

Here, *modularisation* is the strive to design subsystems with moderate complexity, minimal coupling, and maximum cohesion. Hence, modularity as the outcome of the modularisation attempt is a *predicate* assigned to systems. A *module* is a subsystem with defined interfaces. A module does not necessarily exhibit the three modularity characteristics.

In case a system has to be changed, its degree of modularity determines the impact of these changes; depending on the modularity of the system, a change will propagate into other subsystems. The adaptability and changeability of a system is determined by the degree of modularity (Brooks, 1995). For example, a control protocol, i.e. the way a control system performs its tasks, settles largely the interdependencies between the various subsystems. Obviously, the interdependencies of the subsystem are directly related to system modularity. In other words, controlling the amount and form of communication between modules is a fundamental step in achieving modular architectures (Meyer, 1988).

For software engineering, Meyer (1988) makes a link between the modular design in the functional, technology, and physical domain. His ‘linguistic modular units’ principle removes any hope that good modular functional architectures may be implemented without the appropriate technology support, i.e. solution concepts offered by the programming language. Sometimes, developers believe that they can apply advanced modular concepts as a guide to functional design but still use whatever language is imposed by the environment. They think they are able to design in an Ada-like (or object-oriented) style, and then implement in C or Pascal (or any other not object-oriented language). Meyer claims that this approach cannot work for significant developments; the gap between ideas and their realisation is too broad, as becomes painfully clear when the system evolves.

Besides the requirements that interfaces have to be small and limited in number, interfaces have to be explicit. Any outside connection should be clearly marked, so that other elements that might be impacted by a change should be obvious. The importance of an explicit definition of interfaces is shown in the modular design of a conceptual schema of a database. The principles of modular design with respect to data structures in information systems have been formalised in (Pels, 1988; Pels and Wortmann, 1990). This approach has been applied to the design of shop floor management systems by Timmermans (1993a). These studies show that the essential properties of a module are a precise definition of its interfaces and a clear distinction between the input interface and the output interface. By explicitly defining interfaces, modules identify precisely the impact of changes: inside or outside the modules.

Gordian project

Before the modularity of packages is treated, decoupling integrations in the technology domain is discussed. A table-table integration is taken as an example. Figure 4-1 (top left) shows a table in a package B, which contains two fields: a key field called ‘code’ and an accompanying description ‘desc’. A package A table refers to the ‘code’ field of the package B table. The referential control delete and update modes for the ‘code’ attribute of package B are ‘restricted’.

In the original situation, the DBMS prevents situations where deletions and updates of the ‘code’ attribute in the package B table would lead to invalid values of the ‘code’ attribute in the package A table; the DBMS prevents referential integrity problems. However, this coupling between two tables (via the DBMS) implies that a customer who bought package A needs to purchase package B as well. To remedy this problem, Baan decided to decouple its packages.

The reference could be resolved in two ways that are indicated by Figure 4-1. In the first solution (Figure 4-1, bottom left), the reference is deleted, and a field that gives the actual description is inserted instead of a field ‘code’ that indicates the *code* for the description. However, this option is not preferred, since consistency between records with the same description is not assured. Records that need the same description, might have differences in the description field.

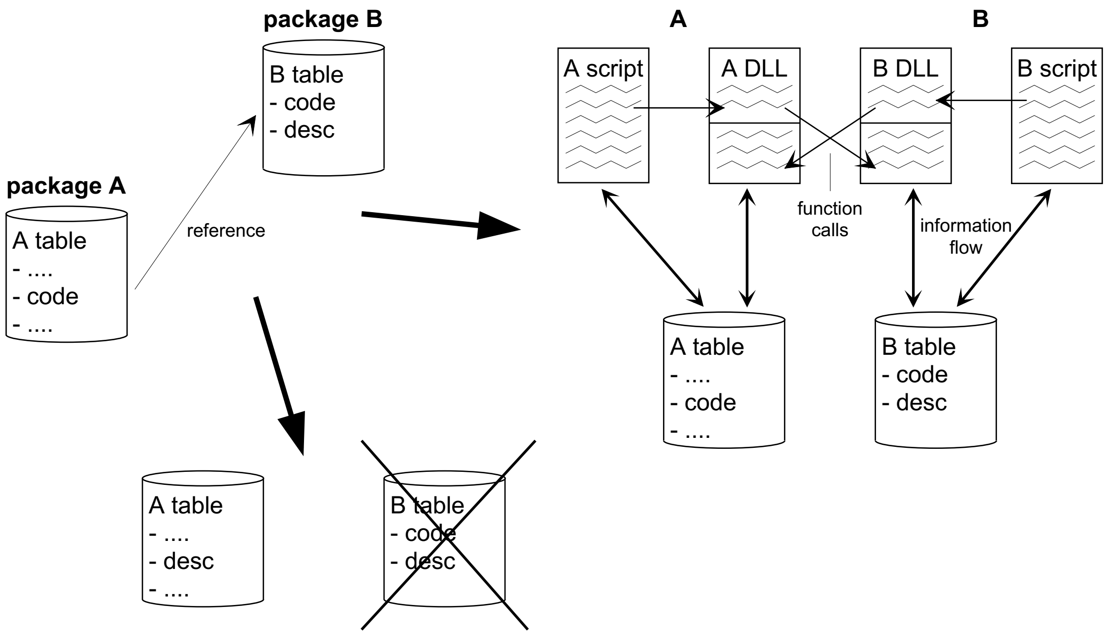


Figure 4-1 Two possible decoupling options

Another option (Figure 4-1, right) is to decouple the two tables – and thus the two packages – by using DLLs. In the current situation, deletion of the parent (a table B record) is prohibited if any child (any table A record) refers to the parent; when the parent is deleted, all references must be checked. A software solution that takes this into account would use two DLLs: one related to package A and serving as the input interface for package B, and another one that is related to package B and serves as the input interface for package A. The latter DLL should provide functions that return the description for a specific code. The first DLL should provide functions that check whether records are present with a specific code. These functions should be invoked when the user deletes a code.

A double DLL has certain advantages over a single DLL. Note that a double DLL is used, where a single call from the A script to the B DLL (without using the A DLL) would suffice (see Figure 4-1). In that case, the A script would have information about the DLLs in other packages. However, using a double DLL gives the advantage that the A script does not need to know all DLLs in other packages; it only has information about its ‘accompanying’ DLL. That DLL takes care of the proper invocation of other functions, whether they belong to package A, another Baan Package, another Baan version, or even another vendor’s software.

In conclusion, the technical decoupling of a table-table integration is illustrated for a situation where the referential integrity checking functionality of a DBMS is replaced by DLL functions. This approach makes interfaces between packages more explicit. Take the DLL belonging to package A in Figure 4-1 as an example. This DLL (like all DLLs) has a double function:

- it contains information about other packages, so that scripts in package A can invoke functions in other packages; these scripts do not need to know the details of the functions in other packages; and,
- it contains functions with which other packages can access information from package A.

Accordingly, the A DLL provides both the outgoing and incoming interface of package A.

By the introduction of a solution for a situation in which package B is not necessarily available, two packages can be fully decoupled concerning the table-table integrations. This solution is presented in the next section.

The decoupling approach using a double DLL construction may be carried out for all three types of integration as discerned in Section 3.5 (see Figure 3-9). Therefore, the other types of integrations are not described in detail in this section. Consider the current integrations between two packages as illustrated by Figure 4-2. The DLLs are put in the outer ring, since they form the interface between packages. However, they write to the tables, just like the scripts. At the moment, there are five kinds of integrations, among which a few subtypes of the previously identified three main types (from top to bottom):

- an ‘include’ of package B is included in a script of package A. This is one type of software-software integration.
- a script or ‘include’ of package A writes (and reads) directly into a table of package B. This is a form of the software-table integration.
- a reference is made from a package A table to an attribute of a package B table. This integration is called the table-table integration.
- an A script calls a DLL function of package B. This is another type of software-software integration.
- a DLL of package A writes (and reads) directly into a table of package B. This is another form of the software-table integration.

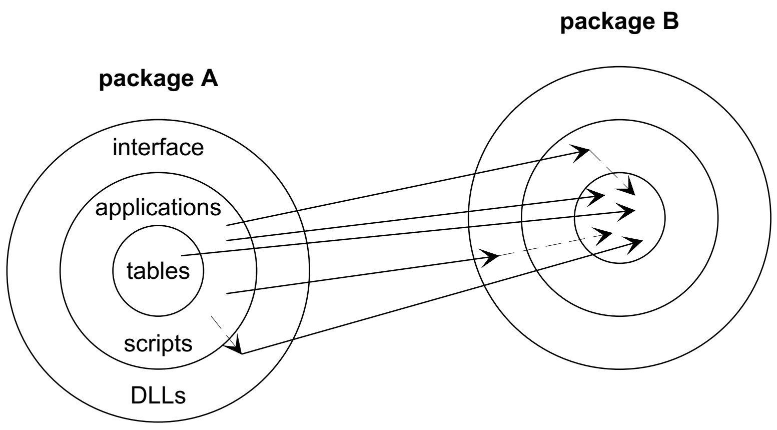


Figure 4-2 Current situation

After removing the five harmful types of integrations as depicted by Figure 4-2, a new, improved situation occurs. Figure 4-3 shows two packages where all table-table, software-table, and software-software integrations have been replaced by double DLL constructions. Note that a double DLL creation is ‘technically’ also a kind of software-software integration, but it is ‘technically’ more flexible than the two versions depicted in Figure 4-2. For instance, compared to the mentioned integration types, a double DLL construction is easy to adapt and extend since it makes the interfaces between packages (more) explicit.

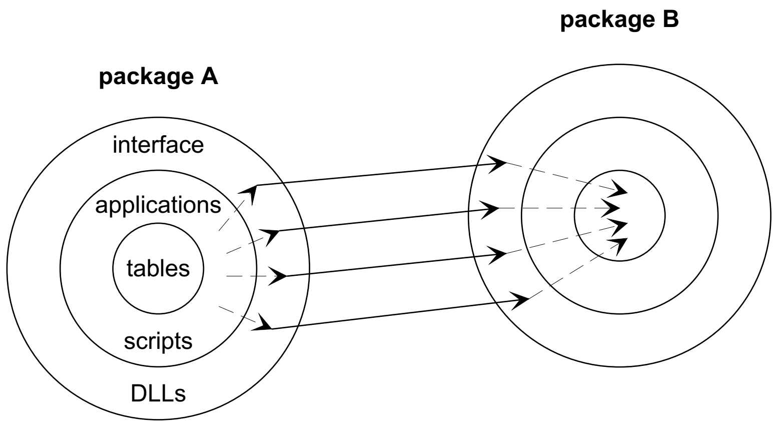


Figure 4-3 Ideal technical situation

The technical decoupling strategy as outlined above is *not* sufficient to realise modular packages. Adequate technical decoupling can significantly improve system maintenance, because it streamlines the integrations between the packages. However, it cannot prevent that one table is updated from various packages, which threatens the integrity of the data. The fact that packages are technically decoupled, does not mean that there are no functional entanglements between packages.

Modularity is improved by decoupling packages in the functional domain. This is supported by Pels' theory on the modular design of the conceptual schema (Pels, 1988; Pels and Wortmann, 1990). An important part of the entanglements between Baan packages lies in the fact that tables in Baan IV can be updated from various places in the system. Pels' theory indicates how this can be improved. Each table should be linked to one own domain only; update authority for a table should be located at only one place in the system. Pels provides a sophisticated approach to assign update authority. He relies heavily on evaluating the constraints which each module assumes about a specific table. Communication clashes result if a module initiates an update that is in conflict with the actual value of the own data of another module. What combinations of values are meaningful and therefore allowed in the information base, is specified by means of constraints. Therefore, constraints play an important role in the prevention of communication clashes.

Problems in finding the constraints complicate the application of Pels' ideas. According to Pels, modular decomposition makes it possible to develop and maintain an integrated information system for arbitrary large organisations. However, the requirement to gather the constraints that are applicable to the identified modules hinders the application of Pels' theory. A crucial assumption is that all constraints can be deduced from the documentation (e.g. functional design documents). In practice, this assumption does not hold and most constraints can only be traced in the code. This limitation is of more importance to applicability of Pels' framework than the number of constraints in a large system such as Baan IV. The mere number of the constraints will only make the application of Pels' approach more time consuming. As long as constraints are hidden, however, application is impossible. Until documentation of constraints is significantly more developed, conceptual decoupling will be very difficult.

4.3 Structural stability

The second architecting principle is structural stability. The need for stability is widely acknowledged in engineering; unless an architect creates stable structures for the system, it may collapse. Less well appreciated, perhaps, is the need for stability as the system is built and evolves. Buildings, for instance, sometimes collapse during construction. Unfortunately, this does not apply to buildings only, but to systems in general. Frequently, the underlying reasons should be searched for in structural inadequacy until the system is fully assembled, i.e. not every intermediate form was able to stand alone. Another reason is instability after assembly, i.e. if the coupling between elements was wrong (Rechtin, 1991).

A system is structurally stable if it can perform its tasks with and without components that — if present and configured — are 'used' by (some of) the system's components. The system in evolution can be seen as an intermediate form; the intermediate form is expected to be changed. For example, it might be foreseen that in the future the system will be extended by certain components. The system as an intermediate form is structurally stable if it can perform its tasks with and without these components; the system does not depend upon them. If these components are present, the system might be configured such that it does not use them. A decomposable system, for instance, is structurally stable; in a decomposable system, intermediate forms or collections of subsystems are naturally stable, since the subsystems do not have relations with other subsystems. A nearly-decomposable system (a modular system) might not be structurally stable.

The way to ensure at least some stability is to build systems only out of stable elements. The architect works from the simplest configuration to the more complex. At each step, he makes sure the system works before he proceeds (Brooks, 1995). Rechtin (1991) points out that around 1990 the same approach was envisaged for the assembly in space of the US space station Freedom; at the end of each assembly flight, a completely functional spacecraft should remain. A similar property is demanded by customers who buy products with optional extensions. With or without extension, the product should function well.

Simon pointed out the issue of evolution from intermediate forms by means of his famous parable of the two watchmakers Hora and Tempus. A conclusion he draws from this parable is that 'the time required for the evolution of a complex form from simple elements depends critically on the numbers and distribution of potential intermediate stable forms' (Simon, 1981; p. 202). Simon's final conclusion is that

> 'complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not. The resulting complex forms in the former case will be hierarchic' (Simon, 1981; p. 209).

However, Simon's conclusion should not be misapplied as a rationale for bottom-up evolution. Bottom-up evolution can have a serious drawback, since it may not be clear ahead of time to what it will lead. Different choices of the first aggregations could result in totally different systems. Although mankind itself proves that trial and error and survival of the fittest results eventually in stable surviving systems, this approach is not suitable for system architects. After all, an architect pursues a given objective in a predictable time frame (Rechtin, 1991).

Complex systems should evolve within an architecture. Simon’s statement was written within the context of an architectural framework with top-down decomposition into stable elements. Therefore, the original statement might be stated as:

> ‘Complex systems will develop and evolve *within an overall architecture* much more rapidly if there are stable intermediate forms than if there are not’ (Rechtin, 1991; p. 91).

The stable intermediate forms are not necessarily architecturally simple. One cannot build on something that was not designed for it. The crucial parameter of intermediate forms is stability, not necessarily simplicity.

It might be sensible to build in options during the architecting process. The system will evolve, and new subsystems will be used in future versions. These versions might put demands on components that differ from the requirements of older versions. If these new demands can be reasonably foreseen during initial architecting, it might be justified to build in the right measures to provide for the demands of future systems. Then, however, components will be more complex, and a compromise has to be made between future demands and component complexity.

In short, a functional (and a technology) architecture should incorporate stable, intermediate components. The components should be enable to function stand-alone without collapsing. This usually implies that these components are quite complex, since they offer the provisions to build upon, and to evolve into more complex systems. The components have to fit within the architecture of the overall system. Architecting is not only structuring (Rechtin, 1991), but also ‘providing’; an architect has to provide adequate measures to provide for future flexibility and evolution. ‘Predicting the future may be impossible, but ignoring it is irresponsible’ (Rechtin, 1991; p. 207).

Gordian project

In addition to the technical decoupling as described in Section 4.2, there should be a provision that allows a user to invoke sessions depending on implemented packages. For example, Figure 4-4 shows that the invocation by a DLL of package A depends on the implemented package B. A parameter determines the functions to be invoked in which DLL, depending on the implemented Triton and Baan versions.

A special solution is introduced for the situation that no package B is available (Figure 4-4, bottom right). This so-called ‘minimal environment’ could for instance consist of some primary tables. By means of those minimal environments, packages could be made independent of each other; if a package B is not available, package A uses some of its own functions and tables that represent the DLLs and tables of the non-existing package B. In general, these functions and tables would not be used if package B were available.

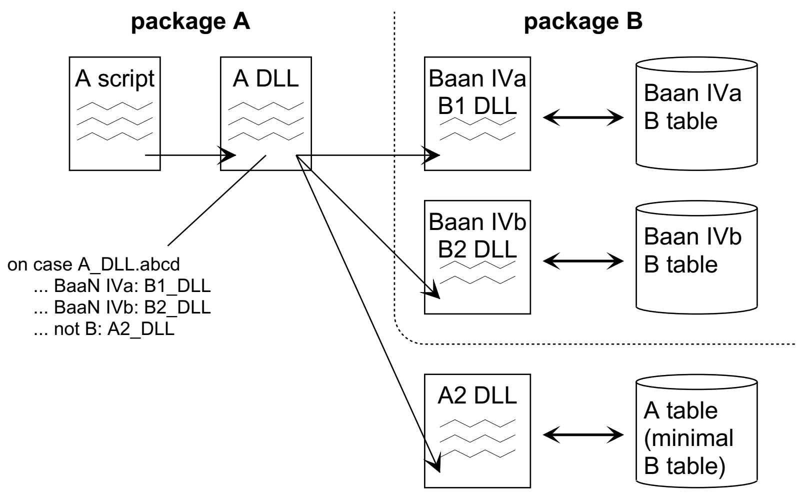


Figure 4-4 Parameter dependent invocation of versions

Obviously, the minimal environment construction enables Baan to introduce new package versions independently of each other. Minimal environments make stable, intermediate forms out of packages. They are stable, because they are able to function on their own; the package as a whole provides functionality without the need for other packages. They are intermediate, because they are part of a larger whole, namely the set of Baan packages; a package is able to cooperate with other packages, and together they provide more functionality. However, the functional interfaces between a package and its neighbours should not change. This is hard to realise in practice; new versions are introduced simply because modifications were needed and these changes often affect the interfaces with other packages.

In the previous section and in this section, the two most important principles to construct future-proof systems are described: modularity and structural stability. In the remaining sections of this chapter, other measures are discussed. Layers, for example, are often a natural consequence of modularity. The indication of changeability is merely a support during evolution, not a guiding principle. Finally, the relation between an evolving system and its development organisation is glanced at.

4.4 Layers

A modular architecture may naturally result in a layered architecture; modules are assigned to specific layers. Layers reflect design decisions based on allowable relations and interfacing constraints. The layers in an architecture represent allowable interfaces among modules. Modules within a layer can communicate with each other. Modules in different layers can communicate with each other only if their respective layers are adjacent (Soni *et al.*, 1995). A layer builds on its underlying layer, which at its turn builds on its underlying layer as well. Consequently, a layer explicitly uses the functionality of its underlying layer, and implicitly uses the functionality of all layers underneath its underlying layer.

Layers are used mainly to solve mapping problems. The mapping task is decomposed in layers: each layer performs a specific part of the mapping. In this sense, the division in layers is part of an architecture. The advantage of layers is the flexibility: changes can be made inside a layer without affecting other layers. A disadvantage of a layered architecture is its rigidity: new layers are hard to be shoved in between existing layers, since this requires a (major) change of interfaces. Examples of the application of layers in mappings are:

- the targets of an enterprise must be mapped on its physical processes; therefore, a strategical, tactical, and operational layer are distinguished;
- data from a database must be mapped on computer screens; therefore, an internal, conceptual, and external layer are distinguished.

A good example of a division in layers is the Open Systems Interconnection (OSI) reference model for network architectures (see Figure 4-5). Messages are mapped from an application via a communication medium to a partner application. Some of the design principles underneath this reference model are that each layer should have a well-defined function, and that the amount of information communicated via the interfaces should be as low as possible. Every layer offers services to the next higher layer (Tanenbaum, 1988).

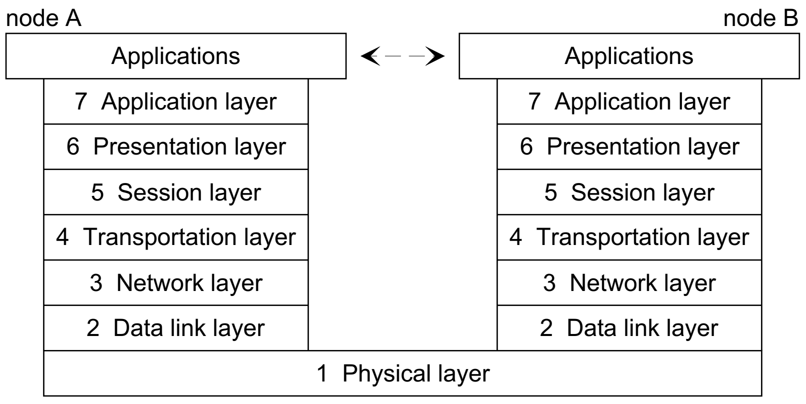


Figure 4-5 OSI reference model for network architectures

Another example is a reference model of information systems for Manufacturing Resource Planning (MRP II) (Bertrand *et al.*, 1990b; Wortmann *et al.*, 1997). Figure 4-6 shows that the reference model constitutes of a number of concentric circles representing various system layers. The innermost circle represents the system platform. The applications that support the registration of *state-independent* data, such as product data and operation data, are found on the second layer. The applications for the registration of *state-dependent* data are found on the third layer which rests on top of the state-independent data. These data describe the state or condition of materials and orders in the transformation process. Finally, the outermost layer is comprised of workflow applications and systems that carry out decision processes with or without human intervention.

Gordian project

Before the application of layers in the Baan applications can be explained, it is necessary to describe the envisaged evolution of the Baan packages as foreseen in 1996. Within a few years, the Triton/Baan product line will be succeeded by software resulting from the Nucleus project. The Nucleus applications will be built from scratch, object-oriented, programmed in

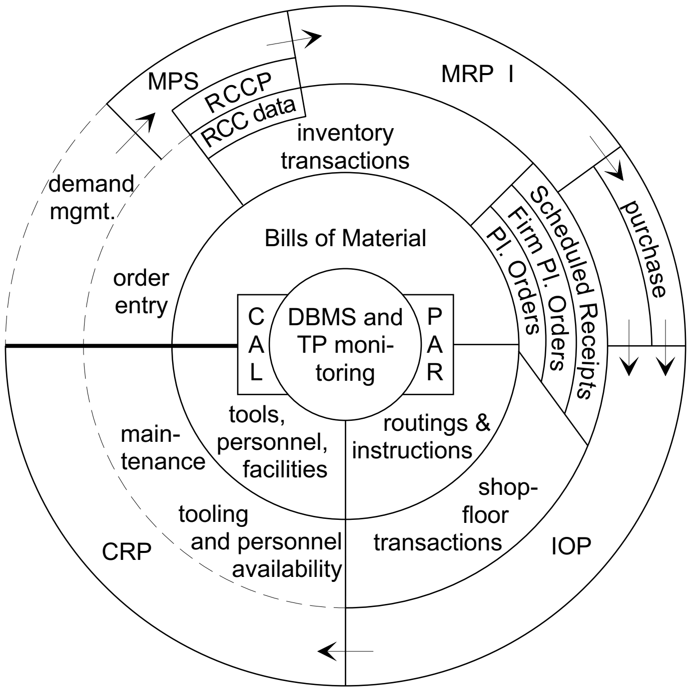


Figure 4-6 Reference model of MRP II systems

Source: Wortmann *et al.* (1997)

an extension of Java, and they will apply some of the possibilities of Internet technology. The question arises whether Baan packages or modules can be integrated with Nucleus packages. As stated before, it is desirable to put Baan packages on the market independently of each other. Evidently, the same holds for Nucleus packages. In the future, customers might want to replace some of their existing Baan packages with newer Nucleus packages, thereby gradually migrating from Baan to Nucleus. These Nucleus packages need to be integrated with the existing Baan packages. In other words, Baan and Nucleus packages have to co-operate with each other.

Although there are big differences in technology between the Baan and Nucleus packages, these differences might be overcome. Baan packages are based on procedural programming languages, whereas Nucleus packages will be object-oriented. However, their interoperability might be assured by standard information buses and wrapping techniques, thereby treating the Baan packages as legacy systems.

Common industrial practice to handle legacy systems is to isolate them. At a certain moment, companies look upon the Baan IV packages as legacy systems. In order to allow the cooperation of Baan and Nucleus packages, the Baan packages need to be isolated. Only the isolated packages are allowed to access their own function calls, and only they are permitted to send messages in their proprietary data formats. Then, the impact of these packages will be minimised, and it will be technically possible to swap applications in and out based on business needs. To accomplish this, the use of Baan's proprietary technology and data formats needs to be restricted by wrapping Baan applications and using standard interfaces.

Part of the solution is using messaging technologies or object request brokers, which mediate between applications. At the end of the 1990's, some CORBA compliant products became available on the market. The Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG), is an answer to the need for interoperability among the rapidly proliferating number of hardware and software products. CORBA allows applications to communicate with one another no matter where they are located.

The Object Request Broker (ORB) is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or distributed across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In doing so, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In addition to the ORB, the OMG specified a language that acts as the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked. The Interface Definition Language (IDL) defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

Wrappers are software implementations that form a layer surrounding an application for the purpose of presenting interface and functional behaviour required by other implementations. The need for wrappers often arises when one migrates existing applications to embrace a new, more advanced approach. This typically occurs when migration is preferred to wholesale replacement for cost/benefit reasons (Sematech, 1996). Object wrappers allow companies to transition to objects gradually from their existing base of legacy system software. The object wrappers make the legacy code appear object-like to the object-oriented programs, tools, and frameworks (Orfali *et al.*, 1996).

Most probably, future Nucleus packages will reside on a CORBA compliant ORB. The legacy Baan packages must reside on the ORB as well. In an ORB-based solution, the Baan packages are modelled using the IDL. Then, 'wrapper code' is written that translates between the standardised bus and the Baan interfaces (see Figure 4-7).

By means of a layered implementation of Baan packages, difficult technical entanglements of the Baan packages with Nucleus products are avoided. All communication with Baan packages use industry-standard formats and interfaces. Only Baan applications have direct access to the Baan function calls or send messages in proprietary Baan formats. For the scripts, the DLLs are the interface to other packages. For the wrapper code, the DLLs are the interface to the functionality of the package in question. Scripts and wrapper code are completely disconnected.

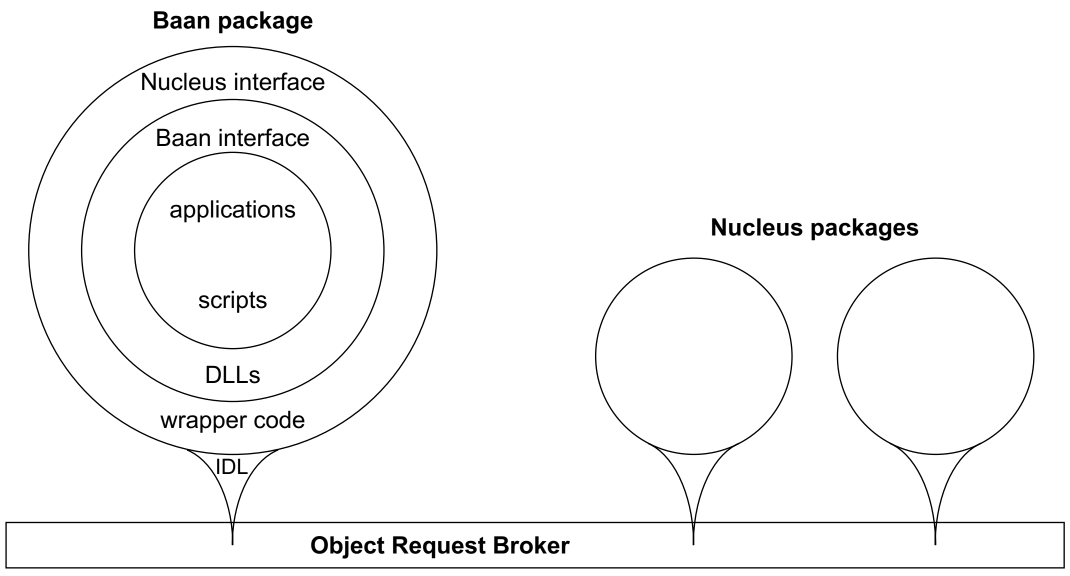


Figure 4-7 Interoperability of Baan and Nucleus packages

4.5 Indication of changeability

Whereas the previous principles aim to facilitate change, this principle states that it should be indicated to what extent change is realisable. After all, every architecture has good features and inherent drawbacks. It takes a lot of effort and objectivity to identify drawbacks inherent to the architecture. If not considered in the beginning, the drawbacks can appear as unpleasant surprises later on. If considered and retained, their consequences need to be understood and accepted (Rechtin, 1991). Therefore, the reasons to make certain architectural choices and their possible future consequences should be well documented. When the system has to be changed, architectural choices will hinder or facilitate these changes. It might prove very useful to have documents explaining past choices when one wants to change a system.

Although it is hard to foresee future changes, one can at least determine to what extent a module is allowed to be changed. Some parts of a system should not be changed at all; the unchangeable parts and the parts that are allowed to be modified should be documented. For example, application frameworks specify the interfaces between components. Since these frameworks act as reference models, architectures that aim to comply with them, are not allowed to change these interfaces. Otherwise, they would not meet the reference model anymore.

Other parts are allowed to be changed, but preferably should not. For instance, one should be very reluctant to change the public interface of an essential module, such as the central controller of a centralised control system; too many other modules would have to be significantly changed as well. The interfaces of such a central module are the equivalents of buildings' load-bearing walls. Preferably, they should not be touched upon.

By measurement of the definitions of a module's interfaces and interdependencies with other modules, one obtains an indication of how easy (or hard) that module can be modified, i.e. how many other modules have to be changed to what extent as well. Even counting the number of modules that use the public interface will give a reasonable impression at least.

Gordian project

As stated in Section 3.2, the objective of the Gordian project was to examine the possibilities and limitations to put the dispersed warehousing functionality into a separate package. However, to realise this objective, an analysis of the entanglement of warehousing functionality with the remainder of the Baan packages had to be performed.

In the entanglement analysis, a number of criteria were applied in accordance with the three types of integrations between packages in 3.5. The first criterion was the number of references from the (intended) Warehousing tables to other packages' tables. Also, the reverse variable was a valid criterion: the number of references from other packages' tables to the warehousing tables. Furthermore, another criterion was the number of write actions to warehousing tables by other packages. Again, the reverse criterion was considered: the number of write actions by warehousing tables to other packages' tables. Finally, the last applied criteria were the number of function calls from warehousing scripts to include or DLL functions of other packages and v.v.

For example, the table-table integrations were analysed by means of the number of references from other packages' tables to the warehousing tables. It appeared that there were only 19 references from other packages' tables to the warehousing tables. To facilitate comparison, there were 343 references from warehousing tables to other tables of other packages. In other words, the coupling between warehousing and other packages by means of references was rather unilateral. This was partly caused by the fact that some warehousing modules had been added recently; new functionality is more likely to use old functionality than vice versa.

4.6 Organisational embedding

This chapter is ended by a discussion on organisations and the systems they develop. This section does not introduce an architecting principle, but indicates some organisational prerequisites for system evolution.

It is generally accepted that the organisation of a development project has a large influence on the outcome. For example, Erens states that 'the development of a new product family with a new architecture could require the restructuring of the existing organisation.' (Erens, 1996; p. 50). Brooks claims that

> 'the organisation chart becomes intertwined with the interface specification, as Conway's Law predicts: "organisations which design systems are constrained to produce systems which are copies of the communication structures of these organisations (Conway, 1968; p. 31)" Conway goes on to point out that the organisation chart will initially reflect the first system design, which is almost surely not the right one. If the system design is to be free to change, the organisation must be prepared to change' (Brooks, 1995; p. 111).

This has led some authors to explicitly discuss ‘development architectures’. For software architectures, for example, Kruchten (1995) writes about a *development view* that describes software’s static organisation in its development environment. It focuses on the organisation of the actual software modules. The software is packaged in small chunks – program libraries or subsystems – that can be developed by one or more developers. Kruchten’s logical and development view are very close, but address different concerns. He concludes that the development view does not have a one-to-one correspondence with the logical view, i.e. the functional architecture. The larger the project, the greater the distance between these views. This is due to constraints such as team organisation, expected magnitude of code, degree of expected reuse, release policy, and configuration management.

A good example of an enterprise with a strict organisation principle is Microsoft: around 1990, its development teams constituted of no more than 6-10 people in order to keep communication within the team minimised. A consequence of this principle is that Microsoft needed mechanisms to support communication among teams. Microsoft’s Object Linking and Embedding (OLE) technology was primarily meant to support communication among its development teams; later, it would be used to support communication among applications.

Brooks (1995) suggests to organise development teams like surgical teams. Instead of each member cutting away on the problem, one does the cutting and the others support him in order to enhance his effectiveness and productivity.

Nevertheless, Brooks acknowledges that the attitude of individual developers largely determines the resulting quality of the system:

> ‘Each (developer) suboptimised his piece to meet his targets; few stopped to think about the total effect on the customer. This breakdown in orientation and communication is a major hazard for large projects. All during implementation, the system architects must maintain continual vigilance to ensure continued system integrity. Beyond this policing mechanism, however, lies the matter of attitude of the implementers themselves. Fostering a total-system, user-oriented attitude may well be the most important function of the programming manager’ (Brooks, 1995; p. 100).

In short, both during initial system development and system evolution, the architect guards the integrity of the system. He should have the right of veto in case violations of the architecture are about to take place. However, it is very hard to refrain individual developers from making architectural violations. Therefore, management should try to provide developers with the right attitude.

4.7 Summary

Architecting principles enable architects to design systems with the flexibility that is needed for system evolution. An architect guards the integrity of the system during evolution. The architect has to build in measures that provide for future flexibility and evolution. Three principles that stimulate the design of such measures are the pursuit of modularity, the pursuit of structural stability, and the formation of layers.

Modularity is a characteristic of moderately complex subsystems with high internal cohesion, and minimal coupling with other subsystems. The interface of modular components is small, precisely defined, and a distinction is made between a module's input and output interface. This way, other components precisely know the possibilities and limitations of using the functionality of a specific module. Furthermore, the interfaces of modular components restrict the impact of changes to few modules; changes are not propagated to other components.

Structural stability is the characteristic of a system in evolution to function stand-alone without collapsing, and with the ability to be part of a larger system or to be extended with new components. This characteristic makes building blocks from complete 'wholes'. Structurally stable systems are easy to extend or modify, simply because they are made to be extended or modified within an overall architecture.

Layers represent allowable interfaces between clusters of components, and are mainly used to solve mapping problems. The mapping task is decomposed in layers, so that each layer performs a specific part of the task. Layers offer the flexibility to make changes in a layer that do not disrupt other layers, as long as its interfaces are not changed.

In addition to the three guiding principles stated above, there are some prerequisites for system evolution. When architectural choices are accounted for, and their consequences for system changeability are known, one knows the possibilities and limitations to change the system. Unfortunately, the motivations for certain architectural choices are frequently unknown, and their implications are not understood. In addition, companies must be aware of the consequence of Conway's Law; if they want the system they develop to change, the organisation must be prepared to change.

5. Reference Architectures for Enterprise Integration

5.1 Introduction

The objective of this chapter is to investigate the practical value of enterprise reference architectures concerning the integration of production systems.* For this, three enterprise reference architectures have been studied, namely CIMOSA, GRAI/GIM, and PERA. In addition, the CIMOSA reference architecture has been applied during an industrial reorganisation project. In Section 5.2, the objectives and ideas behind enterprise reference architectures are outlined. An essential point is that enterprise reference architectures aim to support the whole life cycle of an enterprise integration project rather than just the (architectural) design activities. The GERAM framework is used to indicate how the various components in enterprise integration are related.

CIMOSA is the most well-known and most studied of the three enterprise reference architectures. It is also the one with the most impact at the standardisation committees. Furthermore, as explained in the introduction of this dissertation, CIMOSA is the enterprise reference architecture that is most familiar to the author. CIMOSA is examined more closely in order to obtain a good understanding of the concepts of reference architectures for enterprise integration. Section 5.3 gives an explanation of CIMOSA's view on enterprise integration and how to accomplish it. Furthermore, the two most important elements of CIMOSA, the modelling framework and the integrating infrastructure, are shortly discussed.

In Section 5.4, an application of CIMOSA during a reorganisation project at a machine tool manufacturer is presented. The role of the CIMOSA modelling framework in the design of a functional control architecture is illustrated. Due to immature specifications, the assistance of the CIMOSA integrating infrastructure was rather limited.

Subsequently, the practical value of CIMOSA is discussed by means of the architecting concepts and principles of Chapter 3 and 4 respectively. Although the discussion focuses on CIMOSA, most statements apply to other enterprise reference architectures as well. Section 5.5 shows that the realisation of CIMOSA's true objective, namely a dynamic, flexible, adaptive enterprise by execution of models, is far away.

Finally, Section 5.6 shows an alternative approach in the field of Enterprise Resource Planning (ERP) that does realise some of the targets of enterprise integration thinking. The Dynamic Enterprise Modeller of Baan Company allows users to make changes in models that are directly reflected in changes in the underlying Baan ERP applications.

---

* Parts of this chapter have been published in (Zwegers and Gransier, 1995; Zwegers *et al.*, 1995; Zwegers and Pels, 1998).

5.2 Enterprise reference architectures

Enterprise engineering is the discipline that identifies the need for change in enterprises and carries out that change expediently and professionally. Enterprises face integration problems due to necessary internal adaptations to cope with severe competition in the global marketplace. In order to survive at the global market, efficient operation and innovative management of change are essential. Enterprises need to evolve and be reactive so that change and adaptation should be a natural dynamic state rather than something occasionally forced onto enterprises and their manufacturing and information systems. Enterprise engineering aims to cater for continuous evolution.

Several problems occur during the (re-)design of an integrated manufacturing system. Since such a system cannot be bought off the shelf, each company has to develop its own. Williams *et al.* (1994a) notice that designing an integrated manufacturing system meets with a number of difficulties. Several viewpoints must be taken into account; not only the technical point, but also the economic, social, and human points of view have to be thought about. By definition, CIM systems are very complex, and the development of such systems is often quite expensive and risky. Most systems are not designed from scratch; the existing systems have to be considered in the development process, so that newly developed systems are integrated with the old ones. In addition, Aguiar and Weston (1995) claim that the activities performed during each phase of the life cycle are essentially derived from ad hoc procedures, so that the quality of the resultant system will depend considerably upon the experience of the persons involved. The problem is accentuated due to poor formalism with which those activities are usually carried out. This often leads to solutions that do not adequately address business requirements, low repeatability of successful results, lack of traceability of design decisions, and so on.

The objective of enterprise reference architectures is to offer the framework that solves the problems mentioned above, and with which an enterprise might develop its integrated manufacturing system. The idea behind enterprise reference architectures is that a large part of integration projects is in fact similar and common in every type of enterprise (Williams *et al.*, 1994a). This part could be standardised and utilised instead of developing it again from scratch. Once standardised, generally accepted reference architectures can be supported by tools, methodologies, and a range of compatible products in order to make the entire integration project more time and cost efficient.

Wyns *et al.* (1996) state that the benefits of reference architectures lie among others in the unified and unambiguous terminology, in the envisaged simplicity of designing specific systems, and in the high quality by relying on proven concepts. In addition, a reference architecture models the whole life span of an enterprise integration project. It indicates and justifies how and at what stage in the development process external constraints and engineering design decisions are introduced, thereby providing traceability between solution independent requirements and final realisations. It provides the framework in which enterprise integration methodologies work.

The IFAC/IFIP Task Force on Architectures for Enterprise Integration has developed an overall framework to organise existing enterprise integration knowledge. The proposed framework was entitled GERAM — Generalised (or Generic) Enterprise Reference Architecture and Methodology. GERAM is about those methods, models, and tools which are needed to build and maintain an integrated enterprise. GERAM defines a tool-kit of concepts for designing and maintaining enterprises for their entire life cycle. The scope of GERAM encompasses all knowledge needed for enterprise engineering. Thus, GERAM provides a generalised framework for describing the components needed in all types of enterprise engineering processes. It gives a description of all elements recommended in enterprise engineering and integration (IFAC/IFIP, 1997).

Figure 5-1 shows the components of the GERAM framework. The most important component is the *enterprise reference architecture*, which defines enterprise related concepts recommended for use in enterprise engineering projects. A reference architecture should point toward purposeful organisation of enterprise concepts (Nell, 1996); it should identify and structure concepts for enterprise integration, most notably life cycle activities and enterprise modelling concepts such as views. GERAM distinguishes between the *methodologies for enterprise engineering* and the *modelling languages* which are used by the methodologies to describe and specify the structure, content, and behaviour of the enterprise. These languages will enable the modelling of the human part in the enterprise operation as well as the part of business processes and supporting technologies. The modelling process is supported by guidelines (*modelling methodologies*), and the result of the process are *enterprise models* which represent all or part of the enterprise operations, including its manufacturing or service

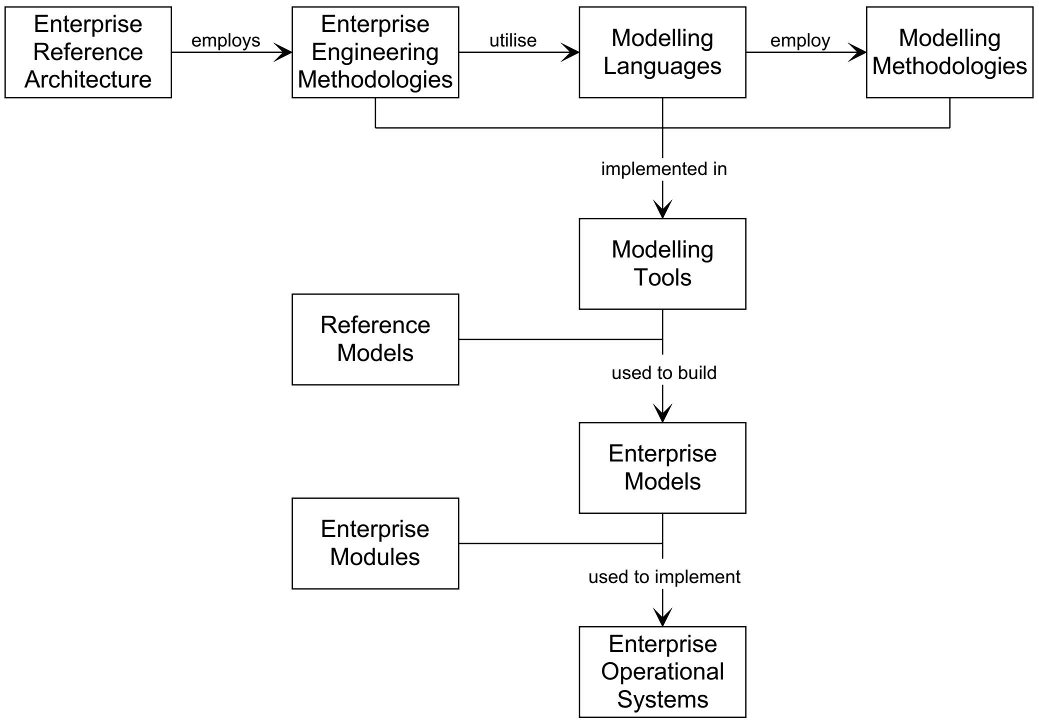


Figure 5-1 Components of the GERAM framework

Adapted from (IFAC/IFIP, 1997)

tasks, its organisation and management, and its control and information systems. These models should be used to guide the implementation of the *operational system of the enterprise* (IFAC/IFIP, 1997).

In the GERAM framework, the methodologies and the languages used for enterprise modelling are supported by *enterprise modelling tools*. The modelling process is enhanced by *reference models* which provide reusable models of human roles, processes, and technologies. The operational use of enterprise models is supported by specific *enterprise modules* which provide prefabricated products such as human skill profiles for specific professions, common business procedures (for instance banking and tax rules), or IT infrastructure services (IFAC/IFIP, 1997).

5.3 CIMOSA

5.3.1 Introduction

The goal of the ESPRIT project AMICE* was to develop an Open System Architecture for CIM (CIMOSA). The project was started in the mid-1980's and finished after some extensions in the mid-1990's. CIMOSA should facilitate continuous enterprise evolution and make necessary operational improvements manageable. At the same time, CIMOSA should provide a strategy to deal with legacy systems in order to protect current and planned investment in an enterprise's production process. CIMOSA aims to offer support for enterprise integration, more precisely for 'business integration' and 'application integration' (AMICE, 1993a).

Business integration is concerned with the coordination of operational and control processes in order to satisfy business objectives. In every enterprise, processes are present that provide supervisory control of the operational processes and that coordinate the everyday execution of activities. CIMOSA aims to integrate these processes by process oriented modelling of enterprises. Modelling these processes and their interrelations can be used in decisions regarding the requested level of business integration.

Application integration, which affects the control of applications, is concerned with the usage of information technology to provide interoperation between manufacturing resources. Cooperation between humans, machines, and software programs has to be established by the supply of information through inter- and intra-system communication. CIMOSA tries to support integration at this level by defining a sufficient infrastructure to permit the system-wide access to all relevant information regardless of where the data reside.

Besides business and application integration, a third level of integration is discerned by the AMICE project, namely physical system integration. This level is concerned with the interconnection of physical systems and has led to a number of standards, such as OSI and MAP. CIMOSA supports this type of integration by adherence to the standards.

* AMICE: European Computer Integrated Manufacturing Architecture (reverse acronym)

5.3.2 CIMOSA approach

CIMOSA provides a framework that guides designers in the design and implementation of CIM systems. In addition, it aims to guide vendors in the development of CIM system components, so that these components can be added and removed at will. It does not provide a standard architecture to be used by the whole manufacturing industry, but rather a reference architecture with which a particular enterprise can derive particular architectures that fulfil its needs. As such, CIMOSA adheres to a descriptive, rather than a prescriptive methodology (AMICE, 1993a). Figure 5-2 gives an overview of the CIMOSA concepts.

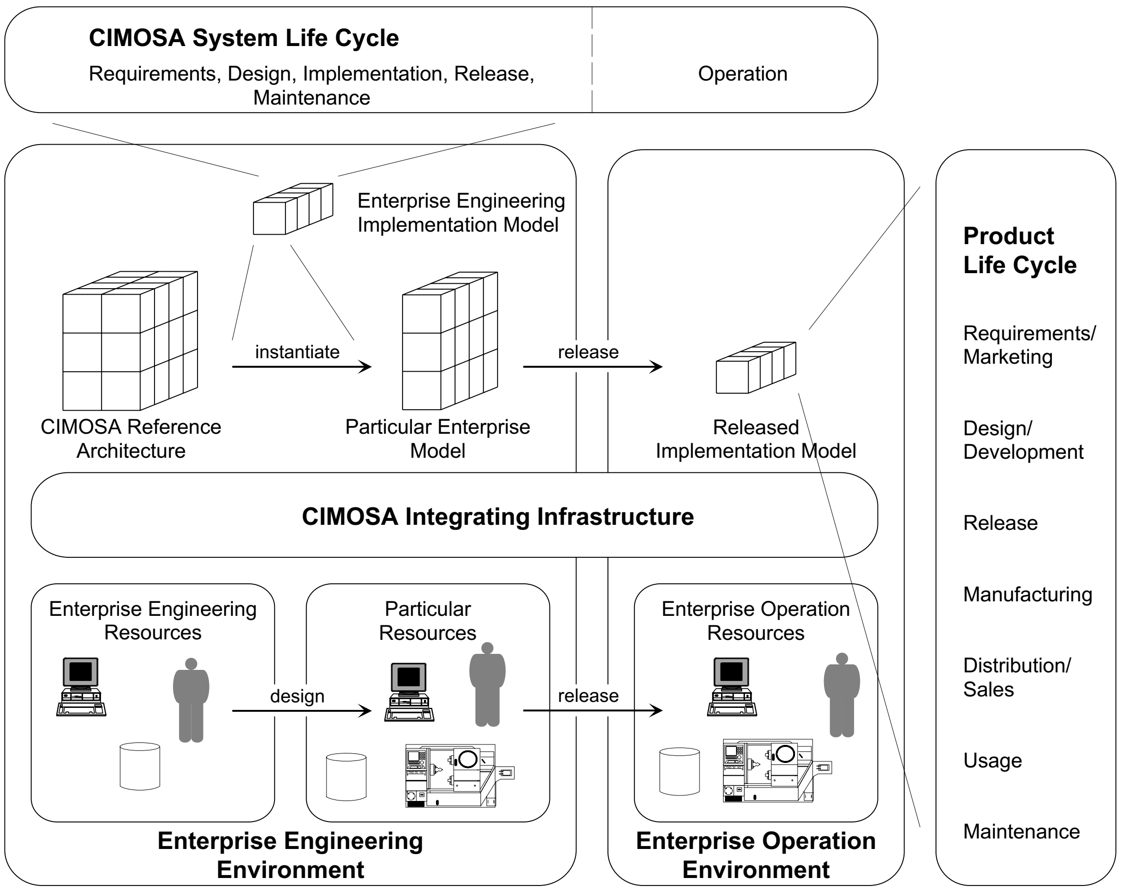

CIMOSA System Life Cycle
Requirements, Design, Implementation, Release, Maintenance
Operation
Enterprise Engineering Implementation Model
Product Life Cycle
Requirements/ Marketing
Design/ Development
Release
Manufacturing
Distribution/ Sales
Usage
Maintenance
instantiate
release
CIMOSA Reference Architecture
Particular Enterprise Model
Released Implementation Model
CIMOSA Integrating Infrastructure
Enterprise Engineering Resources
Particular Resources
Enterprise Operation Resources
design
release
Enterprise Engineering Environment
Enterprise Operation Environment

Figure 5-2 Overview of the CIMOSA concepts

Source: AMICE (1993a)

The CIMOSA *modelling framework* enables one to describe a particular enterprise. Accompanying methods, called ‘model creation processes’, guide engineers in the creation and maintenance process to obtain and maintain a consistent system description. CIMOSA supports the explicit description of enterprise processes at different levels of abstraction for strategic, tactical, and operational decision making. Simulation of alternatives and evaluation of design decisions enhances the decision support. CIMOSA supports incremental modelling of the enterprise rather than following an overall top-down approach. In short, CIMOSA allows the end user to define, to prototype, to design, to implement, and to execute its business processes according to his needs.

CIMOSA facilitates a *system life cycle* which guides the user through model engineering and model execution. The life cycle starts with the collection of business requirements in a requirements definition model, and goes, through the translation of the requirements into a system design model, to the description of the implemented system. These phases are followed by a release of the model for operation and model execution to control and monitor operations. However, various methodologies consisting of various system life cycle phases are possible to instantiate particular models from the reference architecture. These methodologies are supported by tool sets, which are defined by enterprise engineering implementation models.

Model engineering and model execution are supported by the CIMOSA *integrating infrastructure*. This infrastructure provides a set of generic services that process the released implementation model, provide access to information, and connect to resources. In addition, the integrating infrastructure hides the heterogeneity of the underlying manufacturing and information technology.

In the next two subsections, the two most important parts of CIMOSA, namely the CIMOSA modelling framework and the CIMOSA integrating infrastructure are explained in more detail.

5.3.3 CIMOSA modelling framework

In Figure 5-3, the modelling framework is represented by the CIMOSA cube. The cube offers the ability to model different aspects and views of an enterprise (AMICE, 1993a; AMICE, 1993b). This three-dimensional framework has a dimension of genericity, a dimension of enterprise models, and a dimension of views:

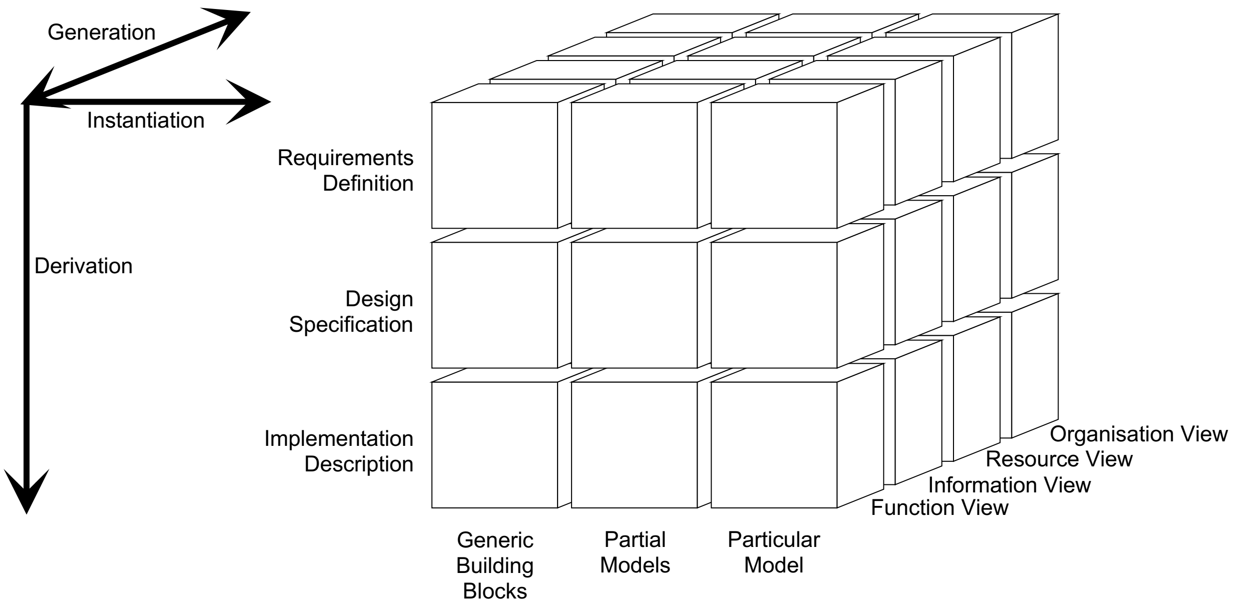


Figure 5-3 CIMOSA modelling framework

- the dimension of *genericity* (and stepwise instantiation) is concerned with the degree of particularisation. It goes from generic building blocks to their aggregation into a model of a specific enterprise domain;
- the dimension of *modelling* (and stepwise derivation) provides the modelling support for the system life cycle, starting from statements of requirements to a description of the system implementation;
- the dimension of *views* (and stepwise generation) offers the possibility to work with sub-models representing different aspects of the enterprise.

5.3.4 CIMOSA integrating infrastructure

The CIMOSA integrating infrastructure enables CIMOSA models to be executed; it allows the control and monitoring of enterprise operations as described in the models. Furthermore, it provides a unifying software platform to achieve integration of heterogeneous hardware and software components of the CIM system. Applications use the generic services of the integrating infrastructure, so that they need no longer contain the specifics of the data processing environment. This provides increased portability and flexibility of the applications and reduces the maintenance tasks considerably.

The integrating infrastructure is made of a number of system-wide, generic services. The business services control the enterprise operations according to the model. Data access, data integration, and data manipulation is provided by the information services. The presentation services act as a standardised interface to humans, machines, and applications. A product that is connected to the presentation services can be attached and removed without changing any other part of the information technology environment. Figure 5-4 displays the integrating infrastructure and the above-mentioned services. Other services are the common services and the system management services, that provide for system-wide data communication and facilities to set up, maintain, and monitor the components of the integrating infrastructure.

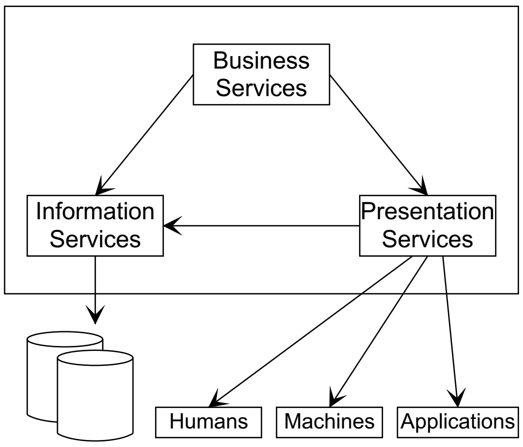


Figure 5-4 CIMOSA integrating infrastructure

5.4 Application of CIMOSA at Traub

The ESPRIT project VOICE* validated the CIMOSA results by the development of manufacturing control and monitoring solutions according to the CIMOSA concepts in three types of industries. The VOICE project constituted of vendors, system integrators, research institutes, and industrial partners. In each of the three industrial pilots, a different aspect of the manufacturing process was addressed. This section focuses on one of these pilots, namely the Traub pilot. Traub's motivation regarding the validation of CIMOSA was to examine whether CIMOSA is a useful tool to support the restructuring process of the existing manufacturing organisation and to support the new organisation with implemented information technology (Gransier and Schönewolf, 1995).

Traub AG is a German manufacturer of machine tools. The products cover conventional machines as well as NC machines for turning and milling. The economic downturn in the market and the world-wide recession forces all machine tool manufacturers to improve their cost situation and shorten the terms of delivery. Customer demands are characterised by an increased diversification of products, resulting in decreasing numbers of series production. Enormous sales efforts and the introduction of new products are not the only guarantee for survival. Enhanced demands to become more flexible and efficient forced Traub to reorganise its production department (Schlotz and Röck, 1995).

Appendix A describes the reorganisation of Traub's production department. In the remainder of this section, the role of CIMOSA during the reorganisation process is summarised.

During the reorganisation of Traub's production department, the role of CIMOSA was twofold: the CIMOSA modelling framework assisted the definition of a functional architecture, whereas the CIMOSA integrating infrastructure supported the design of the technology architecture of the control system's infrastructure.

The CIMOSA models allowed Traub to analyse and re-design its functional control architecture. During architectural design, Traub was able to acquire knowledge of its current manufacturing control system by means of modelling at the requirements definition level. This knowledge was needed in order to analyse the existing system. Then, the model was changed to take required modifications into account. By means of the specification of operational and control functions, their inputs and outputs, and interfaces between functions, a functional architecture was defined.

The definition of a sound functional architecture, upon which the further design of the system is based, is one of the keys to business integration. CIMOSA takes a modular approach to integration, i.e. enterprise operation is modelled as a set of cooperating processes which exchange results and requests. Only this exchanged data needs to have a representation that is common between the cooperating processes. By employing models for identification, analysis, and coordination of existing and new functions, required functional architectures can be designed that define modular components and their interfaces. If the detailed design

* VOICE: Validating OSA in Industrial CIM Environments

and implementation of the CIM system follow the architectural specifications, the requested integration of business processes will be obtained. This way, the required level of business integration is achieved, which is reflected by the coordination of operational and control processes.

CIMOSA aims to support application integration by its integrating infrastructure. Cooperation between humans, software programs, and machines has to be established. Whereas business integration can be regarded as the integration of functional components, application integration affects the integration of technology components. The integrating infrastructure aims to provide services that integrate the enterprise's heterogeneous manufacturing and information systems in order to satisfy the business needs as identified with the help of the modelling framework. The CIMOSA specification of the integrating infrastructure can be seen as a reference model. Components that are designed and implemented according to this reference model should provide desired features such as interoperability, portability, connectivity, and transparency.

Due to the immature specification of most integrating infrastructure services, Traub only considered the communication services. These services are part of the common services. An example of an implementation that fulfils the CIMOSA specifications is the communication platform developed by the Fraunhofer institute IPK. This platform has been developed according to the needs of Traub and the other two industrial VOICE partners, and was based on the communication services of the integrating infrastructure. Lapalus *et al.* (1995) give more information about the communication platform and application integration.

5.5 The practical value of CIMOSA

In this section, the practical value of CIMOSA is central. Its theoretic value is undoubtedly large, since many ideas, developments, and products are inspired by it. In fact, many contemporary products seem to be largely influenced by the CIMOSA ideas. Its value as a means to support the design of flexible systems is evaluated by means of the architecting concepts and principles identified in Chapter 3 and 4 respectively. In addition, CIMOSA's value as a suitable framework for enterprise integration in practice is evaluated by means of the GERAM framework. This part of the evaluation can be found in Appendix B. The conclusion regarding CIMOSA and achievement of model execution is stated in Subsection 5.5.3.

5.5.1 CIMOSA and the architecting concepts

The architecting concepts of hierarchy, views, and domains as distinguished in Chapter 3 are represented in CIMOSA's modelling framework. The functional specifications are *hierarchically decomposed* by means of the domain process, business process, and enterprise activity constructs. Domain processes contain business processes and/or enterprise activities. For example, in the Traub case as described in Appendix A, the domain process 'Order Processing' is decomposed in seven enterprise activities (see Figure A-4). Business processes contain other business processes and/or enterprise activities. Enterprise activities are the leaves of the decomposition tree at the requirements definition level, but can be further decomposed at the design specification level.

The modelling framework provides an open set of four (initially defined) *views* to focus on different enterprise aspects. However, some views are more appropriate for practical usage than others. In the Traub application, Traub practically only used the framework's function and information view; the resource view was barely addressed and the organisation view was not used at all. The organisation view, for example, was seen as a view to manage systems, not to design them.

As for the three *domains*, going from the requirements definition level, via the design specification level, to the implementation description level can be considered as moving from a focus on the functional, via the technology, to the physical domain. However, the level of detail increases as well. In other words, CIMOSA relates a low level in the design process to implementation details (see Figure 3-4, left-hand side). It does not support the specification of a true technology architecture, but only the detailed specification of technology modules. In the design specification level, a relation is made between functions and technology modules. Enterprise activities are decomposed into *functional operations* which are executed by *functional entities*. These functional entities are provided by resources that are chosen on technical grounds.

5.5.2 CIMOSA and the architecting principles

To discuss CIMOSA and the architecting principles as discerned in Chapter 4, a distinction is made between the CIMOSA modelling framework and the integrating infrastructure. The first can be seen as a *layered* framework. Functional specifications are described in the requirements definition level, whereas the physical system is specified by means of the implementation description level. The design specification level provides the mapping between the two previous levels. Clearly, the three levels correspond to three layers. Although layers can be discerned in the framework itself, the modelling framework does not lead to layered systems by necessity.

The modelling framework seems to lead to *modular* systems, but it does not. CIMOSA claims to provide a modular approach to integration. Enterprise operation is modelled as a set of cooperating processes which exchange results and requests, and only the exchanged data needs to have a representation that is common between the cooperating processes. Indeed, the domain processes can be regarded as the functional modules of a system. However, this does not imply that these domain processes are modular. Recall from Chapter 4 that a module, i.e. a subsystem with defined interfaces, does not necessarily have the modularity characteristics of moderate complexity, minimal coupling, and maximum cohesion. CIMOSA does not state how to achieve *modular* domain processes. In other words, users might specify the most bizarre contents of and relations between domain processes. Wrong boundaries between domain processes might be chosen, which would result in rigid and inflexible systems.

The CIMOSA modelling framework does not lead to systems that are constructed according to the architecting principles as discerned in the previous chapter. A similar argument as stated above can be made for *structural stability*. CIMOSA is a descriptive framework; it does not prescribe its users how to design a system. However, the integrating infrastructure does incorporate some design decisions. After all, the specification of the integrating infrastructure can be seen as a reference model for enabling technology, and can be positioned as design specifications at the partial level. However, enabling technologies are not central in this thesis, and therefore the evaluation of the integrating infrastructure regarding the architecting principles is left behind.

5.5.3 CIMOSA and evolution

After time, CIMOSA models produced during an enterprise integration project are almost certain to lose their validity. Enterprise integration is an ongoing process rather than a one-off effort. However, there is little chance for an enterprise to keep an up to date picture of itself as time goes by. Much of the effort initially invested in modelling an enterprise's processes is lost as reality diverges from those models (Bernus *et al.*, 1996). Traub, for instance, foresees a considerable effort in keeping its models consistent with the implementation, also due to the complexity of the CIMOSA modelling framework (Schlotz and Röck, 1995). Note that the consistency aspect is not a CIMOSA related issue, but rather a common problem for all enterprise reference architectures.

In order to remedy this obstacle, enterprise models have to be actually used to drive operation control and monitoring rather than being kept on the shelf. The transition from specification to an operational system requires an infrastructure that supports 'model execution' and therefore has to consist of what CIMOSA calls 'business services'. The realisation of the aims behind the business services as originally conceived might just as well prove to be an illusion for a long time. In 1996, Bernus *et al.* regarded the achievement of business services a goal for the further future.

CIMOSA fails to fulfil the objectives of model execution. The result of the specification phase, i.e. documentation in the form of models, is useful for (onetime) business integration. Without a real integrating infrastructure with services such as the business services, however, it is not sufficient to fulfil the objectives of the enterprise integration philosophy. Applying CIMOSA does not result in operational systems, let alone in flexible, adaptive, efficient systems; the translation from models to a real system still has to be made. Given the fact that it does not offer guidelines and reference models that support designers in the transition from requirements to specification, the practical value of CIMOSA should be merely seen as a framework for the generation of documentation.

Appendix C describes two other enterprise reference architectures, namely GRAI/GIM and PERA. Both provide some solutions to some of CIMOSA's shortcomings. They especially provide well-defined engineering methodologies; GRAI/GIM offers its structured approach, PERA is accompanied by the Purdue Methodology. For enterprise integration, however, it is necessary to keep an up to date picture of an enterprise as time goes by. None of the three reference architectures can guarantee that.

5.6 Baan's Enterprise Modeller

Whereas the previous sections discuss rather theoretic enterprise reference architectures, this section looks at practice. The previous sections show that CIMOSA and other enterprise reference architectures are far from their objective of continuous enterprise evolution. Designers still have to make the translation from models to a real system by themselves. Around 1995, however, some of the concepts of enterprise integration have been implemented in tools for the configuration and implementation of Enterprise Resource Planning (ERP) solutions. For example, the market leader in ERP applications at that time, the German company SAP AG, developed such a tool called 'Business Engineer'. This configuration and implementation tool is based on the R/3 reference model. The R/3 reference model describes business processes that are most commonly needed in practice and that can actually be implemented with SAP's R/3 system (Keller and Detering, 1996). Business Engineer aims to streamline implementation of R/3, and to adapt an existing configuration to new needs or changed circumstances. It allows users to configure their own enterprise model, and to automatically tie the functionality of R/3 into the enterprise model (SAP, 1997).

A commercial product in which some of the concepts of enterprise integration show up is Baan Company's Orgware™. Baan Company recognised that enterprises attempting to implement ERP systems went through a serial process of modelling the enterprise's processes and organisation, and implementation and manual configuration of the actual system. The usual result of this static process is that customer investment in modelling activities has not returned the expected value in the actually implemented system. This is because the statically defined system is no longer in line with the new needs of the business. Therefore, Baan Company developed Orgware, a set of tools and concepts which allows an enterprise to adapt its Baan applications real-time to changing market needs. Even more, it supports a fast implementation and optimisation of a system (Huizinga, 1995).

Dynamic Enterprise Modelling (DEM) concepts comprise the foundation for Orgware. Basically, DEM comes down to developing a model of an enterprise and using this model to adapt the enterprise's processes. In this respect, it does not differ from the objectives of enterprise reference architectures. However, since Orgware is integrated with Baan's ERP product, there is a direct link between the models and their realisation in the Baan software.

The Enterprise Modeller is one of Orgware's main components. Examples of other components are implementation methods and assisting IT services. Originally, a user made three types of models with the Enterprise Modeller, namely business function models, business process models, and organisation models (Huizinga, 1995). Later, a fourth type of models has been added, namely business control models. Figure 5-5 presents an overview of Dynamic Enterprise Modelling with the Enterprise Modeller. From top to bottom, business functions, business processes, and organisation models are pictured.

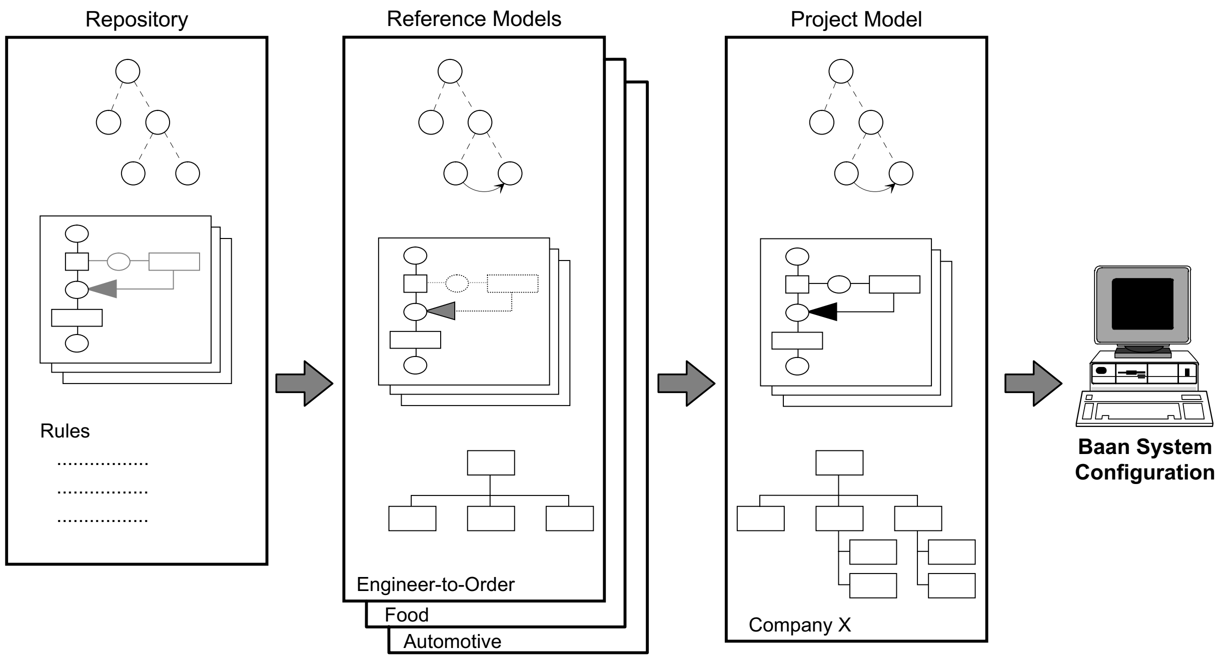


Figure 5-5 Dynamic Enterprise Modelling with the Enterprise Modeller

The *business control model* describes a company's primary process and the manner how it is controlled. A business control model provides modelling teams with a starting point for modelling at a high level. Functions that control the primary process are modelled by constructs of business function models, such as major functions and main functions. The business control model shows the interaction between those functions as well (Boudens, 1997). Note that the business control model is not shown in Figure 5-5.

A *business function model* presents a functional decomposition of business functions, and is the starting point for process selection and configuration. The decomposition tree consists of the company, mega functions, major functions (e.g. Requirements Planning), and main functions (e.g. Material Requirements Planning, and Statistical Inventory Control). Interactions between functions are not modelled. Business function variants are placed beneath the main functions. They are used depending on the phase of the implementation. Optimisation relationships define an optimisation path in an implementation project. Some variants are used after initial implementation (e.g. Statistical Inventory Control); other variants are only used after optimisation phases (e.g. Calculation of SIC Data from History, and Calculation of Economic Order Quantity).

A *business process model* describes formal processes and procedures, and is the basis for configuration of the Baan software. A main process, which is directly related to a main function, is represented by means of a Petri net-like modelling technique. In a business process model, Baan sessions and manual actions are examples of activities. A business process model shows all business function variants of a main function. Depending on the (optimisation) phase of the implementation, variants are used or not ('dimmed' representation).

An *organisation model* is a description of the organisational structure of an enterprise. In a model for a specific company, roles are assigned to persons. A link is made with the business process model, if roles are assigned to (activities of) business processes.

A central feature of the Enterprise Modeller is its use of reference models. In a *repository*, functions, processes, rules, and roles are defined. The rules connect functions and processes to each other. When a new implementation phase starts, the rules state the consequences for the processes. From the components in the repository, *reference models* are assembled that are specifically designed by industry consultants for a particular industry sector. For example, in the reference organisation model for engineer-to-order enterprises, an organisational structure is outlined that is based on the best practices of that type of industry. In the reference business function and process models, optimisation relationships and roles are added respectively (Huizinga, 1995).

Models for a specific enterprise called '*project models*' can be instantiated from the reference models. Phases are added to the project function model, and employees are added to the project organisation model. Subsequently, employees are linked to roles, thereby providing the connection between the process and organisation models. Finally, based on complete project models, a configurator generates user-specific menus and authorisations, and automatically configures the Baan system by setting parameters.

A clear advantage of Baan's Enterprise Modeller compared to enterprise reference architectures is its linkage with the underlying Baan ERP applications. The enterprise models mirror the implementation and configuration of the Baan system. Even more, changes in the models are directly reflected in changes in the Baan system. On the other hand, models made by enterprise reference architectures might reflect the enterprise's current systems and processes, but changes in these models have to be 'manually translated' to changes of the modelled processes and applications.

The Enterprise Modeller's advantage of being linked with the underlying Baan ERP applications is also its disadvantage; it covers only Baan products. It allows one to model manual procedures and applications by other vendors besides Baan sessions. Clearly, it cannot automatically configure these other applications. Although ERP products tend to cover more and more functionality, they do not comprehend the processes of an entire enterprise. Shop floor control processes, for instance, were not covered when this dissertation was written.

In addition, the Enterprise Modeller does not model a company's IT infrastructure; for example, it cannot support distributed Baan applications. It only models the functional domain, not the technology or physical domain.

Nevertheless, a product such as Orgware is a major improvement for the implementation of ERP applications, and as such it provides a contribution to the realisation of the enterprise integration philosophy. For the objectives of enterprise integration to become true, however, more is needed than that: model executability by an infrastructure with real business and presentation services.

5.7 Summary

Reference architectures for enterprise integration aim to provide the necessary frameworks with which companies might adapt their operation due to changing internal and external conditions. The reference architecture CIMOSA strives for the facilitation of continuous enterprise evolution. It intends to offer support for business integration by means of its modelling framework, and for application integration by means of its integrating infrastructure. CIMOSA was applied during a reorganisation project at Traub AG, a German tool manufacturer. The modelling framework assisted Traub in the definition of a functional control architecture. Due to immature specification of the integrating infrastructure, only the specification of the communication services was helpful to Traub.

The architecting concepts of domains, hierarchy, and views are represented in CIMOSA's modelling framework. The modelling framework supports designers during the specification and analysis of functional architectures of production control systems, but it does not support the specification of a true technology architecture.

The architecting principles of modularity, structural stability, and layers are not incorporated in the CIMOSA modelling framework. CIMOSA does not prescribe its users how to design a system; it is a descriptive framework.

The CIMOSA reference architecture does not cover a full life cycle and lacks an accompanying engineering methodology. However, CIMOSA does provide an eligible, though quite complex, framework for the specification and analysis of functional architectures for production control systems. It prescribes how to make a specification of a system, but it does not prescribe how to design the system. In addition, a designer has to make the translation from models to a real system himself. Therefore, CIMOSA should be merely seen as a framework for the generation of documentation.

A product which ensures up to date models of an enterprise is Baan Company's Enterprise Modeller. Based on a repository and reference models, a designer makes models of its enterprise. Since the Enterprise Modeller is integrated with the Baan applications, it automatically configures these applications based on the models. Changes in the models are immediately reflected by changes in the configuration of the applications.

6. Reference Models for Shop Floor Control

6.1 Introduction

The objective of this chapter is to discuss the suitability of reference models for the design of flexible shop floor control systems. Chapter 5 states that CIMOSA is merely a framework for the generation of specifications. It does not provide assistance in the architectural choices that have to be made. Other enterprise reference architectures do not provide domain specific support either. Reference models for shop floor control do provide application domain specific support. However, the question arises whether these reference models lead to flexible shop floor control systems. In this chapter, various reference models and their impact on the flexibility of shop floor control systems are examined by means of the architecting principles of Chapter 4.

In Section 6.2, an important property of dynamic systems and a central point of this chapter, namely control (and coordination), is introduced. This section discusses various control levels that might be present in a manufacturing company. Furthermore, it presents the close relation between control applications and enabling technology.

Section 6.3 gives an overview of the evolution of research into control architectures. It shows that advances in enabling technologies allowed a gradual transition from centralised control forms to more distributed forms.

The following sections present a basic control form, and an evaluation of the control form on the basis of the architecting principles of modularity, structural stability, and layers. Examples of reference models of the basic forms are given in Appendix D.

Section 6.4 starts with the first basic control form, namely the proper hierarchical control form. This form is characterised by vertical master-slave relations, a pyramidal structure, and the absence of horizontal peer-to-peer control relations. Whereas layers are present, modularity is usually low, and structural stability is absent. The ideas developed by the ESPRIT project COSIMA is an example of a reference model of the proper hierarchical form.

The heterarchical control form in Section 6.5 is characterised by the absence of master-slave relations, a flat structure, and the presence of horizontal peer-to-peer control relations. No reference models are known for this form.

The proper hierarchical control form and the heterarchical form are two extreme basic architectural forms. In addition, the modified hierarchical form and holonic manufacturing systems are discussed. Section 6.6 considers the modified hierarchical form as a hybrid form, since it combines hierarchical master/slave relations and heterarchical peer-to-peer relations. Holonic manufacturing systems are able to exhibit all kinds of control behaviour that range from proper hierarchical to heterarchical control. Holonic manufacturing systems are presented in Section 6.7.

6.2 Control architecture and enabling technology

6.2.1 Control levels and granularity of resources

This chapter focuses on control in shop floor control systems. In contrast to static systems, such as houses, dynamic systems need to be controlled. In some dynamic systems, such as the Baan applications, control is a minor aspect; they only need to be started, and do not really have to be controlled. However, manufacturing shop floors have to be controlled. Control is the raison-d'être for shop floor control systems. The behaviours of the individual subsystems on the shop floor need to be coordinated in order to achieve overall system performance.

In general, a number of control responsibilities can be discerned in production control systems. These responsibilities are grouped in control levels, which at their turn are derived from the *spatial decomposition* of the shop floor into collections of one or more pieces of equipment. Traub, for example, distinguishes the following control levels:

- production planning level, responsible for rough planning of orders on machine groups;
- area control level, responsible for machine-oriented fine-planning, order progress, and tool management;
- machine group (or cell) control level, responsible for execution of orders from area control, monitoring of the shop floor operations, and the collection of machine and operation data.
- machine level, responsible for the control of an individual machine (see Appendix A and (Schlotz and Röck (1995)).

Another example, the AMRF Reference Model by the US National Bureau of Standards distinguishes the following five levels (see Figure 6-1, from top to bottom):

The facility is the highest level in the structure and comprises of three major sub-systems: manufacturing engineering, information management, and production management. The shop is responsible for the real time management of jobs and resources on the shop floor and achieves this through two major modules, namely those of task management and resource management. The cell level manages the sequencing of batches and materials handling facilities. The workstation directs and coordinates a set of equipment on the shop floor. A workstation consists of a set of equipment set up to realise a particular task; a typical workstation might consist of a machine tool, a robot, a material handling buffer, and a control computer. The equipment controllers are linked directly to individual pieces of equipment within a workstation (Albus *et al.*, 1981; Simpson *et al.*, 1982; Jones and McLean, 1986).

Besides the AMRF reference model, several other control models have been developed to date. The most well-known of these models are the CAM Reference Model developed by Philips and Digital Equipment Corporation (Philips CFT, 1987), the Manufacturing Planning and Control Systems reference model (Biemans, 1989), and the Factory Automation Model (Graefe and Thomson, 1989). Figure 6-1 shows the production control levels as distinguished by the four reference models and by Traub. For more information on control levels in reference models for production control, the reader is referred to (Micklei, 1993; Arentsen, 1995).

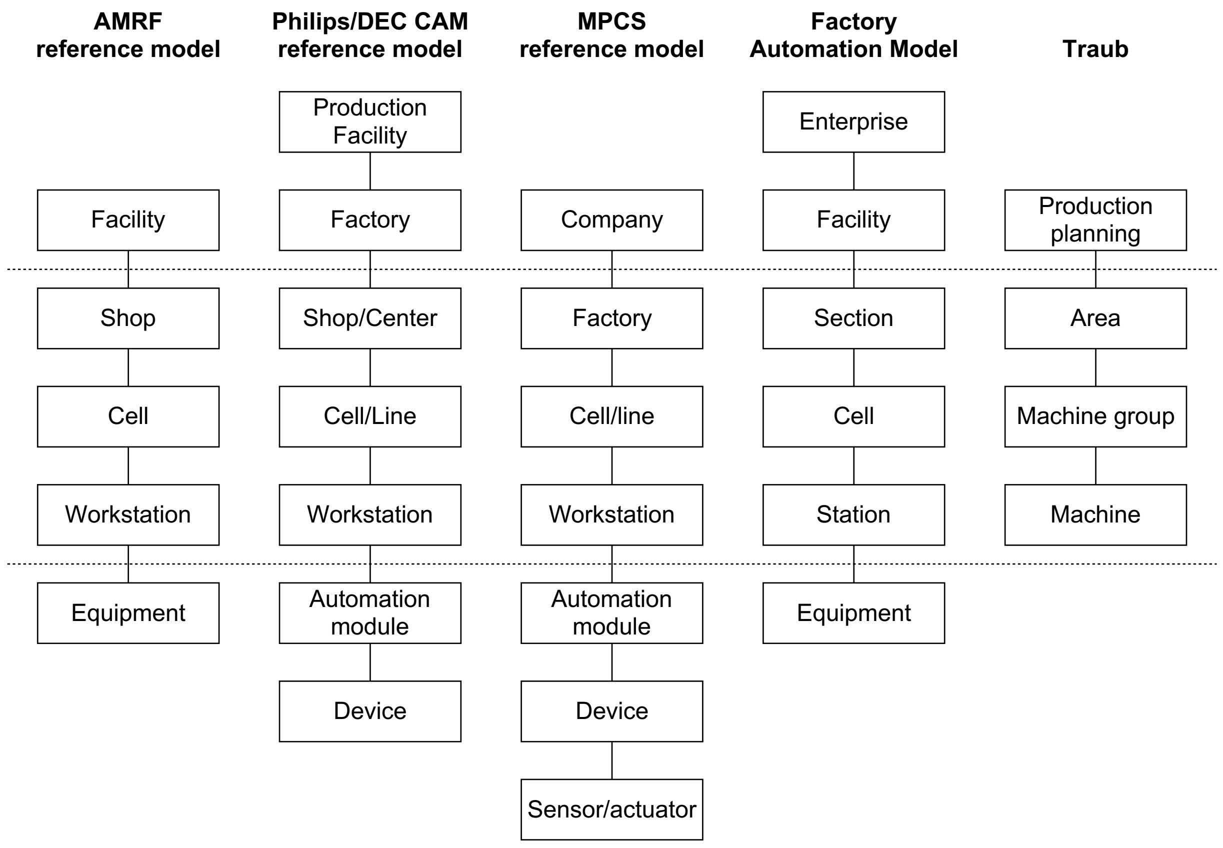


Figure 6-1 Reference models for production control

Adapted from Plasschaert (1996)

Although different reference models propose a different number of control levels, the various reference models in Figure 6-1 discern more or less the same control levels. After all, they are based on the same groupings of resources as seen in factories. This thesis regards only the shop level, the cell or line level, and the workstation level. Higher control levels are the responsibility of production planning functions; lower levels are the responsibility of equipment control functions. The needed shop floor control functionality is defined by the shop, cell/line, and workstation control levels.

Section 6.3 presents the evolution of architectural forms that structure the functionality in the control levels as described above. First, however, the relation between the defined control levels and accompanying infrastructure services is discussed.

6.2.2 Control levels and enabling technology

If certain technologies are chosen to implement certain functions, some additional technology might be needed to enable a proper functioning of the first technologies. This is illustrated by means of an example. Consider a shop floor control system where a function ‘scheduling manufacturing cell’ is defined. This function is a component of the functional architecture of the shop floor control system. The functional architecture defines other components and the interrelations between those components and ‘scheduling manufacturing cell’ as well.

Components of the functional architecture are mapped to components of the technology architecture. The function ‘scheduling manufacturing cell’ could be carried out by a human scheduler supported by Gantt charts. Then, a choice is made for the technology ‘human’. One could also choose for software technology and implement the scheduling function in software. Such a choice would demand several other choices for enabling technologies. Software needs hardware to be executed upon; choices have to be made for computer technology, database technology, network technology, and so on.

It is important to make a distinction between the technology architecture of a shop floor control system and the enabling technology offered by an infrastructure upon which an implementation resides. Furthermore, it should be clear that the (technology architecture of an) infrastructure is not the realisation of the functional architecture of a shop floor control system in the technology domain (or in the physical domain for that matter). Rather, this infrastructure *enables* the realisation of the functional architecture, for instance by providing the platform for software to run.

Figure 6-2 shows the relationship between control applications and their accompanying infrastructure. The control system’s technology architecture implements its functional architecture. Due to choices for certain technologies, infrastructure components might be needed; the control applications are enabled by the infrastructure. This relation between control applications and infrastructure reveals itself most at the technology domain, since the need for infrastructure components becomes apparent in this domain.

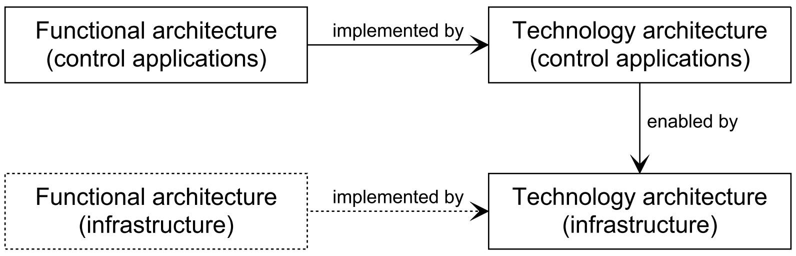

Functional architecture (control applications)
implemented by
Technology architecture (control applications)
enabled by
Functional architecture (infrastructure)
implemented by
Technology architecture (infrastructure)

Figure 6-2 Control applications and infrastructure

Historically speaking, shop floor control applications have been interwoven with components of the information technology infrastructure. Rather, there has been a fixed mapping between the functional control architecture and the technology architecture of the IT infrastructure. Certain shop floor control functionality has traditionally been implemented on certain enabling infrastructure components. For example, Traub implemented its production planning, area, machine group, and machine level control functionality on an IBM mainframe, IBM RS6000, 486-PC’s, and machine specific control equipment respectively. A frequently occurring mapping between the control levels as defined in the previous subsection and enabling technology components is given in Table 6-I, which is based on the AMRF Reference Model by the former US National Bureau of Standards.

A control system’s functional architecture and infrastructure should be decided upon separately, i.e. design decisions about either one of them should not be influenced by the

Table 6-I Mapping between control levels and enabling technology

Control applications		Enabling technology
Functional domain	Technology domain	Technology domain
Facility	ERP applications	Mainframe
Shop	Shop control software	Mini computer
Cell	Cell control software	PC
Workstation	Workstation control software	PLC / Machine-specific control hardware
Equipment	Device-specific control logic	Device-specific hardware

other. Above, the historical entanglement between functional control architectures and enabling infrastructure components is illustrated. Until the 1990's, there was a common mapping between the functional control architecture and the technology architecture of the enabling infrastructure. At the end of the 1990's, infrastructure technology had reached such a level that it did not restrict the implementation of all kinds of functional control architectures anymore; since then, there has been more and more freedom in defining functional control architectures.

Various infrastructural configurations can be used to enable the proper functioning of control applications. Figure 6-3 shows various options regarding the infrastructure for Traub's testbed. Appendix A explains this testbed's hierarchical control architecture, which is the

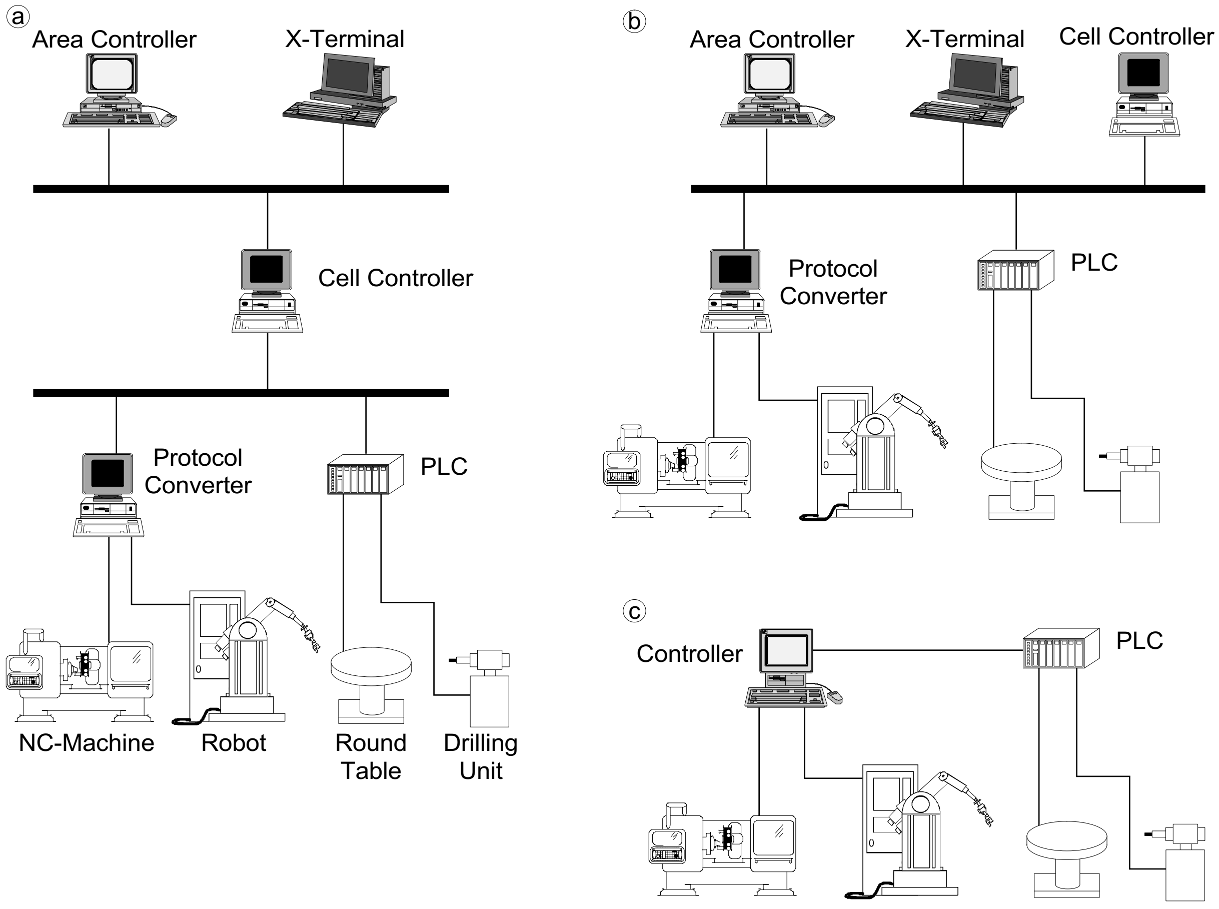


Figure 6-3 Various infrastructural configurations for Traub's testbed

same for all three infrastructural configurations. The left-hand side of the figure shows the original structure as outlined by Traub's partner FhG-IPK. This configuration clearly resembles the hierarchical control architecture. However, the functional control architecture and the technology architecture of the infrastructure do not need to have similar forms. Another option would be to use only one network to connect the various computing resources (Figure 6-3b, top right-hand side). This configuration seems to correspond to a more distributed control architecture. Finally, one computer could be used to enable the execution of software for various control functionalities, such as area control and cell control (Figure 6-3c, bottom right-hand side).

The question arises how to make decisions regarding the technology architecture of the infrastructure. Since infrastructure is not the topic of this thesis, only the considerations of costs, compatibility, and scalability are given. Firstly, companies regard costs as a major issue; if they have existing computing and communication equipment available, they will be less inclined to invest their money in new equipment (no matter how old the equipment is). Then, there are issues of compatibility, e.g. regarding execution of software on various platforms or protocols between computing and manufacturing equipment. Finally, scalability concerns issues such as whether the same hardware can be properly used when the number of manufacturing resources increases. For more information on decision making about infrastructures, the reader is referred to Renkema (1996) who takes a broader perspective on infrastructures and who gives guidelines for investments in information infrastructures.

This section shows that control levels have been interwoven with levels in enabling computing technology. The next section presents the evolution of research in control architectures at the end of the twentieth century which was accompanied by an evolution of enabling technology.

6.3 Evolution of control architectures

From the 1970's to the 1990's, an evolution is noted in research in control architectures. Technological advances in computing and communication technology have made it possible to consider a wide range of possibilities in the design of control architectures. This rapid growth in technology in addition to an increasingly demanding manufacturing environment has been accompanied by an evolutionary growth in control architectures. Figure 6-4 shows the evolution of research in control architectures, which is characterised by an increase in the autonomy of the controllers, a reduction of the use of aggregated information, and a relaxation of master-slave relations (Dilts *et al.*, 1991). Note that Figure 6-4 displays a research evolution, rather than an evolution of the application of control architectures in industry. At the end of the 1990's, the heterarchical form was still hardly applied in industry, although it was extensively studied by universities and research institutes.

Centralised form

Early control architectures employed a centralised approach. Before this approach, control types were based on the primary process and without much supervisory control. The conventional approach at that time was decentralistic in nature (Meal, 1984;

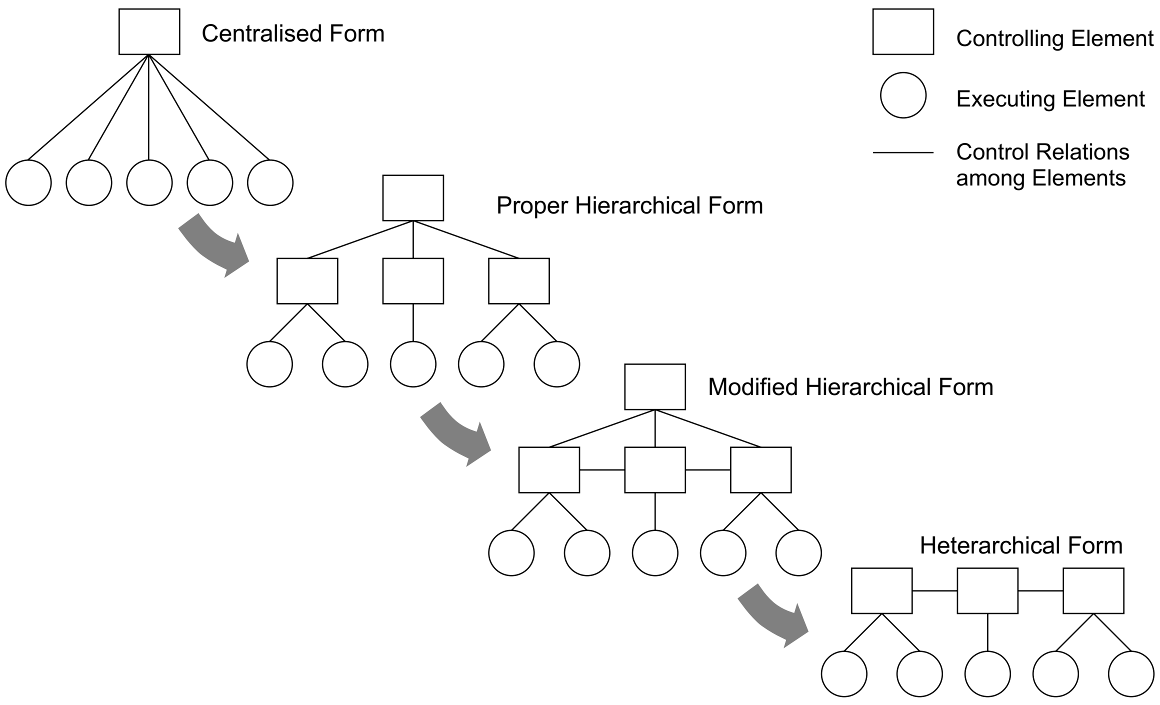


Figure 6-4 Evolution of research in control forms

Source: Dilts *et al.* (1991)

Timmermans, 1993a). In the 1970's, industry became aware of the possibilities of information technology, and decided to put all control logic in a central point. In the centralised form, shop-level and cell-level responsibilities are concentrated in a single software application, which is usually executed on a powerful mainframe. Simple workstation controllers are dispersed throughout the factory. The central controller issues commands in order to coordinate the manufacturing process, and it receives monitoring information from the workstation controllers to make global control decisions.

Proper hierarchical form

After the centralised form, the proper hierarchical form emerged. The dependence on a single controller and the difficulty to make modifications or extensions required that designers considered other options in the development of control architectures. In order to reduce the complexity of the centralised form, control functionality was distributed over several controllers that were organised in a hierarchy. Hierarchical control architectures were developed primarily because designers had been conditioned through training and experience to approach complex system designs from a hierarchical control standpoint. After all, many organisations are designed according to the hierarchical paradigm.

Modified hierarchical form

Galbraith (1973) notices that hierarchies may become overloaded as more exceptions are referred upward. As the organisation's subtasks increase in uncertainty, more exceptions arise which must be referred upward in the hierarchy. Local controllers typically do not have sophisticated capabilities, whereas higher level controllers do not have access to detailed information. Due to each controller's finite capacity for handling information, serious delays may develop between the upward transmission of information about new situations and a response to that information downward. In this situation, the organisation must develop new processes to supplement rules and hierarchy. Besides goal setting, Galbraith proposes four design strategies that aim to reduce the number of exceptional cases referred upward into the organisation through hierarchical channels (see Figure 6-5).

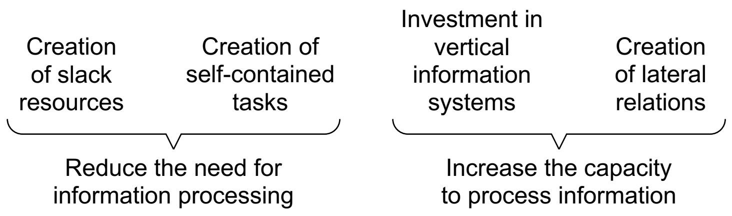

Creation of slack resources
Creation of self-contained tasks
Investment in vertical information systems
Creation of lateral relations
Reduce the need for information processing
Increase the capacity to process information

Figure 6-5 Organisation design strategies

Source: Galbraith (1973)

The modified hierarchical form results from the employment of lateral decision processes. This strategy moves the level of decision making authority down to where the information exists rather than bringing it up to the points of decision. The use of vertical master/slave relations and horizontal peer-to-peer relations characterise the modified hierarchical form. It is enabled by technological advancement in the area of distributed computing and the rapidly declining price/performance ratio of hardware.

Heterarchical form

The heterarchical control form is characterised by coordination among controllers that is purely done by lateral relations. A heterarchical control architecture creates a flat architecture that divides control responsibilities among cooperating controllers. It is an attempt to overcome the disadvantages associated with each of the previous control forms (Dilts *et al.*, 1991). The heterarchical control form shows that control levels are separated from controllers. In other words, the necessity of some form of coordination between components (such as individual workstations) is recognised, but the need for a higher level controller (such as a cell/line controller) by definition is rejected. A cell/line controller is not needed if the coordination between the individual workstations is done by the workstations themselves.

Up to this point, this chapter describes the relation between control applications and enabling technology, and the evolution of both. History has shown an entanglement between control forms and enabling information and communication technology. The rest of this chapter discusses the various functional control forms, and their impact on the flexibility of shop floor control systems. The interested reader may find more details in (Dilts *et al.*, 1991; Solberg and Heim, 1989; Duffie *et al.*, 1988; Duffie, 1990; Smit, 1992; Veeramani *et al.*, 1993; Duffie and Prabhu, 1996). It should be noted, however, that most authors have studied the heterarchical control concepts. Most of them have taken the disadvantages of the hierarchical control form and the promises of heterarchical control architectures as their points of departure.

To illustrate the various control architectures, an example is used, which is based on the flexible assembly system of the Catholic University of Leuven, Belgium. This assembly system comprises four assembly robots and a transport system. The latter consists of a central loop and a buffer at each station. Besides the workstations and the transport system (which is ignored in this example), a scheduler, a controller, and order modules make up the control system. Figure 6-6 shows the various components. Order modules are components that are responsible for the execution of a certain order. The scheduler defines schedules in which operations of a certain order are planned and allocated to the four workstations. The workstations present themselves to the controller, requesting new operations to carry out. Similarly, order modules present themselves to the controller, offering the next operations to be carried out. Finally, the controller may act as a dispatcher or as a broker; it may allocate the operations of the orders to the workstations, or it may provide the place where orders and workstations meet to allocate operations themselves (Valckenaers *et al.*, 1994; Bongaerts *et al.*, 1995; Bongaerts *et al.*, 1997).*

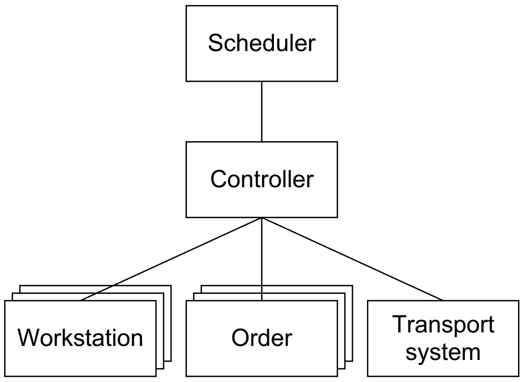


Figure 6-6 Various components of the flexible assembly cell

Note that in Figure 6-6, order modules are separate entities with own intelligence. In contrast, in Figure 6-4 orders or jobs are passive entities; controllers command or negotiate with other controllers about jobs. The separate order modules do not influence the forms possible to control the flexible assembly cell.

The next sections present the basic control forms of Figure 6-4 (except the centralised form, and with holonic manufacturing systems). In addition, their suitability to lead to flexible shop floor control systems is evaluated by means of the architecting principles of modularity, structural stability, and layers. Examples of reference models that correspond to the control forms are given in Appendix D.

* Parts of this chapter have been published in an article by Zwegers *et al.* (1997c). The author wishes to thank the Catholic University of Leuven for their cooperation in this work.

6.4 Proper hierarchical control

Before the proper hierarchical control concept is discussed, a distinction should be made between two properties: *hierarchy* and *stratification*. Van Aken (1978) notices that next to *hierarchy*, the parts-within-parts or boxes-within-boxes structure, organisational structures usually have a second property, namely *stratification*, the bosses-above-bosses structure. He defines a stratified system as follows:

> ‘A *stratified system* is a system the elements of which are ordered, individually or combined to subsets, according to a given priority criterion’ (Van Aken, 1978; p. 35). For example, Traub’s manufacturing cells consist of a cell controller and various machines. The former is responsible for the control of the latter.

Van Aken recognises that organisational structures usually involve a combination of stratification and hierarchy. In such cases, the structure can be described as a stratified hierarchical system, for which he gives the following definition:

> ‘A *stratified hierarchical system* is a hierarchical system having at each level one or more subsystems which have priority over the other subsystems at that level’ (Van Aken, 1978; p. 36). In Traub’s big part manufacturing hall, for instance, an area controller and various machine groups (cells) can be discerned. The area controller controls the cells, which at their turn constitute of cell controllers and various machines.

Hierarchy and stratification form two distinct design issues, which should not be confused. Hierarchy can be used to cope with complexity; stratification can be used to cope with conflicts between subsystem interests and the interests of the system as a whole (Van Aken, 1978).

In this chapter, the terminology as in (Dilts *et al.*, 1991) is adopted. Where the words ‘hierarchy’ or ‘control hierarchy’ are used, Van Aken’s stratification or bosses-above-bosses structure is meant. Note that in Chapter 3, ‘hierarchy’ and ‘decomposition hierarchy’ stand for the boxes-within-boxes structure.

6.4.1 Characteristics of the proper hierarchical control form

In a proper hierarchical control system, a specific controller dictates all activities of the subordinate level. The subordinates – whether they are production modules or lower level controllers – are not allowed to refuse the commands from the upper level controller. Controllers at each level make decisions based on commands received from the level above, and feedback received from the level below. Master/slave relationships are created between levels with command information flowing ‘downward’ through the hierarchy and feedback information flowing ‘upward’ through the hierarchy (Duffie, 1988). Note that there is a distinction between control paths and communication paths (Jones and Saleh, 1990). In the proper hierarchical form, peer-to-peer communication is allowed for the transfer of data, but not for commanding peer controllers.

A configuration of hierarchical controllers is characterised by a philosophy of ‘control levels’ and contains several control modules arranged in a pyramidal structure. Each level has its own purpose and function. At the top of the hierarchy is a single controller which is responsible for setting global goals and formulating long-range strategies. These strategies commit the entire hierarchical structure to coordinated actions which would result in achievement of the selected goals. Aggregate decisions are made at the highest levels. These decisions are decomposed into more detailed commands and passed on to the next lower level in the hierarchy. Detail of information increases with each lower control level, whereas the time period for its consideration decreases. Classical examples of a proper hierarchical control system are the army and the Roman Catholic Church.

A proper hierarchical control strategy in the example of the flexible assembly system of the University of Leuven functions as follows. The controller follows exactly the sequence of tasks as prescribed by the scheduler. If necessary, the controller waits until the planned workstations become available or until an operator has restored the system such that it can continue along the prescribed schedule. The workstations and the order modules are obliged to report to the controller, and to wait for permission to continue.

6.4.2 Evaluation

Compared to the centralised control architecture, the distribution of the control tasks over various *layers* reduces the functionality and complexity of an individual controller. Therefore, large amounts of data can be handled efficiently; fast response times are achieved, since each (controller within a) layer has its own tasks. In normal, stable circumstances the control system operates effectively and efficiently, resulting in near-optimal performance. Adaptive behaviour may be obtained since the status information from subordinates can be used to close a control loop in a controller. Another advantage of proper hierarchical control is the possibility to incrementally add vertical slices of the control architecture. For example, new workstations may easily be added to existing cells (Dilts *et al.*, 1991; Duffie *et al.*, 1988). Note that adding vertical slices means adding new modules to existing layers, and that interfaces between layers remain unchanged. On the other hand, it is very hard to add an extra layer, since interfaces between layers have to be changed.

However, Conant shows that ‘the requirements on a system for *selection* of appropriate information (and therefore blockage of irrelevant information), internal *coordination* of parts, and *throughput* are essentially additive and therefore compete for the computational resources of the system’ (Conant, 1976; p. 240). Hierarchical controllers are inclined to spend a large part of their computational resources in the selection of appropriate information and passing it to other controllers. According to Conant, this is at the expense of the true function of the system, the throughput. The result is an information overload and bad overall system performance. Galbraith (1973) reaches a similar conclusion for hierarchically structured organisations.

Some disadvantages of the proper hierarchical control concept relate to changing conditions. The hierarchy is unable to flexibly handle changes as a result of rush orders, machine breakdowns, or other disturbances. It often takes (too) long before the new circumstances are known at the right controllers. If a workstation in a line breaks down, workstations downstream are waiting for input (starvation), and workstations upstream are blocked. Individual workstations are not *structurally stable*; they depend on other components to make decisions for them. Without those orders, the workstations become helpless.

Moreover, the hierarchy is hardly modifiable due to lack of *modularity*. Entanglements between the various controllers are especially created by fault-tolerance measures. If a controller breaks down, its subordinates are also down, unless extensive fault-tolerance measures have been taken. In that case, a common solution is to give a controller at a given level in the hierarchy substantial knowledge of the controller above it in the hierarchy as well as the controllers below it. If a controller has to be changed, its neighbours have to be changed as well. Even more, it is quite likely that also the neighbours of the neighbours have to be modified. Changes propagate through the system.

Hatvany (1985) and Solberg and Heim (1989) claim that hierarchical systems are ‘resistant to evolution’ due to their structural rigidity. A hierarchical system requires relatively complete design prior to implementation and, once completed, is not easily changed. In most cases, there is no single, obviously correct decomposition of authority into units, so it is quite likely that any first attempts could be improved upon if modifications were not so difficult because of absence of *modularity*. Duffie and Piper conclude that ‘the complexity of computer integrated manufacturing systems with hierarchical architectures grows rapidly with size, resulting in accompanying high costs of development, maintenance, operation, and modification’ (Duffie and Piper, 1986; p. 137).

6.5 Heterarchical control

6.5.1 Characteristics of the heterarchical control form

In a heterarchical control system, distributed, autonomous subsystems communicate with each other without the master/slave relationship. Full local autonomy and a cooperative approach to decision making are the main features. Supervisory decision making is located locally at the point of information gathering rather than in a central location.

Cooperation between subsystems is arranged via a cooperation protocol. This protocol is often called a negotiation procedure, although there is no real negotiation involved, but only allocations based on a set of simple rules (Van Brussel, 1995). Each subsystem must conform to certain rules, in order to obtain certain privileges. Hatvany states that for a heterarchical system ‘to function in a manner that permits the maximum of autonomy and flexibility to its members, great care must be taken first to establish and codify these rules, whose observance is absolutely mandatory’ (Hatvany, 1985; p. 104).

Each subsystem has a dual set of goals: one related to its internal behaviour, the other related to the operation of the overall system. The latter should be dominant, thereby assuring that the subsystem cooperates with other subsystems to arrange operations such as scheduling and routing of work parts. However, to obtain full autonomy, subsystems are allowed to refuse requests from other subsystems based on their own status. A classical example of heterarchical control is a cattle market, where suppliers and demanders fully autonomously try to achieve their goals.

A heterarchical control strategy in the flexible assembly system of the Catholic University of Leuven is as follows. Order modules request the execution of operations. Workstations respond to these requests by offering bids. In a bid, a ‘price’ is mentioned that is determined by factors such as operation costs, lead time, and quality. Then, a negotiation process takes place between order modules and candidate workstations. Order modules choose the workstation that issued the best ‘price’. A schedule results from this negotiation process. In most cases, this takes place opportunistically by scheduling the next operation when the previous one is finished. The controller acts as a broker, where supply (orders) and demand (workstations) of operations meet.

6.5.2 Evaluation

Heterarchical control systems are supposed to be more robust, adaptable, and extensible than hierarchical control systems. This is only true if the heterarchical control system is properly organised. Officially, a heterarchical control system is a control system without hierarchical relations. This means that the possible rigidity of the hierarchy is absent. However, the control system might still be rigid, for instance if each individual controller only knows and negotiates with its successor and predecessor in the line or has no negotiation capabilities to handle disturbances. Then, the system is heterarchical, but not robust. If a controller breaks down, the whole line is down. A controller does not know what to do if a neighbour breaks down; it is not *structurally stable*.

Another example is a situation where each workstation knows the operational capabilities of each other workstation. This way, each controller knows which workstations are able to execute the next operation. If a workstation breaks down, other workstations know the alternatives; the overall system is robust. However, if a new workstation is added to the manufacturing system, all workstations have to be updated with information about the capabilities of the new workstation. In this case, the control system is not *modular*.

However, such information as described above does not have to be incorporated in a component. It can also be made explicit. For example, suppose a workstation reports to a central component that it finished operation on a job, and that it wants a new job. The workstation provides enough information to the central component, so that the latter knows what type of operation is wanted next to complete a job, and what type of operation the workstation wants next. The central component matches demands with offers of operations, and acts as a broker. Note that workstations do not have information about other workstations anymore. The key is to remove such type of global information from individual subsystems; by doing so, the system becomes more *modular*.

An advantage of a heterarchical control system with properly implemented negotiation capabilities is its flexible reaction to changing conditions. Schedules and routings are not fixed but are opportunistically established. A control system with autonomously functioning components that coordinate their operations by negotiation, should be able to function as usual if a component fails. In addition, the implicit fault-tolerance and the reduced coupling among control components leads to reduced control software complexity. For example, Kompass (1993) reports a reduction from 500 to four print pages of control software for robotised paint booths, when a General Motors plant shifted from a deterministic scheduling approach to a heterarchical, negotiation-based approach.

Moreover, the improved modularity leads to higher maintainability and modifiability of individual components. However, reconfigurability and adaptability of the overall system by modification of the negotiation protocol is hard. The negotiation protocol is incorporated in each workstation controller. Every workstation knows the negotiation protocol. Changes in the negotiation protocol require changes in possibly every module.

The main disadvantage of the heterarchical control concept is that normal operation may not be effective and efficient; individual controllers do not have an overview of the complete process and are likely to make suboptimal decisions. In addition, Prabhu and Duffie state that 'systems consisting of highly autonomous entities cooperating through communication without master-slave relationships and with minimal global information have been noted to be seemingly chaotic and sometimes unstable from a controls point of view' (Prabhu and Duffie, 1995; p. 425).

Another frequently quoted disadvantage is that lead times and due dates are quite unpredictable, since schedules and routings are not known when a work order is released. When an order is launched, the system cannot assure when it will finish (Van Brussel *et al.*, 1993). Arentsen concludes that 'because there is no higher level control, an off-line scheduling function may principally not be implemented within a heterarchical control system. As a consequence of the fact that there is no reference against which the order progress can be monitored, it is impossible to deal with inaccuracies and disturbances in an adequate way. The heterarchical control system is not able to predict whether the due dates of the orders can (still) be met, nor is it possible to control adequately the execution of the manufacturing tasks' (Arentsen, 1995; p. 22). However, this is not a valid argument. Central, off-line schedulers may be present in a heterarchical control system, as long as they are used to provide reference information rather than to control the operations.

Both the proper hierarchical and the heterarchical control architecture are two extremes on the wide spectrum of possible control architectures. If master/slave relations are taken to the extreme, the result is a proper hierarchical control architecture. Similarly, only peer-to-peer negotiations lead to a heterarchical control architecture. If master/slave and peer-to-peer relations are combined in one control architecture, a hybrid form occurs.

6.6 Modified hierarchical control

6.6.1 Characteristics of the modified hierarchical control form

A hybrid control model is the modified hierarchical control architecture as described by Dilts *et al.* (1991).* Just like in the proper hierarchical control form, supervisor/subordinate relations exist between controllers. However, controllers in the modified form are equipped with a certain degree of autonomy with respect to higher level controllers. This relative autonomy loosens the master/slave relationships between controllers; a controller acts as an intelligent assistant to the host and not as a slave.

An example of the modified hierarchical control form is a situation where a higher level controller passes an order to a controller. Besides the order, additional information is given to the controller, with which it can cooperate with peer controllers in order to carry out a sequence of activities to complete the job. Exceptions are referred upward in the hierarchy where appropriate actions are taken. This is the control strategy that is usually associated with the modified hierarchical form. Other strategies, however, are also possible in this form, as long as there is a combination of master/slave and peer-to-peer relations.

6.6.2 Evaluation

As with all basic forms, the *modularity* and *structural stability* of the modified hierarchical form depend largely on the specific control strategy chosen. If the prevailing interpretation is chosen, the advantages and disadvantages are roughly equal to those of the proper hierarchical form. In case the system has to react to changed circumstances, it still uses supervisor/subordinate relationships. These relationships reduce the modularity of the system. Because subordinates are more intelligent and because tasks are shifted to these subordinates, however, a certain controller might react faster to requests from its subordinates. Nevertheless, the prevailing interpretation of the modified hierarchical form performs badly on modularity and structural stability.

A control strategy as chosen by FACT (Arentsen, 1995) might perform better on modularity and structural stability. In FACT's modified hierarchical control architecture, stations have the possibility to solve a problem by mutual arrangement. 'Direct requests' are sent from one station to the other (see Appendix D). Information about other workstations that is needed for 'direct requests' should not be stored inside the workstations, but either in a central place or organised by a broker. Then, global information is made explicit, modularity is high, and the system is easily modifiable and extensible. Workstations are no longer dependent on other workstations. By means of the direct requests, workstations are also less dependent on their supervisors; structural stability is higher.

* It could be argued that the description of the behaviour of the modified hierarchical control form as in (Dilts *et al.*, 1991) is contradictory. Therefore, here an own interpretation is given.

6.7 Holonic manufacturing systems

6.7.1 Characteristics of holonic manufacturing systems

During the 1990's, the Holonic Manufacturing Systems (HMS) project was part of the world-wide research programme Intelligent Manufacturing Systems (IMS). IMS was one of the largest research programmes in manufacturing in its time. Initiated by Prof. Yoshikawa from Tokyo University, the IMS programme was envisaged to create a manufacturing science that would meet the needs of the 21st century. The concept of 'holonic systems' had been put forward as the paradigm likely to satisfy the requirements of expected future generations of manufacturing systems. A holonic manufacturing architecture should enable easy (self-)configuration, easy extension and modification of the system, and allow more flexibility and a larger decision space for higher control levels (Valckenaers *et al.*, 1994; Wyns *et al.*, 1996).

The ideas on holonic systems originated from Koestler (1967). He noted that a universal characteristic of hierarchies, i.e. 'boxes-within-boxes' structures, is the relativity and ambiguity of the terms 'part' and 'whole' when applied to any of the sub-assemblies. 'Part' and 'whole' suggest something absolute; a 'part' means something fragmentary and incomplete, whereas a 'whole' is considered as something complete. However, 'wholes' and 'parts' in this absolute sense do not exist anywhere. Intermediary structures are found: 'sub-wholes' which display, according to the way one looks at them, some of the characteristics commonly attributed to wholes and some of the characteristics commonly attributed to parts. Just like the Roman deity Janus who was the keeper of doorways, the members of a hierarchy have two faces looking in opposite directions: the face turned towards the subordinate levels is that of a self-contained whole; the face turned upward towards the apex, that of a dependent part. One is the face of the master, the other the face of the servant. Since Koestler did not find a satisfactory word to refer to these Janus-faced entities, he proposed the term 'holon'. The word 'holon' is a contraction of the Greek 'holos', which means whole, and the suffix 'on', which suggests a particle or part, as in 'proton' or 'neutron'. As such, the concept of the holon reconciles the atomistic and holistic approaches of looking at sub-assemblies (Koestler, 1967).

A holon exhibits two kinds of tendencies. Its self-assertive tendency is the manifestation of its unique wholeness and independence as a holon. Its integrative tendency expresses its dependence on the larger whole to which it belongs. As such, the integrative tendency is an expression of its 'part-ness'.

The main properties of holons are autonomy and willingness to cooperate. Autonomy concerns the ability of an entity to control the execution of its own plans and strategies. It ensures that holons are stable forms capable of dealing with disturbances, since holons handle contingencies without asking higher authorities for instructions. Cooperation is the process in which a set of entities develop and carry out mutually acceptable plans. Holons are components that provide the right functionality to the larger whole. Holons are subject to control from (multiple) higher authorities (HMS, 1996).

A *holarchy* is a system of holons that can cooperate to achieve a goal or objective. The holarchy defines the basic rules for cooperation of the holons are thereby limits their autonomy. A *holonic manufacturing system* is a holarchy that integrates the entire range of manufacturing activities from order booking through design, production, and marketing to realise the agile manufacturing enterprise (HMS, 1996).

Holonic manufacturing systems aim to combine the best features of various control forms. They employ the stability and optimality of the hierarchy and the dynamic flexibility of the heterarchy (HMS, 1996). Holonic manufacturing systems are designed for flexible customised production (Linssen, 1989), and as such, they are a possible next step in the evolution of (research in) control architectures as depicted by Figure 6-4. An example of a commercial product that appeared on the market is the “Holonical Cell” by Hitachi Seiki Co. (Hitachi, 1996).

At the time of writing this thesis, there was no ultimate architecture which fully described holonic behaviour in all its aspects. Various HMS partners have different views on the way holonic manufacturing systems should be controlled. For example, Bongaerts *et al.* (1995) and Tönshoff and Winkler (1995) differ on the role of a central scheduler. Moreover, many HMS partners seem to abolish hierarchical control levels completely, although HMS defines that holons can also receive instructions from and, to a certain extent, be controlled by higher level holons. The control architectures suggested by these HMS members resemble the heterarchical form, rather than that of a holonic manufacturing system. Even worse, the hype around holonic manufacturing systems made many researchers call their heterarchical designs ‘holonic’. This dissertation adopts the views which it considers closest to the original conception, namely those of the Catholic University of Leuven and the Osaka Prefecture University (Valckenaers *et al.*, 1994; Bongaerts *et al.*, 1995; Sugimura, 1996).

A specific feature of holonic manufacturing systems is that the control strategy they apply is not fixed, but rather adaptable to the manufacturing situation. Therefore, both hierarchical control and heterarchical control can be performed with the same hardware and software. Even more, a whole range of intermediate strategies can be used, depending on the situation on the shop floor.

In a holonic architecture in the flexible assembly system of the Catholic University of Leuven, a combination of the hierarchical and heterarchical control strategies takes place. The following hybrid control strategy *could* take place. The controller and the scheduler cooperate; the controller regards the schedules as advises, which it will follow in principle. If the scheduling advises are unfeasible or clearly suboptimal (for instance because of rush orders or workstation breakdowns), the controller adapts itself autonomously to the prevailing situation, and it tries to approximate the advised schedule as good as possible. This is accomplished by using the negotiation capabilities of the order modules and the workstations in a heterarchical manner. Meanwhile, the scheduler takes more time to consider the consequences of these disturbances to the globally optimised schedule and adapts its schedule. When ready, the scheduler provides the controller with the new schedule.

The holonic approach looks similar to the modified hierarchical form (Arentsen, 1995), but it is not. Both approaches are characterised by the combination of peer-to-peer and (relaxed) master-slave relations. However, since the control strategy of a holonic manufacturing system is adaptable, a modified hierarchical control strategy could be applied as one of the options. In that case, the behaviour of a holonic manufacturing system resembles that of the modified hierarchical form. Many other control strategies could be employed just as well, depending on the situation on the shop floor. The control strategy of the modified hierarchical form is (usually) not adaptable; only one control strategy can be applied.

Furthermore, the possible control strategy as described above results in a behaviour that is in fact the opposite of most modified hierarchical control strategies. After all, the control behaviour that is usually associated with the modified hierarchical form is that controllers cooperate in principle heterarchically, whereas exceptions are solved hierarchically. The described strategy makes a holonic manufacturing system function hierarchically in principle, whereas it deals with exceptions in a heterarchical way. It can be compared with the brains of a chicken that are distributed down its spinal column, and not totally in its head. When its head (hierarchical controller) is cut off, the rest of the body still runs around, controlled by the remaining intelligence (heterarchical controllers). Obviously, running around will be less smoothly without the head.

6.7.2 Evaluation

In a holonic manufacturing system, the (control) behaviour of the system is decoupled from the (control) structure. In previous chapters is stated that a major part of system architecting is the definition of the structure of the system. It is argued as well that during system architecting the desired behaviour of the system is specified. This chapter focuses on the control aspect in shop floor control systems. Architecting shop floor control systems involves the definition of a control structure and (accompanying) control strategy in order to obtain certain desired control behaviour. Usually, the control strategy is interwoven with the control structure, and only one behaviour is possible. If other system behaviour is desired, both the control strategy and the control structure need to be updated, which is quite hard in practice. In holonic manufacturing systems, however, each holon contains the abilities needed to perform its tasks under various control strategies, ranging from hierarchical to heterarchical approaches. Therefore, the same basic building blocks can be used together with various control strategies, resulting in various system behaviours. The control behaviour is practically decoupled from the control structure.

A combination of different holons should improve the flexibility and optimality of the control system. Appendix D describes two types of holons: *basic holons*, such as resource holons and order holons, and *staff holons*, such as a central scheduler. Staff holons assist the basic holons in performing their tasks. They allow for the presence of central elements in the architecture without introducing a hierarchical rigidity into the system. The basic holons still have the last word. Van Brussel *et al.* show that the combination of basic and staff holons results in system flexibility and system optimisation: ‘Due to the distributed basic architecture, the holonic manufacturing system delivers robustness and agility and is simple to extend and reconfigure.

The holonic manufacturing system can be optimised by optional staff functions, such as centralised schedulers. When the staff holons provide good advise, the basic holons will follow this advice as well as possible. When disturbances and changes in the system cause bad performances of the hierarchical staff holons, the advice may be ignored by the basic holons, which again take autonomous actions to do their work. On the other hand, when disturbances are absent, a holonic manufacturing system can be configured such that the basic holons follow (in a hierarchical way) the advice of the staff holon. This configuration is determined by the basic rules that determine the cooperation of the holons and thereby limit the autonomy of the individual holons' (Van Brussel *et al*., 1998). Whatever control strategy is chosen, the choice is not restricted by the structure of the control system.

A holon should be *structurally stable* because of its autonomy and willingness to cooperate. A holon is able to control the execution of its own plans and strategies, yet providing the right functionality to a larger whole. Because holons are able to handle contingencies without asking other holons for instructions, they are stable forms capable of dealing with disturbances.

The rules for cooperation of the holons restrict the autonomy of a holon, and largely determine the *modularity* of a holonic manufacturing system. These rules could decrease a holon's modularity by defining many interfaces between the holon and other holons.

6.8 Summary

Reference models for shop floor control aim to support one in the design of effective shop floor control systems. Compared to enterprise reference architectures, reference models contain application domain specific knowledge, and they do prescribe how to make certain architectural choices. The reference models for shop floor control can be categorised into basic forms. This chapter evaluates three such basic forms and a paradigm called 'holonic manufacturing systems' on the basis of the architecting principles of Chapter 4.

During the 1970's and the two decades afterwards, the focus of (research in) control architectures was gradually shifting from centralised to more distributed forms. This evolution was accompanied by an evolution in enabling technology, which gave designers more freedom in the definition of technology and functional architectures.

The overall conclusion of the evaluation of the basic forms is that the basic forms do influence system flexibility, but that within the forms a lot of variation is possible. For instance, a factor that influences system flexibility more than the basic forms, is how an architecture deals with global information. Whether or not global information is made explicit or incorporated within each component influences a system's future flexibility considerably.

7. Shop Floor Control Architecting

7.1 Introduction

The objective of this chapter is to demonstrate the (practical) value of the previously described architecting concepts and principles for the shop floor control domain. The architecting concepts of domains, decomposition hierarchy, and views are presented in Chapter 3, whereas the architecting principles of modularity, structural stability, and layers are introduced in Chapter 4. Chapter 5 shows that enterprise reference architectures do not lead to flexible systems. The basic control forms of Chapter 6 are too rough for a precise evaluation; too many architectural decisions that influence system flexibility have to be taken which are not prescribed by the basic forms. This chapter shows an example in which those decisions are taken, guided by the architecting principles.*

To demonstrate the value of the architecting concepts and principles for shop floor control, a control architecture based on autonomous agents is defined for a model factory. The control system is completely specified and simulated by means of a formalism called χ. The model factory is shortly introduced in the next section.

In addition, this chapter discusses the suitability of a control form which is not commonly used in industry, namely one based on autonomous agents and negotiation. In the previous chapter, the evolution of (research in) control architectures is outlined. The evolution ended with the heterarchical control form. Architectures based on autonomous agents can be classified under this control form. Section 7.3 elaborates on the ideas behind the application of autonomous agents in shop floor control systems, and describes the properties of agents and their accompanying negotiation protocol.

The next sections describe the design of the agent based control system for the model factory. Section 7.4 defines the system boundaries. In Section 7.5, the design of the system in the functional and technology domain is described. Section 7.6 identifies the workstation agents, i.e. the components with negotiation capabilities. The control strategy and a specific feature of this strategy, namely subcontracting, are outlined in Sections 7.7 and 7.8. Section 7.9 presents the components and the structure of a workstation agent.

Finally, the design is evaluated and compared to a previous design in Section 7.10. Compared to the agent based control system, the previous design did not incorporate negotiation capabilities in the workstation controllers.

* Parts of this chapter have been published in (Zwegers *et al.*, 1996; Zwegers *et al.*, 1997a; Zwegers *et al.*, 1997b).

7.2 The model factory

The model factory is a miniaturised, though still complex, model of a real Printed Circuit Board (PCB) assembly and test plant. The function of the model factory is to assemble and test pseudo PCBs. Each PCB consists of a board and a maximum of six components. Currently, two different types of boards and three types of components are used in the model factory.*

The model factory emulates operations which are performed on real PCBs during the manufacturing process. The operations of the model factory have been derived from case studies of real PCB manufacturing facilities. These operations are:

- *screen printing*: the bare PCB is positioned in the workstation. Then, a PCB-specific screen is selected and moved into position, and a squeegee is reciprocated horizontally over the screen in order to simulate the attachment of solder paste.
- *component placement*: the pasted PCB is positioned in the workstation, and components are placed on positions with imaginary solder paste according to the component-placement recipes for that product.
- *reflow and cleaning*: PCBs are passed through an oven and cleaning station
- *test and repair*: the PCB is inspected to see if it contains the components in the designated position, and component and functional tests are performed. If the PCB fails the test, it is routed to an off-line diagnosis and repair workstation. Upon successful repair, the PCB is routed back to the test station.

In addition to the operations described above, the model factory contains some other features. Raw material and components are automatically supplied from a centralised raw material store and component store respectively. The model factory can support mixed model flow production, where different types of products can be manufactured at the same time. The model factory is designed for batch production, but the batch size can vary from batch to batch, as well as product to product. The maximum batch size in the model factory is three.

The process layout is depicted in Figure 7-1. The operations are indicated by square boxes, whereas stock points are indicated by triangles. The first stock point contains the two different empty board types. All products pass the screen printer, but alternative routings are possible between the two component placement stations. After the reflow and cleaning station, the batches may be stored in the in-process-store which consists of three locations for three products each. Here, a batch can be split or concatenated with other batches. Then, products are tested and – if necessary – repaired. In the test and repair cycle, a maximum of one batch can reside in the buffer. Finally, nine individual products can be stored in the final-product-store.

An additional feature is a loop from the in-process-store to the screen printer. This loop is necessary to manufacture PCBs that have components on both sides. These products have to pass the process twice, since only one side can be finished in one pass. The buffer in this second-side loop may contain only one batch.

* The description of the model factory is based on (Timmermans, 1993a).

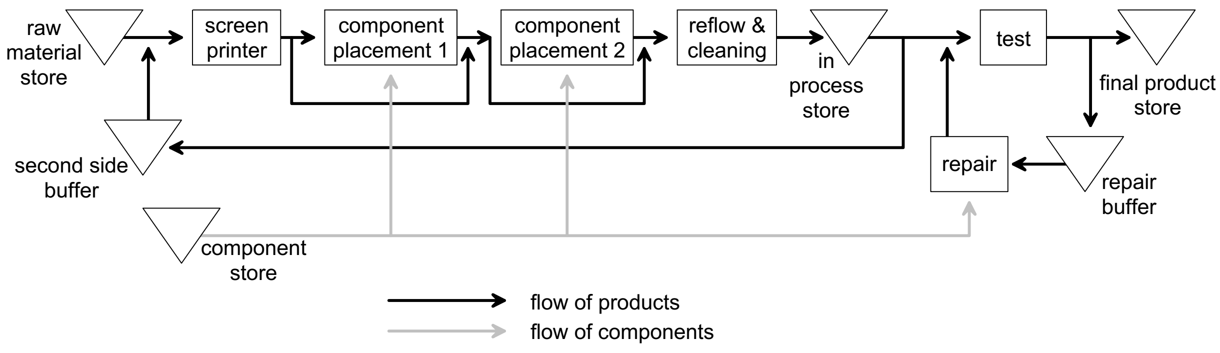


Figure 7-1 Primary process of the model factory

All workstations in the model factory are fully automated, with the exception of the repair station, where a manual operator is required. Besides the actual operation, each workstation has to manage temporary storage and retrieval of PCBs, indexing of PCBs through the process, inventory of raw materials, and so on. This necessitates many sensors in the model factory in addition to solenoid stops, motors, lights, conveyers, pneumatics, etc. All logical I/O signals to and from these sensors and actuators are controlled by a PLC program. In addition to the PLC program, a higher level supervisory system manages the overall production process. This latter control software is executed on a VAX computer (Timmermans, 1993a; Timmermans, 1993b; Timmermans and Szakal, 1996).

Past research on control architectures with the model factory include the specification and implementation of a hierarchical and a distributed control architecture. The latter architecture is shown in Figure 7-2. It is characterised by autonomous controllers for each self-contained unit of the model factory, and by a pull-oriented control strategy. The last controller in the line receives a work order which is consecutively passed to other controllers as requests for production. This architecture can be considered as an example of the heterarchical form

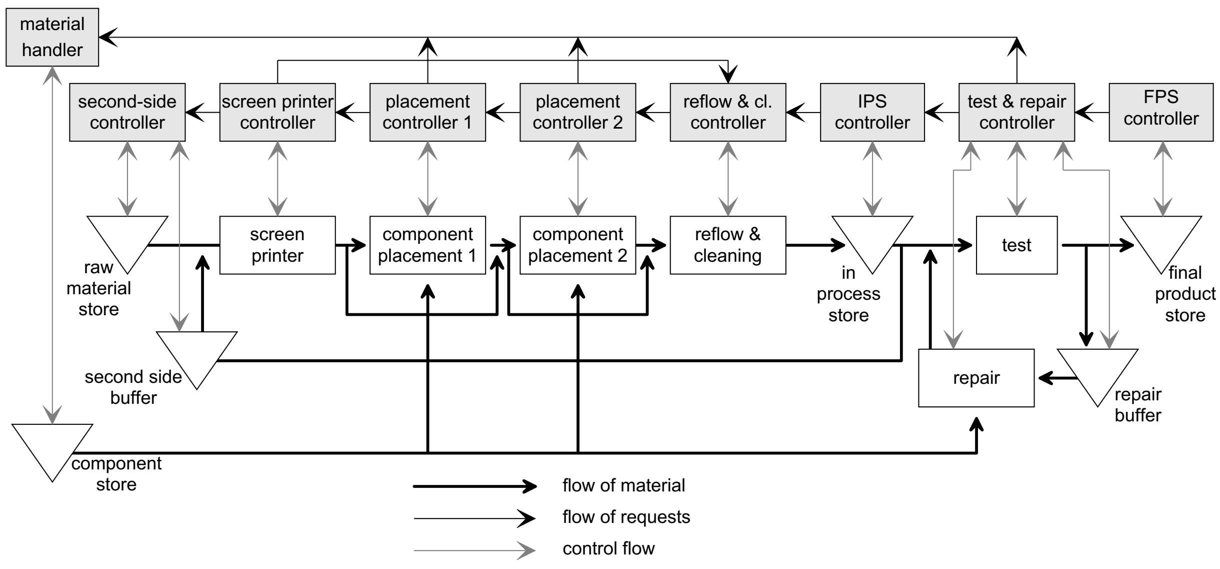


Figure 7-2 Heterarchical control architecture

Source: Timmermans (1993a)

without negotiation. Compared to the hierarchical control system, the heterarchical approach brought faster design and better adaptability (Timmermans, 1993a). However, the routing of products can only follow the communication links between the workstations as in Figure 7-2.

After the specification and implementation of a hierarchical and distributed control architecture, a logical next step in the research on control architectures with the model factory is the specification and implementation of an agent based control architecture. An agent based control architecture is an example of the heterarchical control form with negotiation.

In 1995, the Eindhoven University of Technology decided to restructure the model factory's control software, mainly the PLC program. This program contained information about product routings and Bills of Material. Preferably, execution functions that depend on the physical equipment configuration had to be separated from decision making functions that depend on the specific part types and the production mix. In the PLC program, however, execution functions were entwined with decision making functions. Changes in for instance routings had to be programmed in the PLC software. Therefore, a new PLC program without unwanted information was specified. However, at the time of writing this thesis, the new PLC software was not realised yet. An agent based control system with variable, product type independent routings could not be realised. A feasibility study with simulation experiments was conducted to examine the possibilities to use an agent based approach to control the model factory. This chapter describes the feasibility study. The next section focuses on the principles behind agent based control systems.

7.3 Agent based control systems

Decision responsibilities are more and more distributed to lower control levels. Manufacturing processes are highly dynamic and unpredictable; it is difficult to completely separate the planning and sequencing of required activities from their execution. Detailed time plans are often disrupted by unpredictable delays and other unanticipated events. As a result, a tendency exists within manufacturing systems to decentralise the ownership of the tasks, information, and resources involved in the various processes. Different groups within manufacturing systems become relatively autonomous; how their resources are consumed, by whom, at what cost, and in which time frame lies within their own prerogative.

Given these characteristics, it is quite natural to model the processes in a manufacturing system as a collection of autonomous, problem solving agents which interact when they have interdependencies. In such a context, an agent can be seen as an encapsulated problem solving entity that exhibits the following properties:

- *Autonomy*: agents perform the majority of their problem solving tasks without the direct intervention of other agents; they control their own actions and their own internal state.
- *Social ability*: when they deem appropriate, agents interact with other agents in order to complete their problem solving and to help others with their tasks. This implies that agents have a means by which they can communicate their requirements to others and an internal mechanism to decide what and when social interactions are appropriate (both in terms of generating requests and judging incoming requests).

- *Responsiveness*: agents perceive their environment and respond in a timely fashion to changes that occur in it.
- *Proactiveness*: in addition to responding to their environment, agents exhibit opportunistic behaviour and take the initiative where appropriate (Jennings *et al.*, 1996).

Agents use negotiation to coordinate their interactions. Each agent is able to perform one or more services or tasks. If an agent requires a service that is managed by another agent, it cannot simply instruct the other agent to start the service; agents are autonomous, and control dependencies between them do not exist. Instead, the agents must come to a mutually acceptable agreement about the terms and conditions under which the desired service will be performed. The mechanism for making these agreements is negotiation, a joint decision making process in which the parties verbalise their demands and then move towards agreement by a process of concession.

To negotiate with one another, agents need a protocol that specifies the role of the current message interchange, e.g. whether the agent is making a proposal or responding with a counterproposal, or whether it is accepting or rejecting a proposal. A well-known example of such a protocol is the Contract Net protocol (Smith, 1980). According to this protocol, agents decide upon their actions by exchanging demand and offer for services among themselves, together with varying amounts of status information. As noticed in the previous chapter, however, this protocol is often called a negotiation procedure, but there is no real negotiation involved. The protocol only defines a set of rules that state how agents react to events (Van Brussel, 1995).

Note that the agents mentioned here should not be confused with the mobile agents concept in software engineering. Mobile agents can be transported over a network. If considerable network traffic is needed to transport data between an agent and a database on a different network node, it might be more effective to transport the agent instead of the data (Orfali *et al.*, 1996).

The agent based approach is a logical step after Timmermans' heterarchy, which is characterised by a rigid routing along the stations. The negotiation mechanism results in a variable routing. The price is increased complexity within the agent, but the simplicity of interfaces is maintained.

7.4 Scope

The agent based control system provides for the control of and the coordination among the various workstations in the model factory. It processes a set of jobs and it reports the completion of this set. Its interfaces with the outside world are quite simple. It is assumed that a higher level control system is available, for example an MRP II based planning system. This system gives orders by releasing a set of jobs with process plans to the model factory. It does not provide the model factory with a schedule for the jobs. It is the responsibility of the model factory's control system to make such a schedule.

Figure 7-3 presents the system that is controlled by agents. To demonstrate the applicability of the autonomous agent concept, only part of the model factory is considered. It is the part from the Raw Material Store to the Final Product Store, the loop for second side printed circuit boards, and the component delivery subline. The In-Process-Store is not used as such, and is therefore disregarded, just like the Test & Repair loop. Only the workstations that perform operations are taken into account; movers between those workstations are ignored and decisions about the material flow are entrusted to the workstations.

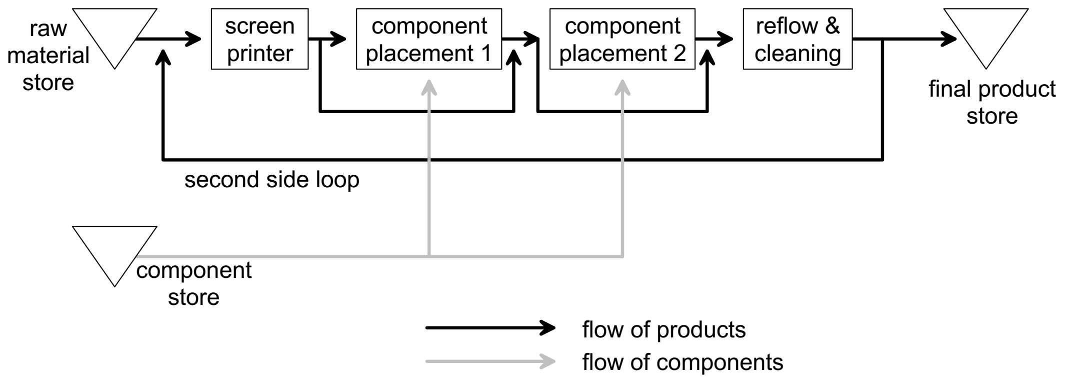


Figure 7-3 Part of the model factory to be controlled by agents

The agent based control system is compared to a previously designed heterarchical control system (see Figure 7-2), and is evaluated on the basis of performance, flexibility, and robustness criteria. The performance concerns the throughput of the system, and the jobs' cycle times. These characteristics are measured by simulation of various samples of a large number of jobs. Routings are determined on the basis of the actual situation in the system, rather than being determined by the product type. Therefore, the agent based control system should outperform the previously designed control system, i.e. its throughput should be higher and cycle times lower. The flexibility concerns the 'ease' with which the control system can be extended, cut down, or modified, if a workstation is added, deleted, or changed. Information about product routings and Bills of Material is no longer inside the various controllers, but is negotiated between agents, and transferred from higher level controllers. Therefore, the agent based control system should be less reluctant to change than its preceding control system. Robustness concerns the ability to deal with disturbances. The robustness of the control system has not been evaluated by means of simulation.

The next sections describe different parts of the model factory's agent based control architecture. Section 7.5 illustrates the use of the functional and technology domain in the agent based control architecture.

7.5 Control functionality and enabling technology

Domains
Design decisions are made in the functional domain and the technology domain. In the functional domain, Hakkesteegt (1993) defined three layers for the model factory: the physical

layer, the logic layer, and the application layer. These three names are adopted in this chapter, although they are clearly influenced by implementation matters. Figure 7-4 shows the three layers of an individual workstation. The physical layer comprises the functionality that operates on boards, such as screen printing and component placement. The other two layers make up the control system which controls the physical layer. The logic layer contains functions that command the manufacturing functions in the physical layer and receive sensory information from that layer. The application layer provides for the coordination among workstations, commands the logic layer, and receives status information from the logic layer.

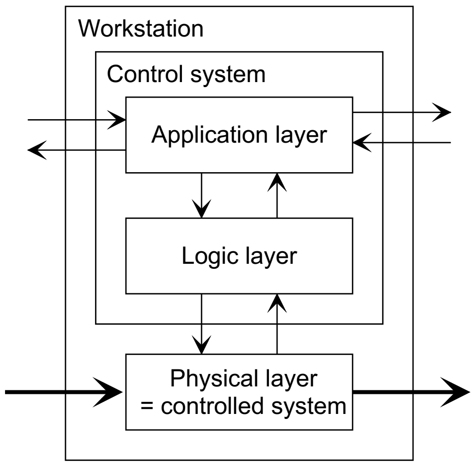


Figure 7-4 Layered functional architecture of the model factory's control system

At the physical layer, a decomposition in stations is evident. An architectural decision is made to follow this division at the logic layer and to handle control per station. For the application layer, various architectural choices are possible. Timmermans (1993a) advocates a heterarchical approach, but he defines fixed routings that depend on the product types. His control system cannot adapt to disturbances of the flow. In the feasibility study, agents are supposed to be robust to disturbances, since they decide upon routings by means of actual system status. However, the robustness aspect is not investigated in the feasibility study.

Besides the functional architecture, the technology architecture and enabling technologies are defined (see Figure 7-5). The functional architecture is mapped to the technology architecture. For instance, the application layer is implemented in control software written in C++, and needs a computer to be executed. Here, a VAX computer is chosen. The logic layer is implemented in PLC (Programmable Logic Controller) software, obviously enabled by a PLC. To connect the control software in the VAX with that in the PLC, an integration platform is used. This platform only provides connectivity, and no control functionality. The choices for enabling technologies are inspired by historical reasons; previous control systems were enabled by the VAX computer, the PLC, and the integration platform as well.

The advantage of using two domains for the definition of the control system is the separation of concerns. The two domains make a distinction between essential functionality and the way to achieve that functionality. It is a difference between the 'what' and 'how' of control. Furthermore, within the technology domain a distinction can be made between technologies

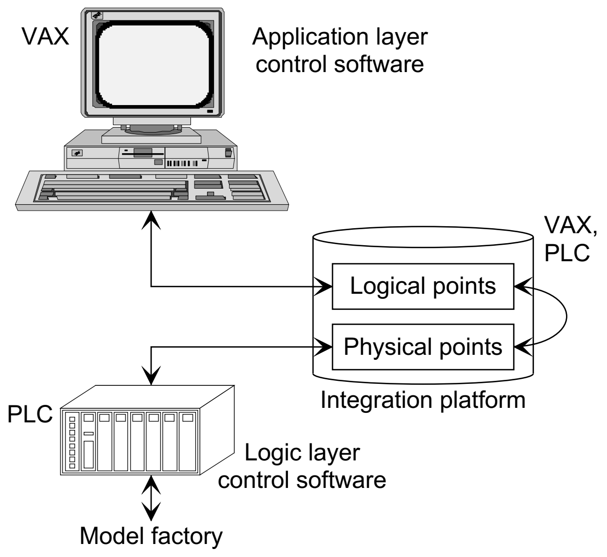


Figure 7-5 Technology architecture of the model factory's control system

that realise the desired functionality and enabling technologies. For example, in Figure 7-5 the PLC program and the PLC itself are indicated.

Layers

The identified layers separate concerns between (clusters of) components within a domain. Within a layer, changes can be made that do not affect other layers, as long as the interfaces remain unchanged. In the functional domain, each layer is responsible for specific control functionalities (see Figure 7-4). The identified layers are clearly associated to the identified control levels in the previous chapter. The application layer corresponds to the workstation level, whereas the logic layer corresponds to the automation module level. Cell/line level control functionality is incorporated in the application layer as well.

Also, layers can be identified in the technology domain. Besides the two control layers, an intermediate layer is present that provides for connectivity between the two software programs. This 'connectivity' layer is realised by an integration platform. By means of this platform, the PLC program is independent of the application layer control software. Two different kinds of data points exist in the integration platform, namely physical points and logical points. Physical points are directly coupled to addresses, such as memory addresses in the PLC. Logical points are memory locations (i.e. variables) which can be accessed by an application program. These logical points can be connected to physical points. In that case, a change of the value of a logical point will result in a value change of the corresponding physical point (Van Stipdonk, 1997).

In this section, the architecture of the control system in the functional and technology domain is outlined. The division in various layers is a characteristic of the architecture. The next section describes the identification of the workstation agents which is inspired by the pursuit of modularity.

7.6 Identification of agents

The control system consists of workstation agents, a component store, a generator, and a network. The expression ‘workstation agent’ indicates a workstation in which the application layer is equipped with capabilities to negotiate about jobs. The identified workstation agents are all workstations in Figure 7-3, except the component store. The component store does not have capabilities to negotiate, and acts merely as a server for the workstation agents. For simulation purposes, the generator is introduced to simulate the transfer of sets of orders from a higher level control system by the generation of these orders.

A network is used for message transfer among agents. Figure 7-6 shows that the network connects the workstation agents. The component store and the generator are drawn with dashed lines; they are connected to the network too, but they are no agents. Compared to direct channels between every pair of workstations, the network significantly reduces the number of channels. The network consists of a switch element, which is connected to the network interfaces for each of the connected entities. A network interface acts as an interface between the switch element and a workstation. For a connected agent, the network interface arranges the message reception from and transmission to other agents. It decouples the agents from the switch element and vice versa. This way, deadlocks are avoided where the switch element and (the controller of) an agent are waiting for each other (Coenen, 1995).

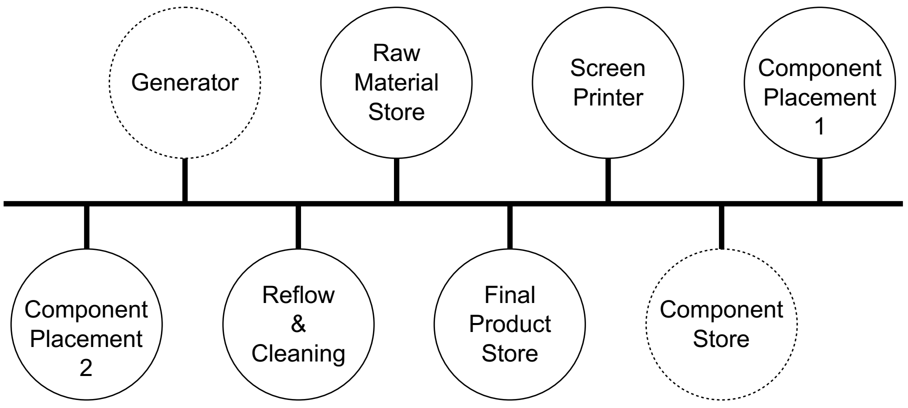


Figure 7-6 Agents connected by a network

Only the workstations are considered as autonomous agents. The workstations negotiate among each other about the execution of the jobs. Batches/jobs are passive entities flowing through the system; they do not have agent-like capabilities, and therefore are incapable of negotiation. This choice is made in particular because the used specification language assumes fixed communication channels, so components such as job agents cannot be created and deleted, but have to exist permanently. Nevertheless, some tricks might get around this restriction, since the number of jobs simultaneously being operated upon cannot exceed a certain maximum (see e.g. (Coenen, 1995)).

Modularity

The identification of the agents is inspired by the pursuit of a modular system. In order to achieve a modular control architecture, workstation agents have as little information about other workstations as possible. The aim is to minimise the coupling between workstations. The line-layout of the model factory allows little variation in the product routings. The only variations are the distribution of component placing process steps among the two component placers and the option of the second side loop. A workstation such as the raw material store, however, always sends processed batches to the screen printer. The raw material store does not need to negotiate with other workstations about the question which station will perform the next operation; it is known beforehand. A design decision could be made to incorporate this knowledge in the specification of the raw material store, thereby avoiding a negotiation procedure. However, this decision was not made. Such type of information would have to be modified in the future if batches would not necessarily have to go from the raw material store to the screen printer anymore, or if another screen printer would be added.

Before the structure of the workstation agents is explained, it is necessary to define the control strategy. The minimisation of information that workstation agents have about other workstations has some consequences for the control strategy.

7.7 Control strategy

Batches can be pushed through or pulled out of the factory. With a pull-approach, a schedule is made in advance. The last station in the line, in casu the final product store, is requested to deliver a batch of finished products at a certain due date. Then, the last station requests the appropriate batch from its preceding station, which – at its turn – asks for semi-finished batches to its predecessors, and so on. When a complete schedule is made, the order is released and production starts. Wiendahl and Ahrens (1995) give an example of such a system.

For the model factory, an opportunistic push-approach is chosen. Within this strategy, a schedule is not made in advance. The job is brought into the system at the first point of the line, namely the raw material store. Subsequently, the job finds its way through the system. However, in the model factory a convergent material flow is present at the component placement stations; both the boards and the component trays lead to these assembly stations. In general, since operations are not planned before job dispatch, stock points should be created in order to decouple the main stream from the branches. These buffers can be replenished by means of simple inventory control heuristics. Just in front of the model factory’s component placement stations, small buffers are located in which two component trays are stored, each containing four components. If the first tray in a buffer is out of components, a new tray is ordered from the central component store. Upon arrival of the new tray at the buffer, the empty tray is removed. In the feasibility study, new components are ordered if the number of present components is not sufficient to fulfil a job.

The main advantage of an opportunistic dispatch method is that decisions concerning the distribution of work on the shop floor are based on the prevailing system status rather than on some projection of that status (as would be the case with a pull approach). Disadvantages include the fact that opportunistic schedulers are myopic, that they may ignore interactions with other components, and that they may only handle priorities in a rather cumbersome way

(Upton *et al.*, 1991). Advantages and disadvantages of both opportunistic and planned approaches are displayed in Table 7-I.

Table 7-I Advantages and disadvantages of opportunistic and planning approaches

	Advantages	Disadvantages
Opportunistic	• Robust, capable of dealing with disturbances • Based on actual status; routing flexibility	• Less suitable for convergent material flows • Only short-term vision, possibly myopic
Planning	• Suitable for convergent material flows • Foresees the future	• Excessive planning needed • More sensitive to disturbances

In order to coordinate the negotiations between workstation agents, a protocol is needed. The points of departure are the opportunistic strategy and the fact that agents only have information at their disposal about the workstations they represent; they have no information about other workstations. Furthermore, the physical construction of the model factory imposes some restraints; workstations do not have output buffers, so they have to put processed boards in the input buffer of the next workstation. A workstation agent that is about to execute an operation has to search for a workstation that can execute the next operation first. The next workstation has to be known before a workstation can start its operation.

The negotiation protocol is as follows. A workstation agent sends a task announcement for the next operation of the job that is about to be executed by that workstation. The task announcement is sent to all workstation agents connected to the network, a so-called '*broadcast*'. Note that a broadcast to all workstation agents is needed since an agent does not have any information about other workstations in the system. A workstation agent that receives a task announcement only replies with a bid if the operation can be performed at that workstation. The message saying that a workstation cannot perform an operation would only cause more communication via the network, and is not sent. After a certain period of time, the agent that sent the task announcement chooses the best bid that has been received up to that moment. The time limit is needed because a workstation agent does not know how many agents will react to a task announcement. Another option would be to choose the agent that has sent the first incoming bid. It is clear that this might cause a suboptimal overall system performance. Due to the assumption that the production system is capable of executing the process steps of a job, an agent that sent a task announcement will always receive at least one bid. After a bid has been selected, a task offer is sent to the workstation agent with the best bid. Compared to the original version of the Contract Net Protocol by Smith (1980), the task offer acceptance message is omitted (see Figure 7-7).

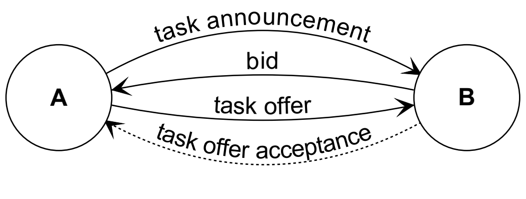


Figure 7-7 Negotiation protocol

7.8 Subcontracting

A specific feature of the control strategy is the possibility to subcontract process steps. A job is made up from a number of operations. An operation might consist of multiple process steps that are all of the same type of operation. In that case, a workstation might execute all process steps in an operation, or it might distribute execution of the process steps among itself and another workstation.

Because of subcontracting, the negotiation protocol changes. Figure 7-8 shows the messages that agents exchange as function of time. If an agent receives a task announcement for an operation that consists of multiple process steps, it sends subcontracting task announcements by means of a broadcast throughout the network. Note that both agents B and C can fulfil the task announcement from agent A in Figure 7-8. The subcontracting task announcements are meant for all combinations of half or less of the total number of process steps to be contracted. Main contractors never perform fewer process steps than subcontractors. Possible subcontractors are not allowed to subcontract the subcontracted process steps again. The agent (e.g. agent C) receives one or more subcontracting bids. It combines the best subcontracting bid (to be executed by agent B) with the accompanying main contract (to be executed by agent C itself). Agent C compares this bid with the bid in which it performs all process steps itself. The best bid is sent to the workstation agent that broadcasted the task announcement. Task offers are only sent to main contractors. So, if a task offer is received for a bid with subcontracting, the main contractor (agent C) notifies the subcontractor (agent B) with a subcontracting task offer.

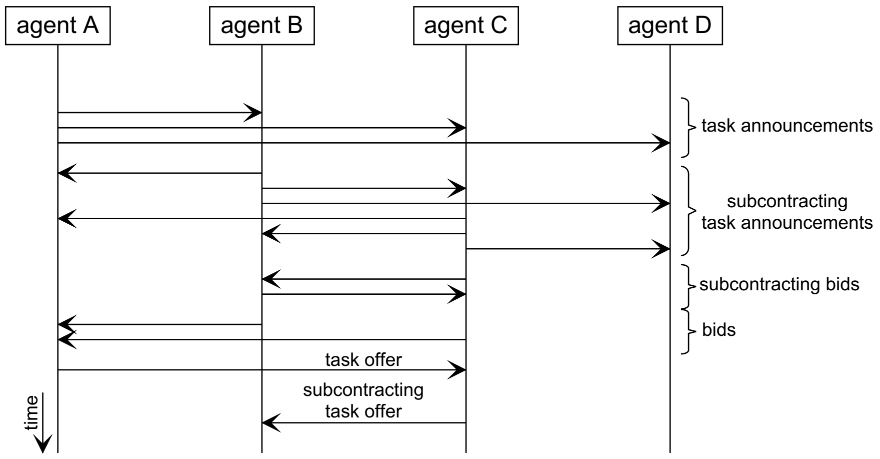


Figure 7-8 Messages between agents as function of time for the negotiation protocol

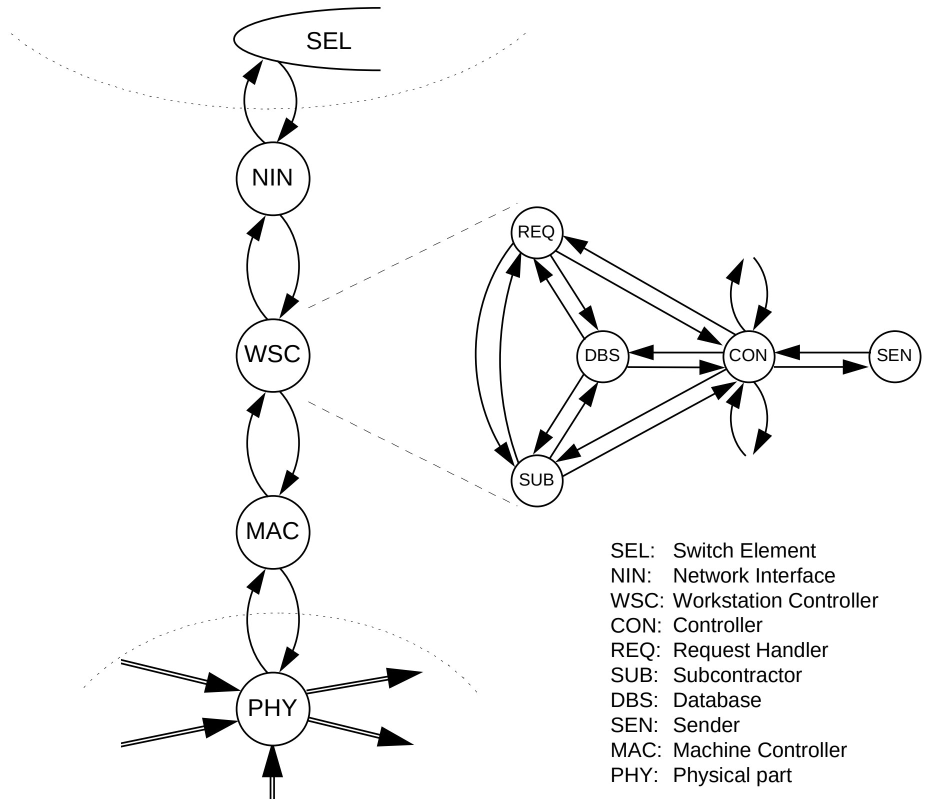
Structural stability

The subcontracting option can be considered as an extension to the original negotiation protocol. With or without the subcontracting option, the control system is stable. The system evolved from a configuration without the subcontracting option to a more complex (with the subcontracting option). Before the subcontracting option was added, it was made sure that the system worked. Before and after adding subcontracting, a complete control system was functional.

To illustrate how stability was achieved with or without subcontracting, it is necessary to focus on the structure of an agents. The next section shows that the design of the structure is guided by the pursuit of structural stability. The various components can be mapped on the identified layers.

7.9 Structure of a workstation agent

All workstation agents are based on the same generic structure. Figure 7-9 is a simplified picture of the workstation components and their relations. The following components are distinguished: a request handler, a subcontractor, a controller, a database, a sender, a machine controller, and a network interface. The only differences among workstations are their capabilities, their operation times, possible component supply and its duration, their place in the manufacturing system, and the conveyors with which they are connected to other workstations. All differences are instantiated, except the physical part of the workstation. More information is found in Appendix E.



SEL:	Switch Element
NIN:	Network Interface
WSC:	Workstation Controller
CON:	Controller
REQ:	Request Handler
SUB:	Subcontractor
DBS:	Database
SEN:	Sender
MAC:	Machine Controller
PHY:	Physical part

Figure 7-9 Internal structure and components of a workstation (simplified)

Hierarchy

Decomposition of the workstation agents results in a hierarchical structure. Figure 7-9 shows a decomposition of a workstation agent as represented in Figure 7-6. The components in Figure 7-9 could be further decomposed. The Sender, for example, consists of subcomponents that are responsible for the initialisation of a new task announcement, the broadcast of the task announcement, the processing of incoming bids, and the sending of a task offer. By means of the hierarchical structure, the overall complexity of a workstation agent is made more manageable. The subcomponents of the Sender are less complex than the Sender itself. In addition, the hierarchical structure stimulates modularity. The four subcomponents of the Sender have to comply to the interface between the Sender and the Controller. They may ignore the remainder of the agent's components, let alone other agents.

Structural stability

The subcontracting option was added in an iterative way. The option to subcontract process steps is an extension of the original negotiation protocol. A fully 'operational' component, the Subcontractor, was needed. In the initial simulations, however, the specification of the Subcontractor was kept at a minimum, i.e. only incoming and outgoing channels were defined. Provisions were taken in the Request Handler to invoke the Subcontractor, and in the Controller to direct incoming messages to the Subcontractor as if it were 'operational'. Furthermore, similar measures were taken in the Request Handler, Database, and Controller to handle messages from the Subcontractor. These measures allowed to add subcontracting functionality step by step. After each step, an agent was able to perform its tasks adequately, although an agent which could subcontract process steps was only available after all steps had been taken. An agent was stable during 'assembly'.

Furthermore, a control system consisting of a number of agents is stable as well. Structural stability concerns the stability during development but also during evolution. Workstations can easily be added to or removed from the control system. The structure of an agent as depicted in Figure 7-9 is the same for all workstation agents, except for the physical part PHY. There, workstation specific information is present such as the place of the workstation in the PCB assembly line. The remainder can easily be instantiated from a generic definition. The structure of an agent may not be simple. However, it allows one to easily add and remove workstations; it is designed to build upon. Furthermore, an agent with subcontracting capabilities can be added to a control system where subcontracting is no part of the control strategy.

Layers

The components in Figure 7-9 can be easily mapped to the various layers in Figure 7-4. The physical layer consists of the physical part of the workstation, PHY. The logic layer consists of the Machine Controller. In the feasibility study, the Machine Controller is identical for all workstations. They are not identical in reality, simply because for instance the Machine Controller of the Screen Printer needs information about screens, and the Machine Controllers of the Component Placers need information about components. All other components are part of the application layer. They control the underlying layers and coordinate the interactions among workstations.

7.10 Evaluation

In this section, the design of the control system is evaluated. Appendix E presents the results of the simulation experiments. In Section 7.4, three evaluation criteria are given, namely performance, robustness, and flexibility. The robustness of the agent based control system can not be evaluated, since disturbances are not simulated. However, it is expected that robustness in the agent based control system is increased compared to the distributed system without negotiation, due to the fact that routings were fixed in the latter system, whereas they are opportunistically 'composed' during operation in the agent based system. Defective workstations can be avoided. The effect is largely determined by the possibilities the manufacturing system offers. The effect is only marginally in the model factory, since in the present situation only the component placers can be interchanged to deal with malfunctions. If another station breaks down, the up-stream part of the system will be blocked, and the down-stream part will 'starve'.

The system extensibility is better in the agent based system than in the previously implemented heterarchical control system. Stations in the implemented heterarchical control system have knowledge about other stations. For instance, each station knows its direct 'neighbours'. If the factory is extended with a new workstation, the information its neighbours have of other stations needs to be updated. This is not necessary in the agent based system, since the agents communicate messages via the network. If a new agent is added to the system, the switch element in the communication network needs to be updated.

It is hard to compare the two control systems in terms of modifiability. Workstation controllers in the previously implemented heterarchical control system have information about other controllers, which hampers extensibility as well as modifiability. However, the controllers in the agent based control system probably have more states and make more assumptions about the behaviour of other agents. This is not determined into detail.

The *performance* of the agent based control system is compared to that of the heterarchical control system without negotiation. Appendix E shows that throughput and cycle times of the agent based system are slightly better than those of the heterarchical system without negotiation. Whereas the latter does not have routing flexibility at all, the possibilities of the agent based system to avoid a busy station and direct the batch to a less busy station are limited. This is caused by the absence of alternative workstations, except for placing components.

As compared to hierarchical control systems, studies show that the overall cycle times of agent based systems are worse than those of hierarchical control systems. After all, hierarchical control systems do not have the myopic view of agents; a hierarchical controller overlooks a larger area than an individual controller and is capable of making more global optimal decisions. The result of such a myopic view can be a deadlock, which could have been prevented by a global system view.

The characteristics of the physical production system cause the agent based control system to perform only slightly 'better' or even worse than other control systems. This leads to the conclusion that the model factory is not a suitable production system for the application of an agent based control system. Obviously, agent based systems are more suitable in systems with many interchangeable workstations.

7.11 Summary

This chapter gives an example of the design of an architecture for a shop floor control system. The role of the architecting concepts and principles as described in Chapters 3 and 4 is illustrated. It is indicated how the architecting concepts support the structuring of the system and how the architecting principles affect its design.

The specified system controls the operation of a model factory by means of autonomous, cooperating agents. A specific characteristic of this form of control systems is the use of negotiation; agents negotiate with each other to coordinate their actions. When this thesis was written, agent based control systems were hardly applied in practice. The model factory's agent based control system can be characterised as a logical successor of earlier control systems. Earlier control systems are a hierarchical one and a heterarchical control system without negotiation.

The agent based control system is specified and simulated by means of the specification language χ. Simulation experiments show that this relatively new type of control system performs better than a heterarchical control system without negotiation. The throughput and cycle time of the agent based control system for a specific work in process level are 'better' than those of the earlier designed heterarchical control system. In addition, system extensibility is improved in the agent based system compared to the heterarchical control system without negotiation. Stations in the first system have no knowledge about other workstations, whereas stations in the latter system know their direct 'neighbours'.

8. Conclusions and Suggestions

8.1 Conclusions

The subject of this thesis is systems architecting. Originally, it was conceived that this research project should examine several basic forms for shop floor control. However, during the project it became clear that more research was needed to clarify the essence of architecting. The research focus shifted from architectural forms to architecting. In addition, this research project studied various reference models and reference architectures.

The concept of an 'architecture' shifted from a user-oriented perspective in building science and digital systems engineering to a more system-oriented perspective in software engineering and Computer Integrated Manufacturing. Similarly, architecting was no longer considered as a one-time activity in which a system architecture was specified; nowadays, architecting is regarded as a continuous process that preserves the integrity of a system.

In Chapter 2, a system architecture is defined as the manner in which the components of a specific system are organised and integrated. An architecture, a reference model, and a reference architecture are three system-related terms with increasing genericity. An architecture is related to a specific system, a reference model to a set of systems in a certain domain, and a reference architecture to a framework to analyse and design systems.

Architecting concepts and principles

Domains, decomposition hierarchy, and views are introduced in Chapter 3 as the main architecting concepts. These three architecting concepts enable architects to manage overall system complexity.

Domains can be used to represent typical design problems. Three domains are distinguished: the functional, technology, and physical domain. As such, system complexity is distributed over three domains. The functional and technology architectures are most important for architecting, since it is part of an architect's role to span the boundary between what his principal wants and what engineers are able to build. Furthermore, architectures in the functional and technology domain should 'match' in order to prevent maintenance problems when the system changes.

Hierarchical decomposition can be used to divide systems into separate subsystems. The complexity of each subsystem is more manageable than that of the original system. Hierarchical decomposition is applicable to all domains. For the architecting process, it implies that an architect should simultaneously refine its models in the functional and technology domains. Then, the decomposition of the functional model should be validated by the possibilities to realise the decomposed functions by means of technology modules.

Views can be used to emphasise particular aspects of a system and hide the complexity of other aspects. Views discern aspect systems, whereas hierarchical decomposition discerns subsystems. Architects can focus on the issues that are important for their purposes by means of views.

In addition to architecting concepts, architecting principles are needed to achieve future flexibility. The architecting concepts enable architects to manage the complexity of architectural problems. With only architecting concepts, however, architects cannot create flexible systems. Guiding principles are needed with which the system flexibility can be obtained that is needed for evolution. An architect guards the integrity of a system during evolution. These systems have to last for a long time, and they have to be adapted to changing requirements. The architect has to build in measures that provide for future flexibility and evolution. Future flexibility is an architectural criterion; the value of a ‘good’ architecture compared to other architectures is in its future flexibility. Chapter 4 introduces modularity, structural stability, and layers as the architecting principles that stimulate the design of measures providing for future flexibility.

Modularity is a characteristic of a system which consists of moderately complex subsystems with high internal cohesion, and minimal coupling among the subsystems. In modular systems, the impact of changes is restricted to few modules. Changes in a component are propagated to only few components, and thereby modularity increases the future flexibility of a system.

Structural stability is the characteristic of a system in evolution to function stand-alone without collapsing, and with the ability to be part of a larger system or to be extended with future components. Complex systems are implemented step-by-step, and evolve regularly after initial implementation. At every point in their evolution, the system has to function well. The structural stability characteristic makes building blocks from complete ‘wholes’, and ‘wholes’ from building blocks. Structurally stable elements are made to function within an overall architecture.

Layers represent possible interfaces between clusters of components, and are mainly used to solve mapping problems. The mapping task is decomposed in layers, so that each layer performs a specific part of the task. Layers offer the flexibility to make changes in a layer that do not disrupt other layers, as long as its interfaces are not changed. Layers build upon underlying layers, and provide a guide for structuring a hierarchical decomposition process.

Architecting is an answer to increasingly complex and rigid systems. The architecting concepts and principles are suitable ‘tools’ for supporting architects in their activities. The architecting concepts allow one to deal with system complexity. The architecting principles guide one to obtain the flexibility that is needed for the system to be changed in the future. The concepts and principles were applied in the Gordian project. This project showed that the application of these ideas contributed to the realisation of Baan applications that have the flexibility to be adapted to future requirements.

Reference architectures and reference models
Chapter 5 discusses reference architectures for enterprise integration, which aim to provide the necessary frameworks with which companies might adapt their operation. The reference architecture CIMOSA strives for the facilitation of continuous enterprise evolution. It intends to offer support in the functional and technology domain by means of its modelling framework and its integrating infrastructure, respectively. The modelling framework assists enterprises in the definition of a functional control architecture. Support in the technology domain is insufficient; due to immature specification of the integrating infrastructure, only the specification of the communication services might be helpful.

CIMOSA is a suitable framework for managing system complexity. The architecting concepts of domains, hierarchy, and views are represented in CIMOSA’s modelling framework. CIMOSA does provide an eligible, though quite complex, framework for the specification and analysis of functional architectures for production control systems. It does not support the specification of a true technology architecture.

The architecting principles of modularity, structural stability, and layers are not incorporated in the CIMOSA modelling framework. CIMOSA does not prescribe its users how to design a system; it is a descriptive framework. It prescribes how to make a specification of a system, but it does not prescribe how to design the system. The translation from models to a real system has to be made by a designer. Therefore, CIMOSA should be merely seen as a framework for the generation of documentation.

In Chapter 6, reference models for shop floor control are presented, which aim to support architects in the design of effective shop floor control systems. Unlike reference architectures, they contain application domain specific knowledge, and they prescribe how to make certain architectural choices. The reference models for shop floor control can be categorised into basic forms, such as the centralised form, the proper hierarchical form, the modified hierarchical control form, and the heterarchical form. During the 1970’s and the two decades afterwards, the focus in (research on) control architectures was gradually shifting from the centralised form to more distributed forms.

Enabling technology has evolved as well. In the past, it used to be a limiting factor in the design of functional control architectures. Nowadays, it gives designers more freedom in the definition of technology and functional architectures. Within the technology domain, a distinction can be made between technologies that realise the desired control functionality and enabling technologies.

The basic forms influence system flexibility to a certain extent, but a lot of variation is possible within the forms. The modularity of a system, for example, is determined more by how the architecture deals with global information than by its basic form. Whether or not global information is made explicit or incorporated within each component influences a system’s future flexibility considerably.

An agent based control system for a model factory is architected in Chapter 7. Global information was made more explicit in its control architecture; the various agents had no knowledge of other agents and had to ask other agents about their capabilities. The extensibility of the agent based control system was improved compared to a previously implemented heterarchical control system. The agent based control system scored better regarding future flexibility. However, a lot of message traffic was needed to compensate for the absence of global information. Therefore, an agent based control system is a quite inefficient system compared to other forms.

Reference architectures and reference models support an architect, but are no panacea. Especially reference architectures tend to promise more than they offer. Reference architectures are useful during the specification and analysis of functional architectures for production control systems. By applying a reference architecture, an architect adopts a ‘mental framework’ with which he can deal with system complexity. A reference architecture does not prescribe how to design a system, and there is no ‘magic button’ with which one can turn models into operational systems.

Reference models provide templates with which architects can design specific systems. Still many architectural choices have to be made, which are not prescribed by the reference models. Both reference models and reference architectures are first steps, but not the final answers in architecting. Together with the architecting concepts and principles, they are necessary ‘tools’ for every architect.

8.2 Suggestions for further research

More architecting concepts and principles

In this thesis, a number of architecting concepts and principles are formulated which help architects in managing complexity and designing future-proof systems. These concepts and principles are an initial attempt. It is not claimed that these concepts and principles are sufficient for architecting; other (and perhaps ‘better’) concepts and principles might be formulated. In addition, concepts and principles might be formulated for specific domains or types of systems, such as real-time systems.

Behavioural aspects

Most research on shop floor control (including this thesis) concentrates on structural aspects of shop floor control systems. Behavioural, dynamic aspects are barely treated. It is very well possible that an architectural description is implemented in a manner where components make assumptions about the state and behaviour of other components. Such a dependency might hinder system operation and evolution. Timmermans (1993a) proposes to use dynamic constraints in the specification of behaviour. Nevertheless, the consequences of concurrency need to be understood.

Human involvement
Shop floor control systems are combinations of production resources, information technology, and humans. Most research focuses on the automated control systems and/or their relation with the production equipment. The role of the human has not been studied in-depth, but is essential for a learning organisation. Humans need IT tools rather than IT systems, so that continuous changing is possible in a learning organisation.

Dealing with legacy systems
In the scientific world, hardly any attention is paid to dealing with legacy systems. When this thesis was written, industry focused on the ‘Millennium’ problem. Science, however, takes a green-field approach: new systems are developed, rather than old systems are modified or extended. Researchers often do not take into account that legacy systems are present. Some technologies have been developed that ease the problem of migration.

Reference models for heterarchical control systems
As stated in Appendix D, no reference models for the heterarchical control form existed at the time of writing this thesis. Several applications had been published, but no reference models had been defined. If reference models for heterarchical systems are to be defined, they should be based on the architecting principles formulated in this dissertation.

Shop floor control in the extended enterprise
The consequences of the extended enterprise for shop floor control need to be examined. Traditionally, CIM has been discussed mainly in terms of the ‘four walls’ of the manufacturing process. However, much more emphasis is being placed on closer interaction with the suppliers and the customers (Higgins *et al*., 1996). The extended enterprise becomes reality. An example of a possible consequence of the extended enterprise for shop floor control is the requirement that certain test data is available in multiple places throughout the chain.

The profession of systems architecting
Universities should attempt to find ways to teach systems architecting to students. The University of South California is the first to offer a graduate degree in systems architecting and engineering with the focus on systems architecting (Rechtin and Maier, 1997). When this research project was conducted, systems architecting was a young, immature discipline. Chapter 1 states that systems architecting was a science in a ‘pre-paradigm phase’ (Kuhn, 1970). One could not learn it from study books. More research is needed to transform the ‘art’ of systems architecting to a true discipline.

References

Aalst, W.M.P. van der. (1994). Putting high-level Petri nets to work in industry. *Computers in Industry*, Vol. 25, No. 1, pp. 45-54.

Aguiar, M.W.C., I. Coutts, and R.H. Weston. (1994). Model Enactment as a Basis for Rapid Prototyping of Manufacturing Systems. In: *Proceedings of the European Workshop on Integrated Manufacturing Systems Engineering (IMSE ’94)*, pp. 86-96.

Aguiar, M.W.C., and R.H. Weston. (1995). A model-driven approach to enterprise integration. *International Journal of Computer Integrated Manufacturing*, Vol. 8, No. 3, pp. 210-224.

Aken, J.E. van. (1978). *On the control of complex industrial organizations*. PhD Thesis Eindhoven University of Technology. Martinus Nijhoff, Leiden.

Albus, J.S., A.J. Barbera, and R.N. Nagel. (1981). Theory and Practice of Hierarchical Control. In: *Proceedings 23rd IEEE Computer Society International Conference*, September 1981, pp. 18-39.

Amdahl, G.M., G.A. Blaauw, and F.P. Brooks jr. (1964). Architecture of the IBM System/360. *IBM Journal*, April 1964, pp. 87-101.

AMICE Consortium. (1989). *Open System Architecture for CIM*. Springer, Berlin.

AMICE Consortium. (1993a). *CIMOSA: Open System Architecture for CIM*. Springer, Berlin.

AMICE Consortium. (1993b). *CIMOSA: Open System Architecture for CIM – Technical Base Line*, Version 2.0. Esprit Consortium AMICE

Arends, N.W.A. (1996). *A Systems Engineering Specification Formalism*. PhD Thesis. Eindhoven University of Technology, Eindhoven.

Arentsen, A.L. (1995). *A generic architecture for Factory Activity Control*. PhD Thesis. Twente University of Technology, Enschede.

Bakker, J.J.A. (1989). *DFMS: Architecture and implementation of a distributed control system for FMS*. PhD Thesis. Delft University of Technology, Delft.

Bauer, A., R. Bowden, J. Browne, J. Duggan, and G. Lyons. (1991). *Shop Floor Control Systems – From design to implementation*. Chapman & Hall, London.

Bax, M.F.T. (1996). *De rijkdom van architectuur : Terug naar de Bouwkunde*. Eindhoven University of Technology, Eindhoven. (in Dutch)

Bemelmans, T. (1990). Invoering Case tools: Een langdurig en kostenintensief proces. *CA Techniek*, Vol. 8, No. 2, pp. 8-13. (in Dutch)

Berg, R.J. van den, and A.J.R. Zwegers. (1996). Decoupling Functionality to Facilitate Controlled Growth. In: *Proceedings of ASI ’96*, (P.P. Groumpos (Ed.)). Also published in *Studies in Informatics and Control*, Vol. 6, No. 1, March 1997, pp. 57-64.

Berg, R.J. van den, and A.J.R. Zwegers. (1997). *Gordian Project*. Report EUT/BDK/83. Eindhoven University of Technology, Eindhoven.

Berg, R.J. van den. (1998). *Rigour and Relevance in information management*. PhD Thesis. Eindhoven University of Technology, Eindhoven (to be published).

Bernus, P., L. Nemes, and T.J. Williams. (1996). *Architectures for Enterprise Integration*. Chapman & Hall, London.

Bertrand, J.W.M., J.C. Wortmann, and J. Wijngaard. (1990a). *Production control : a structural and design oriented approach*. Elsevier, Amsterdam.

Bertrand, J.W.M., J.C. Wortmann, and J. Wijngaard. (1990b). *Produktiebeheersing en material management*. Stenferd Kroese, Leiden. (in Dutch)

Biemans, F.P.M. (1989). *A Reference Model for Manufacturing Planning and Control*. PhD Thesis. Twente University of Technology, Enschede.

Biemans, F., and C.A. Vissers. (1989). Reference Model for Manufacturing Planning and Control Systems. *Journal of Manufacturing Systems*, Vol. 8, No. 1, pp. 35-46.

Blaauw, G.A. (1966). *Door de vingers zien*. Inaugural address, Technische Hogeschool Twente. Twente college, Enschede. (in Dutch)

Blaauw, G.A. (1971). The use of APL in computer design. *Proceedings of the MC-25 Informatica Symposium*. Mathematisch Centrum, Amsterdam.

Blaauw, G.A. (1976). *Beschrijven en begrijpen*. Technische Hogeschool Twente, Enschede. (in Dutch)

Black, J.T. (1983). Cellular Manufacturing Systems Reduce Setup Time, Make Small Lot Production Economical. *Industrial Engineering*, Vol. 15, No. 11, pp. 36-48.

Böhms, H.M. (1991). *Reference Models for Industrial Automation*. PhD Thesis. Delft University of Technology, Delft.

Bongaerts, L., P. Valckenaers, H. Van Brussel, and J. Wyns. (1995). Schedule Execution for a Holonic Shop Floor Control System. In: *Pre-prints of ASI '95*, (P.P. Groumpos and A. de Oliveira (Eds.)).

Bongaerts, L., J. Wyns, J. Detand, H. Van Brussel, and P. Valckenaers. (1996). Identification of Manufacturing Holons. In: *Pre-Proceedings of the European Workshop on Agent-Oriented Systems in Manufacturing*, pp. 13-29.

Bongaerts, L., H. Van Brussel, P. Valckenaers, and P. Peeters. (1997). Reactive Scheduling in Holonic Manufacturing Systems: Architecture, Dynamic Model and Co-operation Strategy. In: *Proceedings of ASI '97*, (P.P. Groumpos and G.L. Kovács (Eds.)), pp. 1-8.

Boudens, G. (1997). *Eenduidig opstellen van het business control model in de dynamic enterprise modeler*. MSc Thesis. Eindhoven University of Technology, Eindhoven. (in Dutch)

Brandts, L.E.M.W. (1993). *Design of industrial systems.* PhD Thesis. Eindhoven University of Technology, Eindhoven.

Brooks, F.P. jr. (1995). *The Mythical Man-Month : Essays on Software Engineering*, Anniversary Edition. Addison-Wesley, Amsterdam.

Browne, J. (1988). Production activity control—a key aspect of production control. *International Journal of Production Research*, Vol. 26, No. 3, pp. 415-427.

Brussel, H. Van, P. Valckenaers, and F. Bonneville. (1993). Programming, Scheduling, and Control of Flexible Assembly Systems. In: *Proceedings of the 25th CIRP International Seminar on Manufacturing Systems*. Also published in *Manufacturing Systems*, Vol. 23, No. 1, pp. 25-36, 1994.

Brussel, H. Van. (1995). “Navigation” issues in intelligent autonomous systems. In: *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS ’95)*, (U. Rembold, R. Dillmann, L.O. Hertzberger, and T. Kanade (Eds.)), pp. 42-52. IOS Press, Amsterdam.

Brussel, H. Van, J. Wyns, P. Valckenaers, L. Bongaerts, and P. Peeters. (1998). Reference Architecture for Holonic Manufacturing Systems. (to be published).

Cantamessa, M. (1995). A few notes upon Agent-based Modelling of Manufacturing Systems. In: *Proceedings of the CIM at Work conference*, (J.C. Wortmann (Ed.)), pp. 301-317.

Cantamessa, M. (1997). Agent-based modeling and management of manufacturing systems. *Computers in Industry*, Vol. 34, No. 2, pp. 173-186.

Chen, D., B. Vallespir, and G. Doumeingts. (1990a). An Integrated CIM Architecture – A Proposal. In: *Proceedings of the CIMCON ’90*, (A. Jones (Ed.)), pp. 153-165. NIST Special Publication 785, Gaithersburg.

Chen, D, G. Doumeingts, and L. Pun. (1990b). An Integrated Inventory Model based upon GRAI Tools. *Engineering Costs and Production Economics*, Vol. 19, pp. 313-318.

Chi. (1996). Example available at the Chi homepage. URL: http://se.wtb.tue.nl/

Coenen, F.W.J. (1995). *A Heterarchical Control structure for Flexible Production Systems* (in Dutch). MSc Thesis. Eindhoven University of Technology, Eindhoven.

Conant, R.C. (1976). Laws of Information which Govern systems. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 6, No. 4, pp. 240-255.

Conway, M.E. (1968). How do committees invent? *Datamation*, Vol. 14, No. 4, pp. 28-31.

CRISC. (1997). Various WWW pages on http://www.pe.chalmers.se/projects/crisc/

Crowther, J. (1995). *Oxford Advanced Learner’s Dictionary of Current English*. Fifth edition. Oxford University Press, Oxford.

Didic, M.M., F. Couffin, E. Holler, S. Lampérière, F. Neuscheler, J. Rogier, and M. de Vries. (1995). Open engineering and operational environment for CIMOSA. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 167-178.

Dilts, D.M., N.P. Boyd, and H.H. Whorms. (1991). The evolution of control architectures for automated manufacturing systems. *Journal of Manufacturing Systems*, Vol. 10, No. 1, pp. 79-93.

Dolan, T., R. Weterings, and J.C. Wortmann. (1998). Stakeholders in Software-system Family Architectures. In: *Proceedings of the second international workshop on the Development and Evolution of Software Architectures for Product Families*. Springer-Verlag (to be published).

Doumeingts, G., B. Vallespir, D. Darricau, and M. Roboam. (1987). Design Methodology for Advanced Manufacturing Systems. *Computers in Industry*, Vol. 9, No. 4, pp. 271-296.

Doumeingts, G., D. Chen, and F. Marcotte. (1992). Concepts, Models and Methods for the Design of Production Management Systems. *Computers in Industry*, Vol. 19, No. 1, pp. 89-111.

Doumeingts, G., B. Vallespir, and D. Chen. (1995). Methodologies for designing CIM systems: A survey. *Computers in Industry; special issue on CIM in the Extended Enterprise*, Vol. 25, pp. 263-280.

Duffie, N.A., and R.S. Piper. (1986). Nonhierarchical Control of Manufacturing Systems. *Journal of Manufacturing Systems*, Vol. 5, No. 2, pp. 137-139.

Duffie, N.A., R. Chitturi, and J. Mou. (1988). Fault-Tolerant Heterarchical Control of Heterogeneous Manufacturing System Entities. *Journal of Manufacturing Systems*, Vol. 7, No. 4, pp. 315-327.

Duffie, N.A. (1990). Synthesis of Heterarchical Manufacturing Systems. *Computers in Industry; József Hatvany Memorial: Total Integration — Analysis and Synthesis*, Vol. 14, No. 1-3, pp. 167-174.

Duffie, N.A., and V.V. Prabhu. (1994). Real-Time Distributed Scheduling of Heterarchical Manufacturing Systems. *Journal of Manufacturing Systems*, Vol. 13, No. 2, pp. 94-107.

Duffie, N.A., and Prabhu, V.V. (1996). Heterarchical control of highly distributed manufacturing systems. *International Journal of Computer Integrated Manufacturing*, Vol. 9, No. 4, pp. 270-281.

Erens, F.-J. (1996). *The Synthesis of Variety: Developing Product Families*. PhD Thesis. Eindhoven University of Technology, Eindhoven.

Erens, F., and K. Verhulst. (1997). Architectures for Product Families. *Computers in Industry*, Vol. 33, No. 2-3, pp. 165-178.

Esch, J. (1995). A Fine MES. *Byte*, Dec. 1995, pp. 67-75.

FACT. (1997). Various WWW pages on http://www.wb.utwente.nl/pt/projects/shopfloor/

Faure, J.M., A. Bassand, F. Couffin, and S. Lampérière. (1995). Business process engineering with partial models. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 111-122.

Galbraith, J. (1973). *Designing Complex Organizations*. Addison-Wesley, Reading, Massachusetts.

Garlan, D., R. Allen, and J. Ockerbloom. (1995). Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software; special issue on Architecture*, Vol. 12, No. 6, pp. 17-26.

Garlan, D., and D.E. Perry. (1995). Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering; special issue on Software Architecture*, Vol. 21, No. 4, pp. 269-274.

Germann, G. (1980). *Einführung in die Geschichte der Architekturtheorie*. Wissenschaftliche Buchgesellschaft, Darmstadt. (in German)

Graefe, U., and V. Thomson. (1989). A reference model for production control. *International Journal of Computer Integrated Manufacturing; special issue on CIM Architecture*, Vol. 2, No. 2, pp. 86-93.

Gransier, T., and W. Schönewolf. (1995). Editorial: Validation of CIMOSA. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 95-100.

Hakkesteegt, R. (1993). *Client-server computing in shop floor management* (in Dutch). MSc thesis. Eindhoven University of Technology, Eindhoven.

Hamer, P. van den, and K. Lepoeter. (1996). Managing Design Data: The Five Dimensions of CAD Frameworks, Configuration Management, and Product Data Management. *Proceedings of the IEEE*, Vol. 84, No. 1, pp. 42-56.

Hammer, D.K. (1997). IT-Architecture: A Challenging Mix of Aspects. In: *Workshop on Engineering of Computer Based Systems (ECBS) – Architecture, Metrics and Measurements*, (J. Rozenblit, T. Ewing, and S. Schulz (Eds.)), pp. 304-311. IEEE Computer Society Press, Brussels.

Hatvany, J. (1985). Intelligence and cooperation in heterarchic manufacturing systems. *Robotics & Computer-Integrated Manufacturing*, Vol. 2, No. 2, pp. 101-104.

Hee, K.M. van. (1993). *Systems engineering : a formal approach*. Eindhoven University of Technology, Eindhoven.

Higgins, P., and J. Browne. (1990). The monitor in production activity control systems. *Production Planning & Control*, Vol. 1, No. 1, pp. 17-26.

Higgins, P., P. Le Roy, and L. Tierney. (1996). *Manufacturing Planning and Control : beyond MRP II*. Chapman & Hall, London.

Hill, S. (1995). Is one enough? Can a single vendor meet your shop-floor information needs? *Manufacturing Systems*, Vol. 13, No. 1, pp. 42-54.

Hionides, H.T. (1978). *Collins contemporary Greek dictionary*. Collins, London.

Hitachi. (1996). *Holonical Cell*. Hitachi Seiki commercial leaflet.

HMS. (1996). Various WWW pages on http://hms.ncms.org/

Hopp, W.J., and M.L. Spearman. (1996). *Factory Physics : Foundations of Manufacturing Management*. Irwin, Chicago.

Huizinga, J. (1995). Een gereedschapkist vol software- en businesskennis. *Software Magazine*, Vol. 12, No. 10, pp. 68-71. (in Dutch)

IFAC/IFIP Task Force on Architectures for Enterprise Integration. (1997). *GERAM: Generalised Enterprise Reference Architecture and Methodology*. (available to download from http://www.cit.gu.edu.au/~bernus/taskforce/geram/V1_2.html)

Janusz, B. (1996). Modeling and Reorganizing of Process Chains Using CIMOSA. In: *Proceedings of the IFIP TC5/WG5.3/WG5.7 international conference on the Design of Information Infrastructure Systems for Manufacturing (DIISM '96)*, (J. Goossenaerts, F. Kimura, and H. Wortmann (Eds.)), pp. 140-151. Chapman & Hall, London.

Janusz, B. (1997). Model based business process redesign. In: *Proceedings of ASI '97*, (P.P. Groumpos and G.L. Kovács (Eds.)), pp. 106-111.

Jennings, N.R., P. Faratin, M.J. Johnson, P. O'Brien, and M.E. Wiegand. (1996). Using Intelligent Agents to Manage Business Processes. In: *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-agent Technology (PAAM96)*, pp. 345-360.

Jones, A.T., and C.R. McLean. (1986). A Proposed Hierarchical Control Model for Automated Manufacturing Systems. *Journal of Manufacturing Systems*, Vol. 5, No. 1, pp. 15-25.

Jones, A., E. Barkmeyer, and W. Davis. (1989). Issues in the design and implementation of a system architecture for computer integrated manufacturing. *International Journal of Computer Integrated Manufacturing; special issue on CIM Architecture*, Vol. 2, No. 2, pp. 65-76.

Jones, A. (1990). *Proceedings of the CIMCON '90*. NIST Special Publication 785. National Institute of Standards and Technology, U.S. Department of Commerce, Gaithersburg.

Jones, A., and A. Saleh. (1990). A multi-level/multi-layer architecture for intelligent shopfloor control. *International Journal of Computer Integrated Manufacturing*, Vol. 3, No. 1, pp. 60-70.

Keller, G., and S. Detering. (1996). Process-Oriented Modeling and Analysis of Business Processes using the R/3 Reference Model. In: *Proceedings of the IFIP TC5 Working Conference on Models and Methodologies for Enterprise Integration* (P. Bernus and L. Nemes (Eds.)), pp. 69-87. Chapman & Hall, London.

Koestler, A. (1967). *The ghost in the machine*. Hutchinson, London.

Kompass, E.J. (1993). Can Control Run Better At The Edge of Chaos? *Control Engineering*, Vol. 40, No. 1, p. 127.

Kroonenberg, H.H. van den. (1975a). Een bijdrage voor een algemene ontwerpmethode (1). *De Constructeur*, No. 9, pp. 51-59. (in Dutch)

Kroonenberg, H.H. van den. (1975b). Een bijdrage voor een algemene ontwerpmethode (2). *De Constructeur*, No. 10, pp. 55-61. (in Dutch)

Kruchten, P.B. (1995). The 4+1 View Model of Architecture. *IEEE Software; special issue on Architecture*, Vol. 12, No. 6, 1995, pp. 42-50.

Kuhn, T.S. (1970). *The structure of scientific revolutions*. Second edition. University of Chicago Press, Chicago.

Lapalus, E., S.G. Fang, C. Rang, and R.J. van Gerwen. (1995). Manufacturing integration. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 155-165.

Linssen, M. (1989). *Assemblage-trends in Japan*. Philips CFT report. (in Dutch)

Maglica, R. (1995). Improving the PAC Shop-floor Control Architecture to Better Support Implementation. In: *Pre-prints of the CIM at Work Conference*, (J.C. Wortmann (Ed.)).

Mazzatenta, O.L. (1996). China's Warriors Rise From the Earth. *National Geographic*, October 1996, pp. 68-85.

Meal, H.C. (1984). Putting proudction decisions where they belong. *Harvard Business Review*, Vol. 62, No. 2, pp. 102-111.

Meester, G.J. (1996). *Multi-Resource Shop Floor Scheduling*. PhD Thesis. Twente University of Technology, Enschede.

Merriam. (1993). *Merriam-Webster's collegiate dictionary*, Tenth edition. Merriam-Webster, Springfield.

MESA. (1997). Various WWW pages on http://www.mesa.org/

Meyer, B. (1988). *Object-oriented software construction*. Prentice-Hall, London.

Micklei, E.M. (1993). *Methodisch ontwerpen van de besturingsstructuur*. MSc Thesis. Eindhoven University of Technology, Eindhoven. (in Dutch)

Morgan, M.H. (1960). *Vitruvius : The ten books on architecture*. Translated by Morris Hicky Morgan. Dover, New York.

Morris, C.R. and C.H. Ferguson. (1993). How Architecture Wins Technology Wars. *Harvard Business Review*, March-April 1993, pp. 86-96.

Mortel-Fronczak, J.M. van de, J.E. Rooda, and N.J.M. van den Nieuwelaar. (1995). Specification of a Flexible Manufacturing System Using Concurrent Programming. *Concurrent Engineering: Research and Applications*, Vol. 3, No. 3, pp. 187-194.

Mortel-Fronczak, J.M. van de, and J.E. Rooda. (1996). On the integral modelling of control and production management systems. In: *Proceedings of the Advances in Production Management Systems conference (APMS ’96)*, (N. Okino, H. Tamura, and S. Fujii (Eds.)), pp. 171-176. IFIP, Laxenburg.

Mortel-Fronczak, J.M. van de, and J.E. Rooda. (1997). A Case Study in the Design of Control Systems for Flexible Production Cells. In: *Proceedings of the Workshop on Manufacturing Systems: Modelling, Management and Control (MIM ’97).*

Nell, J.G. (1996). Enterprise Representation: An Analysis of Standards Issues. In: *Proceedings of the IFIP TC5 Working Conference on Models and Methodologies for Enterprise Integration* (P. Bernus and L. Nemes (Eds.)), pp. 56-68. Chapman & Hall, London.

Orfali, R., D. Harkey, and J. Edwards. (1996). *The Essential Distributed Objects Survival Guide*. Wiley, Chichester.

Parnas, D.L. (1972). On the criteria to be used to decompose systems into modules. *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058.

Pels, H.J. (1988). *Geïntegreerde informatiebanken : Modulair ontwerp van het conceptuele schema*. PhD Thesis Eindhoven University of Technology. Stenfert Kroese, Leiden. (in Dutch)

Pels, H.J., and J.C. Wortmann. (1990). Modular design of integrated databases in production management systems. *Production Planning and Control*, Vol. 1, No. 3, pp. 132-146.

Perry, D.E., and A.L. Wolf. (1992). Foundations for the Study of Software Architecture. *ACM SIGSOFT*, Vol. 17, No. 4, pp. 40-52.

Philips CFT. (1987). *CAM Reference Model*. CFT Report 13/87.

Plasschaert, A.W.A. (1996). *A Study on Shop Floor Control : Principles, Practice and Products*. Internal EUT report.

Prabhu, V.V., and N.A. Duffie. (1995). Modelling and Analysis of Nonlinear Dynamics in Autonomous Heterarchical Manufacturing Systems Control. *Annals of the CIRP*, Vol. 44, No. 1, pp. 425-428.

Rappaport, A.S. and S. Halevi. (1991). The Computerless Computer Company. *Harvard Business Review*, July-August 1991, pp. 69-80.

Rathwell, G.A., and T.J. Williams. (1996). Use of the Purdue Enterprise Reference Architecture and Methodology in industry (the Fluor Daniel example). In: *Proceedings of the IFIP TC5 Working Conference on Models and Methodologies for Enterprise Integration* (P. Bernus and L. Nemes (Eds.)), pp. 12-44. Chapman & Hall, London.

Rational. (1997). *Unified Modelling Language*. Version 1.0. (available to download from http://www.rational.com/)

Rechtin, E. (1991). *Systems Architecting: Creating & Building Complex Systems*. PTR Prentice Hall, Englewood Cliffs.

Rechtin, E. (1992). The art of systems architecting. *IEEE Spectrum*, Vol. 29, No. 10, pp. 66-69.

Rechtin, E., and M.W. Maier. (1997). *The art of systems architecting.* CRC Press, Boca Raton.

Reithofer, W. (1996). Bottom-up Modelling with CIMOSA. In: *Proceedings of the IFIP TC5/WG5.3/WG5.7 international conference on the Design of Information Infrastructure Systems for Manufacturing (DIISM ’96)*, (J. Goossenaerts, F. Kimura, and H. Wortmann (Eds.)), pp. 128-139. Chapman & Hall, London.

Renkema, T.J.W. (1996). *Investeren in de informatie-infrastructuur : richtlijnen voor besluitvorming in organisaties*. PhD Thesis. Eindhoven University of Technoloy, Eindhoven. (in Dutch)

Rooda, J.E. (1996). *The Modelling of Industrial Systems*. Uncorrected preliminary version, lecture notes, Eindhoven University of Technology.

SAP AG. (1997). *R/3 Business Engineer : Knowledge-based, interactive R/3 configuration and continuous change management*. (available to download from http://www.sap.com/products/imple/media/pdf/50014850.pdf)

Schlotz, C., and M. Röck. (1995). Reorganization of a production department according to the CIMOSA concepts. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 179-189.

Schönewolf, W., C. Rang, W. Schebesta, and M. Röck. (1992). *TRAUB/IPK Pilot/Testbed Volume – Specification Phase*. VOICE report R92073.

SEI, the Software Engineering Institute. (1997). *What is software architecture?* (available to download from http://www.sei.cmu.edu/technology/architecture/definitions.html)

Sematech. (1996). *Computer Integrated Manufacturing (CIM) Application Framework*. Specification 1.3. Sematech Technology Transfer # 93061697F-ENG.

Simon, H.A. (1981). *The Sciences of the Artificial*. Second edition. MIT Press, Cambridge.

Simpson, J.A., R.J. Hocken, and J.S. Albus. (1982). The Automated Manufacturing Research Facility of the National Bureau of Standards. *Journal of Manufacturing Systems*, Vol. 1, No. 1, pp. 17-32.

Smit, H. (1992). *A Hierarchical Control Architecture for Job-Shop Manufacturing Systems*. PhD Thesis. Eindhoven University of Technology, Eindhoven.

Smith, R.G. (1980). The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, Vol. C-29, No. 12, pp. 1104-1113.

Solberg, J.J., and J.A. Heim. (1989). Managing Information Complexity in Material Flow Systems. In: *Advanced Information Technologies for Industrial Material Flow Systems*, (S.Y. Nof and C.L. Moodie (Eds.)), pp. 3-20. Springer Verlag, Berlin Heidelberg.

Soni, D., R.L. Nord, and C. Hofmeier. (1995). Software Architecture in Industrial Applications. In: *Proceedings of the International Conference on Software Engineering*, (R. Jeffrey and D. Notkin (Eds)), pp. 196-207. ACM, New York.

Sorge, W. van. (1997). Fact biedt integrale aansturing. *CA Techniek*, Vol. 16, No. 3, pp. 74-79. (in Dutch)

Spur, G., K. Mertins, and W. Süssenguth. (1990). Integrated Information Modelling for CIM. In: *Proceedings of the CIMCON ’90*, (A. Jones (Ed.)), pp. 373-389. NIST Special Publication 785, Gaithersburg.

Stevens, M., P. van der Putten, and R. van Weert. (1994). *Systematisch specificeren van electronica*. Centrum voor Micro-Elektronica, Veenendaal. (in Dutch)

Stipdonk, G.J.H. van. (1997). *Specification for the PLC-program of the Model Factory*. MSc Thesis. Eindhoven University of Technology, Eindhoven.

Sugimura, N. (1996). Personal communication.

Suh, N.P. (1990). *The Principles of Design*. Oxford series on advanced manufacturing. Oxford University Press, Oxford.

Tanenbaum, A.S. (1988). *Computer networks*. Second edition. Prentice-Hall, Englewood Cliffs.

Timmermans, P.J.M. (1993a). *Modular Design of Information Systems for Shop Floor Control*. PhD Thesis. Eindhoven University of Technology, Eindhoven.

Timmermans, P. (1993b). Control architectures and modular information systems: a comparative experiment. In: *Proceedings of the international conference on Advances in Production Management Systems (APMS '93)*, (I.A. Pappas and I.P. Tatsiopoulos (Eds.)), pp. 387-394. Elsevier Science Publishers.

Timmermans, P., and L. Szakal. (1996). A comparative experiment of control architectures. *Computers in Industry*, Vol. 28, No. 3, pp. 185-193.

Tönshoff, H.K., and M. Winkler. (1995). Shop Control for Holonic Manufacturing Systems. In: *Proceedings of CIRP '95*, (Y. Koren (Ed.)), pp. 329-336. Also published in *Manufacturing Systems*, Vol. 25, No. 3, pp. 277-281, 1996.

Upton, D.M., M.M. Barash, and A.M. Matheson. (1991). Architectures and auctions in manufacturing. *International Journal of Computer Integrated Manufacturing,* Vol. 4, No. 1, pp. 23-33.

Valckenaers, P., F. Bonneville, H. Van Brussel, L. Bongaerts, and J. Wyns. (1994). Results of the Holonic Control System Benchmark at KULeuven. In: *Proceedings of the Rensselaer's 4th International Conference on Computer Integrated Manufacturing and Automation Technology (CIMAT),* pp. 128-133. IEEE Computer Society Press, Los Alamitos.

Veeramani, D., B. Bhargava, and M.M. Barash. (1993). Information system architecture for heterarchical control of large FMSs. *Computer Integrated Manufacturing Systems,* Vol. 6, No. 2, pp. 76-92.

Waes, R.M.C. van. (1991). *Architectures for Information Management.* PhD Thesis, Tinbergen Institute research series no. 11. Thesis Publishers, Amsterdam.

Ward, P.T., and S.J. Mellor. (1985). *Structured Development for Real-Time Systems,* Volumes I, II, and III. Yourdon, London.

Wiendahl, H.-P., and V. Ahrens. (1995). Knowledge-Based Support for Planning and Control in Distributed Production Systems. In: *Proceedings of the IFIP 5.7 Working Conference on Managing Concurrent Manufacturing to Improve Industrial Performance*, pp. 429-443.

Williams, T.J. (on behalf of the CIM Reference Model Committee, Purdue University). (1989). A reference model for computer integrated manufacturing from the viewpoint of industrial automation. *International Journal of Computer Integrated Manufacturing; special issue on CIM Architecture,* Vol. 2, No. 2, 1989, pp. 114-127.

Williams, T.J. (1994). The Purdue Enterprise Reference Architecture. *Computers in Industry; special issue on CIM Architectures,* Vol. 24, No. 2-3, pp. 141-158.

Williams, T.J., P. Bernus, J. Brosvic, D. Chen, G. Doumeingts, L. Nemes, J.L. Nevins, B. Vallespir, J. Vlietstra, and D. Zoetekouw. (1994a). Architectures for integrating manufacturing activities and enterprises. *Computers in Industry; special issue on CIM Architectures*, Vol. 24, No. 2-3, pp. 111-139.

Williams, T.J., J.P. Shewchuk, and C.L. Moodie. (1994b). The role of CIM architectures in flexible manufacturing systems. In: *Computer control of flexible manufacturing systems* (S.B. Joshi and J.S. Smith (Eds.)), pp. 1-30. Chapman & Hall, London.

Wortmann, J.C., D.R. Muntslag, and P.J.M. Timmermans. (1997). *Customer-driven Manufacturing*. Chapman & Hall, London.

Wyns, J., H. van Brussel, P. Valckenaers, and L. Bongaerts. (1996). Workstation Architecture in Holonic Manufacturing Systems. In: *Proceedings of the 28th CIRP International Seminar on Manufacturing Systems*, Johannesburg, South Africa.

Yourdon, E., and L.L. Constantine. (1979). *Structured design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs.

Zachman, J.A. (1987). A framework for information systems architecture. *IBM Systems Journal*, Vol. 26, No. 3, pp. 276-292.

Zelm, M., F.B. Vernadat, and K. Kosanke. (1995). The CIMOSA business modelling process. *Computers in Industry; special issue on Validation of CIMOSA*, Vol. 27, No. 2, pp. 123-142.

Zwegers, A.J.R., and T.A.G. Gransier. (1995). Managing re-engineering with the CIMOSA architectural framework. *Computers in Industry*, Vol. 27, No. 2, pp. 143-153.

Zwegers, A.J.R., S.G. Fang, and H.J. Pels. (1995). Evaluation of Architecture Design with CIMOSA. In: *Pre-prints of ASI '95* (P.P. Groumpos and A. de Oliveira (Eds.)). Also published in *Computers in Industry*, Vol. 34, No. 2, 1997, pp. 187-200.

Zwegers, A.J.R., H.J. Pels, R.L.J. Schrijver, and R.J. van den Berg. (1996). An agent based control system for a model factory. In: *Proceedings of the Advances in Production Management Systems conference (APMS '96)*, (N. Okino, H. Tamura, and S. Fujii (Eds.)), pp. 293-298. IFIP, Laxenburg.

Zwegers, A., R. Schrijver, and A. Santana Alguacil. (1997a). *Modelling of an agent based control system for a model factory with the specification language χ*. Report EUT/BDK/88. Eindhoven University of Technology, Eindhoven.

Zwegers, A.J.R., L.H.Th.M. van Beukering, D. van Schenk Brill, and H.J. Pels. (1997b). A Comparison of Three Agent Based Control Systems. In: *Proceedings of ASI '97* (P.P. Groumpos and G.L. Kovács (Eds.)), pp. 209-216.

Zwegers, A., M. Verweij, and R. van den Berg. (1997c). Flexibel produceren door middel van holonic manufacturing. *CA Techniek*, Vol. 16, No. 5, pp. 22-26. (in Dutch)

Zwegers, A.J.R., and H.J. Pels. (1998). Application of Reference Architectures for Enterprise Integration. In: *Computer Aided and Integrated Manufacturing Systems Techniques and Applications* (C.T. Leondes (Ed.)). Gordon and Breach, Newark (to be published).

A. Application of CIMOSA at Traub

A.1 Introduction

This appendix describes the project, in which Traub AG reorganised its production department. The main reason to initiate the project was the desire to obtain a more flexible and efficient production department, and hence to react in a better way to changing customer demands, and to contribute to the company's competitiveness in the global market.

Figure A-1 shows Traub's production control system before the reorganisation. At that time, it consisted of a mainframe with the Production Planning System, an IBM RS6000 for area control functionalities, and several cell controllers. Traub made a global production planning for a year and a half in advance. In this planning, machine types, options and number of

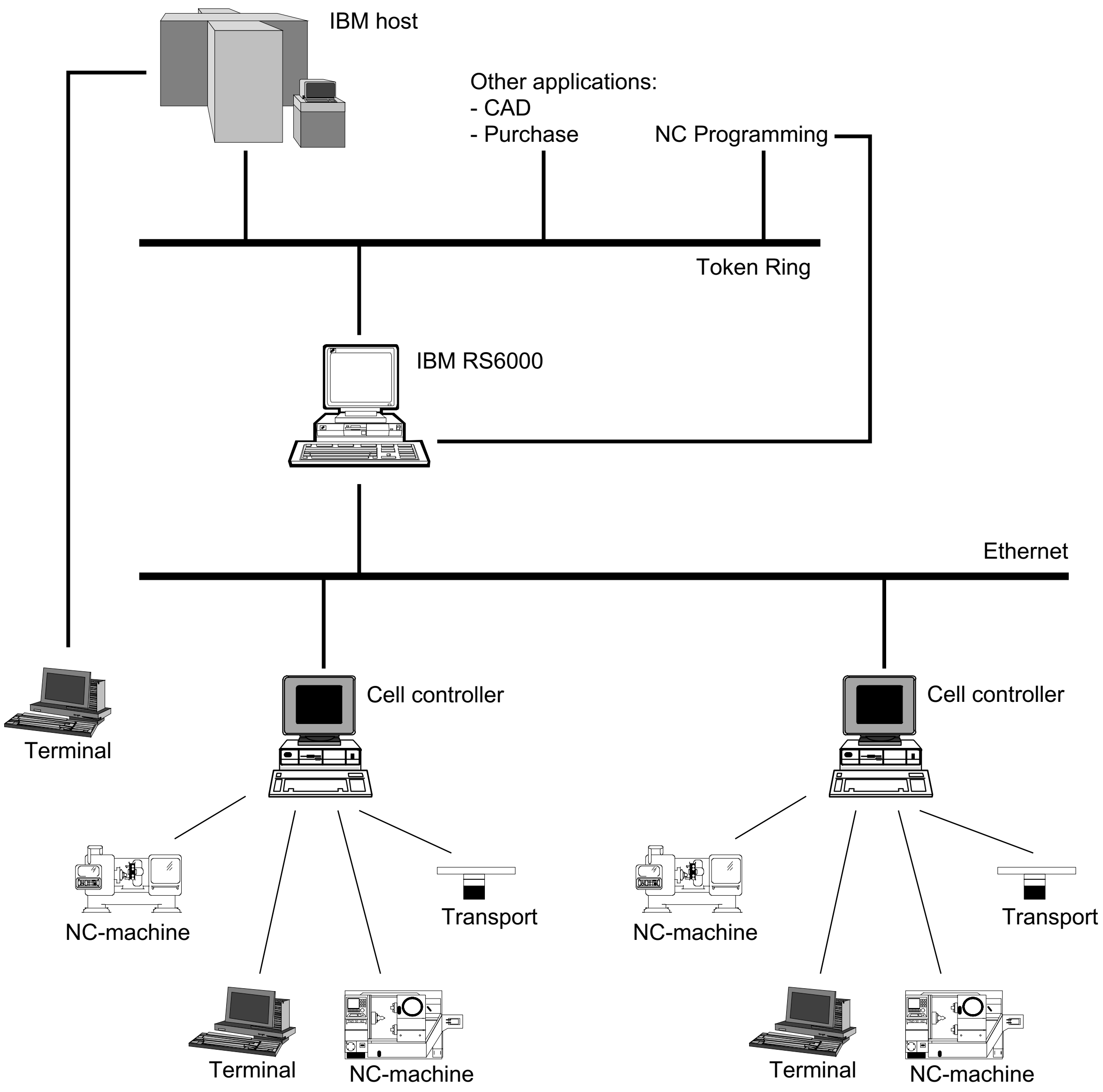


Figure A-1 Traub's production control system before the reorganisation

products were incorporated. Every ten days, a partial planning was made, which consisted of timed orders for design, purchase, and production control. It was possible to control the order flow from the area controller by releasing orders, standing in a ‘ten days order pool’. Dedicated applications established the transmission of NC-programs between the area controller and the cell controllers. The worker at the NC-machine got a list of order data, and could transfer and edit NC-programs from and to the machines, and transmit the optimised programs to the area controller. Order progress and other monitoring data could be automatically sent from the machines to the area controller or by entering a message on a terminal. Monitoring information was sent to the mainframe at the production planning level from terminals on the shop floor located at several places near the machines.

One of the problems associated with the old situation at Traub was its flexibility to respond to changes in customer needs. Shorter delivery times with simultaneous reduction of stocks were already forcing Traub to shorten machining times and to link all processes in design, planning, and shop floor more closely. Even more, it became frequently necessary to rearrange plans at short notice because of changes in demand, machine downtimes, missing production facilities, urgent contracts with large customers, absence of staff, or rejects. Re-scheduling of manufacturing jobs became difficult due to limited feedback from the shop floor, and expenditure for short-notice re-work and machine modifications increased sharply.

Especially the preparatory sectors were the areas where increased deadline pressure and the tendency towards smaller and smaller batch sizes added to planning complexity and expenditure. Most notably, tool management became problematic, since the lack of up to date information concerning the availability of machine tools and their components also reduced the production capacity while operators waited for resources. Furthermore, production management had to cope with a growing amount of specialised tools necessary to produce individually designed machine parts. There were nine machine centres that each had a stock of approximately 120 tools. Each tool consists of eight to ten components. At that time, there was no registration of which tool or component was in which machine (Schönewolf *et al.*, 1992).

A.2 Reorganisation objective

The most important target of the reorganisation was to optimise the order throughput time. In order to achieve this, a number of changes had to be implemented related to the cooperation on optimised order scheduling between the area controller and the cell controllers. Other changes concern tool management, monitoring data acquisition, and so on. In this subsection, only a few major changes are discussed.

The old, process-oriented manufacturing departments such as turning, milling, and grinding, were changed for a more flexible organisation. The new organisation is based on the principles of cellular manufacturing, which involved the creation of clusters of machines that are designed and arranged to produce a specific group of component parts (Black, 1983). In particular, Traub expected the cellular manufacturing principles to result in a decrease of set-up times.

An area control system was required that had to perform the fine planning of manufacturing device utilisation under rescheduling conditions with the help of a production scheduling simulation tool. This area control system had to be linked enterprise-wide with existing production planning, CAD, and CAM applications. It had to control not only the manufacturing processes but also the delivery of material, tools, and equipment. Furthermore, it had to take into account the actual status of each NC machine. This information had to be transferred on-line – without any influence of the working people – directly from the machine controller to the planning application of the area controller.

Fine-planning had to be supported by efficient tool management to get a tool-optimised order sequence in order to decrease the set-up time for each machine. This system would provide information of the location and states of tools by means of a tool identification system and a central database with tool information. It was necessary to integrate the tool logistics application with the entire existing infrastructure (Schlotz and Röck, 1995).

A.3 Requirements definition

Conform the three domains identified in Chapter 3, Traub defined requirements in a functional sense for the application to be developed, and in a technological/physical sense for the application’s underlying infrastructure. In other words, requirements were defined for both the functional and technology aspects of the various components in the manufacturing system. In addition, requirements were imposed on the re-engineering process, influenced by financial and time aspects.

Traub described its requirements from a functional point of view in terms of a scenario. In this scenario, the needed functions and information flows, and their places in the total manufacturing system were outlined. For instance, part of the scenario for the fine-planning activity (precision planning) was described as follows:

‘The work sequences released are assigned to the individual machines during the automatic precision planning. During a planning sequence, only the machine capacity is taken into account. Starting from the terminating time given by the order pool system, timing is done backwards. If the calculated starting time lies in the past, the timing is done forwards while considering the transition time. If the given terminating time is not observed during this forward scheduling, this is explicitly displayed in the system. No alternative machine groups are considered in the automatic precision planning’ (Schönewolf *et al.*, 1992).

Requirements for the technology to be used were given as well. Besides Traub’s current needs, requirements took into consideration the existing manufacturing system, strategic aspects like standards, the factory environment, and production process constraints. Traub mainly defined its requirements of the infrastructure in terms of the CIMOSA integrating infrastructure, for instance:

- ‘multiple machines must be connected to a homogeneous system’ (presentation services),
- ‘connectivity to multiple databases on heterogeneous networks must be achieved from different kinds of computer systems’ (information services),
- ‘the network must be transparent’ (common services) (Schönewolf *et al.*, 1992).

A.4 Architectural design

After requirements were defined, the design of the CIM system commenced. The design activities of Traub’s reorganisation project could be distributed over two phases: architectural design and detailed design. In the architectural design phase, the system’s functional and technology architecture were defined, supported by the CIMOSA framework. In the detailed design phase, the system was worked out in more detail, based on the defined architectures.

Area control was positioned between the production planning level (not in the scope of the reorganisation project) and the shop floor. As an autonomous decision centre in a distributed order planning structure, it processes the order pool coming from the production planning system, and performs scheduling and management tasks for machine level planning. The incoming orders are scheduled for a ten days period on the various machine groups. For each machine group, the area controller performs a daily planning to achieve an optimised order schedule with time slices of two days. The new order sequence on a machine is calculated on the basis of the available tools and the list of needed tools, which is extracted from the NC programs for the new orders. Tools available at NC machines or in the tool store can be requested. Tool handling is supported by the tool management process.

Being intermediate between production planning and shop floor, area control not only has to provide the shop floor with orders, it also allows to feed back information from the shop floor to the production planning system. Then, this system may use data that reflect the actual situation in the shop floor in order to optimise the planning process. Based on on-line messages from the cell control level, the area controller also supports processing and visualisation of data such as the actual status of orders.

The scheduled orders in the time frame of two days are sent to the cell controller for execution. Tool management has allocated the tools required for the orders, and the NC programs are downloaded from the NC program server. Subsequently, the cell controller acts as an autonomous decision centre, responsible for the execution of the orders, for the collection of machine and operational data, and for monitoring the shop floor processes.

Traub defined its production control and tool management system by means of modelling this system and its environment with the CIMOSA requirements level. Firstly, Traub built a user model with the Systems Analysis and Design Technique, since a tool and knowledge covering this modelling method was available. Later, Traub converted the user model to a CIMOSA model at requirements definition level. For modelling at requirements level, Traub used the modelling tool ‘GtVOICE’, which is described by Didic *et al.* (1995). Traub specified the functions and their interrelations for both the tool logistic system and its direct neighbours. By modelling, Traub structured its system component functions, defined the components' allowed inputs and outputs, and specified the relations between these components. In other words, by modelling at requirements definition level a functional architecture was designed.

Figure A-2 shows a CIMOSA model that presents a global view on Traub's production control functions. The principal control functions are captured in CIMOSA *domains*, the constructs that embrace the main enterprise processes. Figure A-2 gives the identified domains and their relations in the form of *domain relationships*. Non-CIMOSA domains are parts of the enterprise which have not been considered for the moment and which could be detailed in the future such as 'Purchase' and 'DNC-Node', or which are closed applications that can not be described such as the 'Area Control' application. Usually, these applications are legacy systems.

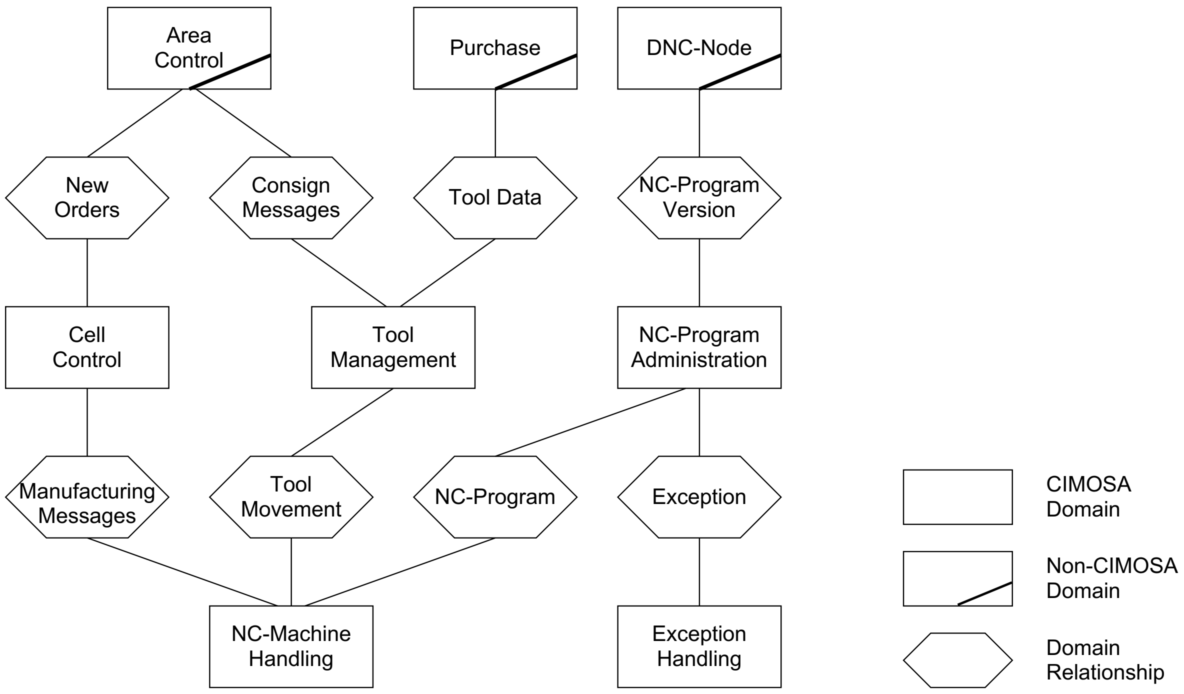


Figure A-2 Overview of Traub's production control functions

Traub also made some models that showed more details than Figure A-2. Figure A-3 is a partial refinement of the previous figure, showing domains and *domain processes*. CIMOSA represents main enterprise functionalities as *domain processes*. CIMOSA offers a modular modelling approach; the system can be extended with new domain processes, and system modifications can be limited to few domain processes. Enterprise operation is structured into a set of interoperating domain processes exchanging results and requests. They encapsulate a well-defined set of enterprise functionality and behaviour to realise certain business objectives under given constraints. In the Traub case, examples of concurrent processes are the processing of tool data and the preparation of tools. These processes are represented in domain 'Tool Management' by two independent domain processes, namely 'Tool Data Processing' and 'Prepare Tools'. Note that the domain processes in Figure A-3 influence each other's behaviour.

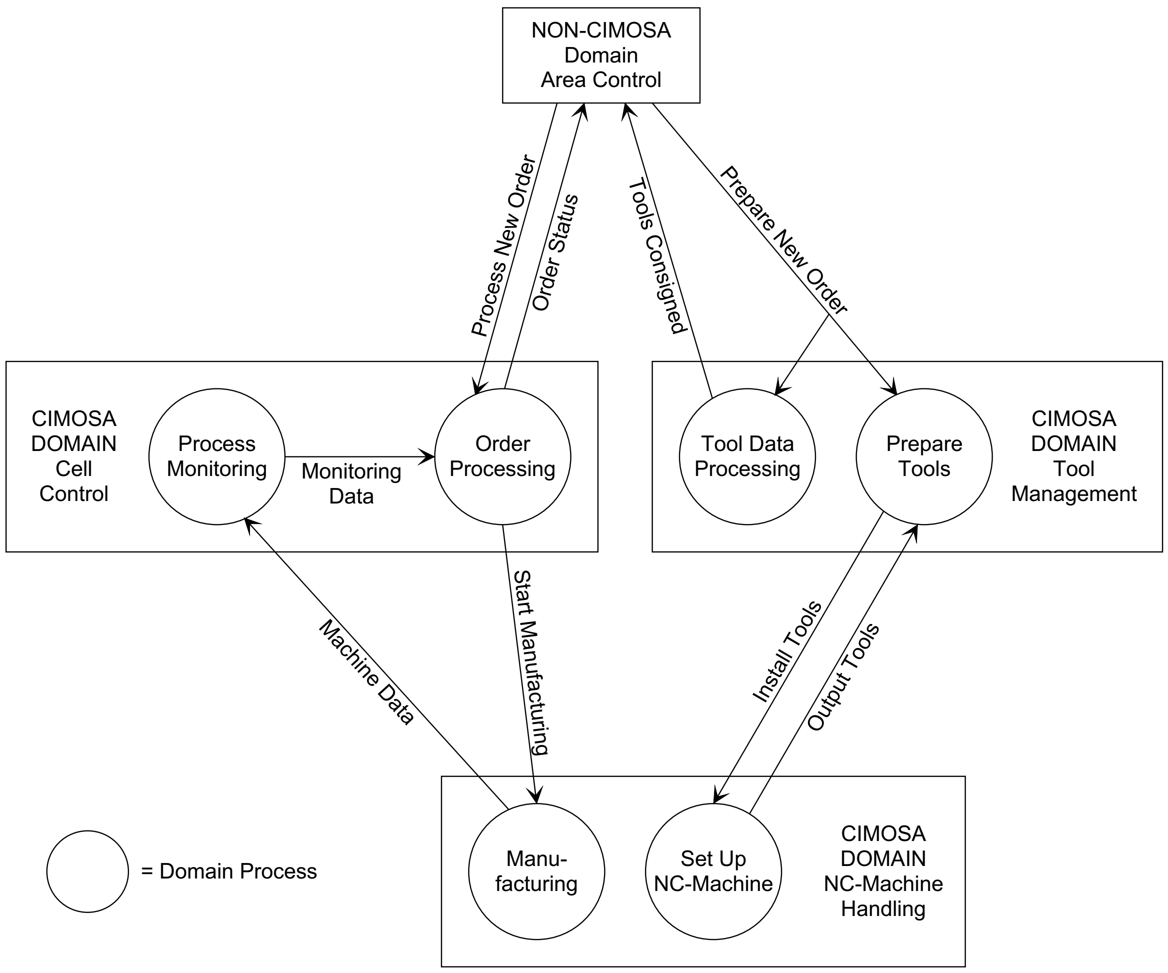


Figure A-3 Partial refinement of Traub's domains

Domain processes are the root of the decomposition tree; they employ *business processes* which are in the middle of the tree. The leaves are named *enterprise activities* and are employed by business processes or, if there are no business processes, by domain processes. The behaviour of a certain business or domain process is defined by rules, according to which enterprise activities belonging to this process are carried out. Enterprise activities represent the enterprise functionality as elementary tasks, and they are defined by their inputs, their outputs, their function and their required capabilities. Figure A-4 presents a part of the behaviour of domain process 'Order Processing'; for clarity, some relationships have been deleted. Events, which are either received from or sent to other domain processes, are represented by a Z-shape; enterprise activities are shown as boxes labelled 'EA'. Note that there are no business processes specified for this domain process. The large triangles indicate behavioural rules according to which enterprise activities or business processes are carried out. Figure A-4 was made by using the GtVOICE tool.

Note that the requirements definition level of the CIMOSA modelling framework is used at architectural design activities and not during the requirements definition process. The reason is that CIMOSA does not support a 'true' definition of requirements in the sense as described in the previous subsection. Instead, the CIMOSA requirements definition level offers support in structuring a manufacturing system, i.e. in defining a functional architecture.

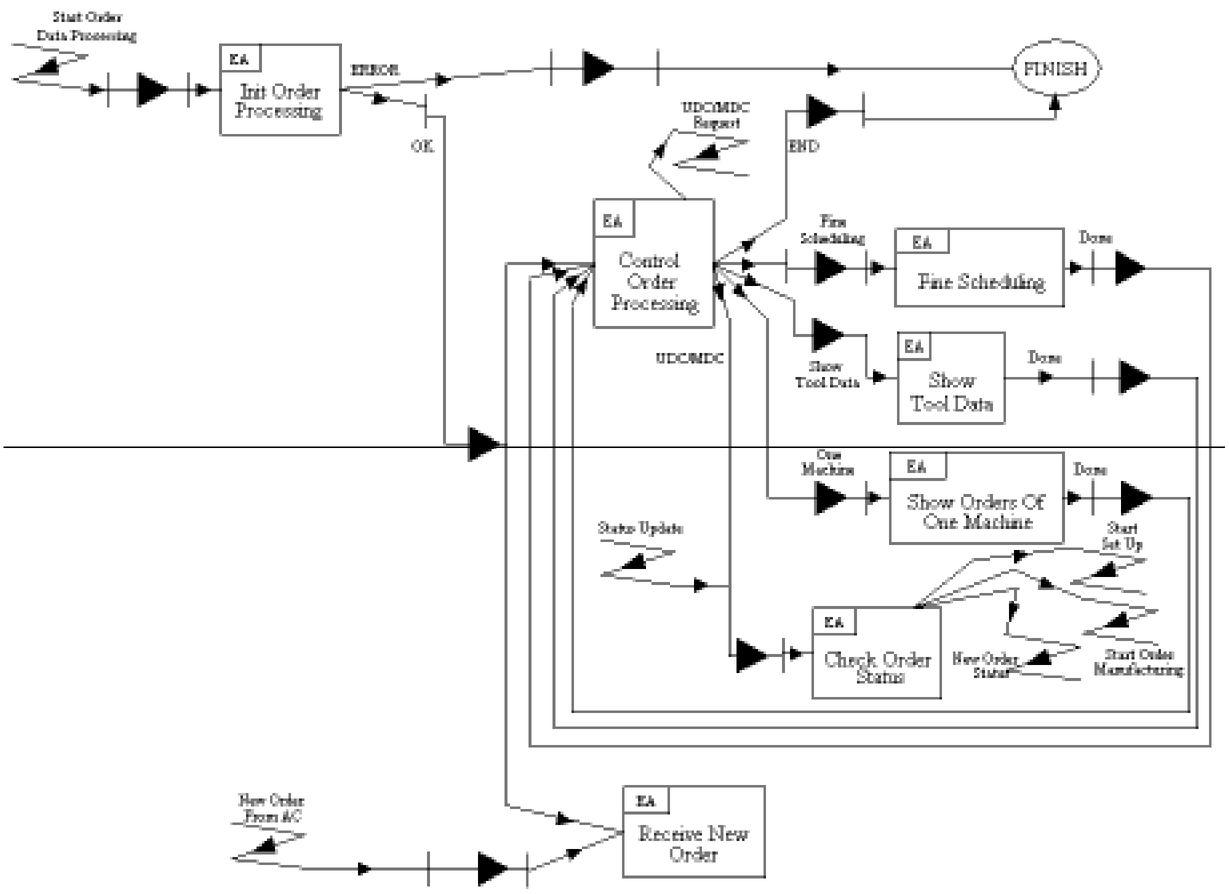


Figure A-4 Process behaviour of domain process ‘Order Processing’ (partly)

Along with defining a functional architecture, Traub also outlined a technology architecture, which was influenced by the existing infrastructure. When defining functional components, one immediately maps these components on technological ones; the functional architecture is depicted on the technology architecture. For instance, when a designer defines a control function, he decides to execute this function by a certain type of software module. In addition, the designer determines the interaction of this component with other technology components.

Together with its partner, FhG-IPK* in Berlin, Traub built a testbed for demonstrating the viability of the new production concepts. Traub initially defined three production management domains which it considered in its manufacturing system (or rather in its testbed), namely tool management, order planning (‘cell control’ in Figure A-2), and machine handling. Subsequently, Traub distributed these three functions over an area controller, a cell controller, and attached machines.

The technology architecture of the testbed is shown by Figure A-5. The area control level comprised of a DecStation 5100 for long term order planning and tool management. The area controller’s user interface was implemented via the X.11 protocol on a terminal that

* FhG-IPK: Fraunhofer-Institut für Produktionsanlagen und Konstruktionstechnik (IPK)

was connected to the area controller via TCP/IP on Ethernet. Communication with the cell controller was established via MMS. The cell controller was a 486 PC running OS/2, enabling cell controller applications to run in a multitasking environment. A MAP network connected the cell controller with shop floor devices. The cell controller received orders from the area controller, after which it processed the orders, controlling the shop floor devices via MMS. These devices consisted of a robot controller, which was connected to an industrial robot, a Traub NC machine, and a PLC that controlled a conveyor and a round-table. Whereas the PLC had a MAP interface and connected directly to the MAP network, the NC machine and the robot controller communicated with the network via a ‘protocol converter’. The converter presented functionalities of both the NC machine and the robot controller as MMS Virtual Machine Devices to the MAP network. The machine and the robot controller connected to the protocol converter via V.24/LSV2.

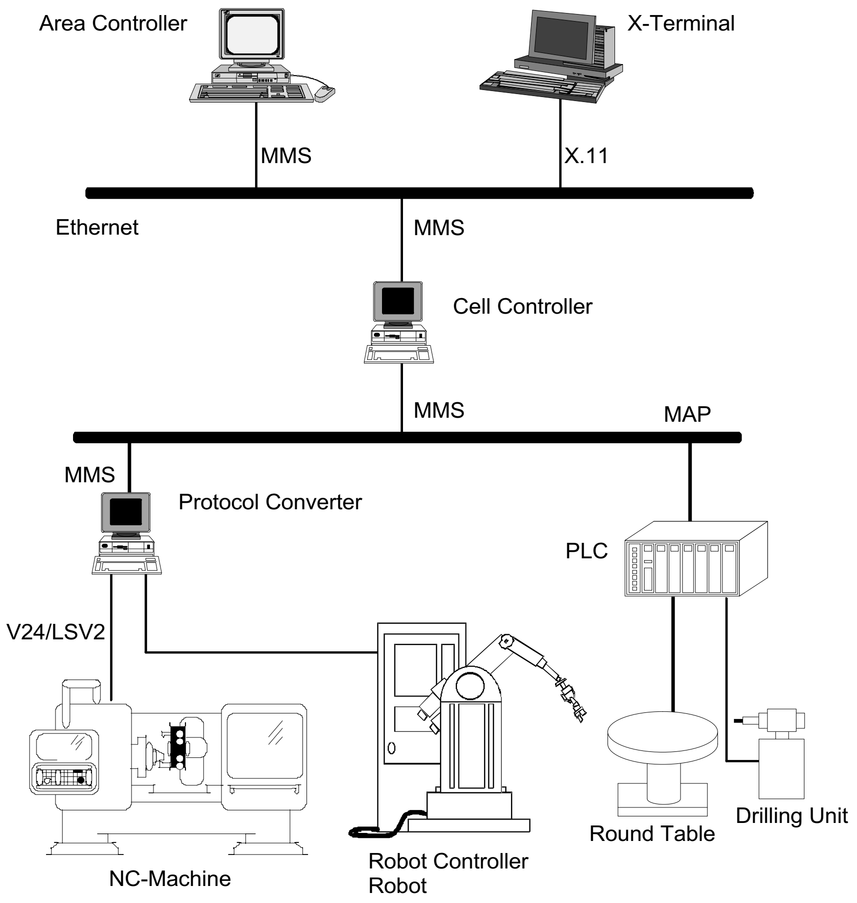


Figure A-5 Technology architecture of Traub’s testbed

A.5 Detailed design

In the second design phase, Traub specified its production control system in more detail, elaborating on the architectures that were defined in the architectural design phase. Both architecture definitions were further decomposed and worked out into detailed system specifications.

By means of the CIMOSA design specification level, a designer is able to detail a manufacturing system’s functionality, taking the model at requirements level as starting point.

The requirements definition modelling level supports a user in the definition of his system's functional architecture, whereas the role of the design level is to restructure, detail, and optimise the required functionality in a consistent model. System optimisation can be supported by simulation, taking all business and technical constraints into account. By the specification of a model at design level, a designer describes the full functionality of a CIM system, while staying within the definition of the functional architecture. If the model at design specification level reveals inconsistencies, or the model lacks optimisation, it might be necessary to adjust the model at requirements definition level.

In addition to detailing the functionality of the system, the designer specifies the technology to be employed in order to achieve the required system functionality. Simply stated, CIMOSA prescribes that for each of the most detailed specified functions, called *functional operations*, a specified resource is assigned that provides the required capabilities. The prime task of the design specification modelling level is to establish a set of (logical) resources that together provide the total set of required capabilities. Some of these required capabilities are offered by the generic services of the integrating infrastructure.

For instance, Traub made a model at design specification level, elaborating on the previously made model at requirements definition level. Traub specified the required information structure, it defined its most elementary functions, and it assigned resources to these functions. During the creation of the models, the function and information view were extensively used, whereas the resource and organisation view were barely addressed. It was not sensible to consider other factors such as responsibility (specified by constructs of the organisation view) before Traub was satisfied with the new control structure regarding functionality and information flow. Part of the specified information structure is given by Figure A-6.

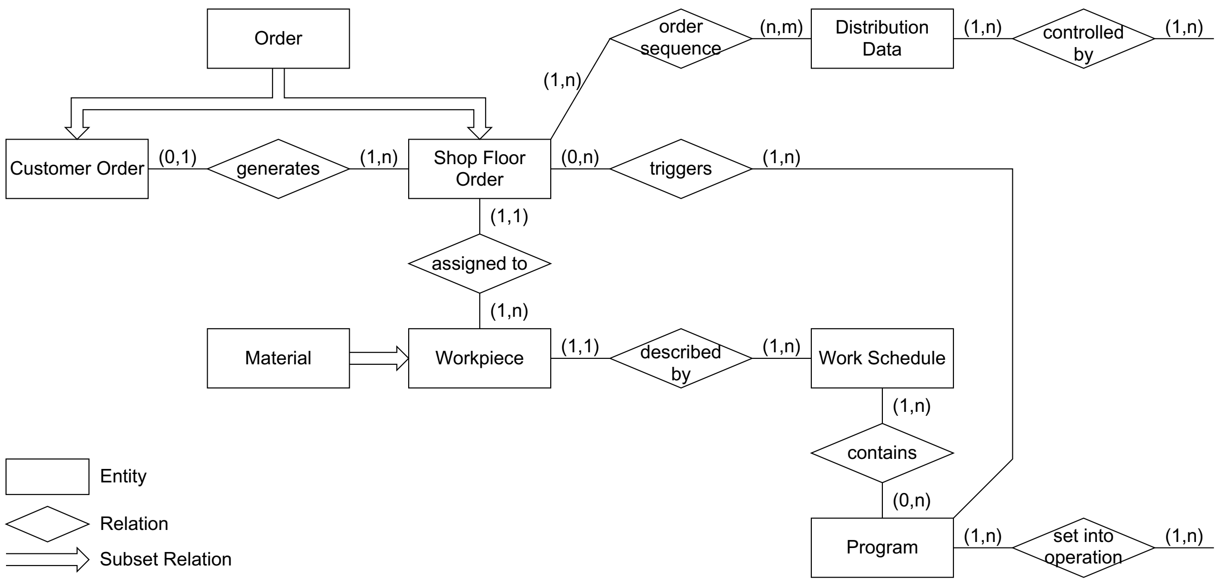


Figure A-6 Model at Particular Design Specification Level, Information View (partially)

A refinement of specified functions is accompanied by a refinement of the technology that provides the desired functions. An enterprise looks for adequate products, either commercial ones or user developed. The technology components, which are defined in the architectural design phase, are specified completely. The definitions of the CIMOSA integrating infrastructure support this specification. Then, the products that fulfil the design specifications are selected, considering enterprise policies and constraints. CIMOSA states that the final build/buy decisions should result in a model that describes the implemented system. Appropriately, CIMOSA calls this model the implementation description model. However, since the AMICE consortium defined this part of the modelling framework after Traub reorganised its production department, Traub does not have any experience with it.

In order to implement a prototype, Traub identified candidates that might help to implement software modules or that might be used as complete software modules within the testbed. From the system specification and the analysis of possible candidates, Traub defined products, tool kits, and tools to implement the functionalities of the area controller and the cell controller. Furthermore, network interfaces were adopted and products fulfilling integrating infrastructure services were chosen. For example, Oracle 6 was selected as the product that had to provide the information services. It was connected by an SQL-gateway to EasyMAP, which was used to provide the testbed's communication services. The communication services are part of the common services. In a later stage, FhG-IPK's communication platform was chosen in order to offer the desired communication services to the operational system.

A.6 Implementation

The final phase contains the implementation and release for operation of the specified CIM system. Implementation is based upon the results and decisions of the previous phases.

Implementation activities concern those tasks needed to bring the system into operation. During the implementation phase, Traub procured and built the necessary new physical components. These components were tested on the testbed and the correctness of the underlying logical model was verified. Traub decided to implement its physical system in an evolutionary way, with stable, intermediate forms. This ways, Traub hoped to achieve a stable implementation of all products and their interactions. When the system passed the tests with satisfactory results, the system was prepared for transfer to production.

Traub felt that user acceptance was a major concern. It realised that the complex changes of the manufacturing system had to be put carefully to the operators in the production environment. Therefore, after testing single components as prototypes in the test environment, training sessions for the users of the area controller and the cell controller were held. In order to make the workers familiar with the new system, parts were transferred to the production without being integrated in the existing system. In addition, the graphical user interface was adapted according to the users' needs, which was done to get a greater acceptance. When Traub believed that the operators were ready to use the new technologies, the new components were integrated in the existing system and released for operation.
Finally, the accepted production control system was released for operation. Figure A-7 shows the technology architecture of Traub's production control system as it was implemented during the reorganisation project.

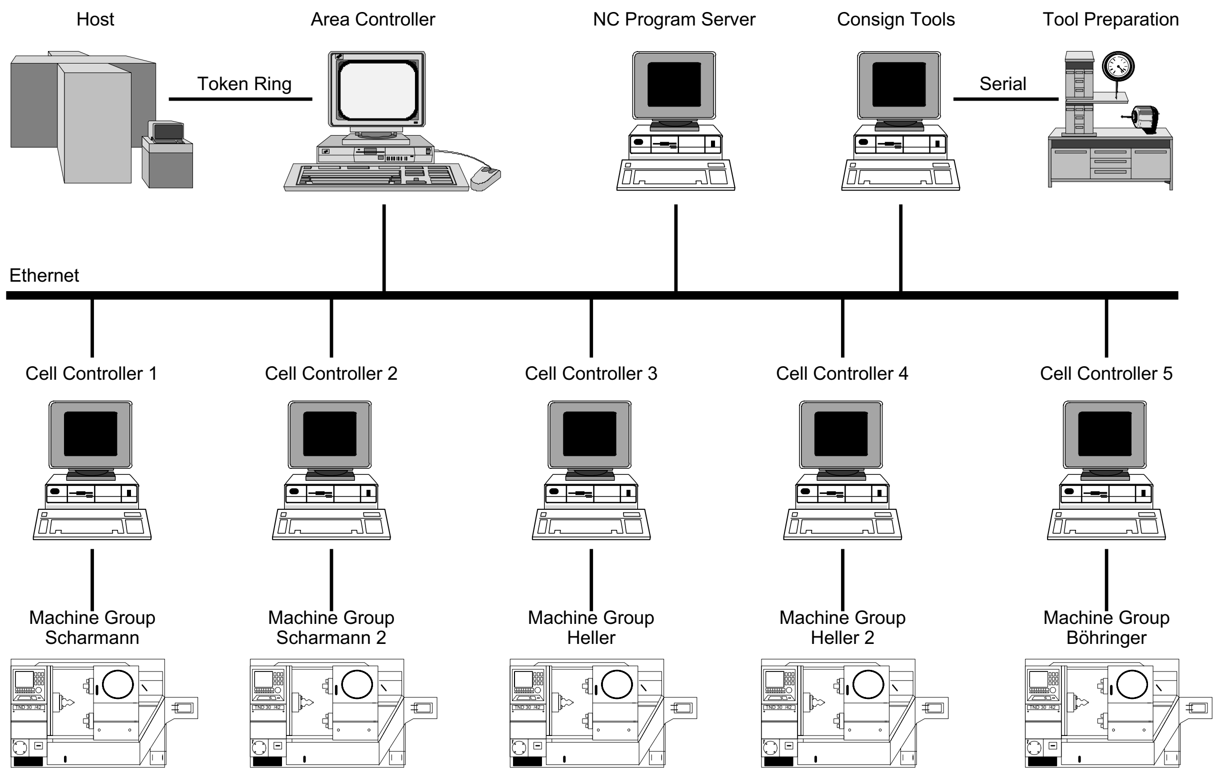


Figure A-7 Technology architecture of Traub’s new production control system

B. CIMOSA and the GERAM Framework

B.1 Introduction

By means of the GERAM framework (see Figure B-1), the overlaps and differences of enterprise reference architectures can be identified. After all, the ways enterprise reference architectures try to achieve their objectives differ from one reference architecture to the other. The following components are the minimal set of elements a reference architecture should be accompanied with:

- enterprise engineering methodologies, which can be thought of as road-maps and instructions of how to use a reference architecture in an enterprise integration project;
- modelling languages, which are needed to support enterprise integration, and which should be placed in relation to each other by means of the reference architecture;
- a modelling methodology, which comprises a set of guidelines that define the steps to be followed during a modelling activity.

In addition, the following components are elements that should preferably accompany an enterprise reference architecture:

- modelling tools, which are computer programs that help the construction, analysis, and, if applicable, the execution of enterprise models as expressed in enterprise modelling languages;
- reference models, which contain a formal description of a type (or part of an) enterprise;
- enterprise modules, which are products that implement (parts of) a reference model, for example an integrating infrastructure, or components thereof.

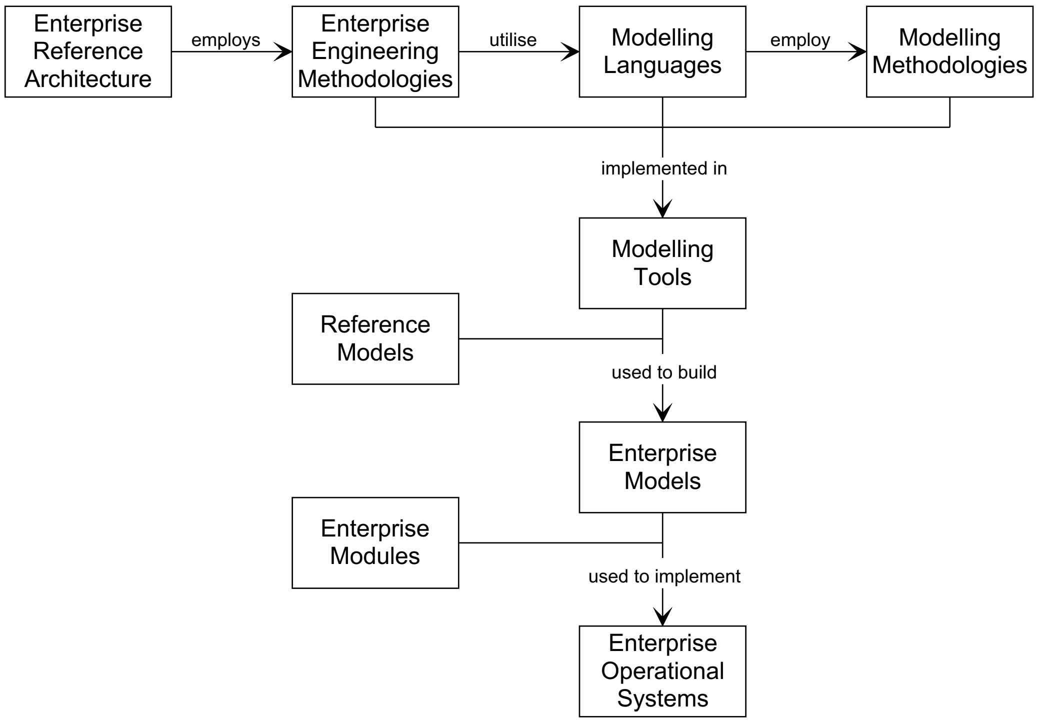


Figure B-1 Components of the GERAM framework (reprinted for convenience)

B.2 CIMOSA and the GERAM framework

This section evaluates CIMOSA's value as a suitable framework for enterprise integration in practice. For this, the sets of essential and desirable elements that should accompany reference architectures are used.

On the eligibility of the reference architecture

Subsection 5.5.1 states that the architecting concepts of domains, hierarchy, and views are represented in CIMOSA's modelling framework. A reference architecture should allow the specification of functional control architectures, either by refinement of reference models, or by design from scratch. Reference models were not used during the Traub reorganisation project; instead, models were made from scratch. These models proved their usefulness by revealing some bottlenecks and inconsistencies in the organisation (Schlotz and Röck, 1995). The Traub case shows that the CIMOSA modelling framework supports designers during the specification and analysis of functional architectures of production control systems.

Most characteristics of functional control architectures can be specified by means of the CIMOSA modelling framework (Zwegers *et al.*, 1995). Domain processes, events, and object views are adequate constructs to specify concurrent processes and the exchange of information and products between these processes. By defining domain processes and by the establishment of relations between them, any type of functional control architecture may be modelled. However, CIMOSA does not support the specification of a true technology architecture, as shown in Subsection 5.5.1.

On the inadequacy of the enterprise engineering methodology

CIMOSA lacks a 'true' engineering methodology, which provides instructions of how to go about an enterprise integration project or programme. Williams *et al.* (1994a) notice that CIMOSA does have a 'life history' for CIM systems, namely the CIMOSA system life cycle, but that this description has not been extended into a 'true' methodology. Zwegers and Gransier (1995) give a description of the engineering approaches adopted by the three industrial partners of the VOICE project, which used CIMOSA during their re-engineering trajectories. However, these engineering approaches have not been extended into a methodology either. Possible users are not supported by knowledge on how to apply CIMOSA to carry out integration projects. This point cannot be emphasised too much; an enterprise reference architecture without matching methodology defeats its own object.

On the complexity of the modelling languages

Industry almost unanimously regards the modelling languages in the modelling framework as too complex. In Traub's view, the high complexity of the CIMOSA modelling framework requires well-trained engineers (Schlotz and Röck, 1995). For example, Traub intensively used the modelling framework's function and information view, but the resource view and organisation view were barely addressed. They were of little use to Traub. In addition, the great number of constructs and their sometimes ambiguous definitions hamper a practical application. A tool was needed to unambiguously define the meaning of constructs.

On the novelty of the modelling methodologies

A modelling framework should be accompanied by guidelines, called ‘model creation processes’ by CIMOSA. A designer must be guided to navigate through the modelling framework in a consistent and optimised path, in order to ensure complete, consistent, and optimal models. During the modelling process for the Traub application, no such modelling guidelines were available. After the AMICE project finished, some guidelines have been defined, but their practical value for industry had not been established when this thesis was written.

At the moment, CIMOSA provides a methodology to guide users in the application of its modelling framework. Zelm *et al.* (1995) describe this so-called CIMOSA Business Modelling Process. As for modelling at the requirements definition level, for example, the top-down modelling process starts with the identification of domains, the definition of domain objectives and constraints, and the identification of relationships between the domains. Afterwards, the domains are decomposed into domain processes, which are further decomposed in business processes and enterprise activities. The modelling process at requirements definition level ends with some analyses and consistency checking.

Recently, new methodologies were proposed that aim to be improvements or alternatives to the CIMOSA Business Modelling Process. For instance, Reithofer (1996) proposes a bottom-up modelling methodology for the design of CIMOSA models. He claims that the CIMOSA Business Modelling Process can hardly be focused on processes and activities that are relevant to solve a concrete problem. His bottom-up modelling approach should not have this disadvantage. In addition, Janusz (1996, 1997) asserts that existing CIMOSA models of particular enterprises are not complete, not consistent, and not optimal. Also, these models often describe only functions or sub-processes limited by department borders of an enterprise. Therefore, she developed an algorithm that filters out process chains of an existing CIMOSA model. Using the algorithm, the completeness and the consistency of the considered process chains can be checked.

On the availability of the modelling tools

A modelling language should be supported by a software tool. During the creation of models, problems might occur regarding the size and complexity of the architectural models. Models become too big to be overlooked by the user, and they become inconsistent. Furthermore, the modelling process without tool support is tardy, and maintainability and extendibility of the models are jeopardised.

Some tools have been developed for modelling with CIMOSA, such as ‘GtVOICE’ (Didic *et al*, 1995) and ‘SEW-OSA’ (Aguiar *et al.*, 1994). GtVOICE was developed by the VOICE project. This tool ensures model consistency and reduces modelling time by easy model modification and extension. In addition, it provides non-ambiguous interpretation of CIMOSA constructs and a uniform way to present models among CIMOSA users. Finally, GtVOICE makes communication with other tools feasible; interfaces were made to an SQL data server and a rapid prototyping tool kit.

SEW-OSA (System Engineering Workbench for CIMOSA) was developed at the Loughborough University of Technology in England. It combines the CIMOSA concepts with Petri net theories, object-oriented design, and the services of an infrastructure called CIM-BIOSYS (Aguiar *et al.*, 1994). Both SEW-OSA and GtVOICE support the design of CIMOSA models according to the CIMOSA Business Modelling Process as described by Zelm *et al.* (1995).

On the absence of reference models

Perhaps most important for industry are guidelines that support designers to make the right architectural choices, i.e. choices that result in flexible, effective systems. Industry needs guidelines for the architectural design of systems in order to discard inadequate options early in the design process, and to enable the selection of the best options. Clearly, such guidelines are not present at the moment. However, it is not an objective of CIMOSA to provide such a prescriptive methodology. The CIMOSA modelling framework aims to support system designers with descriptive modelling of enterprise operation; it is a descriptive rather than a prescriptive framework.

Nevertheless, the CIMOSA modelling framework offers the ability to develop reference models with its partial modelling level. Best-practice reference models are the encapsulations of the prescriptive guidelines mentioned above. As such, they are recognised as major tools to support CIM system development projects (Faure *et al.*, 1995). They should be based on the architecting principles of modularity, structural stability, and layers as distinguished in Chapter 4. Aguiar and Weston (1995) signal the need to populate workbenches with a library of reference models. However, virtually no CIMOSA compliant reference models are available at this moment.

CIMOSA lacks the guidelines and reference models that are needed to make a transition from requirements to specification. After all, it prescribes how to make a specification of a system, but is does not prescribe how to design the system. In terms of Chapter 3 and 4, it offers the architecting concepts, not the principles or the business knowledge. It gives the ruler and compass to draw a house, but it does not give the construction theory.

On the promises of the integrating infrastructure

The promises of the CIMOSA integrating infrastructure seem too good to be true. Integration of hardware and software components, model execution, vendor independence, reduced maintenance, and increased application portability and flexibility appeal to companies facing a heterogeneous world. However, the CIMOSA specifications of the services as used by Traub (AMICE, 1993b) reveal many gaps. Even more, no commercial product supporting the CIMOSA specifications is present at the moment. Nevertheless, enterprises appear to be more attracted by application integration promised by the integrating infrastructure than by business integration as actually supported by the modelling framework.

C. GRAI/GIM and PERA

This appendix discusses two other enterprise reference architectures besides CIMOSA, namely the GRAI Integrated Methodology and the Purdue Enterprise Reference Architecture.

C.1 GRAI/GIM

The GRAI laboratory of the University of Bordeaux, France, has been rather active in the field of enterprise reference architectures. Besides developing their own ideas, the GRAI laboratory contributed to the ESPRIT projects IMPACS* and AMICE. Here, the main elements of what has become known as the GRAI Integrated Methodology (GIM) are described, namely a 'global model', a modelling framework, and a structured approach to guide the application of the methodology.

The global model describes the invariant parts of a CIM system: the subsystems, their relationships and their behaviour. The global model (sometimes called 'macro reference model' or 'reference model') is based upon the concepts of three activity types and their corresponding executional subsystems. These three subsystems are:

- the physical subsystem, which performs the activities of product transformation using human and technical resources;
- the decisional subsystem, which guides production towards its goals;
- the informational subsystem, which feeds the other subsystems with information.

Sometimes, a fourth subsystem is distinguished, namely the functional subsystem (Doumeingts *et al.*, 1987; Doumeingts *et al*., 1992).

The modelling framework uses $IDEF_0$ formalisms to model the physical and functional subsystems, GRAI formalisms for the decisional subsystem, and MERISE formalisms for the informational subsystem. The GRAI formalisms are shortly described; for the other formalisms, the reader is referred to (Doumeingts *et al.*, 1995). The GRAI formalisms consist of the GRAI grid and the GRAI nets. The GRAI grid allows to model a decision system. It is displayed as a table-like frame, and it uses a functional criterion to identify production management functions and a decision cycle criterion to identify decisional levels. Each function is decomposed into several levels according to the decision horizon *H* and revision period *P*. A decision period is a time interval through which decisions are valid; a revision period is a time interval at the end of which decisions are revised. The building block of a grid is a decision centre which is the intersection of a production management function and a decisional level. Decision centres are mutually connected by decision links and information links. The GRAI nets allow to model the various activities of each decision centre identified in the GRAI grid. The results of one discrete activity can be connected with the

* IMPACS: Integrated Manufacturing Planning And Control System. IMPACS was one of the first attempts to design integrated planning tools that would bridge the gap between production planning and production control.

support of another discrete activity. Since this is done for each decision centre, the links between decision centres are shown (Chen *et al.*, 1990b).

GIM's structured approach (see Figure C-1) aims to cover the entire life cycle of the manufacturing system. The approach consists of four phases: initialisation, analysis, design, and implementation. Initialisation consists of defining company objectives, the domain of the study, the personnel involved, and so on. The analysis phase results in the definition of the characteristics of the existing system in terms of four user-oriented views. The design phase is

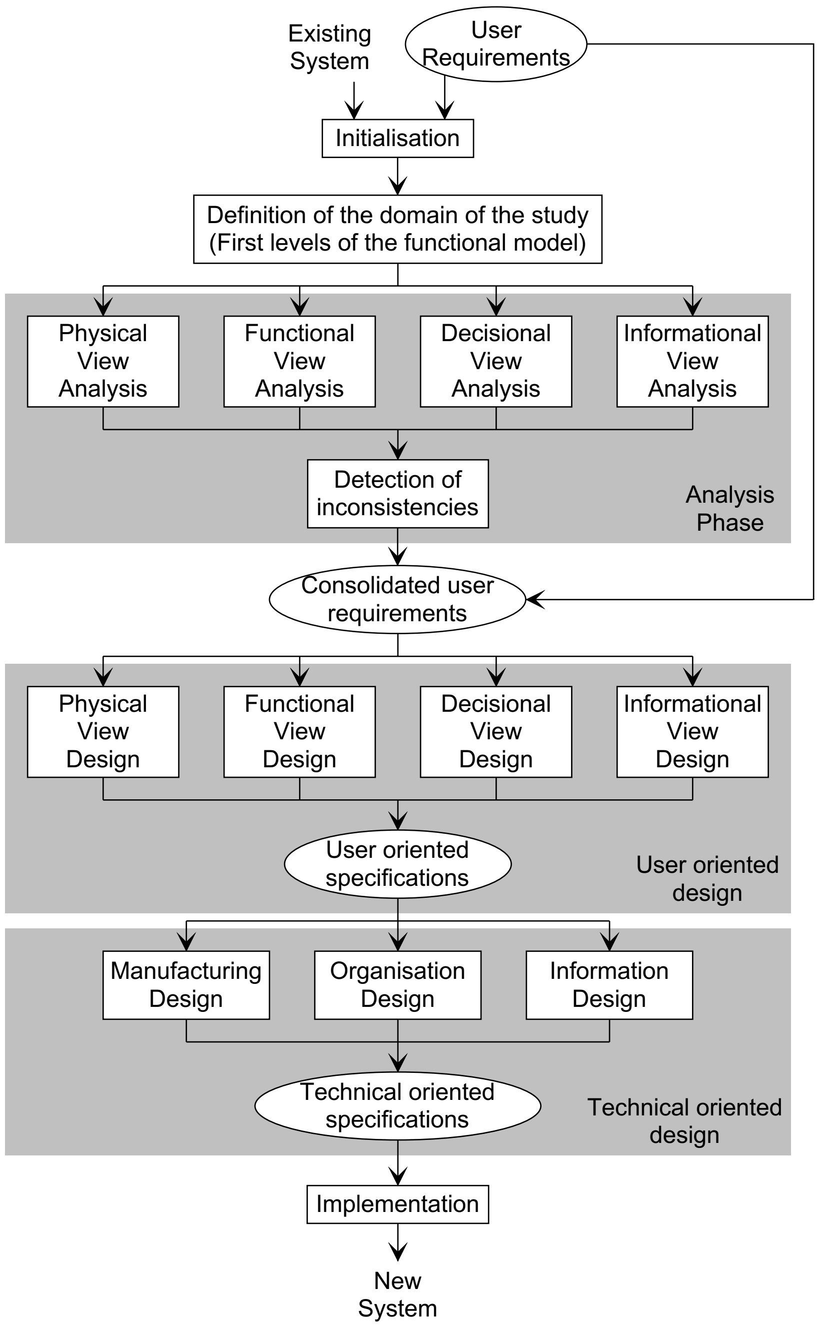


Figure C-1 GIM structured approach

Source: Doumeingts *et al.* (1995)

performed in two stages: user oriented design and technical oriented design. User-oriented design uses the results of the analysis phase to establish requirements for the new system, again in terms of the four user-oriented views. Technical-oriented design consists of transforming the user-oriented models of the new system design to technical-oriented models. These models express the system requirements in terms of the required organisation, information technology and manufacturing technology. Finally, the new system is implemented (Doumeingts *et al.*, 1995).

GRAI/GIM covers the whole life cycle of a manufacturing system, except the operation and decommission phases. Its four views differ from CIMOSA's; it introduces a decisional and physical view. However, the models of the physical view do not describe physical attributes; they describe functional attributes of physical elements. The four views used during the analysis and user oriented design phases are translated to three implementation views during the technical oriented design phase. Concerning modelling languages, GRAI/GIM is less formal than CIMOSA. After all, CIMOSA aims to achieve model execution, and that requires formal modelling languages. GRAI/GIM uses several known modelling techniques such as $IDEF_0$ and MERISE, and it developed the GRAI grids and nets. Although the GRAI laboratory has completed many projects with its modelling framework, there is no modelling methodology known in literature. Obviously, this does not imply that there is no such methodology. GIM's structured approach offers an enterprise engineering methodology. However, this structured approach is focused on the initial phases of a system life cycle; GRAI/GIM mainly supports designers during the analysis and design phases.

As for modelling tools, there is no tool known in literature that supports modelling with the GRAI/GIM modelling framework. The same applies to reference models. GRAI/GIM is based on a 'global model', a kind of generic model of an enterprise. However, no reference models that encapsulate industry type or area specific knowledge are known. GRAI/GIM does not aim to provide generic enterprise modules such as parts of an integrating infrastructure.

C.2 PERA

The Purdue Enterprise Reference Architecture (PERA) and its accompanying Purdue Methodology were developed at the Purdue University, USA. This university took a leading role in the definition of reference models for computer integrated manufacturing as well.

The Purdue Methodology is based on an extensive Instructional Manual that guides the preparation of Master Plans. According to the methodology, an overall Master Plan is necessary before attempting to implement any CIM programme. A Master Plan includes a CIM Program Proposal, i.e. a set of integrated projects whose completion will assure the success of the desired integration of the enterprise. The Purdue Enterprise Reference Architecture provides the framework for the development and use of the Instructional Manual, the resulting Master Plan, and the ultimately implemented CIM Program Proposal. PERA is the glue that holds all aspects of the CIM programme together (Williams, 1994).

Figure C-2 shows that the Purdue Enterprise Reference Architecture is characterised by the layered structure of its life cycle diagram. Starting with the Enterprise Business Entity, it leads to a description of the management's mission, vision, and values for the entity under consideration. From these, the operational policies are derived for all elements of concern. Two kinds of requirements are derived from the policies, namely those defining information-type tasks and those defining physical manufacturing tasks. Tasks become collected into modules or functions, which at their turn are connected into networks of information or of material and energy flow. Then, technology choices are made; the place of the human in the information architecture and in the manufacturing architecture is defined. The result of the technology choices are three implementation architectures, namely the information systems architecture, the human and organisational architecture, and the manufacturing equipment architecture. After functional and detailed design, the life cycle goes through the construction and operation phases, after which it is disposed of (Williams, 1994).

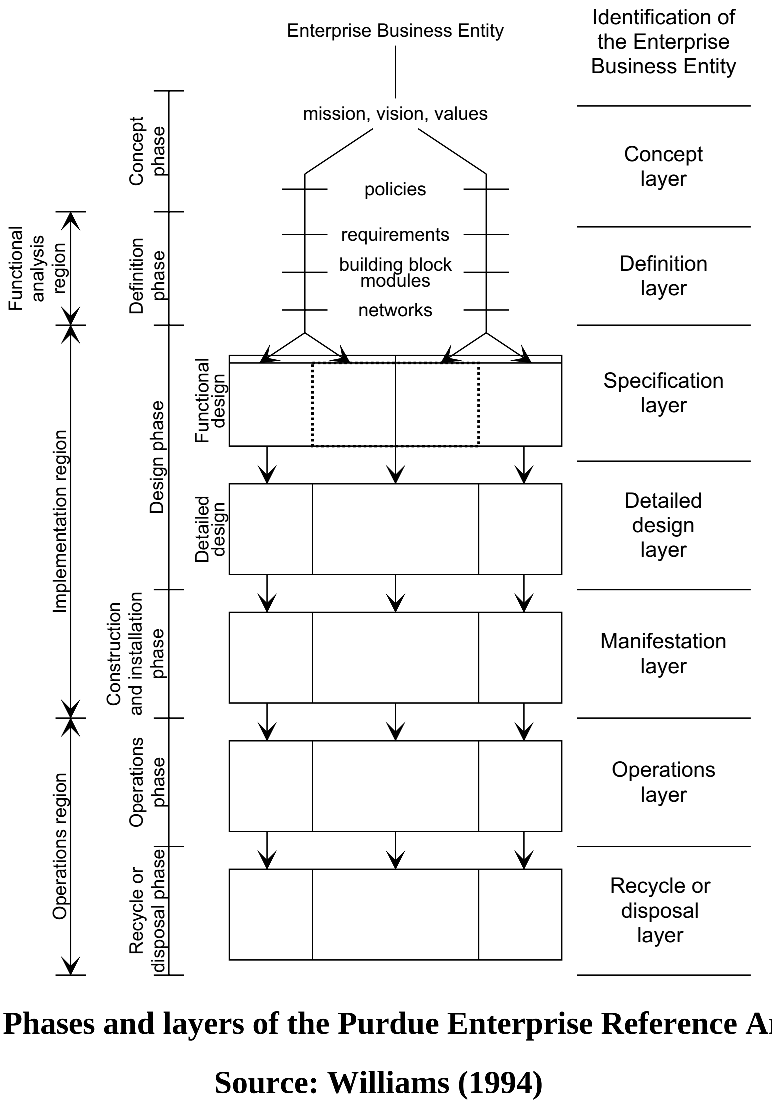


Figure C-2 Phases and layers of the Purdue Enterprise Reference Architecture

Source: Williams (1994)

A specific feature of PERA is the emphasis it puts on the role of humans. By defining the functions that will be performed by humans, the information and manufacturing architecture are converted in the information systems architecture, the human and organisational architecture, and the manufacturing equipment architecture. Figure C-3 illustrates that this definition involves three 'lines'. The automatability line shows the absolute extent of pure technological capability to automate tasks and functions, and is limited by the fact that many tasks and functions require human innovation and cannot be automated with presently available technology. The humanisability line shows the maximum extent to which humans can be used to implement tasks and functions, and is limited by human abilities in speed of response, physical strength, and so on. The extent of automation line shows the actual degree of automation carried out or planned in the system. This third line defines the boundary between the human and organisational architecture and the information systems architecture on the one hand, and the boundary between the human and organisational architecture and the manufacturing equipment architecture on the other side. Its location is influenced by economic, political, social, and technological factors (Rathwell and Williams, 1996).

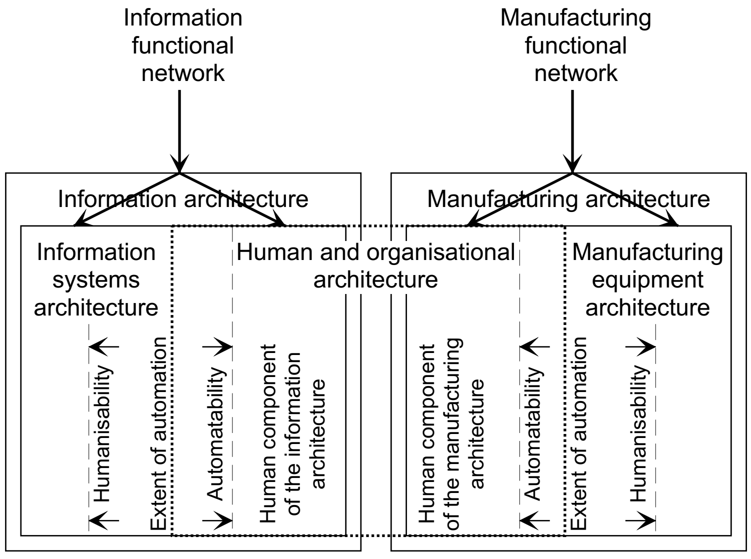

Information functional network
Manufacturing functional network
Information architecture
Manufacturing architecture
Information systems architecture
Human and organisational architecture
Manufacturing equipment architecture
Humanisability
Extent of automation
Automatability
Human component of the information architecture
Human component of the manufacturing architecture
Automatability
Extent of automation
Humanisability

Figure C-3 Humanisability, automatability, and extent of automation to define the three implementation architectures

Source: Rathwell and Williams (1996)

PERA explicitly takes into account the role of the human in a manufacturing system. In addition, it does not distinguish between model content views such as function, information, and resource views, but rather between purpose views (information and manufacturing architecture) and implementation views (human and organisational architecture, information systems architecture, and manufacturing equipment architecture). The Purdue Methodology offers an enterprise engineering methodology which covers all phases of a system life cycle. Of the three enterprise reference architectures described in this chapter, PERA is clearly accompanied by the most extensive methodology. As for modelling languages, PERA offers only the task module construct. Information system tasks, manufacturing tasks, and human-based tasks are modelled by means of this construct. By combining the various task modules into networks, a type of data-flow or material and energy flow diagram results. There are no modelling methodologies known.

In addition, no modelling tool is known in literature. Models might become large and without a tool their accuracy over time is jeopardised. Earlier, the developers of PERA had defined a reference model for computer integrated manufacturing (Williams, 1989). This reference model is restricted to the automatable functions of an enterprise and all functions that might require human innovation are considered as external entities. PERA does not provide components of an integrating infrastructure or any other generic enterprise modules.

D. Controller Architectures

D.1 Introduction

In this appendix, the architecture of the controllers in Section 6.3 (see Figure 6.4) is central. Coming from a high perspective of control levels and basic forms, this appendix zooms into the building blocks that make up the various basic forms. It focuses on three sets of building blocks, namely those of the Production Activity Control and Factory Coordination modules as outlined by Bauer *et al.* (1991), the Factory Activity Control concepts as described by Arentsen (1995), and the reference model for holonic manufacturing systems as defined by the PMA group of the Catholic University of Leuven (Van Brussel *et al.*, 1998).

The Production Activity Control and Factory Coordination modules as developed by the ESPRIT project COSIMA are quite well-known proper hierarchical control concepts. Many applications based on COSIMA's ideas have been reported. The ideas are mainly focused on the make-to-stock and assemble-to-order production environments.

Less well-known are the Factory Activity Control concepts from the University of Twente, the Netherlands. However, their ideas to achieve predictive and reactive behaviour from a shop floor control system can be characterised as a completion of the modified hierarchical control form. Factory Activity Control mainly focuses on the make- and engineer-to-order production environments.

The reference model for holonic manufacturing systems is relatively new. It is an example of the state-of-the-art in manufacturing control. The reference model does not focus on a specific type of production environment, although the ideas have been inspired by discrete manufacturing.

D.2 Production Activity Control (PAC) and Factory Coordination (FC)

The ESPRIT project COSIMA (Control System for Integrated Manufacturing) defined a hierarchical architecture for shop floor control consisting of Production Activity Control modules and the Factory Coordination module (Bauer *et al.*, 1991). Production planning and control is seen in terms of three hierarchical levels of activity: strategic activities, tactical activities, and operational activities.

Shop floor control involves the operational activities of Factory Coordination and Production Activity Control (PAC). Bauer *et al.* consider the factory to be composed of a series of group technology-based manufacturing cells, where each cell is responsible for a family of its products, components, or processes, and each cell is controlled by a PAC system. A PAC system provides the necessary functions to control the flow of products within a cell. The Factory Coordination layer ensures that the individual cells interact to meet an overall production plan; it coordinates a group of cells on a shop floor. This overall production plan relates to the work orders that need to be scheduled and executed in the cells. The work orders

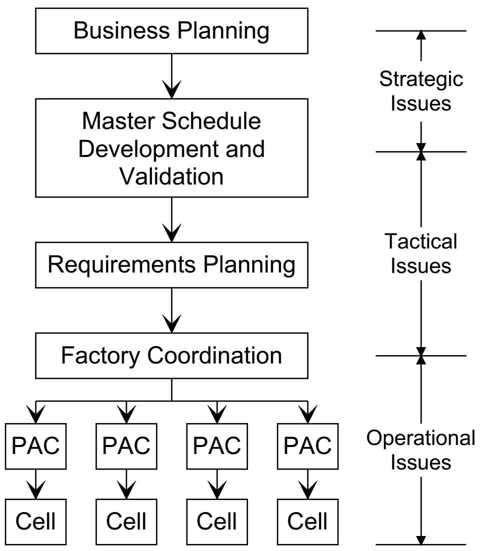


Figure D-1 Architecture for production management systems

Source: Bauer *et al.* (1991)

are input from a requirements planning layer that in most cases includes an MRP application (Higgins *et al.*, 1996).

Originally, PAC was seen as residing at the next level below an MRP system (Browne, 1988). Later, the designers of PAC envisioned a Factory Coordination system between the requirements planning and PAC system. The original view resembles the ideas of Bertrand *et al.* (1990a), who define a Goods Flow Control level and a Production Unit Control level that correspond respectively to the tactical and the Production Activity Control levels of Bauer *et al.* (1991).

The five building blocks of the Production Activity Control architecture are shown in Figure D-2. The Scheduler accepts the production requirements from a higher planning system (i.e. a Factory Coordination system), and develops a schedule over a specified time period for the manufacturing cell, based on known process routings and expected capacity at the cell. The Dispatcher issues commands to the movers and producers, based on the schedule, the current state of the production environment, and manufacturing data describing how tasks are to be performed. In a sense, the dispatching function is a real-time scheduler which assigns jobs to resources based on real-time information on the state of the jobs and the resources (Browne, 1988). Based on the instructions from the Dispatcher, the Producer controls the execution of the various operations in the cell. It isolates the physical level of production devices from the control level by translating different instructions from the Dispatcher into specific device instructions. A Mover organises the handling of materials between workstations within a cell by following the Dispatcher’s commands. The Monitor provides real-time and historical feedback to the other PAC control functions. The Monitor’s feedback allows for real-time dispatching of parts and/or materials, reporting on work in progress and inventory levels, and reporting on how well the schedule is being adhered to (Higgins and Browne, 1990).

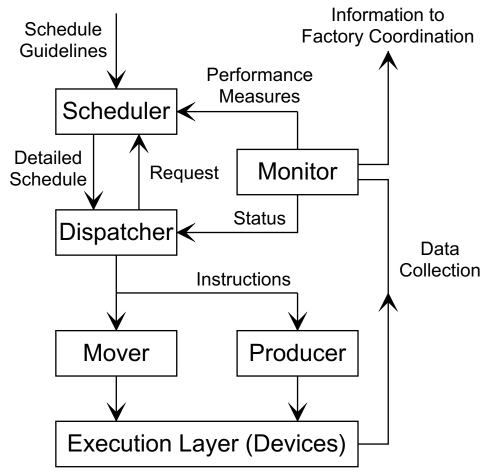


Figure D-2 Production Activity Control

Source: Bauer *et al.* (1991)

The Factory Coordination (FC) layer is divided into two tasks, namely the Production Environment Design task, and the Control task. The Production Environment Design task reorganises the product-based layout of the manufacturing system in order to simplify it and to accommodate new products coming into production. The main purpose of the Control task is to coordinate the activities of each PAC system through the provision of schedule and real-time control guidelines, while recognising that each PAC system is responsible for the activities within its own cell. Figure D-3 shows that the Factory Coordination control task is in many ways a higher level recursion of a PAC system. It involves developing schedule guidelines using a factory level Scheduler, implementing these guidelines and providing real-time guidelines for each of the PAC systems using a factory level Dispatcher, and monitoring the progress of the schedule using a factory level Monitor (Bauer *et al.*, 1991).

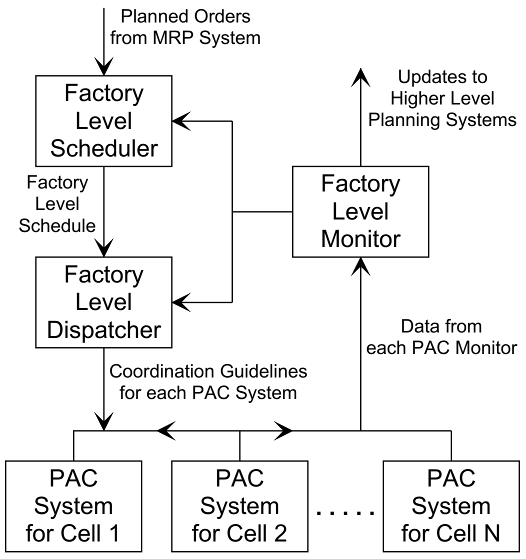


Figure D-3 Factory Coordination

Source: Bauer *et al.* (1991)

Some authors suggest modifications to the standard work of Bauer *et al.* For example, studies by Chalmers University of Technology in Göteborg, Sweden, notice two weak points: PAC’s information handling is highly centralised due to the way the Monitor is defined, and the mapping of the PAC system on the physical manufacturing system is weak due to the definition of the Mover and the Producer (Maglica, 1995). Therefore, they suggest PAC+ where the Producer and Mover are split into several activities, each responsible for the control of one workstation. Furthermore, the Producers and Movers are enabled to communicate directly with each other, permitting them to synchronise directly without involving the Dispatcher or the Monitor. In PAC+, the Monitor function is disconnected from the equipment level, forcing data collection by the Monitor through the Produce and Move functions. Chalmers University also defines PAC++ which aims to extend its predecessors with a generic data model that must be reusable in various applications (CRISC, 1997).

To conclude, the architecture for production management systems as outlined by Bauer *et al.* (1991) is based on the proper hierarchical form. Commands are passed from top to bottom and status reporting is flowing in the reverse direction. The architectures for PAC and FC are possible completions of cell and shop level controllers respectively. An architecture for controllers on workstation level is lacking. The PAC architecture is generally acknowledged as a sound basis for a cell controller, mainly because of its clear distinction between Scheduling, Dispatching, and Monitoring. However, the modifications as suggested by Maglica (1995) provide an improved version of PAC. Splitting the Mover and the Producer into several such entities that correspond to individual workstations in the cell, and moving parts of the information-handling from the Monitor to the new workstation controllers results in a more solid workstation level.

D.3 Reference models for the heterarchical control form

No reference models for the heterarchical control form were known in literature when this thesis was written. Some applications had been reported, but no reference models had been defined.

Nevertheless, some research efforts might provide a suitable basis for a reference model. For example, Duffie and Prabhu (1994; 1996) formulate a number of excellent design principles for heterarchical systems. Cantamessa (1995; 1997) gives a classification of agent based control systems, which belong to the heterarchical control form. Reference models might be defined on the basis of these design principles and classification.

D.4 Factory Activity Control

The Factory Activity Control concepts from the University of Twente, the Netherlands, can be characterised as a completion of the modified hierarchical control form. At this university, various projects have been carried out aimed at the development of a generic concept for Factory Activity Control in part manufacturing and assembly. Factory Activity Control refers to the operational control of all activities which take place on the shop floor and in the factory office. The developed concept should meet the requirements of both a predictive and a reactive behaviour which is needed for the make- or engineer-to-order manufacturing environments. An essential feature of the concept is that it addresses both the main and auxiliary tasks which have to be performed for the execution of the production jobs. The main tasks concern the actual manufacturing of the products. The auxiliary tasks deal with the preparation and the management of the auxiliary resources and the control of inspection tasks. For example, technological planning functions are also dealt with. These concern the generation of technical information that is required to manufacture the products, for instance process planning in an engineer-to-order environment. A concrete result of the projects at the University of Twente, is FACT, a prototype that implements the developed ideas on shop floor control (FACT, 1997; Van Sorge, 1997).

FACT (Factory Activity Control Technology) is based on four hierarchical production planning and control levels. The highest level, the company level, consists of functions for client order entrance and master production planning. The next highest level, the factory level, deals amongst others with capacity planning on the basis of the available manufacturing resources. FACT is an implementation of the cell and station levels, which concern the operational planning and execution of all manufacturing activities.

Figure D-4 shows that the cell control level consists of the functions Scheduling, Dispatching, Monitoring, and Diagnostics (Arentsen, 1995). The Scheduling function transforms production suborders into production jobs, allocates the jobs to the stations, and determines the sequence and both the starting and completion times of the jobs. It is capable of multi-resource scheduling, i.e. workstations as well as tools, fixtures, operators, and so on, are scheduled (see for example (Meester, 1996)). After a workplan has been generated, the resulting workload of the individual workstations has to be checked. In order to allow workload balancing, the technical information of the jobs specifies alternative resources within the same manufacturing cell. The Dispatching function releases jobs to the stations, based on the workplan. Before it releases a job, it has to check the availability of the required technical information. The Monitoring function receives status information about the jobs and the stations, and informs the Dispatching function of unexpected changes in the status.

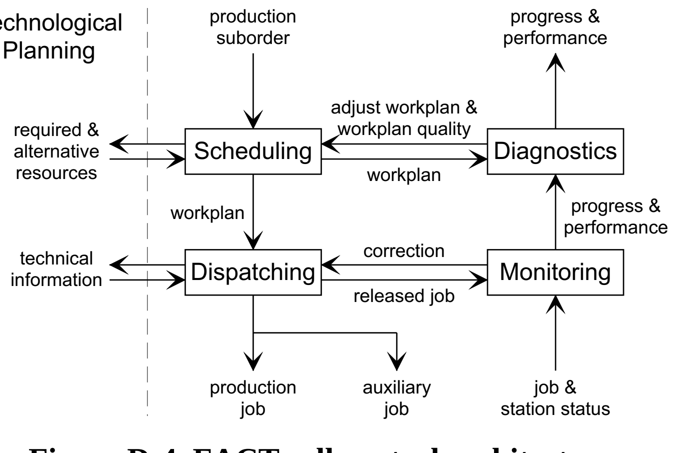


Figure D-4 FACT cell control architecture

Source: Arentsen (1995)

It passes information on both the progress and the disturbances to the Diagnostics function. The Diagnostics function evaluates the quality of the generated workplans. Furthermore, it informs the Scheduling function about expected consequences of a disturbance for the workplan. The Diagnostics function also performs the feedback to the factory control level.

The station control architecture consists of the four control functions Sequencing, Dispatching, Monitoring, and Diagnostics, as is shown in Figure D-5. Station control is actually a lower level recursion of cell control, and therefore not treated into detail. Station control provides cell control with an on-line connection to the shop floor and closes the loop between the planning and the execution phases of a job. Two types of stations are distinguished: workstations and auxiliary stations, which perform production and supporting activities respectively.

Stations have the possibility to solve a problem by mutual arrangement without bothering the cell level about it. For this reason, the Diagnostics function can send a so-called ‘direct request’ to the Sequencing function of another station. Although it can formally be considered as a job, it specifically deals with a high priority request for help in an unexpected situation. For example, in the case of a tool breakage, a direct request is sent to the auxiliary station Tool Station Control. Only if the other station is unable to solve the problem adequately, the higher level must be informed. This way, FACT’s control structure claims to meet the requirements of both a predictive and reactive behaviour needed for the make- or engineer-to-order manufacturing environments.

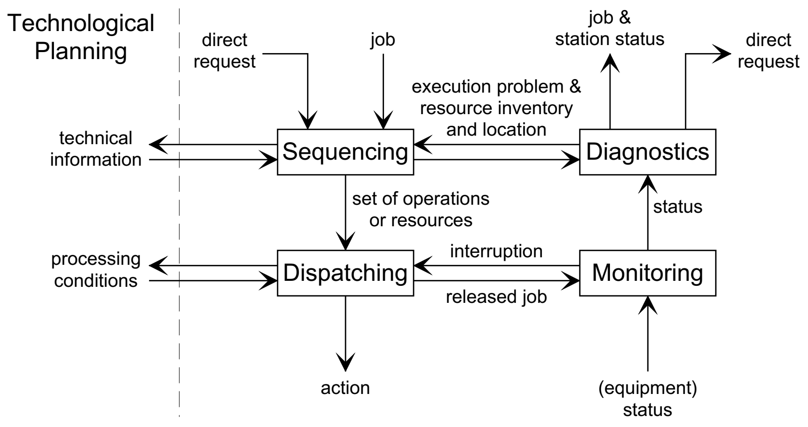


Figure D-5 FACT station control architecture

Source: Arentsen (1995)

D.5 Building blocks of a holonic manufacturing system

The Catholic University of Leuven distinguishes three types of basic building blocks that make up a holonic manufacturing system: resource holons, product holons, and order holons. Figure D-6 shows the three basic types of holons and the knowledge about the manufacturing system they exchange (Bongaerts *et al.*, 1996; Van Brussel *et al.*, 1998).

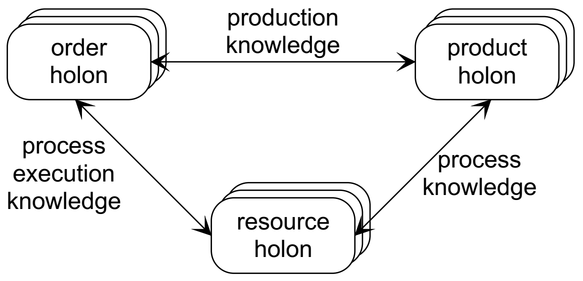


Figure D-6 Basic building blocks of a holonic manufacturing system

Source: Van Brussel *et al.* (1998)

Resource holons contain the knowledge to organise, use, and control production resources to drive production. A production holon holds the process and product knowledge to assure the correct making of the product with sufficient quality. An order holon represents a task in the manufacturing system, and is responsible for performing the assigned work correctly and on time.

Specialisation and aggregation relations exist between the basic holons. For example, robots, NC-machines, and conveyors might be defined as specialisations of the equipment holon, which at its turn is a specialisation of the basic resource holon. Aggregation relationships might be defined as well; for example, a shop holon might consist of workstation holons, which at their turn consist of equipment holons.

In addition to the basic holons, *staff holons* might be present in the holonic manufacturing system. Staff holons, such as a scheduler for a shop, assist the basic holons in performing their work. They allow for the presence of central elements in the architecture. If the staff holons provide good advise, the basic holons will follow this advice as well as possible. The extent to which the basic holons follow the advice of staff holons is determined by the basic rules that determine the cooperation of the holons and thereby limit the autonomy of the individual holons. For more information, the reader is referred to (Van Brussel *et al.*, 1998).

E. Simulation of the Model Factory's Agent Based Control System

This appendix describes the simulation model of the agent based control system of the model factory. Furthermore, simulation experiments of this control system and a control system without negotiation are presented.

E.1 Introduction to the specification language χ

Arends (1996) describes the syntax and semantics of a systems engineering specification formalism, called χ. This formalism has been developed at Eindhoven University of Technology, department of Mechanical Engineering. It can be used for the specification and simulation of industrial systems. It supports modularity and allows separate descriptions of the structure and of the components' behaviour. A specific feature of χ is the clear representation and unambiguous specification of interfaces between components. Furthermore, it is well suited for the description of autonomous components cooperating with each other by exchanging information (Arends, 1996; Van de Mortel-Fronczak *et al.*, 1995; Van de Mortel-Fronczak and Rooda, 1996; Van de Mortel-Fronczak and Rooda, 1997).

A system is treated as a collection of concurrently operating sequential components. A system component is modelled by a process as a sequential program where changes in the state of a process are accomplished by performing actions. Interaction between components is modelled by 'send' and 'receive' actions along fixed communication channels. A process is specified by a program in a CSP-like specification language preceded by Pascal-like declarations of local variables and statistical distributions. Processes do not share variables – they interact exclusively by using the communication and synchronisation primitives (synchronous message-passing). Extensive examples of the specification language χ may be found in (Van de Mortel-Fronczak *et al.*, 1995; Van de Mortel-Fronczak and Rooda, 1996; Chi, 1996; Rooda, 1996; Van de Mortel-Fronczak and Rooda, 1997).

E.2 Simulation model

Figure E-1 shows the simulation model of the model factory. It consists of a generator (GEN), a network (SEL), a component store (COS), and six workstations, namely the raw material store (RMS), the screen printer (SCP), two component placement stations (CP1 and CP2), a reflow and cleaning station (RCL), and the final product store (FPS). The workstations are divided in two parts: a physical part operating on batches, and a control process controlling the physical part. A control process sends station numbers to the physical part, so the latter is able to forward a batch to its next destination. The control process receives batch numbers from the physical part, if the latter detects a batch arriving at the workstation. Messages are exchanged between the various (control) processes via the network. The thick lines represent the physical flow of batches through the model factory. Components flow from the component store to the component placement stations.

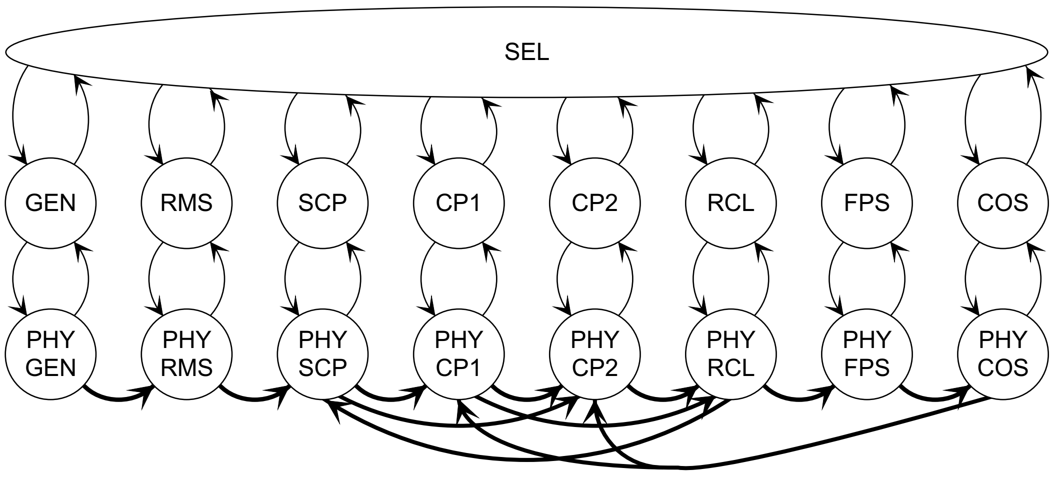


Figure E-1 Simulation model of the model factory

Figure E-2 shows that the control part of a workstation consists of a network interface (NIN), a machine controller (MAC), and a process termed the workstation controller (WSC), which incorporates the negotiation capabilities of an agent. The workstation controller determines the actions taken by the workstation, depending on the type of message received from other workstations via the switch element and the network interface. For instance, if a task announcement is received, and if the workstation is able to perform the requested operation, the workstation controller returns a bid.

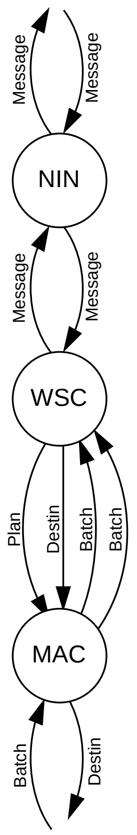


Figure E-2 Control part of a workstation

The machine controller controls the physical part of the workstation, which is evidently dependent on the type of station. The physical part performs the actual operations. The machine controller receives notifications of batch arrivals from the physical part of the workstation, and forwards these messages to the workstation controller. The latter sends a work plan to the machine controller. The machine controller notifies the workstation controller when the operation is finished. Then, the workstation controller sends the number of the next workstation to the machine controller, which forwards this information to the physical part.

Figure E-3 represents the generic structure of the workstation controllers. The following components are distinguished: a request handler (REQ), a subcontractor (SUB), a controller (CON), a database (DBS), and a sender (SEN).

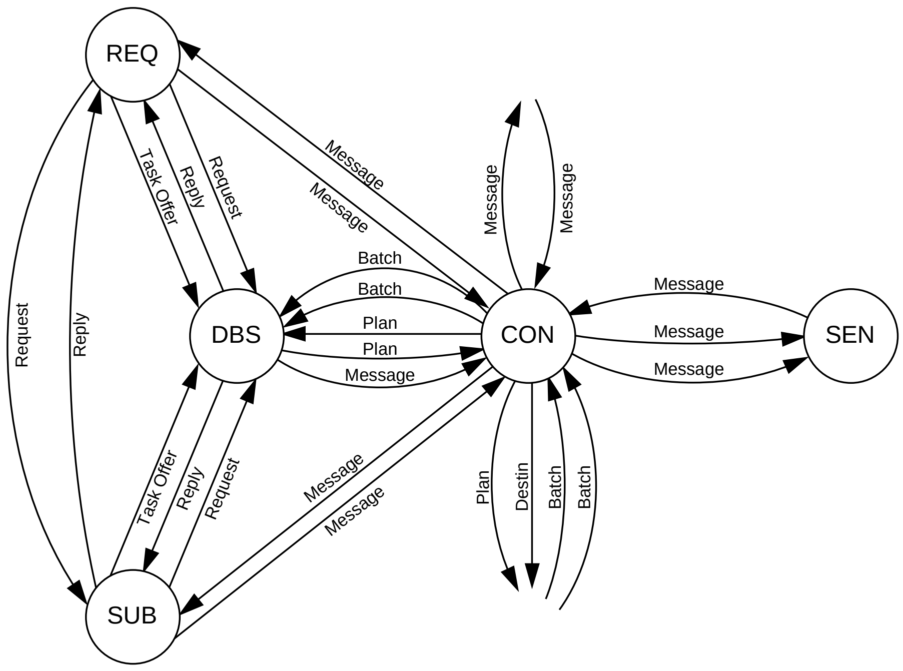


Figure E-3 Workstation controller model

The request handler issues bids as replies to incoming task announcements. Both bids and task announcements are examples of data exchanges called 'Message' in Figure E-3. To issue a bid, the request handler needs information from the database and possible subcontractors. The subcontractor puts subcontracts out to tender to other agents, in order to divide the current process steps among the agent itself and other agents. The controller coordinates the various components of an agent. Together with the network interface, it also provides the interface with the outside world; all messages from/to other agents, such as negotiations between a predecessor and the request handler, pass through the controller. Furthermore, it commands the machine controller to start the operation on a batch. The database stores run-time information of the agent. It sends information to the request handler, subcontractor, and controller upon request. The sender is responsible for the continuation of the batch. It sends task announcements, receives and evaluates incoming bids, and sends a task offer to the agent with the best bid. For a complete functional specification, the reader is referred to (Zwegers *et al.*, 1997a).

For instance, Figure E-4 shows the messages between the controller, the request handler, and the database if a task announcement message arrives. If the controller receives a task announcement from another agent via the network interface, it forwards the announcement to the request handler. This process forwards a request to the database. If the workstation is able to perform the requested operation, the database checks the workstation's schedule, and replies with a provisional start and end date for the job. The request handler determines whether the job can be subcontracted. If subcontracting could be possible, a request is sent to the subcontractor that issues subcontracting task announcements, and receives and evaluates incoming bids. The subcontractor sends the best subcontracting bid to the request handler. Note that subcontracting is not shown in Figure E-4. The request handler sends a bid via the controller to another agent from which the original task announcement was received. If the bid is accepted, a task offer is received. Again, the reader is referred to (Zwegers *et al.*, 1997a) for a complete specification.

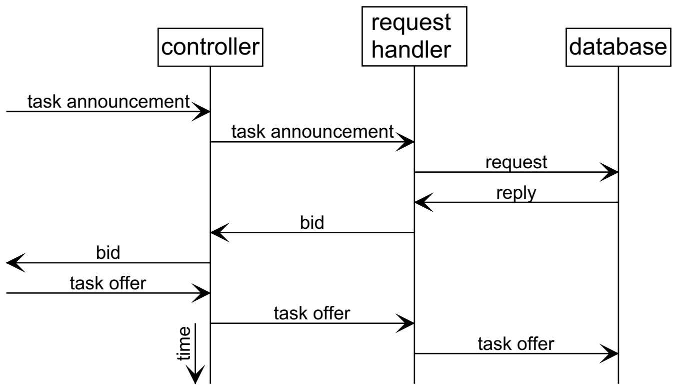


Figure E-4 Message sequence chart for a task announcement arrival

E.3 Simulation results

In this appendix, only the most important experiments are described. Before that, it is necessary to explain a few terms. The following definitions are based on (Hopp and Spearman, 1996):

- The *work in process (WIP)* is the inventory between the start and end points of a product routing. Here, the work in process is not expressed in number of jobs, but in total required operation time (that is, the sum of the work contents of all released jobs that are not finished yet). The WIP norm is the maximum WIP allowed in the system at a certain moment.
- The *throughput (TH)* is the average quantity of good parts produced per unit of time. The upper limit on the throughput of any workstation is its capacity. Here, the throughput is expressed in total required operation time per unit of time.

- The *cycle time (CT)* of a given routing is the average time from release of a job at the beginning of the routing until it reaches an inventory point at the end of the routing (that is, the time the part spends as WIP).

In the experiments, the WIP norm is the independent variable. Jobs are released into the system depending on the current level of work in process. Figure E-5 shows the throughput and cycle time versus the work in process for the model factory.

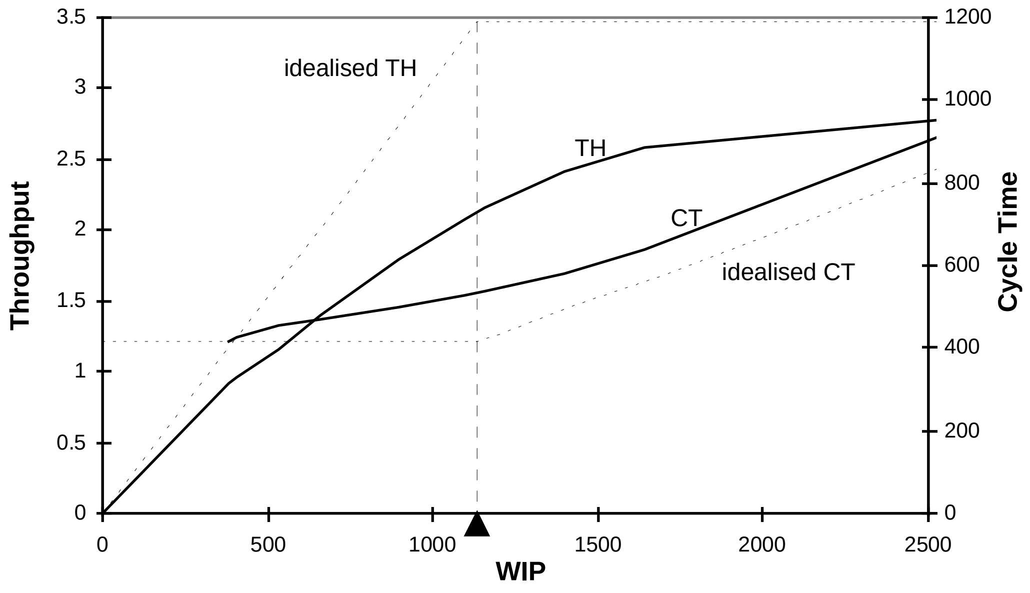


Figure E-5 Throughput and cycle time versus WIP

Companies strive for high throughputs and low cycle times. Figure E-5 displays the idealised throughput and cycle time as well. For ideal systems with zero variability, increasing the work in process results in more throughput while maintaining a constant cycle time. However, after a certain point the throughput remains constant and the cycle time increases if work in process is increased. This point is termed the *critical work in process*, and is indicated by the triangle in Figure E-5. The (idealised) throughput is limited by the capacity of the system. Similarly, the (idealised) cycle time is limited by the sum of operation and transportation times needed for a job.

Figure E-5 reveals a fundamental relationship between work in process, cycle time, and throughput. At every WIP level, the ratio of work in process to cycle time yields throughput. This relation holds for both the real and idealised characteristics, and is known as Little's Law (Smit, 1992; Hopp and Spearman, 1996):

$$TH = \frac{WIP}{CT}$$

In another experiment, the effect of negotiation is determined. Compared to the first experiment, there is no negotiation between workstation agents, and the generator decides which workstation executes which operation. The results of the experiment are shown in Figure E-6. The figure also presents the results with subcontracting (identical to Figure E-5).

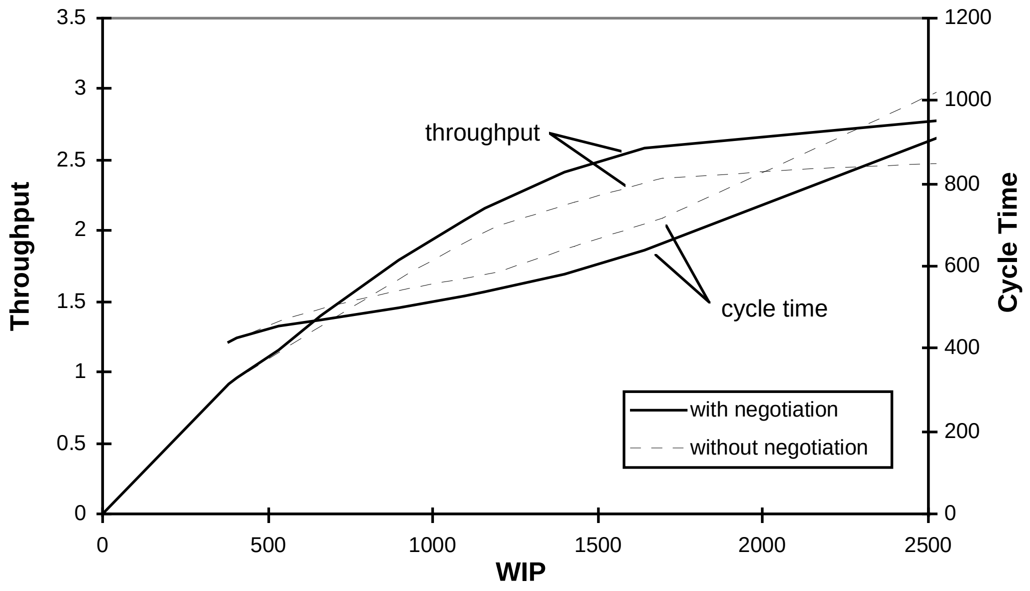


Figure E-6 The effect of negotiation on throughput and cycle time

As expected, a clear difference is shown in the results with and without negotiation. Both the throughput and the cycle time curves of the experiment with negotiation are 'better' than those of the experiment without negotiation. However, the agent based control system demands a lot of message traffic via the network. For descriptions of more experiments, the reader is referred to (Zwegers *et al.*, 1997a).

This appendix describes a χ model of the agent based control system of the model factory. The simulation experiments conducted with the model show that this relatively new type of control system performs better than a heterarchical control system without negotiation. For a certain WIP level, the throughput and cycle time of the experiment with negotiation are closer to the idealised throughput and cycle time than those of the experiment without negotiation.

Summary

Shop floor control systems are becoming more complex, and more difficult to manage. They can hardly be maintained or modified anymore. A change in a component might lead to a chain reaction, i.e. changes in more components. This thesis aims to provide a contribution to solving or avoiding these problems. System architects establish the limitations or possibilities for changing systems in the future. This thesis is an attempt to formulate guidelines with which architects are able to design more flexible systems.

As a basis for the remainder of this thesis, various terms are defined, especially the term ‘architecture’. In many publications, authors assume that their readers understand what is meant by that term. As a consequence, the term remains undefined, and different readers and authors interpret the term differently. The interpretation of the term ‘architecture’ in various engineering disciplines and in various time periods is studied.

A system architecture is defined as the manner in which the components of a specific system are organised and integrated. Related terms such as ‘reference model’ and ‘reference architecture’ are explained by means of the same notions, i.e. organisation and integration of system components. The main roles of an architecture are to manage system complexity and to provide for future system flexibility. The main role of architecting is to preserve system integrity, i.e. compliance with the architecture, while the system is developed or adapted. Architecting is an ongoing process, since many systems evolve after initial delivery.

As stated above, one of the main roles of an architecture is to manage the complexity of a system. Architecting concepts are defined that enable architects to manage overall system complexity. Three architecting concepts are proposed as part of an open set. Three other concepts might be suitable as well in certain occasions, namely status, versions, and variants. Nevertheless, the three most important concepts are domains, decomposition hierarchy, and views.

Three domains are distinguished; each of them represents a typical design problem: the functional, technology, and physical domains. They must be separated, in order that designers may focus on one domain, without dealing with design problems in other domains. Hierarchical (or nested) systems are formed by means of decomposition. Systems can be decomposed in smaller parts, which are more manageable than the original system. Views emphasise particular aspects of a system and hide the complexity of other aspects.

In addition to the architecting concepts, three architecting principles are formulated. An architecture determines the possibilities to change a system in the future, so architects have to design future-proof architectures. Three architecting principles are given that stimulate the design of such architectures, namely modularity, structural stability, and layers.

Modularity is a characteristic of systems consisting of moderately complex subsystems with maximum cohesion and minimal coupling. The interfaces of modular components restrict the

impact of changes to few modules; changes are propagated to at most a few other components. Structural stability is the characteristic of a system to function stand-alone without collapsing, and with the ability to be part of a larger system or to be extended with new components. Some stability can be assured by building systems out of stable elements. These elements should be both building blocks as complete ‘wholes’. Layers represent allowable interfaces among modules. Modules within a layer can communicate with each other. Modules in different layers can communicate with each other only if their respective layers are adjacent. Layers build upon underlying layers.

The architecting concepts and principles are illustrated by the Gordian project. This project focused on the decoupling of warehousing functionality in the Baan packages. Baan is a supplier of a set of standard software packages for Enterprise Resource Planning. The objective of the Gordian project was to study the possibilities to put the dispersed warehousing functionality into a separate package.

In addition to architecting concepts and principles, this thesis evaluates the suitability of several theories for the design of flexible systems. The evaluation is carried out by means of the discerned architecting concepts and principles. The application domain is the area of shop floor control.

Reference architectures for enterprise integration aim to provide the necessary frameworks with which companies might adapt their operation. CIMOSA (Computer Integrated Manufacturing — Open System Architecture) is such a reference architecture. It strives for the facilitation of continuous enterprise evolution. CIMOSA consists of a modelling framework, an integrating infrastructure, and a system life cycle.

CIMOSA was used during a reorganisation project at Traub AG, a German tool manufacturer. Traub restructured its manufacturing organisation. It used the CIMOSA modelling framework to define a functional control architecture, and it used the CIMOSA integrating infrastructure to support the design of the technology architecture of the control system’s infrastructure.

CIMOSA prescribes how to make a specification of a system, rather than how to design a system. In addition, the translation from models to a real system has to be made by a designer. Therefore, CIMOSA should be merely seen as a framework for the generation of documentation.

On the other hand, reference models for shop floor control contain application domain specific knowledge, and prescribe how to make certain architectural choices. They aim to support architects in the design of effective shop floor control systems. The reference models for shop floor control can be categorised into a number of basic forms. During the 1970’s and the two decades afterwards, the focus in (research on) control architectures was gradually shifting from centralised to more distributed forms.

Enabling technology has evolved as well. In the past, it used to be a limiting factor in the design of functional control architectures. Nowadays, it gives designers more freedom in the definition of technology and functional architectures.

An agent based control system for a model factory was designed by means of the architecting concepts and principles. Its control architecture belongs to the heterarchical form. Global information was made explicit in this architecture; the various agents had no knowledge of other agents and had to ask other agents about their capabilities. Negotiation among agents was used to determine the routings of parts. The extensibility of the agent based control system was improved compared to a previously implemented heterarchical control system without negotiation capabilities. The agent based control system scored better regarding future flexibility. However, a lot of message traffic was needed to compensate for the absence of global information.

Samenvatting (Summary in Dutch)

Informatiesystemen voor shop floor control worden steeds complexer en moeilijker te beheersen. Ze kunnen nauwelijks nog onderhouden of gewijzigd worden. Een verandering in een onderdeel kan een kettingreactie van veranderingen in andere delen veroorzaken. Dit proefschrift beoogt een bijdrage te leveren aan het oplossen of vermijden van dergelijke problemen. Systeem-architecten bepalen de beperkingen en mogelijkheden tot het wijzigen van systemen in de toekomst. Deze dissertatie is een poging om richtlijnen te formuleren waarmee architecten in staat zijn om flexibelere systemen te ontwerpen.

Enkele termen zijn gedefinieerd die dienen als basis voor de rest van het proefschrift, met name de term ‘architectuur’. In veel publicaties wordt door de auteurs verondersteld dat hun lezers begrijpen wat met de term bedoeld wordt. Als gevolg daarvan blijft de term ongedefinieerd en leggen verschillende lezers en auteurs de term anders uit. In dit proefschrift wordt de interpretatie van de term ‘architectuur’ in verscheidene ingenieursdisciplines en in verschillende tijdsperiodes onderzocht.

Een systeem-architectuur wordt gedefinieerd als de wijze waarop de componenten van een specifiek systeem worden georganiseerd en geïntegreerd. Verwante termen zoals ‘referentiemodel’ en ‘referentie-architectuur’ worden uitgelegd met behulp van dezelfde begrippen, dat wil zeggen de organisatie en integratie van systeem-componenten. De belangrijkste rollen van een architectuur zijn het beheersen van de complexiteit van het systeem en het zorgen voor toekomst-flexibiliteit voor het systeem. De belangrijkste rol van ‘het architecten’ (Engels: ‘architecting’) is het beschermen van de integriteit van het systeem, dat wil zeggen de overeenstemming met de architectuur, terwijl het systeem wordt ontwikkeld of aangepast. Het architecten is een voortdurend proces, aangezien veel systemen verder ontwikkeld worden na initiële oplevering.

Zoals hierboven reeds genoemd, is het beheersen van de systeem-complexiteit één van de voornaamste rollen van een architectuur. Dit proefschrift definieert architectuur-concepten (Engels: ‘architecting concepts’) waarmee architecten in staat worden gesteld de totale complexiteit van een systeem te beheersen. Drie architectuur-concepten worden voorgesteld die onderdeel vormen van een open verzameling. Drie andere concepten zouden in bepaalde gevallen eveneens geschikt kunnen zijn, namelijk status, versies en varianten. Niettemin zijn de drie voornaamste concepten domeinen, decompositie-hiërarchie en ‘views’.

Drie domeinen worden onderscheiden: het functioneel, technologie en fysiek domein. Elk domein stelt een bepaald ontwerp-probleem voor. Ze moeten worden gescheiden, opdat ontwerpers zich op één domein kunnen richten zonder rekening te houden met ontwerp-problemen in andere domeinen. Hiërarchische (of geneste) systemen worden gevormd door middel van decompositie. Systemen kunnen worden gedecomponeerd in kleinere delen, die beter hanteerbaar zijn dan het oorspronkelijke systeem. Views benadrukken bepaalde aspecten van het systeem en verbergen de complexiteit van andere aspecten.

Naast de architectuur-concepten worden er drie architectuur-principes (Engels: ‘architecting principles’) geformuleerd. Een architectuur bepaalt de mogelijkheden om een systeem in de toekomst te wijzigen. Architecten dienen derhalve ‘toekomst-bestendige’ architecturen te ontwerpen. Er worden drie architectuur-principes gegeven die het ontwerp van dergelijke architecturen stimuleren, namelijk modulariteit, structurele stabiliteit, en lagen.

Modulariteit is een kenmerk van systemen die bestaan uit gematigd complexe subsystemen met een maximale inwendige samenhang en minimale koppeling met andere subsystemen. De interfaces tussen modulaire componenten beperken het effect van veranderingen tot enkele modules; veranderingen planten zich voort in hooguit enkele andere componenten. Structurele stabiliteit is het kenmerk van een systeem om op een correcte manier alleenstaand te functioneren, en met de mogelijkheid een onderdeel van een groter systeem te vormen of te worden uitgebreid met nieuwe componenten. Enige stabiliteit kan worden bereikt door systemen te bouwen die bestaan uit stabiele elementen. Deze elementen zouden zowel bouwblokken als complete ‘gehelen’ moeten zijn. Lagen stellen toegestane interfaces tussen modules voor. Modules binnen een laag kunnen met elkaar communiceren. Modules in verschillende lagen kunnen alleen met elkaar communiceren als hun respectievelijke lagen aan elkaar grenzen.

De architectuur-concepten en -principes worden geïllustreerd aan de hand van het Gordiaan project. Dit project richtte zich op het ontkoppelen van functionaliteit voor magazijnbeheer in de Baan pakketten. Baan is een leverancier van een verzameling standaard Enterprise Resource Planning pakketten. Het doel van het Gordiaan project was de mogelijkheden te onderzoeken om de verspreide functionaliteit voor magazijnbeheer in een apart pakket onder te brengen.

Naast de architectuur-concepten en -principes worden in dit proefschrift de geschiktheid van verscheiden theorieën voor het ontwerpen van flexibele systemen geëvalueerd. De evaluatie wordt uitgevoerd aan de hand van bovenstaande architectuur-concepten en -principes. Het toepassingsgebied is shop floor control.

Referentie-architecturen voor bedrijfsintegratie (Engels: ‘enterprise integration’) beogen de noodzakelijke raamwerken te bieden waarmee ondernemingen hun bedrijfsvoering zouden kunnen aanpassen. Een dergelijke referentie-architectuur is CIMOSA (Computer Integrated Manufacturing — Open System Architecture). Het streeft ernaar een continue evolutie van een onderneming te vergemakkelijken. CIMOSA bestaat uit een modelleringsraamwerk, een integrerende infrastructuur, en een systeem-levenscyclus.

CIMOSA werd toegepast tijdens een reorganisatie-project bij Traub AG, een Duitse fabrikant van gereedschap. Traub herstructureerde zijn fabricage-afdeling. Het maakte daarbij gebruik van het CIMOSA modelleringsraamwerk om een functionele besturingsarchitectuur te definiëren. Traub gebruikte de CIMOSA integrerende infrastructuur om het ontwerp te ondersteunen van de technologie-architectuur van de bij het besturingssysteem behorende infrastructuur.

CIMOSA schrijft voor hoe een specificatie van een systeem gemaakt moet worden in plaats van hoe een systeem ontworpen zou moeten worden. Bovendien moet de slag van modellen naar een echt systeem door de ontwerper gemaakt worden. Daarom zou CIMOSA vooral moeten worden beschouwd als een raamwerk voor de generatie van documentatie.

Daarentegen bevatten referentiemodellen voor shop floor control wel kennis die specifiek is voor het toepassingsgebied en schrijven ze voor hoe bepaalde architectuur-keuzes gemaakt moeten worden. Ze beogen architecten te ondersteunen bij het ontwerpen van effectieve informatiesystemen voor shop floor control. De referentiemodellen voor shop floor control kunnen worden ingedeeld in een aantal basisvormen. In de periode 1970-1995 verschoof het zwaartepunt van (het onderzoek naar) besturingsarchitecturen van de gecentraliseerde naar meer gedistribueerde vormen.

De technologie die het allemaal mogelijk maakt ontwikkelde zich eveneens. In het verleden was het een beperkende factor bij het ontwerpen van functionele besturingsarchitecturen. Tegenwoordig geeft het ontwerpers meer vrijheid in het definiëren van technologie- en functionele architecturen.

Met behulp van de architectuur-concepten en -principes werd een op ‘agents’ gebaseerd besturingssysteem voor een modelfabriek ontworpen. De besturingsarchitectuur van dit systeem behoort tot de heterarchische vorm. Algemene informatie werd expliciet gemaakt in deze architectuur; de verschillende agents wisten niets over andere agents en moesten andere agents naar hun capaciteiten vragen. Er werden onderhandelingen tussen agents gebruikt om de ‘routings’ van onderdelen te bepalen. De uitbreidbaarheid van het op agents gebaseerde besturingssysteem was verbeterd ten opzichte van een heterarchisch besturingssysteem zonder onderhandelingsmogelijkheden dat reeds eerder geïmplementeerd was. Het op agents gebaseerde besturingssysteem scoorde beter met betrekking tot toekomst-flexibiliteit. Er was echter veel berichtenverkeer nodig om het gemis aan algemene informatie te ondervangen.

Curriculum Vitae

Arian Zwegers was born on July 2, 1969, in Asten, The Netherlands. In 1987, he received his VWO diploma from the 'St. Willibrord Gymnasium' in Deurne, after which he studied industrial engineering at the Eindhoven University of Technology. He received his MSc degree cum laude in 1993 after a research project on software cost estimation. The project was carried out at the European Space Research and Technology Centre (ESTEC) in Noordwijk. After graduating, he started his PhD work at the Eindhoven University of Technology as a member of the Manufacturing Technology group in the faculty of Technology Management. During two years, he was detached at TNO-TPD, where he participated in the ESPRIT project VOICE. Another part of the PhD work was carried out at Baan Company, a Dutch ERP vendor. This thesis concludes the research. In January 1998, he started working for the Philips Centre for Manufacturing Technology (CFT).

STELLINGEN

behorende bij het proefschrift

On Systems Architecting

A study in shop floor control to determine architecting concepts and principles

van

Arian Zwegers

Eindhoven, 30 juni 1998

I

Het ‘architecten’ (Engels: ‘architecting’) van systemen is het proces dat de systeem architectuur definieert en dat de integriteit van het systeem over de tijd waarborgt.

Bron: dit proefschrift, hoofdstuk 2.

II

De systeem architectuur discipline (Engels: ‘systems architecting’) heeft architectuur-concepten en -principes nodig. De concepten stellen architecten in staat de complexiteit van een systeem te beheersen. De architectuur-principes stimuleren het ontwerp van systemen die in de toekomst redelijk eenvoudig gewijzigd kunnen worden.

Bron: dit proefschrift, hoofdstuk 3 en 4.

III

CIMOSA moeten worden beschouwd als een raamwerk voor het opstellen van documentatie. Het schrijft niet voor hoe een systeem ontworpen moet worden, maar hoe een specificatie van een systeem gemaakt moet worden.

Bron: dit proefschrift, hoofdstuk 5.

IV

De geschiktheid van een bepaalde besturingsvorm is afhankelijk van de layout van het productiesysteem.

Bron: dit proefschrift, hoofdstuk 7.

V

Gezien de kosten voor het ontwerpen en onderhouden van besturingssoftware dient de architectuur van een productiesysteem in samenhang met de architectuur van het besturingssysteem ontworpen te worden.

VI

Een proefschrift met stellingen is als een loosely-coupled systeem, niet zozeer vanwege het feit dat de stellingen op een los blaadje staan, maar vooral omdat proefschrift en stellingen een losse relatie met elkaar hebben.

VII

Om het gemis aan productiviteit enigszins te verhelpen, is het in het bedrijfsleven gebruikelijk de snelste computers te geven aan die mensen die er het slechtst mee om kunnen gaan.

VIII

Het begrip ‘sport’ is aan een grondige herdefinitie toe. Bezigheden zoals golf, zweefvliegen, stijldansen, paardendressuur, en vele andere moeten niet als ‘sport’ worden beschouwd.

IX

Koestler’s stelling dat ‘the egotism of the social holon feeds on the altruism of its members’ is wellicht het best vertoond bij een groep Hollanders op vakantie.

Bron: Koestler, A. (1967). *The ghost in the machine*. Hutchinson, London, p. 266.

X

De spreekwoordelijke Nederlandse tolerantie is een gevolg van een gebrek aan interesse.

XI

Het is onbegrijpelijk dat er mensen zijn die in ontwikkelingslanden vakantie vieren en die toch menen dat ze niet rijk zijn.

XII

Cultuur is een kwaliteit waar men niet bewust naar kan streven.
Bron: T.S. Eliot geciteerd in Bax, M.F.Th. (1996). *De rijkdom van architectuur : Terug naar de Bouwkunde*. Afscheidsrede Technische Universiteit Eindhoven, p. 3.