

# Modular design of information systems for shop floor control

**Citation for published version (APA):**

Timmermans, P. J. M. (1993). *Modular design of information systems for shop floor control*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Industrial Engineering and Innovation Sciences]. Technische Universiteit Eindhoven. <https://doi.org/10.6100/IR398985>

**DOI:**

[10.6100/IR398985](https://doi.org/10.6100/IR398985)

**Document status and date:**

Published: 01/01/1993

**Document Version:**

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

**Please check the document version of this publication:**

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

**General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

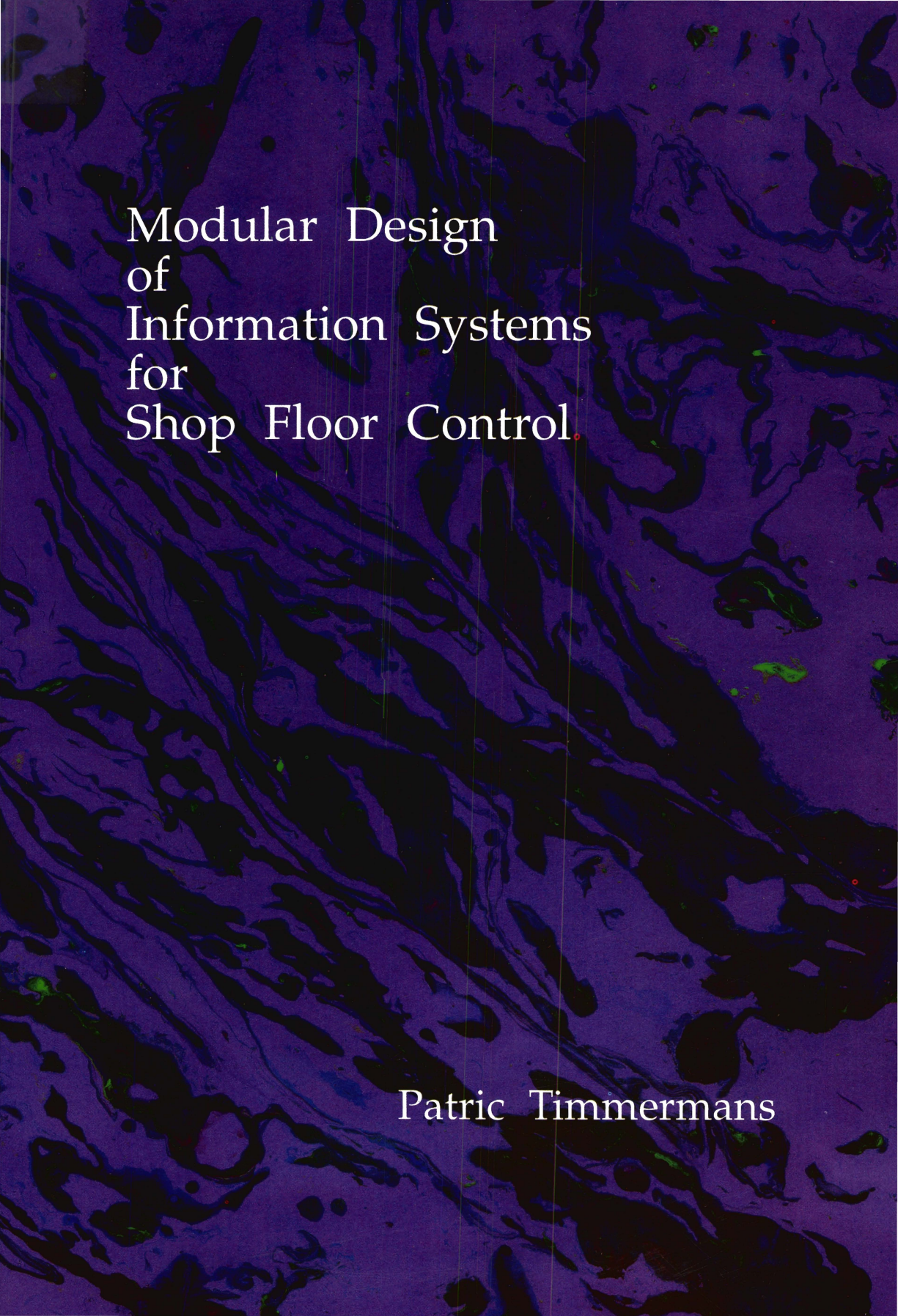
[www.tue.nl/taverne](http://www.tue.nl/taverne)

**Take down policy**

If you believe that this document breaches copyright please contact us at:

[openaccess@tue.nl](mailto:openaccess@tue.nl)

providing details and we will investigate your claim.



Modular Design  
of  
Information Systems  
for  
Shop Floor Control.

Patric Timmermans

# **Modular Design of Information Systems for Shop Floor Control**

## **PROEFSCHRIFT**

ter verkrijging van de graad van doctor aan de Technische  
Universiteit Eindhoven, op gezag van de Rector Magnificus,  
prof. dr. J.H. van Lint, voor een commissie aangewezen door  
het College van Dekanen in het openbaar te verdedigen op  
vrijdag 9 juli 1993 om 16.00 uur

door

**PETRUS JACOBUS MARIA TIMMERMANS**

geboren te Gilze

Dit proefschrift is goedgekeurd door

de promotoren:        prof. dr. ir J.C. Wortmann  
                              prof. dr. ir. E.J. Sol

de co-promotor:        dr. ir. H.J. Pels

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Timmermans, Patric

Modular design of information systems for shop floor control/

Patric Timmermans.

- Eindhoven: Eindhoven University of Technology. -III.

Thesis Eindhoven. - With ref. - With summary in Dutch.

ISBN 90-386-0301-0

NUGI 689

Subject headings: modular design / distributed systems / data modelling

Omslagontwerp: Selma Boumans

Druk: Febo Enschede

© 1993, P.J.M. Timmermans, Eindhoven

Alle rechten voorbehouden. Uit deze uitgave mag niet worden gereproduceerd door middel van boekdruk, fotokopie, microfilm of welk ander medium dan ook, zonder schriftelijke toestemming van de auteur.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form by any means, mechanical, photocopying, recording or otherwise, without the written permission of the author.

---

# Contents

## Chapter 1

|  |   |
|--|---|
| Introduction and problem statement . . . . . | 1 |
| 1.1 Introduction . . . . .                   | 1 |
| 1.2 Modular design . . . . .                 | 1 |
| 1.3 Problem statement . . . . .              | 3 |
| 1.4 History of the research . . . . .        | 7 |
| 1.5 Structure of the thesis . . . . .        | 8 |

## Chapter 2

|  |    |
|--|----|
| Modular design of the conceptual schema . . . . .                | 9  |
| 2.1 Introduction . . . . .                                       | 9  |
| 2.2 Criteria for modularity . . . . .                            | 9  |
| 2.3 Concepts and terminology of the conceptual schema . . . . .  | 10 |
| 2.4 The Modelling Language . . . . .                             | 11 |
| 2.5 Conceptual modelling of information systems . . . . .        | 15 |
| 2.6 Semantic data models versus object-oriented models . . . . . | 16 |
| 2.7 Modular design of the information system . . . . .           | 17 |
| 2.8 Concluding remarks . . . . .                                 | 25 |

## Chapter 3

|   |    |
|---|----|
| The model factory . . . . .                                   | 27 |
| 3.1 Aim and content . . . . .                                 | 27 |
| 3.2 The primary process . . . . .                             | 28 |
| 3.3 The control system . . . . .                              | 30 |
| 3.4 The information system . . . . .                          | 32 |
| 3.5 Complexity and limitations of the model factory . . . . . | 36 |

**Chapter 4**

|   |    |
|---|----|
| Information systems in manufacturing .....                                | 39 |
| 4.1 Introduction .....  | 39 |
| 4.2 Levels in manufacturing .....   | 42 |
| 4.3 Control models .....  | 43 |
| 4.4 Implementation architectures .....                                    | 45 |
| 4.5 Hierarchical control versus layered implementation architecture ..... | 46 |
| 4.6 Redesigning the process .....   | 48 |
| 4.7 Software packages for shop floor control .....                        | 49 |
| 4.8 Concluding remarks .....  | 51 |

**Chapter 5**

|   |    |
|---|----|
| Modular design in manufacturing .....                   | 53 |
| 5.1 Introduction .....                                  | 53 |
| 5.2 Methodology of information system development ..... | 56 |
| 5.3 Determining a module .....                          | 61 |
| 5.4 Implementation of a module .....                    | 63 |
| 5.5 Development of generic modules .....                | 65 |
| 5.6 Concluding remarks .....                            | 66 |

**Chapter 6**

|  |    |
|--|----|
| Architectures for distributed systems .....    | 67 |
| 6.1 Aim and content .....                      | 67 |
| 6.2 Database architectures .....               | 68 |
| 6.3 Distributed system architectures .....     | 74 |
| 6.4 Organisational control architectures ..... | 80 |
| 6.5 Concluding remarks .....                   | 83 |

**Chapter 7**

|                                   |    |
|-----------------------------------|----|
| Generic modules .....             | 85 |
| 7.1 Introduction .....            | 85 |
| 7.2 Reusability of software ..... | 85 |
| 7.3 Reference models .....        | 87 |
| 7.4 Generic modelling .....       | 88 |
| 7.5 Generic modules .....         | 90 |
| 7.6 Generic software .....        | 92 |
| 7.7 Concluding remarks .....      | 94 |

---

|  |     |
|--|-----|
| Chapter 8  |     |
| Discussion and conclusions . . . . .               | 95  |
| 8.1 Summary and conclusions . . . . .              | 95  |
| 8.2 Recommendations for further research . . . . . | 97  |
| References . . . . .                               | 99  |
| Appendix A1: Notational conventions . . . . .      | 107 |
| Appendix A2: Modelling language . . . . .          | 109 |
| Appendix B: Example: the model factory . . . . .   | 111 |
| B.1 Introduction . . . . .                         | 111 |
| B.2 Second-side controller . . . . .               | 111 |
| B.3 Screenprinter controller . . . . .             | 114 |
| B.4 Component placement #1 controller . . . . .    | 117 |
| B.5 Component placement #2 controller . . . . .    | 119 |
| B.6 Reflow & cleaning controller . . . . .         | 120 |
| B.7 In-process store controller . . . . .          | 122 |
| B.8 Test & repair controller . . . . .             | 125 |
| B.9 Final product store controller . . . . .       | 128 |
| B.10 Material handler controller . . . . .         | 131 |
| Appendix C: Glossary . . . . .                     | 133 |
| Summary . . . . .                                  | 137 |
| Samenvatting (summary in Dutch) . . . . .          | 139 |
| Curriculum Vitae . . . . .                         | 141 |

## Chapter 1

# Introduction and problem statement

### 1.1 Introduction

Manufacturing systems have become increasingly dependent on the adequate supply of information that is related to their control. Information technology is employed to provide this information. Initially, information technology was introduced on an ad-hoc basis. Individual problems were solved by individual solutions based on information technology. Rapid changes in both technology and requirements for manufacturing systems necessitated a structured approach to the use of information technology, that addresses issues such as modularity and flexibility of information systems.

More systematic solutions have been proposed in order to avoid the ad-hoc approach that resulted in 'islands of automation'. These solutions included life-cycle models for information system development, starting with information strategy planning. Information strategy planning prescribes the formulation of goals, starting-points and conditions for information services [Greveling 90]. The realisation of one single 'integrated' information system for an organisation is however not feasible [Melkanoff 84] [Scheer 92]. A single integrated information system will exhibit the characteristics of a monolith, i.e., the system will be increasingly difficult to enhance or modify, and the systems learning curve will approach its asymptote. A more promising method seems to be the development of information system in an evolutionary way as federations of autonomous modules. According to this method, an information system is extended by adding modules, and modified through step by step changes in the individual modules.

### 1.2 Modular design

An ideal information system for manufacturing systems would be based on a single global database, which is the heart of the information system. This global database system would allow the definition, updating, retrieval, consistency and communication of data among all functions in the manufacturing system. A complete and detailed conceptual model of the manufacturing system is necessary to be able to specify the complete and precise knowledge of all the necessary information.

Numerous problems stand in the way for the design and implementation of this ideal system.



In [Melkanoff 84] the following non-exhaustive list of problems is given:

- heterogeneous hardware
- heterogeneous software
- heterogeneous models
- complexity of data conversion
- size of the database
- heterogeneity of data
- heterogeneity of users
- update difficulties:
  - integrity constraints
  - temporary violations of constraints e.g. by CAD applications
  - update propagation because of engineering changes
- performance requirements
- graphic I/O requirements
- requirements for heavy numerical computation
- ease and rapidity of launching the database
- control of security, integrity and privacy
- distributed data
- integration of text and graphic information.

The aim of this thesis is to provide a contribution to solving or avoiding these problems by describing a method for the modular design of information systems. The method to be formulated will use conceptual schemas of the module information base as the basis for modular design. The method to be formulated in this thesis originates from [Pels 88]. Pels recognised the problem of an all-embracing conceptual schema, of which every user has to know only a small part. Although it would be possible to define subschemas that present only the relevant information to the user, this would still require an overall conceptual schema, since the subschemas would be derived from it. Moreover, the overall conceptual schema has to change in case a user wants to change its subschema. The method described in [Pels 88] provides guidelines for the decomposition of a conceptual schema into modules that can be changed without being confronted with the complexity of the overall conceptual schema.

## **1.3 Problem statement**

The research question of this thesis is to describe a method for the modular design of information systems for Computer Integrated Manufacturing (CIM). The characteristics of this method should differ in a number of ways from traditional methods for information systems design for CIM. The global scenarios of both methods are now portrayed to characterise these differences.

### **1.3.1 Traditional scenario**

Traditional methods of information system design are often based on the waterfall model. The first phase concerns the definition of requirements of the system to be built. Once these requirements are known, they are considered to be fixed for the following phases in the design process. Changes in the requirements in later phases can be made according to certain procedures, but should be avoided since it is well-known in software engineering that these requirement changes increase the costs of system development drastically. The second phase concerns the global conceptual design. A global conceptual schema is developed in this phase which is successively decomposed in the third phase (detailed design) into modules that can be used and modified autonomously. The decomposition of the global conceptual schema will take place according to the organisational structure of the manufacturing system, probably based on a CIM reference model as described in for example [Biemans 90] [CFT 87] or [Jones et al. 86]. Detailed functional specifications are made in this phase for the applications that need to be developed for each module. It is usually assumed that in this conceptual design phase it is sufficient to consider only the design of the conceptual schema to obtain independent modules. According this assumption it should be possible in this phase to disregard the implementation issues of user interface, applications processing, data management and communication. The fourth phase then concerns the implementation of the detailed conceptual design. The following phases, which will not be discussed here, include the introduction of the information system and the usage and management of the information system.

### **1.3.2 Proposed scenario**

The scenario of the method proposed in this thesis will be based on an evolutionary development process. To begin with, it is assumed that there is an existing information system that needs to be changed, i.e., updated or extended. These changes occur relatively frequently, and sometimes more than one development team is making changes to different modules of the same CIM system. The boundaries and interfaces of these modules are known, and are based on the particularities of the specific primary process and control system. An update of the information system in this scenario

usually involves only one module. The extent to which the module can be changed without affecting other modules is defined by rules. Should other modules be affected, it would only concern the neighbouring modules. These neighbouring modules are identified through the interface definitions. Extending the system concerns the design and implementation of a new module. Emphasis will then lie on the specification of the interfaces to other modules, the architecture of modules at the conceptual level and the location of the module in the implementation architectures. Both updating and extending the system usually concern software packages that are bought from software suppliers. Only very occasionally new software will be developed in this scenario. If that is the case, it will usually concern a modification or enhancement of these software packages.

### **1.3.3 Design from scratch scenario**

Occasionally, it will occur that there is no system yet available, and the complete manufacturing system has to be designed from scratch, which was the situation in the model factory example that will be described in this thesis. The first step in the design of the information system will then be the definition of the boundaries of the modules. This definition will be based on the characteristics of the primary process and the control system, which have to be designed before the information system can be designed. The result of this definition will be a federation of autonomous modules without a global conceptual schema. Then, each module in isolation will be designed in detail and implemented. However, there will be a joint initiative to implement an infrastructure of hardware and software that is shared between modules. Thus, the proposed method abandons the necessity of having a global conceptual schema, and emphasizes the design, re-design and implementation of information systems that consist of federations of autonomous modules without a global conceptual schema.

### **1.3.4 Layered models scenario**

If levels have been defined in the control system, which will often be the case, then the architecture of modules at the conceptual level will be based on it. The interfaces between modules will reflect the different levels, and aggregation of objects can be used to specify these interfaces. Consider for example the situation where there is a distinction between an Aggregate Production Planning level (APP), Goods Flow Control level (GFC), and a Production Unit Control level (PUC) (cf. [Bertrand et al. 90]). At each level one or more modules will be defined. Consider for example the role of the customer order in customer-order driven manufacturing. When a customer order is defined as an object class in an APP module, then this class could be decomposed in the GFC module into customer-order specific workorders, and again in a PUC module into actual operations. However,

each of the three modules can be designed and implemented independently once these interfaces are defined. This is not constrained by the fact that the PUC module uses data from the GFC module (viz. workorder data), and that the GFC module uses data from the APP module (viz. customer order data). Because these interfaces are of crucial importance for the structure of the control system itself, they are not likely to change frequently. The interfaces between the information system modules will therefore not change frequently either. However, if the company intends to implement a new production planning system for Goods Flow Control, it is possible to do so without changing the APP and PUC modules.

### **1.3.5 Further research goals**

Furthermore, the method should be validated, and extended where appropriate. Two areas of extension are the design of CIM architectures and the reuse of software. The research question concerning the extension towards CIM architectures is: what is the relation between modular design of information systems at the conceptual level and the modular implementation of information systems in different CIM architectures. The research question concerning the extension towards the reuse of software is: how can the method of modular design contribute to the reuse of software.

### **1.3.6 Research area**

Industrial companies have to provide greater flexibility and responsiveness, better use of resources, a reduction in inventory levels and faster delivery of customer orders in order to be competitive. Adequate shop floor control systems contribute considerably to these goals. In the research area of manufacturing however, relatively little attention is paid to shop floor control systems. Bauer *et al.* mention that conventional commercially available computer based systems are very weak on shop floor control [Bauer et al. 91]. This thesis is dedicated to the design of shop floor control systems to decrease this deficit. The examples in this thesis are taken from a shop floor control system as well. The method described in this thesis has however a wider application area, as is indicated by a project for the re-design of a production planning system that is not reported in this thesis [Timmermans 92].

### **1.3.7 Starting-point: data modelling**

The main assumption of the method presented in this thesis is that data modelling is a valid starting-point for the modelling of information systems for shop floor control. This assumption is based on

two arguments. First, it is argued by various authors that conceptual data models constitute the skeleton of the information system [Bertrand et al. 90] [Melkanoff 84] [Scheer 92]. Any organisation is subject to continuous change. These changes hold in particular for the operating procedures of the organisation, which are described as the functionality of the organisation. The conceptual data model describes the objects that are recognised to exist in the manufacturing system. Although these are also subject to changes, they are generally more stable than the operating procedures. Moreover, most manufacturing systems are based on a manufacturing database, which is the implementation of the conceptual data model, and many different applications use this database.

The second argument is put forward in [Pels 88]. He makes a distinction between the technological, syntactical, semantical and pragmatical aspects of an information system. Integration of information systems can be considered from each of these aspects. However, the integration of information systems on technological and syntactical level requires a consideration of the semantics of both systems. These semantics are described in the conceptual data model. Pels argues that conceptual data models are appropriate instruments for the analysis of composition and decomposition of information systems.

It is however recognised that conceptual data models are not universally applicable. The possibility of modelling dynamics in data models is limited. It would therefore be more appropriate to use process oriented modelling languages to model highly reactive real-time systems such as programmable logic controllers. However, it is argued here that data modelling is a valid starting-point for the modelling of information systems in database oriented systems such as in shop floor control systems and even more in information systems for production planning and control. The application domain of data models in manufacturing is illustrated by figure 1.1.

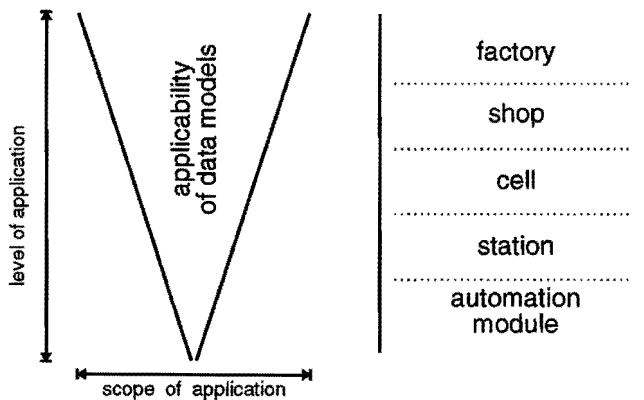


figure 1.1 application domain of data models

## 1.4 History of the research

The research project was started in 1989 in the section Information & Technology of the school of Industrial Engineering & Management Science at Eindhoven University of Technology (TUE). After an initial literature survey, a modular CIM system was designed in the CIM laboratory of the school to get acquainted with the method of modular design.

In 1990, a second experiment was carried out. A design of a shop floor management system for components design and manufacturing was made for a consortium consisting of 7 system integrators and suppliers of CIM components. The goal of this project was to make a design of a manufacturing cell that consisted of existing CIM components. It would not be possible to implement components from scratch because of limited time and resources. The method for modular design was used to specify and integrate existing components, such as CAD/CAM systems, a production planning system, a shop floor management system and a Flexible Manufacturing System.

Almost simultaneously, the CIM laboratory of the Digital Cooperative Engineering Centre (CEC) was set up. The laboratory involves *'the research and development of new tools and techniques for shop floor management and device connection, and the integration of these tools and techniques into one shop floor management system'* [Kearns 90]. The results are applied to a scale model factory of a Printed Circuit Board (PCB) assembly and test plant. From early 1990, research was carried out in co-operation between TUE and the CEC. During the period from 1990 until early 1993, one TUE researcher, successively two Digital program managers, and in total 6 TUE students and one other student have participated in the research in the CIM laboratory that is directly related to this thesis.

During 1991, an additional project was carried out for the redesign and implementation of a production planning system.

Finally, the second half of 1992 and the beginning of 1993 were spent in the preparation of this thesis.

## 1.5 Structure of the thesis

The main structure of the thesis is depicted in figure 1.2.

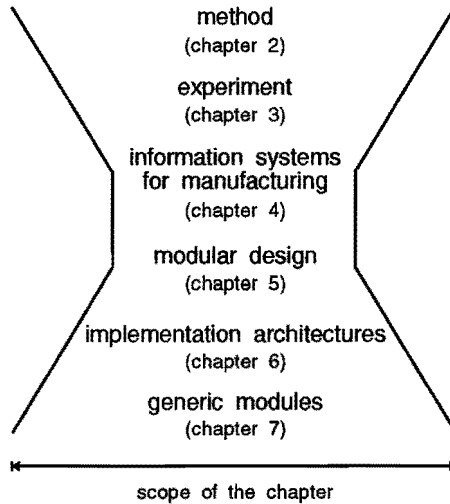


figure 1.2 structure of the thesis

Chapters 2, 3 and 4 present the backgrounds of the research and the research area. The method for modular design of information systems, including concepts and terminology, is introduced in chapter 2. The experiment carried out in the CIM laboratory of the CEC is then presented as an example in chapter 3. Chapter 4 presents an overview of current methods for the design of information systems for manufacturing.

Chapter 5 is the centre of the thesis. This chapter provides principles for the design, redesign and implementation of a modular information system for manufacturing, based on the findings of previous chapters. Emphasis is placed on the design, redesign and implementation of application software for shop floor control.

Chapters 6 and 7 elaborate on two main subjects of chapter 5. In chapter 6 this is the implementation of modular information systems. Different architectures for the implementation of information systems are discussed. Here, an important conclusion will be that distributed implementation architectures provide more adequate conditions for the implementation of modular information systems than for example hierarchical architectures. Chapter 7 concentrates on the reuse of software based on module specifications.

Finally, in chapter 8 the knowledge gained so far in the application of modular design is discussed, which leads to a few suggestions for further research.

## Chapter 2

# Modular design of the conceptual schema

### 2.1 Introduction

This chapter introduces a method for the modular design of information systems. This method plays a central role in this thesis, since one of the research goals has been to validate and extend this method for the application area of manufacturing. The essence of the method is described in [Pels 88] as the method of modular decomposition of the conceptual schema. The main difference between [Pels 88] and this thesis is that the emphasis in the latter is put on design and re-design while the emphasis in [Pels 88] is put on the analysis of a design. Furthermore, the method is described in terms of object modelling instead of the relational model. It is emphasized that one of the strong points of the method for modular design is its independence of a schema specification language.

The basic concepts of the method of modular design are described in this chapter. First, criteria for modularity are discussed in section 2.2. The concepts and terminology of the conceptual schema are described in section 2.3. An appropriate language for the conceptual schema is introduced in section 2.4. Section 2.5 discusses conceptual modelling of information systems. Section 2.6 discusses the difference between a semantic data model and an object-oriented data model. The definition of a module is then presented in section 2.7.

### 2.2 Criteria for modularity

Modularity of an information system has, like quality of an information system, more than one perspective. [Meyer 88] specifies five independent criteria for design methods with respect to modularity. These criteria will be applied throughout the thesis. They are designated as:

- modular decomposability
- modular composability
- modular understandability
- modular continuity
- modular protection



According to Meyer, the modular decomposability criterium is met by a design method if the method helps in the decomposition of a new problem into several sub-problems, whose solution may be pursued separately. In addition to Meyer, the method should also help the re-design process by the decomposition of an existing system into separate components.

A method satisfies the criterium of modular composability if it favours the production of software elements that may freely be combined with each other to produce new systems, possibly in an environment different from the one in which they were initially developed.

A method favours modular understandability if it helps to produce modules that can be separately understood by a human reader. At worst, the reader will have to look at a few neighbouring modules.

A design method satisfies modular continuity if a small change in a problem specification results in a change of just one module. Such changes should not affect the architecture of the system. An architecture is defined as a description of components and their interfaces.

A method satisfies the modular protection criterium if it yields architectures in which the effect of an abnormal condition occurring at run-time in a module will remain confined to this module, or at least will propagate to a few neighbouring modules only.

### 2.3 Concepts and terminology of the conceptual schema

Concepts and terminology of the conceptual schema can be found in [Griethuysen 82]. Some concepts that are of interest for further discussion in this thesis will be introduced in this section.

Specifying an information system involves the modelling of a part of the real world or postulated world, called the universe of discourse. The concrete physical representation of the information will be called a database. The term database system refers to a data processing system dealing with a database.

When designing a database system the primary interests lies in the meaning of the information. This meaning is specified in the conceptual schema. *A conceptual schema comprises a unique central description of the various information contents that may be in a database.* This includes classifications, rules, laws, etcetera, of the universe of discourse. The database itself may be implemented in any of a number of possible ways. For this purpose, the ANSI/SPARC three-schema architecture has been defined (figure 2.1) [Tsichritzis et al. 77]. This architecture provides data independence. A distinction is made between physical and logical data independence. Physical data independence means that users (application programs or end-users) do not need to have knowledge of changes in storage structure and access strategy of the data. Logical data independence means that it should be possible to introduce changes on a logical level without having repercussions on the usage of the data.

While the meaning of the information is specified in the conceptual schema, the physical

storage structure is described by an internal schema. External schemas describe the way users and application programs may view the data in the database system. Every external schema is therefore derived from the common conceptual schema.

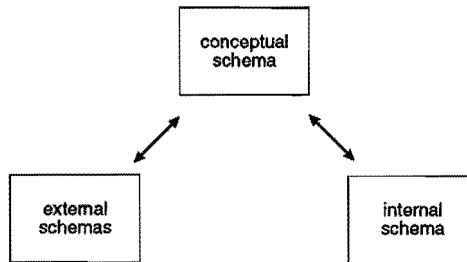


figure 2.1 ANSI/SPARC three-schema architecture

The actual objects that are perceived to exist in the universe of discourse in a specific instant or period of time and their relevant actual states of affairs are described in the information base. Both the conceptual schema and the information base are considered to be at the conceptual level.

## 2.4 The Modelling Language

### 2.4.1 Introduction

The modelling language in this thesis is based on semantic data models. Semantic models are oriented towards the representation of the meaning of data and attempt to provide a structural abstraction. An example of one of the earliest semantic models is the Entity-Relationship model of Chen [Chen 76]. The choice of semantic data models is based on the assertion that semantic data models are more powerful in representing integrity constraints and various relationships than other current data models [Bouzeghoub et al. 91]. Behavioural aspects of the conceptual design are introduced by dynamic integrity constraints.

Another class of data models are object-oriented models. The power of object-oriented data models is highlighted by their ability to describe the dynamic behaviour of the objects by means of methods. A discussion on semantic models and object-oriented models can be found in section 2.6.

The following sections will provide a linguistic description of the language for the conceptual schema.

## 2.4.2 Basic concepts

### Basic building blocks

The basic building blocks of the modelling language are SCHEMA and CLASS. A schema is the means of specifying objects, integrity constraints and domain rules. The schema is used as the syntactic unit to specify a module, which is addressed in section 2.7.2.

A CLASS is a group of objects with similar properties, common behaviour, common relationships and common semantics. Integrity constraints in a schema specify the static and dynamic constraints on or between object classes. Domain rules in a schema specify the interfaces of a module. Other concepts that will be used and explained in this section are generalisation and specialisation.

### Classes

A class describes the structure of a set of objects in terms of their attributes. All objects will be called instances of the class. An object is described by its attributes. There are no other objects than class instances; any object is an instance of at least one class. This class is said to be the type of the object. It is important to notice that each object has its own identity. Thus, two objects are distinct, even if all their attribute values are identical. The following notational convention is used:

- if *i* is an information base state and *b* is a class then *i.b* refers to the set of objects of *i* belonging to class *b*.
- if *o* is an object then *o.a* refers to attribute *a* of that object. Attributes of an object are other objects or sets of objects. Objects can be simple or complex. Simple objects are, for example, integers and strings. They are not specified explicitly in a schema.

An example of a class definition could be:

```
class batch
  attributes
    batch_id : integer;
    creator : station;
    item_type : item_type;
    size : {1..3};
end; -- class batch
```

### Subclasses

A class can have subclasses. Instances of a subclass inherit all attributes of their superclasses in addition to the attributes of their own class. A subclass is a specialisation of its superclass.

The design process also can go the other way. A superclass is created from some existing classes. The superclass contains (a part of) the common attributes of the subclasses. Thus, the generalisation of a number of subclasses into a superclass can be obtained. Both specialisation and generalisation can be specified, if this is required. A simple specialisation definition could be:

```

class consumer
  attributes
    station_name : string;
    produced_requests : set of request;
end; -- class consumer

class station
  subclass of consumer
  attributes
    received_requests : set of request;
    batch_in_process : batch;
    batch_available : {available, non-available};
end; -- class station

```

### 2.4.3 Integrity constraints

#### Static constraints

A conceptual schema should describe all relevant static and dynamic aspects of the universe of discourse [Griethuysen 82]. Static and dynamic constraints describe the permissible information base states and sequences of information base states.

Static constraints are concerned with the consistency and permissibility of a single state of the information base. A static constraint is specified as a first order formula on the set of all information base states. A constraint can refer to attribute domains, the relation between attribute values, the relation between objects, or the relation between classes. The requirement that the batch size of a batch is minimally 1 and maximally 3 is an example of a static constraint. The notation will then be:

$$C(i) = (\forall b: b \in i.\text{batch} : 1 \leq b.\text{size} \leq 3),$$

where  $i \in S$  (the set of all information base states).

The constraint  $C$  is a boolean function on the set of all information base states.  $i.batch$  refers to the set of objects of the type  $batch$  in information base state  $i$ , and  $b.size$  refers to the value of the attribute  $size$  of  $batch$   $b$ .

### Dynamic constraints

Dynamic constraints are concerned with the allowed transitions from one information base state to the next. They specify the possible sequences of information base states. Therefore, these constraints are also called transition constraints. A dynamic constraint is essentially a boolean function on a pair  $\langle old, new \rangle$  of information base states. A change from a state  $old$  to a state  $new$  is permissible if and only if the function, when applied to pair  $\langle old, new \rangle$ , is true. Dynamic constraints are specified in this thesis as first order formulas on the Cartesian product  $S \times S$  of the set  $S$  of all information base states. Examples of dynamic constraints can be found in appendix B.

A distinction can be made between permissive rules and prescriptive rules [Griethuysen 82]. Permissive rules describe the possibilities of an action:  $p$  can occur only if  $g$  is true. Prescriptive rules describe the necessity of an action: if  $p$  is true then  $g$  must occur. Both propositions correspond to the logical implication [Bracchi et al. 79]:

$$p \rightarrow g$$

The way to distinguish between them is to add time  $i$  and  $j$  ( $i$  precedes  $j$ ). This results in:

- a.  $j.p \rightarrow i.g$
- b.  $i.p \rightarrow j.g$

which mean:

- a.  $p$  is allowed if  $g$  was true before (permissive)
- b.  $g$  must occur if  $p$  is true (prescriptive)

It is noted that all static constraints are permissive rules since they are not time dependent. Other examples of static and dynamic constraints can be found in appendix B.

### Temporal Modelling

Sofar a distinction has been made between static and dynamic constraints. Static constraints apply to each of the individual states of the information base. Dynamic constraints apply to a pair  $\langle old, new \rangle$  of consecutive states, thus describing the allowed transitions. When describing a dynamic constraint by a pair of consecutive states one assumes that the history of states can affect permissibility only in as much as the history is reflected in the most recent state. It is therefore

sufficient to consider the most recent state only. [Put 88] however argues that a distinction has to be made between dynamic constraints and temporal constraints. Temporal constraints refer to information base states other than the current or new state. Checking a temporal constraint requires information about one or more states in the past or future.

For reasons of simplicity it will be assumed in this thesis that the history of states is reflected in the most recent state. There will be no further distinction between dynamic and temporal constraints.

### 2.4.4 Diagrammatic Notation

The object model defined in the previous section can be represented by a data structure diagram. In this thesis, the diagrammatic technique described in [Martin et al. 92] is adopted. A class is represented by a rectangle. An attribute type is an association between one class and another class, indicated by a line. The cardinalities of the association are indicated by cardinality symbols. An example of a data structure diagram is given in figure 2.2. Further explanation of the diagrammatic technique is given in Appendix A1.

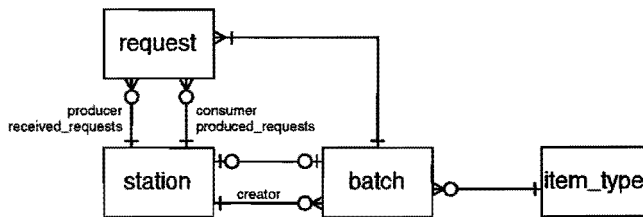


figure 2.2 example of a data structure diagram

## 2.5 Conceptual modelling of information systems

### 2.5.1 Information systems

An information system can be considered as an information base plus a number of information base applications. Information base applications, shorthanded as applications, are formalised procedures for manipulating the information base. This may either be a computer program or a manual operation.

Modular design aims at the elimination of errors that can occur in applications when changing the conceptual schema. Two types of errors can occur: errors in update operations and errors in

retrieval operations. Errors in update operations occur when an intended transition is not allowed in combination with the current information base state. Errors in retrieval operations can occur when the occurrence of an information base state is not foreseen.

An application on a conceptual schema is defined as a combination of retrieval and update operations which are defined for that conceptual schema. The execution of an operation should never violate the integrity constraints that are defined on the information base, or cause errors elsewhere.

### **2.5.2 Transferability of Applications**

Applications can be transferred, in case of re-design, from an old module to a modified module, or, for example in the case of re-use, from one module to another. An application is transferable to a modified module if it satisfies all constraints of the modified module after the module has been changed. An application is transferable to another module if it satisfies all constraints of that module. Transferability of an application is therefore an important property concerning modular continuity.

## **2.6 Semantic data models versus object-oriented models**

A short characterisation of semantic models and object-oriented models is given in [Hull 87]: semantic models encapsulate structural aspects of objects, whereas object-oriented languages encapsulate behavioural aspects of objects. Semantic models are oriented towards the representation of data, whereas object-oriented languages are concerned with the manipulation of data [King 89]. Essentially, semantic models provide constructors for creating complex types, while behavioural issues are often left undefined. In contrast, object-oriented models take an abstract data type approach. However, class hierarchies and inheritance are generally defined likewise in semantic models and object-oriented models. Behaviour of an object in semantic data models can be described by dynamic integrity constraints. Some researchers refer to semantic models as being 'object-oriented' in order to stress the fact that they provide mechanisms for structuring complex objects. Thus, the distinction between the two kinds of modelling is not always well defined. In fact, in [Bouzeghoub et al. 91] a procedure is described to generate an object-oriented model from a semantic model.

The most important difference between both models which is of interest for this thesis is the way integrity constraints are specified. In semantic data models, integrity constraints can be specified in first order predicate calculus (see section 2.4.3). Object-oriented data models do not, except through methods, easily permit specification of integrity constraints on the objects

[Bouzeghoub et al. 91]. I.e., most integrity constraints have to be specified in terms of the behaviour of the objects, and are described as dynamic constraints on the operations of an object. Dynamic constraints can be formulated in pre- and postconditions, which are expressed as predicates. However, pre- and postconditions should be considered as specifications of an operation or method, rather than a property of the object structure. Moreover, there are a number of drawbacks to the use of pre- and postconditions [Put 88]. It is for example not possible to define prescriptive rules, which are based on the arrival of a particular state. Also, when reusing dependent actions (methods) in different action calling patterns, one must be very carefully [Put 88].

## 2.7 Modular design of the information system

### 2.7.1 Introduction

With respect to modularity, five evaluation criteria for design methods are given in section 2.2. Certain design principles follow from this set of criteria which must be observed to ensure proper modularity. [Meyer 88] mentions five principles:

1. linguistic modular units
2. few interfaces
3. small interfaces
4. explicit interfaces
5. information hiding

The principle of linguistic modular units need a little explanation. This principle refers to the requirement that the language used to specify the design must support the notion of modularity. I.e., the grammar of the language should support the notion of modularity.

Furthermore, the principle of *independence* of a module is added as the sixth principle. In this section, language features will be introduced for the realisation of the first, the fourth, the fifth and the sixth principle by means of the conceptual schema. In chapters 5 and 6, guidelines will be given for quantifying the second and third principle.

### 2.7.2 Linguistic modular units

A module refers to a part of an information base that can be used separately [Pels 88]. It is therefore a concept at the conceptual level. A module presents itself to its user as an isolated information base. A module is therefore specified by an isolated conceptual schema, and should be considered



as an instance of that schema. In contrast with the distributed database approach, there is no global conceptual schema which is composed of various 'local' module schemas. A more detailed discussion on the relation between modules and distributed and federalised databases can be found in chapter 6.

The basic building block SCHEMA is the syntactic unit to specify a conceptual schema. The classes and constraints related to a module can be specified in a schema. A schema definition could look like<sup>1</sup>:

**SCHEMA** screenprinter

### **CLASSES**

**class** station

#### **attributes**

station\_name : string;

**end;** -- class station

**class** batch

#### **attributes**

batch\_id : integer;

**end;** -- class batch

**class** request

#### **attributes**

consumer : station;

batch : batch;

**end;** -- class request

### **INTEGRITY CONSTRAINTS**

-- for every information base state *i* must hold that the maximum number of requests created by the second-side for one batch is 1.

$C(i) = (\forall b: b \in i.batch:$

$(\# r: r \in i.request: r.consumer.station\_name = 'screenprinter' \wedge r.batch=b) \leq 1)$

**END;** -- schema screenprinter

---

<sup>1</sup> a complete specification of the example is given in appendix B.

### 2.7.3 Explicit interfaces and information hiding

The specification of a module includes also the interfaces of the module. These interfaces concern the update and retrieval authorisation of the module with respect to the information base. The interfaces will be called *domains*, and are defined as functions on the information base state. Domains specify the update and retrieval authorisation of a module in terms of the retrieval operation 'read' and the three primitive update operations 'create', 'delete' and 'modify'. Thus, the concept of domains satisfies the principle of explicit interfaces and information hiding.

It should be noticed that in [Griethuysen 82] 'modify' is not considered as a separate operation. In stead it is considered as a combination of 'delete' and 'create'. However, in terms of objects this would mean that one object is deleted and another is created. Although these two objects may have the same representation, they have different identities. The operation 'modify' has for this reason to be considered as a separate operation.

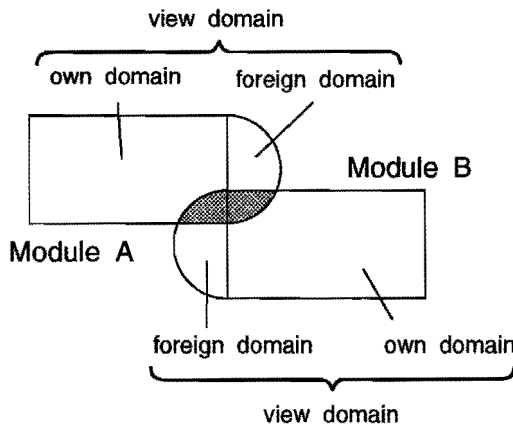


figure 2.3 domains of a module

#### Own domain

The *own domain* of a module contains the objects for which the module has retrieval and update authorization (create, delete and modify). A further refinement can be made by distinguishing a private domain and a public domain, where the private domain contains the objects of the own domain that are not visible to other modules, and where the public domain contains the objects of the own domain that are visible to one or more other modules.

## Foreign domain

The *foreign* domain of a module contains the objects for which the module has retrieval authorization but no update authorization. The foreign domain refers to the objects a module retrieves from other modules. The union of the own and foreign domains is called the *view* domain. The *view domain* of a module contains the objects that are visible for the module. Being visible means that the objects are included in the information base of the module, and that these objects can be retrieved from this information base by the 'read' operation.

Interfaces between modules are explicitly defined by their domains, which is illustrated by figure 2.3. The foreign domain refers to the objects a module retrieves from other modules, whereas the public domain refers to the objects created by the module itself and retrievable by other modules.

The domains are specified by means of domain rules. A specification of domain rules could look like<sup>2</sup>:

## SCHEMA screenprinter

### CLASSES

```

class station
end; -- class station
class batch
end; -- class batch
class request
end; -- class request
class item_type
end; -- class item_type

```

### DOMAIN RULES

-- the own domain of this module consists of the objects of the object types request and station that have 'screenprinter' as the name of the station.

```

own domain (i) = {t ∈ i.request | t.consumer.station_name = 'screenprinter'} ∪
                 {t ∈ i.station | t.station_name='screenprinter'}

```

-- the foreign domain of this module consists of all objects of the object types item\_type and batch, the objects of the object type request with 'screenprinter' as the producer station name, and the objects of the object type station with 'second-side' as the station name.

---

<sup>2</sup> a complete specification of the example is given in appendix B.

$$\text{foreign domain}(i) = \{t \in i.\text{item\_type}\} \cup \{t \in i.\text{batch}\} \cup \\ \{t \in i.\text{request} \mid t.\text{producer.station\_name} = \text{'screenprinter'}\} \cup \\ \{t \in i.\text{station} \mid t.\text{station\_name} = \text{'second-side'}\}$$

END; -- schema screenprinter

### Further refinement of the own domain

The distinction between own and foreign domain as defined above is usually sufficient. Sometimes however, a refinement of the own domain is necessary. As defined earlier, the own domain contains the objects for which the module has update authorization. Update authorization includes the three primitive update operations create, delete and modify. In some cases one module may have the authorization to create objects, while another module has the authorization to delete and modify objects. This leads to a refinement of the own domain in three non-disjunct domains: insert domain, delete domain and modify domain.

The update domains will be specified in the conceptual schema by the terms 'insert domain', 'delete domain' and 'modify domain'. The objects specified in the own domain are included in all three update domains.

### 2.7.4 Horizontal and vertical module fragmentation

When defining the domains of a module, one will often find that not all objects of a particular class can be allotted to a single module. It will often occur that some objects in a class will be allotted to one module, while others will be allotted to another module. Furthermore, it may occur that some attributes of an object are own to a module and other attributes foreign. In analogy with Distributed Database terminology [Elmasri et al. 89], this allotting will be called horizontal module fragmentation respectively vertical module fragmentation respectively. It is remarked here that vertical module fragmentation often indicates a non-optimal definition of classes. Introducing specialisation avoids the necessity of using vertical module fragmentation. A fragmentation example could be:

horizontal module fragmentation

$$\text{own domain } (i) = \{s \in i.\text{station} \mid s.\text{station\_name} = \text{'test\&repair'}\}$$

-- only stations with the name 'test&repair' are in the own domain

vertical module fragmentation

$$\text{foreign domain } (i) = \{\langle s.\text{station\_name}, s.\text{batch\_available} \rangle \mid s \in i.\text{station}\}$$

-- only the attributes 'station\_name' and 'batch\_available' are included in the foreign domain

### 2.7.5 The relation between modules

#### Independence

Modular understandability is determined by the *independence* of a module. A module is independent if the user (application program or end-user) of that module can inspect the validity of an intended transition in every situation. A sufficient condition for independence is: a module is independent if all *applicable constraints* are *visible*<sup>3</sup>. Visibility of an integrity constraint means that the user of a module can determine the logic value of the constraint, which requires that the view domain of the module includes all objects and attributes a constraint refers to. However, for the analysis of independence only those constraints are of interest that might be violated by an update operation in an update domain of the module. These constraints are called *applicable constraints*. For the analysis of the applicable constraints knowledge is needed of:

1. the objects involved in the constraint specification
2. for each involved object, the update operations that might violate the constraint.

A method for performing this analysis is indicated in [Bouzeghoub et al. 91]. The result of the analysis is the applicability of each constraint for each pair ⟨class, update operation⟩. Whether the constraint is applicable to a module can easily be checked by comparing the pairs with the update domains of the module.

#### Applications on schemas

Modular understandability requires also that an application can be developed as an application on a schema without considering other schemas. The discussion on independence shows that if a module is dependent, operations could be initiated that are not allowed, without being able to check the applicable constraints. This resulted in the condition for independence that all applicable constraints have to be visible. This condition holds of course also for the development of an application. Moreover, the following general design rule must be obeyed in order to avoid the situation where it is not possible to check the validity of constraints:

when developing an application for a module, specifications of that module only shall be used.

---

<sup>3</sup> this condition is proven in [Pels 88] where an applicable constraint is defined as a constraint that is involved with either an INSERT, DELETE or MODIFY operation.

**Derived attributes**

Derived attributes are attributes of which the value is calculated from the value of one or more other attributes. Special attention is here required for the definition of constraints. For example, attribute A is defined as the sum of B and C. Changing B or C would change the value of A as well. However, if B and C are owned by two different modules: who owns A? Besides, what happens if the value of A is changed: how does this affect B and C? These situations require a careful definition of constraints and ownership. Prescriptive rules have to be specified to solve this problem.

**Continuity and Composability**

Independence of a module is of high importance for the design criteria of modular decomposability, modular understandability and modular protection. The method proposed here contributes also to modular composability and modular continuity. Modular continuity is satisfied if a small change in the problem specification results in a change of just one module, or a few modules [Meyer 88]. Modular continuity is enhanced by the design principles of small interfaces, few interfaces, explicit interfaces and information hiding. While the former two principles concern a specific design and the latter two concerning the general properties of a module, each principle can be applied through the use of domain specifications. More particularly, any changes in the private domain of a module will affect only that module, and changes in the public domain of a module only affect the modules that interface that module through their foreign domain. Hence, these changes will be limited to a few modules.

The criterium of composability is satisfied by the concept of integration. Integration of modules involves the combination of specifications of two or more modules into one schema. Both object structure and constraints have to be combined. Generalisation of object classes has to be applied in case of differences between classes in different modules. In [Pels 88] is argued that no further conditions have to be met in order to maintain independence when integrating two modules.

The combination of the constraints of two or more schemas will however likely incur changes in what transitions are allowed on the information base. Consequently, the validity of the current transitions may be changed. This may have important consequences in case of update operations, since it will affect the requirement that applications on a conceptual schema should be based on the specifications of that schema. It might be required that applications have to be redesigned as well. Transferability of an application to an integrated schema is however guaranteed if the following conditions are satisfied [Pels 88]:

1. the original module of the application is independent
2. no applicable constraints may be introduced for the integrated module.

### 2.7.6 Other Approaches

The problem of handling large complex schemas is recognized by many researchers and practitioners. A number of proposals have been presented for solving this problem. A few representative solutions will be discussed in this section. Other solutions that are related to distributed databases will be discussed in chapter 6.

A general solution to capture complexity in schemas is to define views on the global schema. In [Rumbaugh et al. 91] for example a 'module' is used for enhancing understandability and capturing a view of a situation. It is defined as a logical construction for grouping classes, associations, and generalisations. The global object model consists of one or more modules, whereas the modules enable the partitioning of an object model into manageable pieces. The same class can be referenced in different modules, which is the mechanism for binding modules. There is however no special notation given for a module, and a module is defined only in the external schema. This approach does not mention a modular conceptual schema, and no guidelines are given for the creation of independent modules in the meaning defined in this chapter. It is therefore concluded that this approach is not sufficient for the development of modular information systems.

Another starting-point for the introduction of modules is to manage schema development. Several approaches from the area of Computer Aided Design address this issue in particular. [Kim et al. 88] for example present a model for version management of schemas for Object-oriented databases. [Andany et al. 91] present a version model that handles database schema changes, and that takes evaluation into account. It allows the development of partial schema versions in the form of external views of a schema. There are rules described for authorised modifications on a schema and for guaranteeing coherence. The main goal of these approaches is to secure modular understandability and continuity. The modularity criteria of composability and decomposability are however largely neglected.

In object-oriented design a module is defined as an abstract data type including the attribute definitions, operations and integrity constraints [Meyer 88] [King 89]. In section 2.6, it is pointed out that a weakness of current object-oriented data models is that, except through the specification of methods, they do not easily permit the specification of integrity constraints. The specification of integrity constraints is however essential for analysing the independence and transferability of applications. Yet another risk of object-orientation is the complexity involved with reusing dependent actions in different calling patterns. For these reasons it is not clear whether current approaches based on object-orientation satisfy the composability and decomposability criteria for programming-in-the-large.

Finally, there are efforts in various standardisation committees to create languages that allow the design of modules. The data modelling language Express for example, proposed by ISO TC184/SC4/WG5 [Spiby et al. 91], introduces the possibility of multiple schemas. The terms 'reference' and 'use' are introduced for interfacing different schemas. These terms are used for

including the specification of one schema into another schema. However, they cannot be used directly for the specification of the domains of a module since the domains are specified in terms of the information base, not in terms of the conceptual schema [Baats 92]. Moreover, Express allows implicit references through chains of relations. This violates the principle of explicit interfaces, which is essential for the definition of module independence.

## **2.8 Concluding remarks**

Five criteria were specified in section 2.2 for the evaluation of a design method with respect to modularity. From these criteria it is possible to derive design principles. Six design principles are mentioned in section 2.7.1: linguistic modular units, few interfaces, small interfaces, explicit interfaces, information hiding, and independence.

An evaluation of the method of modular design according to the five criteria will now complete this chapter. First, the method was initially developed to decompose complex information systems by the definition of modules and assigning domains to these modules [Pels 88]. Thus it satisfies the decomposition criterium. Second, the composition criterium is satisfied since it is possible to combine different modules by combining the domains. Third, the method aims at modules that can be understood separately. This is reflected in the visibility requirement of integrity constraints. Hence, the understandability criterium is also satisfied. Fourth, the continuity criterium is satisfied by the definition of own and foreign domains. These domains specify what part of the module can be changed without implications for other modules. Moreover, these domains provide the means for extending the information system. Fifth, modular protection is guaranteed when applications are specified and implemented as applications on one module. This will prevent unexpected errors to occur in an application due to errors in neighbouring modules.





## Chapter 3

# The model factory

### 3.1 Aim and content

This chapter provides an overview of an experiment that includes the modular design of an information system for shop floor control. This experiment will serve as an example in following chapters, and more details will be revealed. The experiment was carried out in the CIM laboratory of the Cooperative Engineering Centre of Digital Equipment Corporation in Amsterdam. The laboratory consists of a scale model factory of a PCB production line. This scale model factory was designed and implemented as a vehicle for applied research and advanced development in the area of shop floor control. In the laboratory it is possible to use commercially available hardware and software, to test new software, and to design and implement experimental tools and techniques. Research issues of the laboratory are the modular design and implementation of information systems for shop floor control, and the design of generic and reusable modules.

This chapter includes a description of the products of the model factory, the physical hardware, the primary process, the control system, and the information system. The design of the information system will be discussed in more detail in chapter 5. A more complete description of the modular design of the information system of the model factory can be found in appendix B.

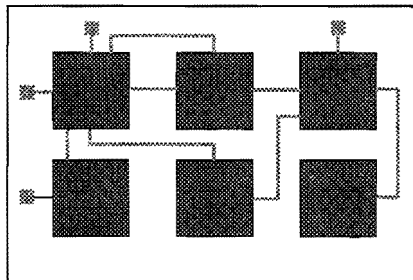


figure 3.1 layout of a PCB

## 3.2 The primary process

### 3.2.1 The Product

The model factory produces printed circuit boards (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 layout of a PCB is depicted in figure 3.1.

### 3.2.2 Operations

The model factory is a miniaturised model of a PCB production line. It emulates operations that are performed on real PCBs during their manufacturing process. The operations have been derived from case studies of real PCB manufacturing facilities [Rozendal 91]. The operations are:

- screen printing: the bare PCB is positioned in the workstation, a PCB-specific screen is selected and moved into position, and a "squeegee" is reciprocated horizontally over the screen to attach imaginary solder paste.
- component placement: the pasted PCB is positioned in the workstation, and components are placed on the positions with the imaginary solder paste according to the component-placement recipes for that product.
- reflow & cleaning: populated PCBs are passed through an oven and cleaning station. For environmental reasons, no actual reflow or cleaning is performed.
- test & repair: the PCB is inspected to see if it contains the components in the designated position (according to the recipe), and component and functional tests are performed. If the PCB fails, it must be routed to an off-line diagnosis and repair workstation. Upon successful repair, the PCB is routed back to the test station.

In addition to these basic operations, the model factory contains other features. Empty boards and components are automatically supplied from a centralised raw material store or component store. The model factory can support mixed model flow production, where different types of products can be manufactured nearly simultaneously. 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.

### 3.2.3 Process layout

The process layout is depicted in figure 3.2. The operations are indicated by square boxes, while stocks and buffers are indicated by triangles. The first stock in line contains the two types of empty boards. The following station is the screenprinter. After the screenprinter, alternative routings are possible between two component placement stations. The next station is reflow and cleaning. The in-process-store consists of three first-in-first-out (FIFO) locations for three products each. The repair buffer in the test & repair cycle can contain one batch. The final-product-store is randomly accessible and can contain nine individual products.

An additional feature is a loop from the in-process-store to the screenprinter. 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 FIFO buffer in this second-side loop can contain nine products.

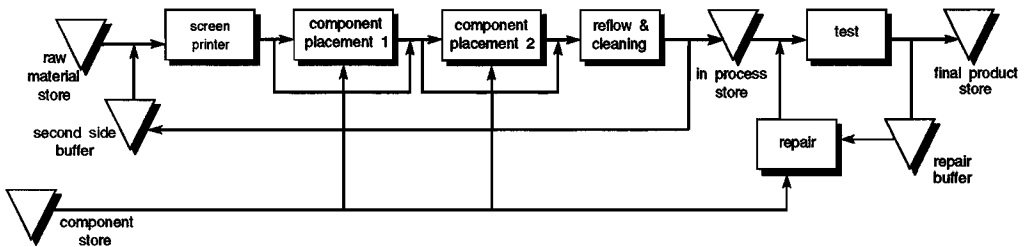


figure 3.2 process layout of the model factory

### 3.2.4 Hardware

All workstations in the model factory are fully automated, with the exception of the repair station, where a human 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, etcetera. This necessitates many sensors in the factory, in addition to solenoid stops, motors, lights, conveyers, pneumatics, etcetera.

Obviously, all the logical i/o signals to and from these sensors and actuators are controlled by a Programmable Logic Controller (PLC) and its associated programs. To get an impression of the size of the system: there are 170 inputs and 150 outputs to the PLC, with a program size of 4 K (700 rungs). There is also a higher level supervisory system to manage the overall production process. Thus, the computer hardware and software are implemented in two levels, a PLC level and a VAX level. In summary, a considerable amount of equipment was needed to realise the i/o control and the supervisory computer system.

### 3.3 The control system

Two different architectures for the control system of the model factory have been specified and implemented: a hierarchical control architecture and a distributed control architecture.

The hierarchical architecture is based on the COSIMA architecture [Duggan 90] [Duggan et al. 91], and is characterised by a two-level scheduling and dispatching. At the highest level a work order is received and planned in a static schedule. The second level provides dispatching and detailed dynamic scheduling. Feedback is obtained by monitoring the status of the model factory. Monitoring is performed on lead-times and transport times.

The distributed architecture is characterised by autonomous controllers for each self-contained unit of the model factory. This results in a pull oriented control of the model factory. The last controller in the line receives a work order which is consecutively passed to the other controllers as requests for production. Each request is related to an individual batch.

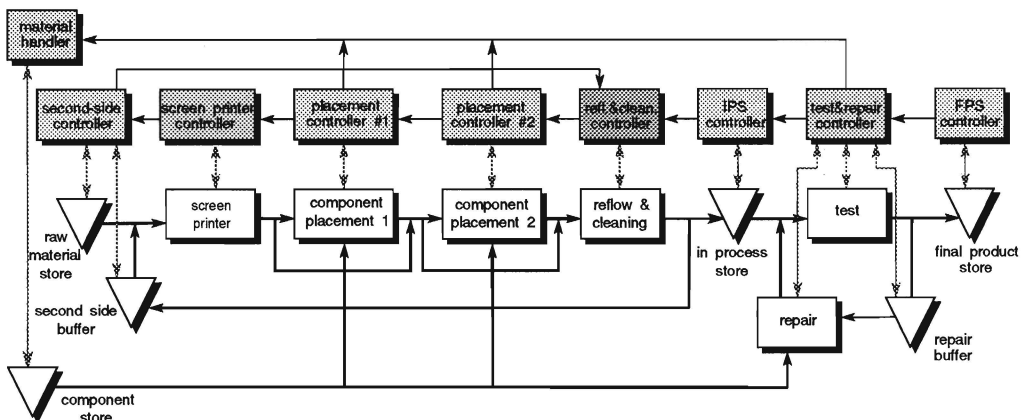


figure 3.3 decentralised control model

Both architectures have been implemented in the model factory, using the same computer hardware, system software and model factory layout. The example in this thesis is limited to the distributed architecture. A discussion of both architectures and their implementations can be found in [Timmermans et al. 93]. The distributed control architecture is depicted in figure 3.3. This figure shows the flow of products through the model factory, as well as the flow of requests between the controllers.

The principal functionality of the implemented control architecture is as follows. The last controller in the line, the final product store controller, receives a work order. This controller creates a batch and a request for that batch based on Statistical Inventory Control. The controller specifies

in the batch definition what type of products have to be produced and how many (the batch size). The request for this batch is then sent to the preceding controller in line, the test & repair controller. This controller in its turn creates its own request for the same batch and forwards it to its preceding controller. This process of receiving and forwarding requests continues until the beginning of the production line is reached. Here, the physical batch is created by releasing empty boards from the raw material store. This physical batch is sent to the requesting station, thereby eliminating the request for that batch. Hence, all requests for the batch will be eliminated when the physical batch arrives at the final product store. Here, the batch definition itself may also be removed from the information base.

The modules in the decentralised control architecture are characterised as follows:

#### *Material handler*

The material handler controls the replenishment of components from the component store to both component placement stations and the repair station. It gets requests for components from the repair controller and both component placement controllers. The material handler must then schedule, and dispatch the replenishment of components.

#### *Second side controller*

The second side controller controls the loop in product flow caused by double sided PCBs. The controller receives requests from the screenprinter controller. When the request concerns the second side of a double sided PCB, the controller forwards the request to the reflow and cleaning controller. If the request concerns an empty board, the second side controller drives the raw material store to satisfy the request. When the screenprinter is available the second side controller moves a batch to this machine.

#### *Screenprinter controller, component placement 1 controller, and component placement 2 controller*

Each of these controllers translates a request coming from the succeeding controller into a request to be sent to the preceding controller. These requests are stored in a queue, and processed when the station becomes available. Once work is received by the station, it is processed immediately.

#### *Reflow & cleaning controller*

This controller handles the requests that come from the in-process-store controller or from the second side controller. It converts the incoming requests into outgoing requests, which are sent to placement controller 2. Once work is received by the station, it is processed immediately. After the reflow and cleaning operation, the batch is moved to the second side buffer or to the in-process-store, depending on the station which requests the batch.

*In-process-store controller*

The in-process-store controller controls the in-process-store. It gets requests from the test & repair controller. The requested products are delivered from stock, if they are available, to the Test & Repair station. The stock is controlled by Statistical Inventory Control (SIC). A request for a replenishing batch is made to the reflow & cleaning controller when the inventory becomes below a minimum stocklevel. The stock of the in-process-store is refilled upon arrival of the batch. Thus, the in-process-store acts as a decoupling point, to allow for fluctuations in the demand for the factory.

*Test & repair controller*

This controller controls the testing operation and the repair operation. The controller receives its requests from the Finished Product Store controller. It converts the incoming request to an outgoing request and sends it to the in-process-store controller. The controller coordinates the flow of batches from the in-process-store and the repair buffer to the test station. A batch from the repair buffer has priority over batches from the in-process-store in order to avoid a deadlock in the material flow. This deadlock can occur when a batch is rejected by the test station, and at the same time there is a batch in the repair buffer.

*Finished product store controller*

This controller controls the finished product store. The controller receives production orders. They can be provided by another planning system. This planning system is however outside the scope of the experiment. On the due date of the production order, the finished product store controller starts delivering the order from stock. Replenishing takes place similar to the procedure of the in-process-store. A request for a batch is created and sent to the test & repair controller based on SIC. When all items for a production order have been delivered, the production order is closed.

### **3.4 The information system**

The information system for the distributed control architecture is designed and implemented according to the method of modular design [Koopmans 92]. An information system module is specified for each of the controllers in the control architecture. The specification consists of a functional description of the module, a conceptual schema, and the domain definitions. The modular design enabled the one-by-one design and implementation of the controllers in the model factory. It is also possible to enhance the system by adding new modules without effecting the existing ones, or to modify one of the modules without effecting the other modules.

A specification of the screenprinter module is given in this section. The specifications of the other modules can be found in appendix B.

### 3.4.1 Functional description

The screenprinter station has the most straightforward controller. The controller receives a request from the component placement 1 controller. This request is converted to a request for the second-side controller. In due time, the screenprinter station receives a batch from the raw material store or the second side buffer. The type of the products of the batch is then identified. Dependent of the type of product a screen-printing mask is selected, and a "squeegee" operation is performed. The batch is forwarded to component placement station 1 after all operations have been performed on all products in a batch.

The control of the physical operations is performed by a PLC program. It is the task of the controller to give orders to the PLC program.

### 3.4.2 Conceptual schema, constraints and domain definitions

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure 3.4. The central object classes in this diagram are request and batch. A request refers to the batch that is requested. Additionally, a request refers to the station that will consume the batch of the request and to the station that will produce the batch of the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. The buffer, which belongs to a certain station, has also an optional relation to batch to indicate the batch it contains.

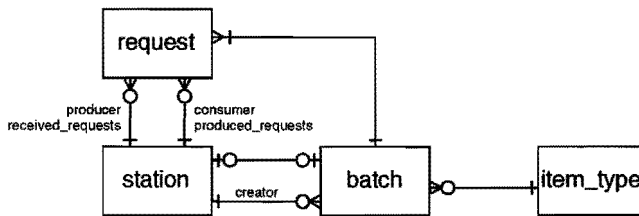


figure 3.4 data structure diagram of screenprinter controller



## SCHEMA screenprinter

## CLASSES

**class** station**attributes**

```
station_name : string;
produced_requests : SET OF request;
received_requests : SET OF request;
batch_available : {available, non-available};
ready_to_receive : {Yes, No};
batch_in_process : batch;
```

**end;** -- class station**class** batch**attributes**

```
batch_id : integer;
creator : station;
item_type : item_type;
size : {1..3};
```

**end;** -- class batch**class** request**attributes**

```
producer : station;      -- the station that will produce the batch,
                          -- i.e. receive the request
consumer : station;     -- the station that will consume the batch,
                          -- i.e. create the request

batch : batch;
item_type : item_type;
```

**end;** -- class request**class** item\_type**attributes**

```
item_type : string;
second_side : {yes, no};
```

**end;** -- class item

**INTEGRITY CONSTRAINTS<sup>1</sup>**

-- for every outgoing request there must be an incoming request with the same item type and batch size: for every information base state  $i$  must hold that for every request that this station produces there must exist a request that it receives, and the item type and batch size of both requests should be identical.

$$C1(i) = (\forall r: r \in i.request \wedge r.consumer = 'screenprinter': \\ (\exists r': r' \in i.request: r'.producer = 'screenprinter' \wedge r'.batch.item\_type = \\ r.batch.item\_type \wedge r'.batch.size = r.batch.size) \\ )$$

-- for every information base state  $i$  must hold that the maximum number of requests created by the screenprinter for one batch is 1.

$$C11(i) = (\forall b: b \in i.batch: \\ (\# r: r \in i.request: r.consumer.station\_name = 'screenprinter' \wedge r.batch=b) \leq 1 \\ )$$

-- for every information base state  $i$  must hold that once a batch is in process in 'screenprinter', there may be no outstanding requests for that batch by 'screenprinter'

$$C12(i) = (\forall s, b: s \in i.station \wedge b \in i.batch \wedge s.batch\_in\_process=b \wedge \\ s.station\_name = 'screenprinter': \\ \neg (\exists r: r \in i.request: r.batch=b \wedge r.consumer=s) \\ )$$

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

$$C20(i) = (\forall s: s \in i.station: s.produced\_requests = \{r \in i.request \mid r.consumer = s\})$$
**DOMAIN RULES**

-- the own domain of the screenprinter module consists of the objects of the object types request and station that have 'screenprinter' as the name of the (associated) station.

$$\text{own domain (i)} = \{t \in i.request \mid t.consumer.station\_name = 'screenprinter'\} \cup \\ \{t \in i.station \mid t.station\_name='screenprinter'\}$$


---

<sup>1</sup> not all constraints have been included here.

-- the foreign domain of the screenprinter module consists of all objects of the object types item type and batch, the objects of the object type request with 'screenprinter' as the station name, and the objects of the object type station with 'second-side' or 'component-placement#1' as the station name.

```
foreign domain(i) =      {t ∈ i.item_type} ∪ {t ∈ i.batch} ∪
                        {t ∈ i.request | t.producer.station_name = 'screenprinter'} ∪
                        {t ∈ i.station | t.station_name = 'second-side' ∨
                          t.station_name = 'component-placement#1'}
```

END; -- schema screenprinter

### 3.5 Complexity and limitations of the model factory

Many elements of the model factory are encountered in real production environments. Such as:

- problems such as rework with their subsequent effect on test & repair equipment utilisation
- alternative routings for particular products where duplicate equipment is available
- loops of products through the same equipment
- complexity brought about by the mix model flow production, as well as the necessity of controlling many independent, cooperating workstations.

There are different types of devices present in the factory, all of which are found in a real industrial environment. The control hardware, i.e., PLCs and supervisory computers, are state-of-the-art for industrial control. Also the commercial software tools and the integration platform corresponds to modern industrial environments.

A number of unexpected events can and do occur because of the complexity of the model factory, which have to be anticipated for in the management of the factory. This reveals some limitations of the model factory and the current implementation. The produced products are simplistic and non-functional, and the amount of different products capable of being manufactured is limited. Some of the operations are only simulated, which means that problems related to operation control will not be encountered. Until now, the emphasis of the model factory implementation has been on the logistics management of the factory. There are many other issues to be considered when managing a real factory, such as quality management, maintenance, work-in-process tracking, product development, process planning, work preparation, cost control, etcetera. Of course, human interaction with shop floor management systems is vitally important in real

situations. All of these aspects have more or less been outside the scope of the project.

Many other systems are required to run a real factory in addition to the facilities and systems to control the shop floor. Examples of these are purchasing systems and invoice systems, MRP systems, shipping and distribution systems. Neither these systems nor their interfaces are considered in this project.

The consistency of the information system in case of exceptions, in particular concerning error detection and error recovery, is another limitation in the present implementation. In the current implementation there are many situations that cannot be dealt with. Examples of these are the manual removal of batches while they are in process, adding products or batches in the production process, or the restart of a single station after breakdown. Inconsistencies of the information base will occur in all these situations. Only a restart of the complete factory is possible now. Extra equipment such as bar code readers and other sensors are necessary to solve this problem. Moreover, it would require a considerable amount of control software to check and recover impermissible or undesired states. For example, the stations in the model factory are equipped with optical sensors to detect the arrival of a product. When a batch arrives however, only the first product is detected. Consequently, it must be assumed that the whole batch has arrived. An error occurs if this is not the case, for example in the situation where the transport system does not handle batches but individual products. The other products of a batch may be delayed, and therefore not yet be present when the station starts an operation. The installation of additional sensors and control software to detect the arrival of all products in a batch would solve this problem. However, it would not solve the situation where products do not arrive at all. Including robustness and error detection and recovery in the conceptual design would therefore increase the complexity drastically, and the necessity for modular information systems would even be greater.



## Chapter 4

# Information systems in manufacturing

### 4.1 Introduction

#### 4.1.1 Outline of this chapter

This chapter presents an overview of current methods for information systems design in manufacturing. The goal is not to make a complete overview, but to provide a background for following chapters.

The outline of this chapter is as follows. Section 4.2 introduces a common method for the design and implementation of control systems, namely the definition of layered architectures. Two architectures are discussed in more detail, the hierarchical control architecture and the layered implementation architecture. It will be concluded that neither architecture suffices for solving the design and implementation problems of modular information systems.

Hierarchical architectures for production control are discussed in section 4.3. It will be explained why hierarchy is introduced and what the consequences of hierarchy are for the design of modular information systems. Section 4.4 will then discuss layered implementation architectures. It will be argued that most implementation architectures are hierarchical in nature, due to historical technical reasons. However, alternatives are now available. The relation between implementation architectures and modular design of information systems is discussed. In section 4.5, a comparison is made between the hierarchical control architecture and the hierarchical implementation architecture, based on the model factory. Section 4.6 then discusses the limitations of both architectures concerning the redesign of systems. Section 4.7 completes this chapter with a short discussion of software packages for shop floor control. The conclusions of this chapter are presented in section 4.8. The most important conclusion will be that both architectures can be useful for the reduction of complexity, but are not sufficient for the design of modular information systems.

### 4.1.2 Problem statement

The design of manufacturing systems includes product design, process design, and the design of the control system. All three aspects are found in the literature on CIM, both in isolation as well as combined. The starting-point in this thesis will be the control system. Later on, it will become clear how the other aspects of manufacturing system design have to be taken into account. Moreover, it will be emphasized that all three aspects have to be considered concurrently in the redesigning of manufacturing systems.

Currently, many design methods for manufacturing and information systems consider mainly, if not solely, the design of these systems from scratch. It is assumed that there is no system available yet, or that it is possible to adjust existing systems according to new requirements. The notion that systems are available refers not only to the existing manufacturing system that has to be replaced, but also to ready-to-buy systems, as for example commercial software packages. In this respect, the starting-point in this thesis is that the design of a manufacturing system involves in most cases a *redesign* process. The more ideal situation of design from scratch will occur only seldom. Moreover, in most cases only a part of the manufacturing system has to be redesigned.

A discussion of a design method of manufacturing systems involves issues as complexity, uncertainty and flexibility. A method has to deal with complexity in terms of the size of the manufacturing and information system, and the integration of different views into the design of these systems. Uncertainty is faced externally in the relation between the manufacturing system and its environment, and internally in the relation between the tasks of the system. Flexibility in manufacturing is defined as the ability to adjust the primary process according to new understandings of the environment [Geraerds et al. 89]. It should be noticed that there is a trade-off between flexibility and complexity: an increase of flexibility will usually also increase the complexity.

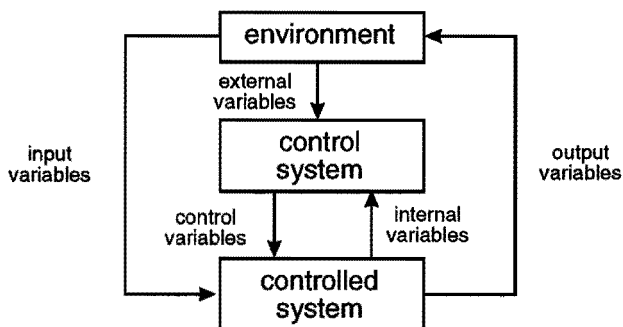


figure 4.1 the control paradigm



figure 4.2 the PCI paradigm

**4.1.3 Process, control and information**

The control paradigm of systems science makes a distinction between three subsystems: the controlled system, the control system and the environment (figure 4.1). In manufacturing the controlled system is also called the primary process, and the control system is called the production control system. The combination of the primary process and the control system is called the manufacturing system. The information system is not explicitly mentioned in systems science since it is considered to be part of the control system. Consequently, the design of information systems is often implicitly included when discussing the design of control systems. In this thesis however it will be stated explicitly when the design of the information system is included.

This thesis is based on the Process-Control-Information (PCI) paradigm for the design and implementation of information systems (figure 4.2) [Bemelmans 84]. The PCI paradigm is derived from the control paradigm and states that the characteristics of a process (and product) determine the suitability of a control system for that process. The selection of a control system is a matter of matching process characteristics with characteristics of the possible control systems. In a similar way, the characteristics of the process and the control system determine the structure of the appropriate information system.

| control levels | implementation levels |
|----------------|-----------------------|
| cell/line      | VAX                   |
| workstation    | PC                    |
| aut. modules   | PLC                   |
| equipment      | devices               |

figure 4.3 an example of levels in manufacturing



## 4.2 Levels in manufacturing

A typical way to decrease the complexity of manufacturing systems is to define levels [Bauer et al. 91]. An example is presented in figure 4.3. At each level two aspects are considered: control and (technical) implementation. The lowest control level consists of manufacturing equipment that is controlled by automation modules. The automation modules are then controlled by workstations, and a number of workstations are controlled by a cell/line controller. If necessary, it is possible to add more levels.

The lowest implementation level consists of devices such as sensors, optical readers, barcode readers, motors etcetera. These devices are controlled by real-time PLCs and PLC software. The PLCs are connected to an industrial PC for downloading programs and to provide supervisory control. The PCs receive their commands from information systems on larger central systems such as VAX stations.

Figure 4.4 shows schematically the design of an information system for production control according to the waterfall model. A traditional way to design is to start with the specification of the 'business problems' and to develop a business model with Yourdon-like techniques for structured design [Yourdon et al. 79]. The goal of the design phase is then to structure the processes, data flows and data stores of the business model in a (hierarchical) control model. Thus, the levels in the control model reduce the complexity that was met in the business model. Then, the control model is implemented in a layered implementation model.

Besides the fact that the waterfall model does not focus on re-design in particular, and that there are no natural rules for the mapping of business model to control model and implementation model, it will be argued in the following sections that the definition of levels does not naturally result in a modular information system. In particular, criteria such as modular decomposability, modular continuity and modular protection could be neglected. These criteria concern the coordination between different levels. In this respect, problems may arise when integrity constraints apply to

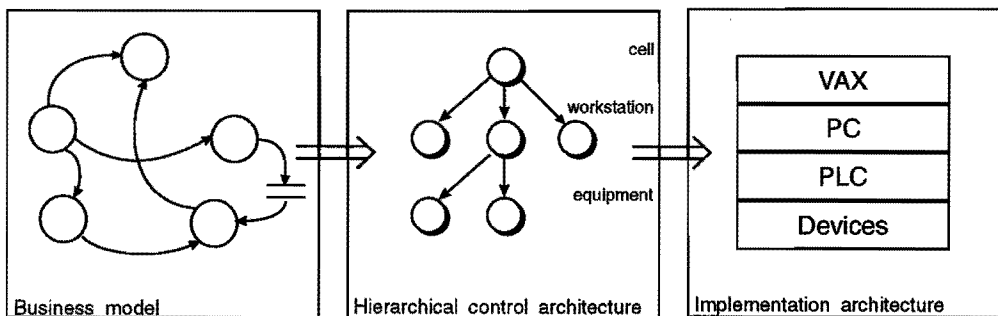


figure 4.4 example of mapping of models

more than one level, and there are no design principles for describing the interfaces between levels.

In addition, different views or aspects may result in different level definitions. Two views or aspects have been given as an example: a production control view and an implementation view. It will be argued that the mapping of these two views is not as trivial as figure 4.3 suggests. This is especially true since different designers of information systems in manufacturing tend to favour one of these views over the other, and consider the other view as derived. An information analyst for example will prefer the control view. A hardware oriented designer however will prefer the implementation view.

## **4.3 Control models**

### **4.3.1 Approaches to production control**

In chronological order, three approaches to the design of production control systems appear according to [Meal 84]: decentral control, central control, and hierarchical control. The first type of control was very much based on the primary process and without much supervisory control. It is therefore decentralistic in nature. Improved information technology initiated then a centralised approach. In this approach there is the tendency to create central decision functions that are given the power to control in detail the operational processes in all parts of the organisation. This approach results typically in large monolithic applications. Third, there is the hierarchical approach which gives the management at the various organisational positions insight into the whole situation.

The hierarchical approach will be discussed in further detail since it is most commonly applied as a starting-point for the design of flexible control systems. The other approaches will return in chapter 5 when the modularity of information systems in manufacturing is discussed. It will then be argued that it is no longer necessary nor desirable to apply hierarchical control. It is now possible to design 'distributed control systems' corresponding the principles of lean manufacturing [Womack et al. 91] and modular information systems.

### **4.3.2 The hierarchical approach**

The hierarchical approach is based on a system science perspective [Mesarovic et al. 70]. It reveals the relation of a control system as a whole and its components in terms of their tasks, functional behaviour, and performance. The hierarchical approach is exhibited in manufacturing through the definition of hierarchical control architectures. Examples of hierarchical control architectures are presented in [Jones et al. 86], [CFT 87], and [Biemans 90]. The starting-point for the design of these architectures is the inherent complexity of monolithic control systems. To avoid these large

monolithic systems, various levels of abstraction are introduced, reducing at each level the span of control, the time horizon and the time period of decision making. Generic components can be defined that can be used at each level by specifying the specific requirements for that level. At each level goals or tasks are decomposed into sequences of subtasks which are passed down to the next (lower) level in the hierarchy. This procedure is repeated at each level until a sequence of primitive tasks is generated that can be executed by simple actions. An architecture that includes this principle is presented in figure 4.5. Such architectures are implemented in pilots and demonstration projects as for instance the CIMphony project at Philips Research [Sol 89].

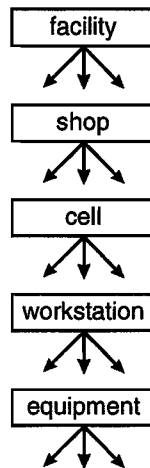


figure 4.5 The NBS hierarchical control architecture [Jones et al. 87]

### 4.3.3 Modularity and flexibility in hierarchical control architectures

A hierarchical decomposition provides a structured method for the reduction of complexity in control systems. There are however a number of important drawbacks to this approach. Five drawbacks are mentioned here.

First, the approach does not provide adequate criteria for the decomposition of the tasks or levels. This is reflected in the fact that different hierarchical models propose a different number of levels [Jones et al. 86], [CFT 87], [Biemans 90].

Second, the hierarchical approaches rely heavily on the rationalisation of the control system. It is assumed that the primary process is fully understood and that its behaviour can be modelled as far it is of relevance to the control system. The validity of this assumption will be questioned in

the next chapter.

A third drawback results from an information system view on organisational control. Galbraith mentions in [Galbraith 73] that 'the weakness of hierarchical communication systems is that each link has a finite capacity for handling information. As the organisation's subtasks increase in uncertainty, more exceptions arise which must be referred upward. As more exceptions are referred upward, the hierarchy becomes overloaded. Serious delays 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.'

Fourth, the applicability of generic components that can be used at each level is rather limited since the behaviour and content of a component is often very specific for certain applications and depends on the position of a component in the hierarchy. In particular when performance requirements play an important role, it is difficult to develop standard software for isolated components.

Fifth, a controller in a hierarchical control architecture must have substantial knowledge of the lower levels to be able to control these levels [Dilts et al. 91]. This is in particular of interest for exception handling, such as start-up, shut down, error detection and error recovery. In these situations, the controller has to know the states of the lower levels. Indeed, it would require a large and complex conceptual model to describe the possible states, which is in direct conflict with the aim of a hierarchical architecture, which is to reduce the complexity.

#### **4.4 Implementation architectures**

While an information analyst type of designer may prefer the control oriented design of an information system for manufacturing, a hardware oriented designer may focus on the implementation architecture and use a bottom-up approach. This bottom-up approach entails a distinction between levels of hardware and its associated software. Functionality is added to the manufacturing system bottom-up and level by level by means of building blocks consisting of hardware and software. The number of levels in this implementation oriented architecture is not definitely determined, as was also the case with hierarchical control architectures discussed in the previous section. An example of an implementation oriented architecture is presented in figure 4.6. At the lowest level one finds the physical hardware of the manufacturing system that specifies the possible transformation processes. The second level consists of a PLC and PLC software. The third level includes workstations and station controller applications. The fourth level is defined by mini-computers for shop floor planning and control, and the fifth level contains mainframe computers for MRP and MPS tasks.

Such a layered implementation architecture provides an adequate method for modular composability and understandability. The functionality of each level can be defined clearly. All

real-time software for example can be implemented at PLC level, and non-time-critical software at higher levels. Workstations are used for hour-to-hour machine control, etc. Thus, understandability between levels is improved to a large extent. The commercial availability of hardware and software is an important advantage of this approach: CIM component manufacturers tend to offer the functionality in a combination of hardware and software. It is therefore possible to select the appropriate components to fill in a level. Furthermore, tools are available to create bridges between different levels. Hence, the modular composability criterion is met by this method.

The disadvantage of implementation oriented architectures is that it tends to result in systems that are difficult to decompose. There are two reasons for this. First, a level by level implementation of modules emphasizes on the one hand the simplification of interfaces between components on one level. On the other hand it often assumes the interfaces between levels to be simple by the nature of the layered architecture itself, which is not necessarily true. This could likely result in many-to-many relations between components at different levels which is a burden for decomposability. Second, there is the danger that each level uses the lower levels as their foundation. Any change in this foundation could immediately affect all the components that rely on it. This issue will be discussed in more detail in section 5.2.3.

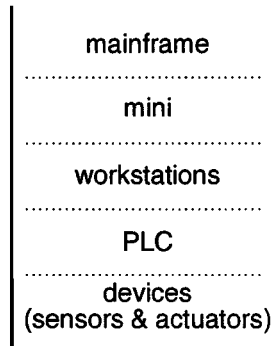


figure 4.6 example of implementation levels

## 4.5 Hierarchical control versus layered implementation architecture

Erroneously, hierarchical control architectures are often confused with layered implementation architectures. Especially at the lowest levels, it might seem promising to make a one-to-one mapping of control levels into implementation levels, as was illustrated in figure 4.3. Some of the problems this could lead to can be illustrated by the design and implementation process of the model factory.

Initially, the control system and the model factory hardware plus the PLC programs were developed concurrently. The functional specifications of the model factory were sent to a manufacturer in California, USA, for the design and implementation of the model factory hardware and the PLC programs. Simultaneously, these specifications were used for the design and implementation of the control system in the Netherlands. The goal of the control system developers was to create a modular information system for the control of the model factory. That is, the information system should be developed according to the principles of modular design as presented in chapter 2.

The distance and time difference between both groups made it difficult to communicate intensively on details. As it appeared, both groups had different perceptions of the mapping of control levels onto implementation levels. Apart from flaws in the design of the model factory hardware and PLC software, differences appeared when the following questions had to be answered:

1. which functionality belongs to which implementation level
2. how are modules defined.

An example of different answers to the first question appears at the implementation of product routings. While the hardware/PLC designers implemented these in the PLC software, the control system designers would probably implement them in the control software. Yet, another example is already given in chapter 3: only the first product in a batch is registered by an optical reader upon arrival at a station. A time delay was built in the PLC software to be sure of the arrival of the complete batch. These delays are highly undesired according to the control system designers. They would have preferred the installation of additional optical readers. This would however make the model factory hardware and the PLC software more complex.

A different answer to the second question is given, for example, concerning the definition and implementation of conveyor belts that serve multiple stations. The hardware designers reduced the complexity of the model factory by implementing relatively large conveyor belts that serve multiple stations. For the control system designers, however, this resulted in the sharing of resources by different independent modules. The synchronisation of modules that share conveyor belts made an increase of the interfaces of the modules necessary.

The conclusion is therefore that the introduction of control levels and implementation levels are two different approaches for relieving the complexity problem. Often, it is not possible to make a one-to-one mapping between both models, as is demonstrated by the model factory example. The problem of mapping is illustrated in figure 4.7 where a hierarchical control model has to be mapped to an implementation model using a local area network.

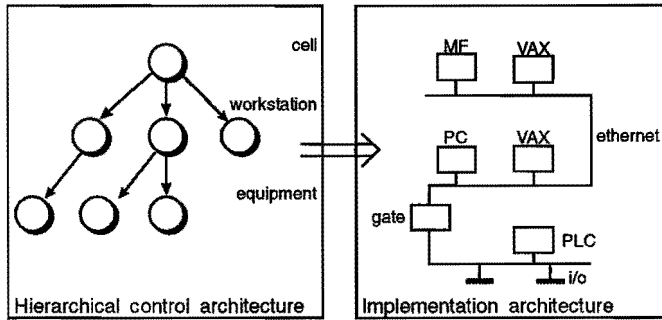


figure 4.7 mapping of control levels to implementation levels

## 4.6 Redesigning the process

Another limitation to both architectures is that they mainly focus on design from scratch and disregard the (physical) limitations of the primary process. When redesigning a manufacturing system, it will often be impossible to change certain parts of the present primary process due to limitations concerning:

- shop floor layout aspects
- social aspects
- technological aspects
- economical aspects

First, shop floor layout aspects refer to the layout of the primary process. There may be constraints in the shop floor layout that prohibit a certain control architecture. A line layout for example cannot provide many features for rerouting, while the flow of products in a system with a job shop oriented layout can be defined freely.

Second, there are the social aspects. These aspects together with the layout aspects are discussed by for example Group Technology (GT) and Production Flow Analysis (PFA) [Burbidge 89]. The consequences of social aspects for control architectures are illustrated by the argument of Burbidge that the correct application of GT and PFA would substitute the use of more complex control systems, most notably MRP. Social aspects may for example also limit the creation of 'virtual cells'. Virtual cells are manufacturing cells that exist only in the production control software for planning reasons. The reason is that virtual cells tend to neglect the human role in manufacturing.

Third, there are technological aspects that influence the possibilities of control systems, such

as communication protocols, differences in data formats and the capacity of a network. These aspects are of particular importance in highly automated shop floor control. The aim of this thesis is not to extend on how to include these aspects as such. However, it is important to aim at a method that allows the consideration of the limitations of the primary process and to address problems related to redesign. In this respect, this thesis will contribute to these issues in the following chapters.

Fourth, there are economical aspects that require that parts of the primary process cannot or will not be changed. The high costs involved with implementing new machines is an example of an economic aspect that might prevent changes in the primary process.

## **4.7 Software packages for shop floor control**

A short list of software packages for shop floor and cell control will be discussed in the section to complete the overview of this chapter on information systems in manufacturing. The goal of this section is to provide the reader with some characteristics of the present software that can readily be bought. Each of the software packages will be discussed in terms of their main functionality and their architectural principles.

Since it was not the main goal of this research to provide a detailed overview of available software, it is not possible to present here the data models underlying the software packages. However, this would have provided a more in-depth knowledge of the functionality of the software package, as is indicated in [Bertrand et al. 90] and [Heij 91].

### **COSIMA**

COSIMA (Control System for Integrated Manufacturing) is a system for Production Activity Control, developed as part of Esprit project 477 [Duggan 90] [Duggan et al. 91]. The system is based on the production management system hierarchy described in [Bauer et al. 91], and is broken down into five distinct building blocks; namely a scheduler, a dispatcher, a monitor, producers and movers. The COSIMA architecture is presented in figure 4.8.

### **PLATO-Z**

PLATO-Z (Production Logistics and Timings OrganiZer) is a framework for production planning and control containing an 'intelligent cell control system' (ICCS) [O'Grady et al. 88]. PLATO-Z is based on a combination of the hierarchical control systems proposed by CAM-I and NBS ([CAM-I 84] and [Jones et al. 86] respectively). The goal of PLATO-Z is to make decisions as low as possible in the hierarchy, where the cell takes over much of the responsibility concerning scheduling, dispatching, error recovery, cell initialisation and termination, communication and networking, as well as the user interface. The ICCS is implemented using a multi-blackboard/actor-



based framework containing several blackboard subsystems each of which performs major cell functions. This model aims to provide an architecture in control is achieved by passing appropriate messages between blackboard subsystems.

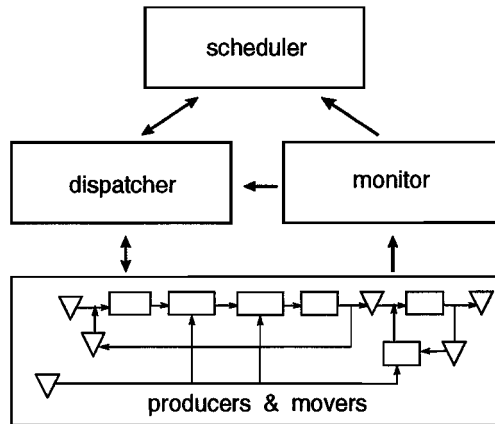


figure 4.8 COSIMA architecture

### MADEMA

MADEMA (MAnufacturing DEcision MAKing) is a system for process planning and scheduling based on the CAM-I hierarchical control architecture [Bunce 88]. It aims at manufacturing decision making at the work centre level which involves the assignment of resources to production tasks. MADEMA involves a database that describes jobs, work centres and resources that make up the factory. In addition, the system consists of a simulator and a decision making module containing different scheduling techniques.

### INFINET

INFINET comprises a set of products based on the CAM reference model [CFT 87]. This product set makes a first distinction between: product process development, production process control, and support layer. One of the products in production process control is the workcell controller, consisting of three major components, the workcell kernel, workcell support and the workcell user interface. The workcell kernel is responsible for making dispatch decisions and providing line setup control. The workcell support is responsible for maintaining the database and performing all automated transactions. The workcell user interface interacts with the operator to allow scheduling, data entry, monitoring and reporting transactions.

**VAXplant**

VAXplant is a shop floor information system for tracking, analysis and on-line monitoring of production data, with capability to manage work order and resource data. It interfaces between the production resources at shop floor level and the Plant Information System (PIS). VAXplant is based on a relational database. The system is provided with orders from an MRP(-like) system which are then released to the shop floor. The orders are tracked and information about the order status is returned to the PIS.

**Workstream**

Workstream is a shop floor control system that offers a data repository specifically designed to support plant floor management, execution, and improvement [Consilium 92]. It is built on a central data base system that includes data on workorders, work-in-process, bills of material, inventory, etc. The system performs data-intensive functions such as scheduling, dispatching, tool management, and work-in-process tracking. As such, the shop floor controller consists of configurable application packages around a standard database package. On the one hand, the system can be integrated with a system for production and inventory control. On the other hand, it can be integrated with a workcell controller that provides equipment interfaces for monitoring and control, operator status and work instructions, real-time quality control, and coordination of manufacturing devices and operators [Pelusi 90].

In addition to these shop floor and cell controllers, numerous systems are available to specific tasks such as production planning and monitoring. FACTOR, CIMPICS, AHP Leitstand, to name a few, are examples of systems for the specific task of production planning.

**4.8 Concluding remarks**

An important aim when designing manufacturing systems is to obtain modularity in such a way that modifications according to environmental changes or internal requirements can easily be implemented. According to the PCI model, this modularity should be reflected in the information system, which is the subject of this thesis. This chapter presented an introduction into two commonly used architectures for the design of manufacturing systems. The goal of this introduction is to provide a background for the modular design of information systems in manufacturing.

This introduction shows that there can be different aims when designing modular systems. Hierarchical control architectures for example focus strongly on the rationalisation of the manufacturing system. However, company specific systems with a low degree of modularity in terms of modular continuity and modular protection may be the result.

Implementation architectures on the other hand tend to result in systems that are difficult to

decompose. The relations between components within a level will be clear and relatively simple. Many interfaces could however be created between levels. These interfaces are difficult to maintain from both a conceptual and a technological view. The result of this is a layered architecture of dependent components. Moreover, there is the danger that each level uses the lower levels as their foundation. Any change in this foundation could immediately affect all the components that rely on it. In addition, both architectures focus mainly on design from scratch and disregard the (physical) limitations of an existing primary process.

The main conclusion of this chapter is therefore that introducing levels is not sufficient to obtain modular manufacturing systems. Other design and redesign principles have to be applied as well. The following chapters will concentrate on principles for the design and redesign of information systems.

---

## Chapter 5

# Modular design in manufacturing

## 5.1 Introduction

### 5.1.1 Contents of the chapter

The aim of this chapter is to provide principles for the design, re-design and implementation of a modular information system for manufacturing, based on the findings of the previous chapters, and with an emphasis on shop floor control systems.

The following two subsections (5.1.2 and 5.1.3) will provide a retrospect in the realisation of this chapter, and the concept 'flexibility' will be explained into more detail, since the meaning of this concept often differs according to different authors. The question to be answered in section 5.2 will then be: "what should be the starting-points for information system design?" It is argued that there is no one-best-way method for the design and implementation of information systems for manufacturing. Every method has its merits and its limitations. It is however worthwhile to consider what the main assumptions of a method are. A variety of assumptions and their validity concerning modern manufacturing systems and technology will be discussed in section 5.2.

The question to be answered in section 5.3 will then be: "what has to be considered when designing a module?" In the previous chapter, some approaches to the design of manufacturing systems and information systems for manufacturing were discussed. It was illustrated that these approaches do not necessarily satisfy all the criteria for modular design as specified in [Meyer 88] and as discussed in chapter 2. Section 5.3 will discuss criteria for determining a module.

Section 5.4 will discuss implementation issues that affect the modularity of an information system. In [Pels 88] it is argued that in the conceptual design phase it is sufficient to consider only the design of the conceptual schema to obtain independent modules. In this phase it should be possible to disregard the implementation issues of user interface, processing, data management and communication. According to Pels, communication between modules is merely considered as the sharing of data between two modules [Pels 88]. When moving to the technical design and implementation however, choices have to be made between different communication protocols, operating systems, database management systems, etcetera. The definition of implementation architectures appears to be critical for the implementation of modular information systems.

The subject of section 5.5 will then be the relation between implementation architectures and

control architectures. Distributed control architectures are considered as an alternative for the more commonly used hierarchical control architectures.

The reuse of software is an issue that should be considered when discussing design and re-design of information systems. In this respect, reuse of software is more than the reuse of programming code only. To allow fast modifications of an information system, one has to consider a method for the reuse at different levels, comprising both design and implementation. Section 5.6 will provide an introduction to this method.

### 5.1.2 Retrospect

The previous three chapters have set the stage for a discussion of principles of modular design and the consequences of these principles for the implementation of information systems.

A method for the modular design of information systems is presented in chapter 2. This method originated in the method described in [Pels 88]. At that time however, there was no experience in the application of the method in the area of CIM yet. The first research goal leading to this thesis was therefore the application and validation of the method in this particular area. An experiment was set up in the area of shop floor control. This experiment has been presented in chapter 3 and appendix B, and results are presented in a range of papers and research reports: [Frissen 91] [Rozendal 91] [Baats 92] [Koopmans 92] [Hakkesteeft 93] [Timmermans et al. 92] [Timmermans et al. 93] [Timmermans 93]. These results indicate that the method provides good guidelines for the modular design of information systems. Among the advantages are:

- the method provides a formal approach to the identification, design, analysis and implementation of modules. It creates much structure in the discussions between project members. Without the method these discussions would likely have resulted in discussions about common-sense design principles. Because of different backgrounds and opinions of project members, these common-sense design principles might not be as common as required.
- the developer does not need to have a total view of the model factory. This made it possible to design and implement the modules one-by-one, only considering the interfaces to the neighbouring modules.
- it is easy to modify one module or to enhance the model factory with new modules.
- it allows the structuring of the information system according to the primary process and the control system, as is prescribed by the PCI paradigm.
- there are possibilities for quantifying modularity in terms of complexity, coupling, and cohesion.
- there are possibilities for defining generic modules, an issue that is elaborated on in chapter 7 of this thesis.

The research also indicated that the modularity of a shop floor control system is not solely determined by the modularity of the conceptual schema. Other aspects that have to be considered are the product, process and control structure. Moreover, modularity of a shop floor control system is also determined by the implementation architecture of the system.

In addition to the experimental research, a survey was carried out of methods for the design and implementation of complex information systems in manufacturing. This survey highlights a widely used solution to the reduction of complexity and the increase of flexibility, namely the definition of layered architectures. It was however concluded in chapter 4 that these architectures are not sufficient for the design of modular information systems.

### **5.1.3 Concepts and limitations**

#### **Flexibility**

Flexibility is a key concept in manufacturing system design [Geraerds et al. 89]. Flexibility in manufacturing is defined as the ability to adjust the primary process according to new requirements of the environment. Examples of flexibility in manufacturing are: flexibility in production volume, in product design, in machine change-over times, in shop floor lay-out, in production planning, in batch sizes, in the availability of personnel, etcetera.

Flexibility in this thesis is restricted to the flexibility of the structure of information systems for shop floor control. This flexibility is defined as the ability to change the structure of the information system. Such a change may be of any size or type. Examples of this type of flexibility are: flexibility to add a station in the control architecture, to introduce new computer hardware for production control, to change the database management system for production control, to change the error recovery procedures of a particular module, etcetera.

Flexibility is strongly related to the concept of 'architecture'. An architecture is defined as a description of components and their interfaces. Architectures determines the future flexibility of a system. For example, if the architecture of a building is being changed, then the building can crash. If changes are made only within a room, for example a room redecoration, then no vital thing is affected, but if a new window is being made then the architecture is affected again.

#### **Conceptual modelling**

Flexibility of the structure of the information system will be realized through the modular design of the conceptual schema. The five modularity criteria introduced in chapter 2 will provide guidelines for realising this flexibility. The conceptual schema is a description of the universe of discourse, modelling the meaning of the information in the information system [Griethuysen 82]. The conceptual schema plays a key role in systems analysis and database design. The conceptual schema should both be an enterprise model and serve as a step between user views and the physical

database design [Griethuysen 82] [Scheer 92]. The main principles of conceptual modelling will be discussed in this chapter.

The main conclusion of this chapter will be that there are good principles for the modular design of information systems for manufacturing. The application of the principles of modular design is sometimes hindered due to (technical) limitations in the implementation of the information system. Architectures play a key role in the implementation of modular information systems. Three types of architectures are distinguished and will be introduced in this chapter: database architectures, system architectures and organisational control architectures.

## **5.2 Methodology of information system development**

There is no one-best-way method for the design and implementation of information systems, as there is no one-best-way method for the design and implementation of manufacturing systems. Moreover, every method involves certain assumptions. A few of these assumptions are discussed in this section.

The limits of conceptual modelling in information systems for manufacturing is a controversial issue that is dealt with in subsection 5.2.1. Data modelling is then discussed in subsection 5.2.2. The problem of modelling information systems for artefacts such as manufacturing systems is discussed in subsection 5.2.3. Assumptions that often appear in other methods are discussed in subsection 5.2.4. Finally, the close relation between functional and non-functional requirements is discussed in subsection 5.2.5.

### **5.2.1 Conceptual modelling**

The ANSI/SPARC three schema architecture prescribes the following two general principles for the conceptual schema [Griethuysen 82]:

1. Conceptualisation principle: a conceptual schema should only include conceptually relevant aspects, both static and dynamic, of the universe of discourse, thus excluding all aspects of (external or internal) data representation, physical data organisation and access as well as all aspects of particular external user representation such as message formats, data structures, etc.
2. 100 percent principle: all relevant general static and dynamic aspects, i.e., all rules, laws, etc., of the universe of discourse should be described in the conceptual schema. The information system cannot be responsible for not meeting those described elsewhere, including those in application programs.

It is however not always possible to meet these principles. Consider for example a shop floor control module. The 100 percent principle requires that all relevant general static and dynamic aspects of the physical hardware, the PLC, the communication software and the application software should be included in the conceptual schema of the module, as well as the functionality of sensors, devices, automation modules and stations. It is doubtful whether all *relevant* information can be specified *in advance*. In many situations this will not be possible, nor desirable.

That it is not desirable is the result from the first principle, the conceptualisation principle. This principle says that a conceptual schema should only include conceptually relevant aspects of the universe of discourse. Over a longer period of time however, it is impossible to include all relevant information and only relevant information, due to uncertainty and complexity of the manufacturing system and its changing environment. It is very probable that information would be included in the conceptual schema that would never be relevant. The 100 percent principle on the other hand requires a full description of all relevant aspects.

The principles of Van Griethuysen should therefore be considered as guidelines, rather than as strict principles. It should not be the question whether it is possible to include all and only relevant aspects in the conceptual schema. The question should be: are there possibilities for updating the conceptual schema by means of gradual changes. These possibilities are provided by a modular design of information systems, where each module is autonomous. In a modular information system it is not necessary that the conceptual schema of a module is stable over a long period of time. It will be possible to design and implement modules that can easily be changed, if the technology allows. The latter precondition will be discussed later in this chapter.

### 5.2.2 Data modelling

Data modelling can be considered as a starting-point for information system development. The reason for this is that a data model is a sound basis for the design of application software and for management decisions which improve business processes [Scheer et al. 92]. Also in [Pels et al. 90] and [Bertrand et al. 90] it is argued that the conceptual data model constitutes the skeleton of the information system. There is however a drawback to the modelling of the control flow in an information system by data models. Although (prescriptive and descriptive) dynamic constraints allow the specification of dynamics, the understandability of specifications may be insufficient in case of complex transactions. Process modelling techniques, as for example ExSpect [Aalst 92], may be used as a supplementary technique for the specification of the control flow. In [Stut 92] the integration of data modelling and process modelling is discussed in more detail. It is argued here that in a modular design of an information system that involves a complex control flow, it is necessary to extend the data models with process models in order to enhance modular understandability.



### 5.2.3 Sciences of the artificial

Layered models have been introduced in the previous chapter to reduce the complexity of a manufacturing system. It was however argued that although this method may result in a modular structure within a level, it will often also imply many dependencies between levels. This section analyses this problem in relation to conceptual modelling.

Concepts and principles for the design of information systems with the conceptual schema as a basis are described in the widely accepted report of Van Griethuysen [Griethuysen 82]. The approach to information system design in this report is illustrated in figure 5.1.

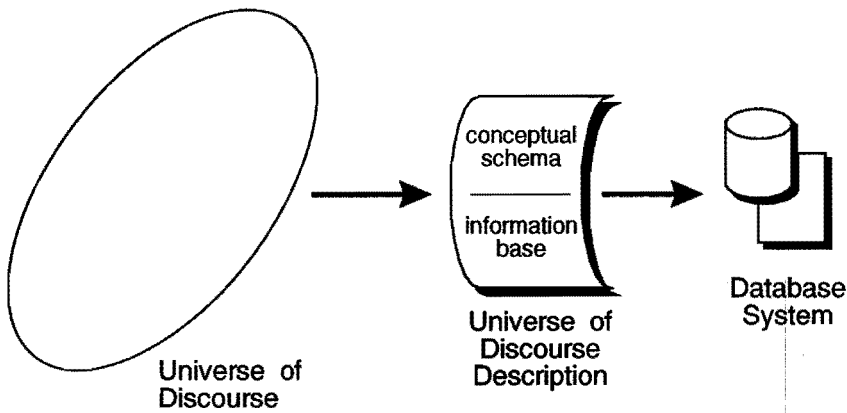


figure 5.1 approach to the design of information systems  
(adapted from [Griethuysen 82])

The universe of discourse is described in the conceptual schema and the information base. Thereupon, the information base is implemented in a database system. Van Griethuysen mentions that it is possible for the database system itself to be one of the subjects that are being modelled, in which case the database system would be included in the universe of discourse. To simplify the discussion, he further assumes that the database system is disjoint from the universe of discourse.

When designing manufacturing systems however, the universe of discourse itself is largely designed as well. This statement requires further enquiry. Consider the case where three levels are defined in the manufacturing system: the physical transformation processes, the PLC and its associated software, and the shop floor control system (including hardware and software). The steps indicated in figure 5.1 can be applied to each of the three levels. Succeedingly, the transformation processes, the PLC level and the shop floor control system are developed and implemented.

This situation corresponds to the situation initially described by Van Griethuysen, but excluded in his further discussion: the universe of discourse of the shop floor control system includes the PLC, the PLC program, the product structure, the machines, etcetera. The universe of discourse of the PLC level includes the machines and the devices, but also the components of the present shop floor control system are included in the universe of discourse of the PLC level. The same holds for the physical transformation level.

The conclusion of this section is that the design of information systems for artefacts, such as manufacturing systems, needs a careful consideration about how the universe of discourse is affected by the design itself. Although this problem is mentioned by van Griethuysen, there is no answer given concerning the manner in which this dilemma should be approached when designing information systems.

The way it was solved in the design of the model factory is to model first those parts that are most stable over a period of time; then to enhance the design in an iterative mode, and to finish with those parts that are most sensitive to future changes. For example, once the primary process of the model factory was implemented, it would be difficult to make any changes in it. A model of the primary process is therefore not likely to change. The specific features of the production planning & control system however can easily be changed, and these features were therefore modeled last. Until publication of this thesis, the primary process of the model factory has indeed not changed, while numerous changes have taken place in the production planning & control software, including the implementation of completely different control architectures.

#### **5.2.4 Assumptions**

All methods for information system design involve certain assumptions. The validity of these assumptions is questionable in particular situations. Three common assumptions that may inhibit the design, re-design and implementation of modular information systems are:

- information system design is a one-off activity
- information systems are designed from scratch
- information system requirements are fixed

First, most projects for information system design nowadays involve either the redesign or the extension of an existing information system. Hence, the applied method should focus on these situations, and the information system architecture should therefore allow redesign and extension. It is a fact that in most situations parts of the existing system cannot or will not be changed for financial or technical reasons. It is therefore a misconception that information system design is a one-off activity. Modular design of the conceptual schema avoids this assumption by emphasizing

a technique for decomposition and composition of modules.

Second, it is a misconception that information systems are designed from scratch. This way of system development is too expensive for most companies. It is more pragmatic to base a solution on the existing system, and to make use of widely available commercial software. An important activity in future information system design will therefore be the selection of commercial software packages. Data modelling provides a good starting-point in the selection of software packages, as is indicated in [Bertrand et al. 90], [Scheer et al. 92] and [Heij 91]. Furthermore, the design of infrastructures requires attention. In chapter 6 the consequences for a modular information system will be discussed.

Third, the general applicability of the waterfall model for information system development has to be discussed. Its key features are a predefined list of deliverables in each phase and the introduction of milestones, usually at the end of each phase. This model however is not applicable to all information development activities [Genuchten 91]. Van Genuchten points out that the conditions for the appropriate use of the waterfall model are relative stability and clearness of specifications. Taking the uncertainty and complexity of a manufacturing system and its changing environment into consideration, it is a false assumption to consider the information requirements as being stable. It is therefore necessary to reconsider the waterfall model, and to include flexible, generic modules based on existing components in the life cycle of an information system. Biggerstaff *et al.* mention that the following four steps have to be included in the life cycle: finding components, understanding components, modifying components and composing components [Biggerstaff et al. 87]. This approach will be discussed in more detail in chapter 7. Furthermore, when considering the trade-off between quality and costs, it might be wise to sacrifice some specific requirements in exchange for a standard solution [Genuchten 91].

As argued above, these three assumptions are no longer applicable in the design of information systems in manufacturing. Therefore, the method of modular design avoids these assumptions.

### 5.2.5 Functional and non-functional requirements

Conceptual design aims at the specification of both functional and non-functional requirements. The functional requirements are described in the conceptual schema. However, the border between functional and non-functional requirements will not always be clear. Examples of requirements that lie on the border are: fault-tolerance of a module, error recoverability, and the possibility of adding a module at run-time. Whether these requirements are considered as functional or non-functional requirements will often depend on the application area. In case of doubt, a requirement should be considered as a functional requirement and should be taken into account in the conceptual schema. This will avoid ad-hoc adjustments of the information system when it is being implemented. Consider for example the updating of the work-in-process database. In many manufacturing

systems it is sufficient to update the database once a day or even once a week. However, a continuous update procedure is required when 'real-time' rescheduling of the on-hand production orders is desired. Assumptions on the work in process have to be made in case it should not be possible to have this continuous update. These assumptions have to be included in the conceptual schema to avoid impermissible states.

The procedure described above is however not realistic for two reasons. First, it necessitates appropriate methods and tools for modelling these requirements in the conceptual design. Powerful tools are needed to model for example an error recovery procedure for an application of significant size. Most commercially available tools are still insufficient for this purpose. Second, the technology to implement a requirement has to be available. Limitations in the technology or the involved costs may prohibit the readily implementation of these requirements.

It is therefore concluded that not all non-functional requirements can always be included in the conceptual design of a system. It is not the aim of this thesis to solve this problem. However, the relation between the conceptual design of information system modules and the implementation of these modules will be discussed extensively in chapter 6. The choice of an implementation architectures contributes considerably to solving this problem. The following two sections will respectively discuss the conceptual design of a module and the implementation of a module.

### **5.3 Determining a module**

The PCI paradigm was introduced in chapter 2 as a paradigm for information systems design. According to this paradigm the information system should reflect the characteristics of the product, the primary process and the control system. The first step in the design of a manufacturing system is therefore the design of the primary process, the product and the control system. This design can include a number of activities in the factory; among those are product design, process design, production planning & control, manufacturing, assembly, physical distribution, quality control, marketing, sales, etcetera.

#### **5.3.1 Modularity criteria**

When designing modular systems, the first question is: what are the units that can or will operate autonomously. The reason to identify these units is that they will be considered as the modules for which a modular information system will be developed. A unit can for example be an individual machine, a manufacturing cell, a FMS or a planning system. The factors influencing size and contents of an autonomous unit may include technological, organisational and historical elements. The following criteria were introduced in chapter 2 for the demarcation of a module in terms of

modularity of the information system: understandability, continuity and protection. It has to be clear what a unit does on its own, changes to a unit should remain local, and run-time error recovery should be handled locally.

Two additional criteria were introduced for the specification of the interfaces of a module. These comprised composability and, often neglected, decomposability. Once the units have been identified, the aim is to create simple interfaces between the units. These interfaces have to be designed in such a way that it is easy to compose an information system from components. Moreover, it should also be easy to decompose an information system again into its components. Simplicity of interfaces is an important design criterion in the design of the information system, the control system, as well as the primary process. A distinction should be made between the volume of communication between modules and the complexity of communication between modules when designing the interfaces. A design guideline is to keep any form of communication as much as possible local. Considering current technologies, the problem is often not so much the volume of communication, but rather the complexity of the communication. Reducing communication in terms of modular information system design involves the reduction of interface specifications (objects, attributes, constraints) in both the foreign and public domains of the modules (see section 2.7).

### 5.3.2 Complexity, coupling and cohesion

Three more criteria for the demarcation of a module and the definition of the interfaces of a module were introduced in [Yourdon et al. 79] and given a specific meaning in [Pels 88] regarding a module: complexity, coupling and coherence.

The size of the module should be determined by the complexity of the module. The complexity should neither be too small nor too large. Pels defines a measure for the complexity of a module as the sum of the number of object classes, the number of attributes and the number of constraints. Although useful to begin with, it should be noticed that this measure is not very accurate, since it does not distinguish between attributes with many or few allowed values and between simple static constraints and complex dynamic constraints, and it does not include a measure for applications that operate on the module.

Coupling of a module is a measure for the knowledge that other modules have about that module. If the specifications of a module are widely used by other modules (in their foreign domain) then it would be difficult for that module to change these specifications. Coupling should therefore be minimized to allow changes in a module to take place without affecting other modules. Pels defines a measure for coupling of a module A as the average of the number of other modules for which each own specification (object class, attribute or constraint) of A is visible.

Cohesion is characterised as the insensitivity of a module for structural changes in its environment. Cohesion should therefore be maximized to create more autonomous units. A measure

for cohesion is defined by Pels as the ratio between the number of specifications in the own domain of a module and the number of visible specification of that module. These measures can be used for the evaluation of alternative designs, as for example in [Baats 92], where alternative designs for modules in the model factory are compared.

These criteria determine the contents of a module at the conceptual level. The implementation of the module determines whether the modularity defined at the conceptual level is actually realised. The following section will therefore discuss the relation between design and implementation in further detail.

## **5.4 Implementation of a module**

### **5.4.1 Introduction**

The method for modular design in chapter 2 concentrates on the conceptual design of a module. In [Pels 88] it is argued that in the conceptual design phase it is sufficient to consider only the design of the conceptual schema to obtain independent modules. In this phase it should be possible to disregard the implementation issues of user interfaces, application processing, data management and communication. Although this proposition might be right from a theoretical viewpoint, practical experience, including the model factory experiment, sometimes seems to contradict it. The independence of a module often relies on the technologies used for the implementation of that module. These are often expressed as non-functional requirements which behave as constraints for the conceptual design. It is therefore of interest to consider the relation between conceptual design and implementation.

### **5.4.2 The problem**

Consider for example the communication protocol between modules. Different commercial software packages may require different protocols. Although it is technically possible to combine these protocols into one network, this does not improve exchangeability of software. Furthermore, functional and non-functional requirements may need the replication and fragmentation of data across the network. The possibilities of this replication and fragmentation depend largely on the availability of a distributed database system. Although it is possible to specify the requirements for this system, it is often difficult to realise them. Commercially available database management systems for example do not always fully support the ANSI/SPARC three schema architecture. This creates problems concerning logical and physical data independence. The alternative of developing a database management system for one specific organisation is in most situations too expensive.

Furthermore, organisations are usually limited by existing systems. They cannot or will not update the existing systems during one giant operation. Therefore, one has to consider the existing systems and technology when designing new systems. Existing systems in this respect involve not only the automated information systems, but include also the organisational control structure.

### 5.4.3 Implementation architectures

The conclusion from the research underlying this thesis is that the definition of the implementation architecture of a manufacturing system is a critical design choice. It determines the conditions for modular (re-)design of information systems at the conceptual level. Three types of implementation architectures are of particular interest concerning the implementation of modular information systems. These are: database architectures, (networking) system architectures and organisational control architectures.

Database architectures describe the dispersion of data across the different sites of an information system. Many different database architectures exist as extensions of the ANSI/SPARC three-schema architecture. However, not all database architectures provide optimal conditions for the implementation of a modular information system.

The (networking) system architecture provides communication facilities between the modules and distribution of processing across the network. This architecture is of great importance for the technological realisation of modular composability and modular decomposability.

Finally, the organisational control architecture may create conditions or opportunities for the implementation of a distributed database system and a distributed networking system. The relation between these architectures and the implementation of modular information systems will be discussed in further detail in chapter 6. It will be illustrated that distributed architectures in particular provide best conditions for the implementation of modular information systems.

To anticipate to chapter 6, a short discussion of the design of distributed control architectures is appropriate here, since hierarchical control architectures were discussed in chapter 4 as the most common control architecture.

### 5.4.4 Distributed control architectures

Most control systems are based on hierarchical control architectures as was discussed in chapter 4 [Bauer et al. 91] [Biemans 90] [Duggan 90] [Jacques 90] [Tiemersma 92]. Although these control systems are currently used to control real manufacturing systems, they have certain disadvantages [Bakker 89] [Timmermans et al. 93]. Among the disadvantages of these 'conventional' control systems mentioned by Bakker are:

- *both the scheduler and the dispatcher are complicated components. (...) For the human operators it is frequently not clear why the system behaves as it does,*
- *the traditional control systems cannot easily be extended (...), and*
- *the high costs of the traditional control systems (...).*

The distributed control architecture is an alternative to the hierarchical control architecture that does not have the disadvantages discussed above [Bakker 89]. Further advantages of a distributed control architecture that are of interest for modular design of information systems are [Dilts et al. 91]:

- *reduced complexity and simplified development*
- *implicit fault-tolerance*
- *reconfigurability and adaptability*

There are still some restrictions associated with the implementation of distributed control architectures. Dilts *et al.* point out that these restrictions arise from inherent deficiencies and current limitations in the technology. They mention differences in internal formatting, differences in communication protocols, and incompatibilities in operating systems, file servers and database systems to cause limitations. Also network capacity and response requirements pose a problem. Sol argues however that these limitations will soon disappear with the availability of current computer power and local area networks in combination with the trend towards 'open systems' (=standard interfaces) in industrial automation [Sol 92]. In addition, Weber *et al.* present a CIM architecture that can be applied to both a hierarchical and a distributed control architecture [Weber et al. 89].

An example of a distributed control architecture is presented by the model factory in chapter 3. A more complete specification of the information system design of the model factory can be found in appendix B.

## 5.5 Development of generic modules

Another issue in the modular design of information systems is the development of components. A key issue here is the reuse of software. Reuse of software is necessary to increasing the productivity of information system development. Simple approaches, like source code reusability, reusability of personnel, reusability of designs and subroutine libraries have experienced some degree of success in specific contexts. They fall short, however, in providing a basis for a systematic attack on the reusability problem [Meyer 88].

Data models will be used in chapter 7 to discuss the following requirements for increasing the reusability of software:



- reusable components at a sufficient high conceptual level
- a policy for reusability that ultimately produces reusable programs
- a technique to describe a complex hierarchy of reusable modules, with different levels of parametrisation.

Ensuing, a method is presented for the definition of generic modules based on data models, integrity constraints and domain rules.

## 5.6 Concluding remarks

The conclusions of the previous chapters have been condensed in this chapter into a number of general principles for the design, re-design and implementation of a modular information system. An important conclusion is that the design, re-design and implementation of a modular information system cannot be considered independent from the design and re-design of the product, primary process and the control of the primary process. Then, one has to consider what type of flexibility is needed, taking into account the trade-off between the increase of flexibility and the reduction of complexity. This thesis focuses on the flexibility of the structure of information system for shop floor control.

Modular design aims at both the increase of flexibility and the decrease of complexity. The principles described in chapter 2 and this chapter should result in the design of modular information systems. There are however a few important new learnings.

The first learning is that the modularity of an information system may be limited by the technology that is used for the implementation. Database architectures, system architectures and organisational control architectures play an important role in providing the conditions for the implementation of modular information systems. These architectures will be the subject of chapter 6. It is expected that appropriate implementation architectures will make it more easy to include the modelling of non-functional requirements at the conceptual level in such a way that these requirements can also be realised in an efficient and effective way.

Second, there is a need for a method and for tools that allow fast modifications of an information system. This requirement is characterized as the need for reusable components, and will be discussed in chapter 7.

Finally, assumptions that may inhibit the implementation of modular information systems should be avoided. A design method should therefore at least address the following concerns:

- information system design is an evolutionary process
- information systems have to be built from components
- information system requirements are continuously changing.

## Chapter 6

# Architectures for distributed systems

### 6.1 Aim and content

The focus of this chapter is on architectures for the implementation of information systems. Previous chapters, in particular chapter 2, concentrated on the conceptual design of independent modules. It was assumed that the implementation of these independent modules would not face many problems. This assumption will be weakened in this chapter. It is argued here that design and implementation are not entirely distinctive phases of information system development.

The rationale of this chapter is as follows. In the conceptual design it is sufficient to consider only the design of the conceptual schema to obtain independent modules, provided that the implementation architecture fully supports the implementation of these modules. This condition will often not be met, as shown in chapter 5. There are many reasons for this, such as the costs involved in implementing these architectures, the (lack of) availability of proper technology, and the presence of a system that can not readily be updated. In those situations one has to reconsider the conceptual design, or one has to take for granted that (temporarily) the implementation does not fully support all modularity principles. It will be indicated in this chapter what implementation architectures provide the best conditions for the implementation of independent modules.

Three types of architectures are described in this chapter: database architectures, (networking) system architectures and organisational control architectures. An architecture describes the components and the relation between components of an information system. The importance of an architecture lies in the fact that it establishes the structure of the information system for a long period of time. The emphasis in this thesis lies on distributed architectures. The trend to distributed architectures is also described in literature [Bell et al. 92] [Dilts et al. 91]. It is demonstrated that distributed architectures provide better conditions for satisfying modularity criteria than for example centralistic architectures or hierarchical architectures.

Database architectures are discussed first. Section 6.2 describes different database architectures as extensions of the ANSI/SPARC three-schema architecture. Emphasis lies on the design of federated database architectures. The term 'federated database system' was introduced in [Heimbigner et al. 85] to indicate a database system with multiple conceptual schemas with local users. An extensive survey of federated database systems is given in [Sheth et al. 90]. Section 6.2 concludes with an architecture of modules as an extension of the ANSI/SPARC three-schema

architecture.

The second type of architectures, system architectures, will be discussed in section 6.3. These architectures involve the distribution of data and applications across a network. Most database researchers take for granted the existence of a reliable data communications facility in much the same way as most software assumes the existence of an operating system which provides certain standard services [Bell et al. 92]. However, when implementing a modular information system, the architecture of the distributed system is of great importance to the technological realisation of modular composability and modular decomposability. The system architecture will to a great extent also determine the non-functional requirements as for example performance and costs. Section 6.3 explains some basic understanding of the issues involved in the definition of these architectures. The client-server architecture (CSA) will be presented after an introduction in computer networks and standardisation in the area of manufacturing. The CSA can be used to implement distributed information systems. A distributed information system is characterised as a modular information system where data and applications are no longer bound to one specific site, but rely on a common infrastructure. This will be summarised by a schematic presentation of a modular CSA.

A key design issue of the system architecture is the infrastructure. Infrastructure is defined here as hardware and software (computers, networks, operating systems, applications, etc.) that is shared between different autonomous units. Benefits and disadvantages of the definition of an infrastructure are discussed in section 6.3.

The organisational control architecture may create conditions or opportunities for the implementation of a distributed database system and a distributed networking system. Section 6.4. therefore discusses the relation between information design principles and the organisational design principles of Galbraith.

Finally, the conclusions in this chapter are summarized in section 6.5. It is emphasized that distributed architectures in particular, although not necessary or sufficient, provide good conditions for the implementation of modular information systems.

## **6.2 Database architectures**

The question that needs to be answered in this section is how modules can be implemented in a database architecture. The starting-point for the conceptual design of information systems was the ANSI/SPARC three-schema architecture [Tsichritzis et al. 77]. The definition of a module is based on this architecture. It was implicitly assumed that this architecture can be extended for distributed environments. Although this may be true, there are several ways to do this. Many investigations have taken place to extend the ANSI/SPARC three-schema architecture for distributed environments. These extensions take place in three directions characterised by [Özsu et al. 91] [Sheth et al. 90]:

- the distribution of data
- the heterogeneity of the database
- the autonomy of local databases.

Distribution of data refers to the dispersion of data over multiple sites. Heterogeneity refers to the integration of different types of database systems. More important for this thesis is the third type of extension, the autonomy of the local database. When considering autonomy, a distinction can be made between distributed database systems and multidatabase systems. Two types of multidatabase systems are of interest here, namely tightly coupled federated database systems and loosely coupled federated database systems<sup>1</sup>. The following sections discuss distributed databases, loosely coupled and tightly coupled federated databases in more detail. [Sheth et al. 90] is recommended for further reading on this subject. Neither of the presented architectures however is satisfying for the implementation of a modular information system. Therefore, this section finishes with the definition of an architecture for modules, in the meaning specified in chapter 2.

### 6.2.1 Transparency

Data independence provided by the ANSI/SPARC three-schema architecture allows the user, whether it be the human end user, a programmer or a program, to consider only the data structure specified in the conceptual schema. The physical storage or the logical access of the data is transparent to the user. That is, the user does not have to know how the physical storage or the logical access is realised. The extension of the three-schema architecture requires also the replication and fragmentation of data over the sites to be transparent. These concepts are therefore discussed in this subsection.

Replication of data is often necessary in a distributed environment for performance reasons. The possibility for replication is especially important in environments with high volume data manipulation, as for example in CAD/CAM applications. These applications may require that the data is located on the node of the network where the processing application resides. Replication of data will then often be inevitable when foreign data are concerned. This replication of data however should be transparent to the user.

Transparency of fragmentation is usually discussed in terms of the relational model. It means that a database relation is divided into smaller fragments. Each fragment is treated as a separate database object. In terms of object modelling, it would mean that various objects of one class are distributed over different sites. The problem that has to be dealt with in this situation is the handling of queries and updates that were specified on entire classes but now have to be performed on

---

<sup>1</sup> the reader should be aware that there is no standard agreement on terminology in this field yet.

subdivisions. Notice that this fragmentation has a slightly different meaning than the horizontal and vertical fragmentation of classes as discussed in section 2.7.4. In the latter, fragmentation refers to the distribution of data over multiple modules where the user has the access to one module only (module fragmentation). Fragmentation in the former meaning refers to the distribution of data over different local schemas where the user has the access to multiple schemas (database fragmentation).

Fragmentation was introduced in section 2.7.4 to assign different objects of one class to different modules. Only the objects in the view domain of a module can be accessed by a user of that module. Consequently, queries and update operations on a module have to be performed solely on those objects that are specified for that module. The query and update operations will therefore be limited to the horizontal and vertical module fragments. However, objects in the view domain of a module may be distributed (fragmented) over different local schemas in a distributed database. Then, query and update operations must extend over all these database fragments.

## 6.2.2 Distributed databases

A distributed database is defined as a database that is distributed over multiple sites, while a single global conceptual schema is provided to the users [Bell et al. 92]. A standard distributed database based on the ANSI/SPARC three-schema architecture could include local conceptual schemas and local internal schemas (figure 6.1). However, these local schemas do not have to be explicitly present in any particular implementation. In practice, most of the homogeneous systems do not have local schemas and have limited data management software at the local level [Bell et al. 92].

The global conceptual schema is defined as the union of the local conceptual schemas. This schema is global because it describes the conceptual structure of the data at all the sites. Fragmentation and replication are handled in the mapping between the global conceptual schema and the local conceptual schemas. External schemas define the access of users to the database. The external schemas are defined on the global conceptual schema.

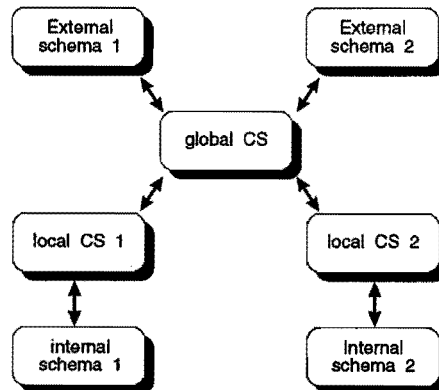


figure 6.1 a distributed database system architecture

### 6.2.3 Tightly coupled federated databases

The architecture of a tightly coupled federated database is shown in figure 6.2. Each local database system in this architecture defines an export schema, which describes the data it is willing to share with others. A 'global' conceptual schema is defined as the union of all export schemas. External schemas can be defined on either the global conceptual schema or one of the local conceptual schemas.

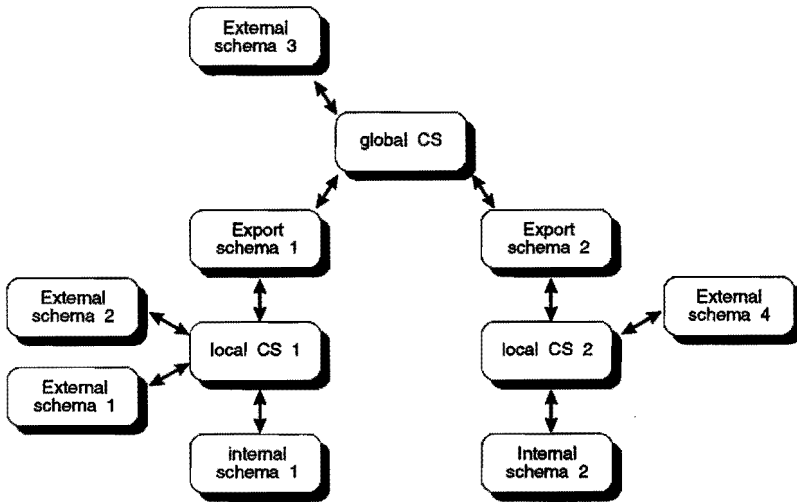


figure 6.2 tightly coupled federated database system architecture

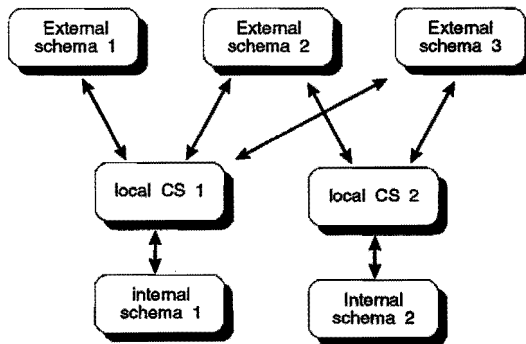


figure 6.3 loosely coupled federated database system architecture [Litwin et al. 90]

### 6.2.4 Loosely coupled federated databases

The existence of a global conceptual schema is a controversial issue [Litwin et al. 90]. A global conceptual schema presumes an organisational unit that is responsible for the central data management. In an organisation with highly autonomous units however, it may be desirable that this responsibility is delegated to these units. This is effectuated by a local responsibility for local conceptual schemas and content of the information base. The merits of a global conceptual schema as the union of the local conceptual schemas would be eliminated since there is no task left for central data management [Özsu et al. 91]. Each unit defines its own internal schema and conceptual schema (figure 6.3), which may be based on heterogeneous databases. External views are constructed on one or more conceptual schemas. Thus the responsibility of providing access to multiple databases is delegated to the mapping between the external schemas and the local conceptual schemas. This is fundamentally different from architectures that use a global conceptual schema. The responsibility in the latter architectures is taken over by the mapping between the global conceptual schema and the local conceptual schema(s).

### 6.2.5 A module architecture

Neither of the three database architectures satisfies the following requirements for the implementation of a modular information system:

- there should be no global conceptual schema
- the interfaces between modules should be defined at the conceptual schema level

In this section, an architecture of modules, based on the ANSI/SPARC three-schema architecture, is described to resolve this deficit. Furthermore, a comparison is made with the three database architectures described earlier.

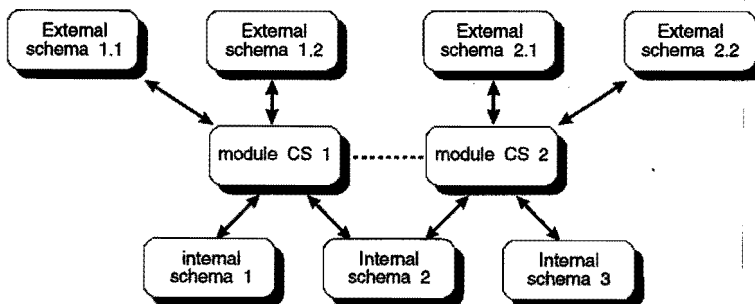


figure 6.4 a module architecture

No global conceptual schema exists in an architecture of modules (figure 6.4). The responsibility and authority for updating a module are entirely in hands of that module. The interfaces between modules are defined by the public and foreign domains in the conceptual schema of the module. This is illustrated in figure 6.4 by the dotted line between the conceptual schemas. An external schema is limited to one conceptual schema only, but may have access to other modules via the foreign domain. However, it would be allowed to design one DBMS to implement the internal schemas of multiple modules.

The characteristics of this module architecture match with a loosely coupled federated database system in the sense that there is no global conceptual schema, and that each site is autonomous in its operation. A major difference is the interfacing between the sites. The interfaces between modules in the module architecture are defined at the conceptual schema level, while the interfaces in the loosely coupled federated database system are defined by allowing an external schema to access multiple conceptual schemas.

The module architecture corresponds in particular with the loosely coupled federated database system described in [Heimbigner et al. 85]. Heimbigner *et al.* describe an architecture where each site has its own private schema. Derived from the private schema, each site has an export schema that specifies the information the site is willing to share with other sites. Finally, each site has an import schema that specifies the information a site desires to use from other sites. The import schema is derived from the export schemas of other components. Thus, this architecture implements interfaces between modules at the conceptual schema level, and matches closely to the module architecture, provided that an external schema is associated to one conceptual schema only. The difference with [Heimbigner et al. 85] is that they do not provide criteria for the modularity of the architecture. The definition of export and import schemas may still result in a complex and rigid system, while the presented method provides modularity criteria by means of the domain definitions.

The design of external schemas for only one conceptual schema is related to the tightly coupled federated database architecture. The distinction is that the export schemas in a tightly coupled federated database are combined in a single global conceptual schema upon which an external schema may be defined. As mentioned before, there exists no global conceptual schema in the module architecture. Interfaces are provided by the domain definitions in the conceptual schemas of each module.

The conclusion of this section is that the proposed module architecture is an extension of the current developments in distributed database architectures. The module architecture should serve as a guideline when designing a modular information system. In many cases however it will not be possible to implement the architecture exactly as it is specified in this section in the near future, due to lack of appropriate commercial database management systems.



### **6.2.6 Current alternatives for the implementation of a module architecture**

There are two alternatives to implement an intermediate solution: either to define a distributed database system architecture according to figure 6.1, or to define a loosely coupled federated database system architecture according to figure 6.2. In the former alternative, a global conceptual schema has to be created artificially by integrating the conceptual schemas of each module. If one of the modules will be changed or if a module will be added, then the global schema has to be updated. The advantage of this alternative is the simplicity of the solution and the broad range of commercial database systems available for it, which was also the reason to use it in the model factory. The disadvantage of this alternative is the amount of effort that it incurs in case changes have to be made to one module schema. This will essentially require a change in the global conceptual schema.

In the second alternative, the loosely coupled federated database system architecture, a local conceptual schema is defined for each individual own domain. The external schemas need to have knowledge of the local conceptual schemas they are related to. Each external schema is then related to one primary local conceptual schema and zero or more secondary local conceptual schemas. The primary local conceptual schema represents the own domain of the module where the external schema belongs to, and the secondary local conceptual schemas represent the foreign domains of the module where the external schema belongs to. The advantages of this solution have been discussed in section 6.2.4. The disadvantage however is that it requires complex data management tasks in each external schema.

## **6.3 Distributed system architectures**

The previous section focused on database architectures. In this section the implementation of distributed system architectures is addressed. After an introduction to the principles of a distributed system, a short introduction will be provided into the standardisation of computer networks and into distributed operating systems. An important development in the area of distributed system architectures is the appearance of client-server architectures (CSA). The principles of a CSA are introduced in this section, and an example is given of an implementation of a modular information system using a CSA.

### **6.3.1 From integrated to distributed information systems**

The approach to the design of complex information systems described so far is probably best characterised as the design of integrated information systems. An integrated information system can be characterised as an information system consisting of a number of independent modules that

exchange information through a dedicated network of interconnections. The emphasis lies on the design of the individual sites that are related to autonomous organisational units, rather than the design of an infrastructure of common facilities.

The goal of this section is to show that this approach to the design of complex information systems may not necessarily result in the best solution in the long term concerning the costs and non-functional requirements such as performance. It will therefore be argued that one has to consider the infrastructure when implementing complex information systems.

New technologies such as distributed operating systems, local and wide area networks, advanced communication protocols, powerful software development tools, and standardisation make it possible to reconsider the design strategy for complex information systems. Essentially, it is possible to make a shift from integrated information systems to distributed information systems. Four phases describing this shift are outlined in [Pels et al. 86].

If the implementation does not put any constraints on the conceptual design, then there will be no distinction between the conceptual design of an integrated information system and the conceptual design of a distributed information system. Both systems would provide the same user functionality. The difference between an integrated information system and a distributed information system appears in the implementation. An integrated information system will be implemented on one or more sites per module. Except for mainframe oriented systems, no two modules will share one site. On the other hand, the data and applications in a distributed information system are no longer bound to one specific site, which is the major advantage of this approach. Different applications of different modules can freely be combined on various sites. Thus, the system architecture of the information system will change considerably. The hardware components, the system software components and the relations between these components will be different in a distributed information system. Various projects and standards bodies are now involved with the definition of distributed systems, as for example OSF, X/Open, ISO/SC18, and ISA/ANSA.

The functional and non-functional requirements of an integrated information system are specified and implemented per site. The interconnections between different sites will receive special attention in case of high integration, i.e., a high degree of data exchange. These interconnections will be implemented based on the specific characteristics of the sites to be interconnected.

The emphasis in the implementation of distributed information systems will be on the infrastructure. Infrastructure is defined as hardware and software (computers, networks, operating systems, applications, etc.) that is shared between different autonomous modules. The combination of functional and non-functional requirements of different modules into requirements for system components will be possible by emphasizing the infrastructure. A system configuration will be specified taking into account the requirements of a number of modules, instead of optimizing a configuration for each individual module. This will result in a configuration that is partially shared and partially owned by the module. Capacity on shared computers will be assigned to the individual

modules. Computer networks and distributed operating systems will play an important role in the design of distributed information systems. They will implement the user requirement that there should be no difference whether they use their own local computer or capacity of shared computers (location transparency).

The following sections discuss the main concepts of a distributed information system: computer networks, distributed operating systems, and client-server architectures. The goal of these sections is to provide a short introduction into standards and developments of technologies for the implementation of distributed information systems in manufacturing. The client-server architecture is discussed in more depth because of the important consequences for the reusability of software and the design of generic modules.

### 6.3.2 Computer networks

For a user there should be no difference between applications that run on a single machine and those that run on a network. This means that the operational details of the network should be transparent for the user. It is desirable to hide even the existence of the network, if possible. This section gives an introduction in the facilities and standards in manufacturing that provide this network transparency. Technological details are omitted. [Tanenbaum 88] is recommended for further reading on this subject.

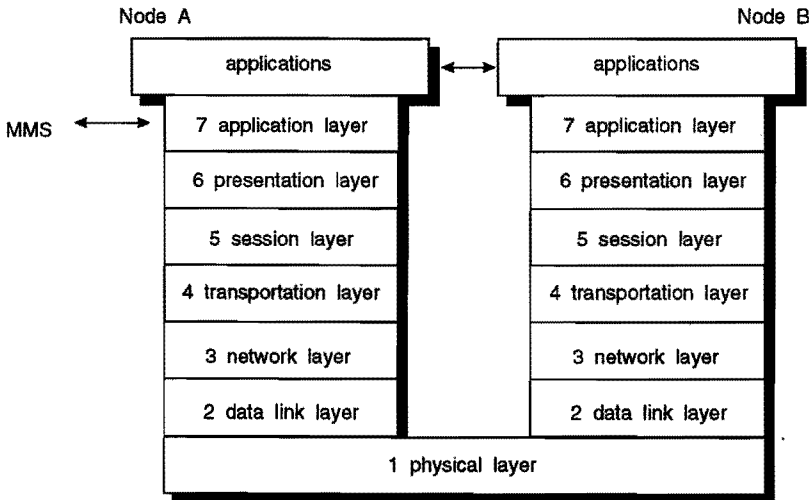


figure 6.5 ISO/OSI 7 layer architecture

## ISO/OSI architecture

To deal with the heterogeneity of equipment in a network, the International Standards Organization has developed the Open Systems Interconnection architecture, referred to as the ISO/OSI 7-layer architecture. The ISO/OSI architecture specifies seven layers of interfaces and protocols for the exchange of data between two sites in a network (figure 6.5). The application layer is of interest when discussing how applications communicate through a network. The application software is developed on top of this layer. Applications access the OSI environment by using the communication services of the application layer. Notice that the ISO/OSI architecture can be used for the realisation of integrated information systems, but is insufficient when one wants to realise fully distributed information systems [Pels et al. 86].

## MAP/MSS

The Manufacturing Automation Protocol (MAP) is based on the seven-layer OSI standard, and is designed to meet the requirements of manufacturers dealing with multi-vendor equipment on the shop floor [Jones 88]. MAP has chosen the broadband, token bus topology as the physical carrier (layer 1). An important component of MAP V3.0 is the MMS (Manufacturing Messaging Service), which is a message based protocol for communications between computers and shop floor devices also available in other OSI implementations using Ethernet, Baseband, etcetera. Currently, vendors and users of robots, PLCs, Numerical Control (NC) machines, and process controllers are writing companion standards for MMS. The companion standards state what subset of the MMS messages is to be used and which objects are predefined for each type of device. When these companion standards have been completed, the task of integrating applications with devices from multiple vendors should be simplified. This has important consequences for the costs of implementation.

MMS provides a set of services for application software, and allows the implementation of client-server relations between controllers at different or the same level of control. In case of shop floor control, it would enable production control systems to communicate with shop floor devices. MMS allows the designer to define objects that can be identified in shop floor equipment. MMS provides a set of convenient services for the application software to create interfaces to device controllers. MMS creates an additional level of portability across different device vendors, thus increasing the user's flexibility in procuring shop floor equipment [Bauer et al. 91].

### 6.3.3 Distributed operating systems

It is customary to run a DBMS as an application on top of a host operating system. However, there is significant evidence that such a mode of operation may not yield the best results in terms of functionality and performance of these systems [Özsu et al. 91]. A distributed DBMS requires additional support from the distributed operating system. Tanenbaum defines a distributed system as one that runs on a collection of machines that do not have shared memory, yet looks to its users like a single computer [Tanenbaum 92]. It should be reconsidered which features are to be provided by the DBMSs and which ones by the distributed operating system. This is especially of great importance for federated database systems that consist of heterogeneous systems. Each of the systems in a heterogeneous environment can provide different features or different implementations. Nowadays, these systems are often based on a network operating system, where each machine has a high degree of autonomy and there are few system-wide requirements. The operating system manages as a minimum the format and meaning of the messages that may potentially be exchanged [Tanenbaum 92]. However, no further research on this subject has been carried out in relation to this thesis, and for further reading on this subject is referred to [Tanenbaum 92].

### 6.3.4 Data servers

The previous chapters suggest the availability of general purpose computers that execute both application programs and data management functions. A reconsideration of the distribution of these functions is however required when taking into account the powerful workstations that are now available. The integration of workstations into a distributed environment makes a more efficient distribution of functions possible. Applications can run on workstations, called application clients, while database functions are dealt with by dedicated computers, called data servers. The same holds for other servers like mail servers, gateway servers, etc. A data server provides a complete database functionality, including persistence, recovery, and concurrency, and should therefore not be compared with the traditional file-server. Furthermore, a server can act as a client as well by calling other services. This leads to a distributed system architecture where sites are organised as specialised client/servers rather than general-purpose computers.

The need to integrate various types of workstations into a local network has resulted in a system architecture referred to as a client-server architecture. The client-server architecture dates from about twenty years ago when a host machine executed the main applications and called dedicated machines, called backend computers, to perform specific operations. The client-server architecture applies to both computer hardware and software. Client software can run on any machine and use server software on the same or another machine. The client manages for example the user interface

and submits service requests to the servers through the communication facilities of the distributed operating system. For example, the client-server architecture of the model factory applications is presented in figure 6.6. Each module in this architecture is designed as a client that can call the other modules, the database server and the factory devices where appropriate.

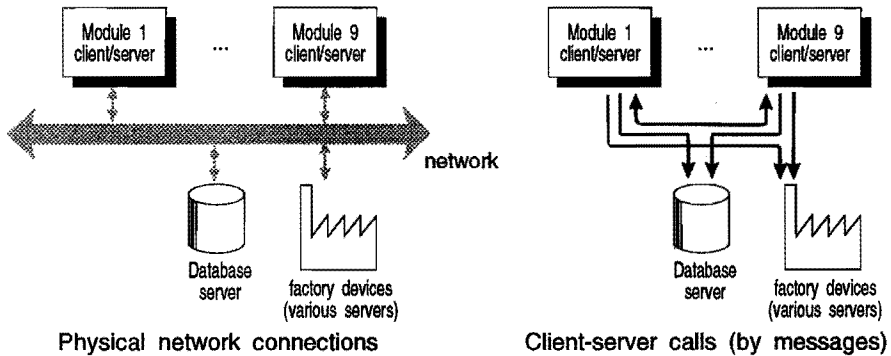
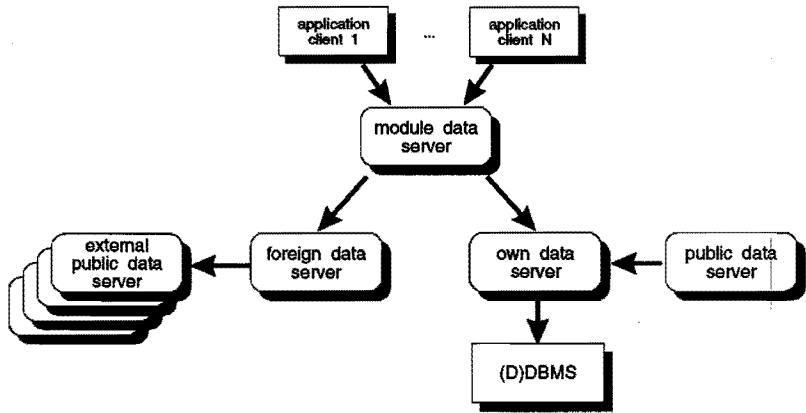


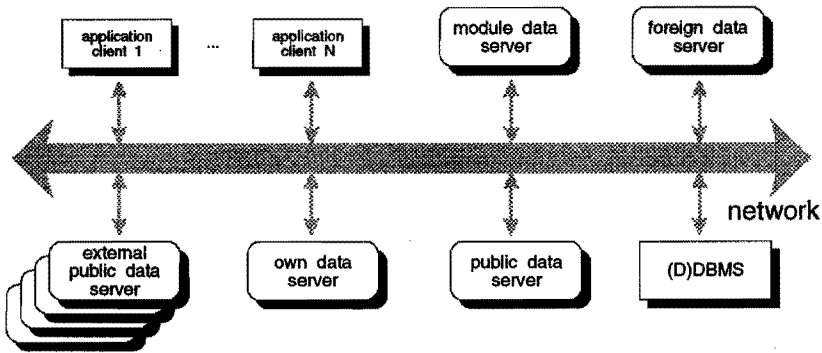
figure 6.6 a client-server architecture

### 6.3.5 Client-server architecture for a module

The client-server architecture of the model factory is reflected in figure 6.6. The right part of the figure illustrates that each module behaves as both a client and a server by calling each others applications. Furthermore, each module calls the factory devices. Also, various modules use the same database server. The principles of modular design however would allow the implementation of a module itself in a client-server architecture (figure 6.7). Each of the applications in this example of a module architecture is defined as a client operating on data of the module. The access to the data is provided by a module data server. The module data comprise own data and foreign data. Two dedicated servers are responsible for providing these data to the module data server. The own data server provides the own data according to the own domain specifications in the module, and a foreign data server provides the foreign data by calling the public data servers of other modules according to the foreign domain specification of the module. Finally, a public data server of the module can be called by other modules.



client/server calls



Physical network connections

figure 6.7 a module CSA

## 6.4 Organisational control architectures

### 6.4.1 Introduction

The complexity, coupling and cohesion criteria can also be applied to the design of organisational control architectures, although these criteria originally refer to information system design. It is however more appropriate to start with organisation design principles when discussing the relation

between information design and the design of control architectures from an organisational viewpoint. Therefore, the four organisation design strategies of Galbraith (figure 6.8) are discussed for the organisational design of control systems. A comparison is made with the modular design of information systems. Two conclusions result from this section. The first conclusion is that the organisational control architecture provides conditions and opportunities for the modular design of information systems. The second conclusion is that the method of modular design provides a technique for analysing the modularity of an organisational control architecture.

Galbraith bases his arguments for the four design strategies on general information processing principles which are elaborated below. He presents them as alternatives for organisation (re-)design when an overloading of the hierarchy occurs. Although Galbraith does not mention it explicitly, an important distinction should be made between the volume of data that is exchanged between different units and the complexity of these data in terms of data structures. The emphasis in this section lies on complexity of the data rather than volume. The reason for this is that complexity is of more importance in most situations when the integration of complex information systems is discussed, especially when emphasizing the design of *automated* information systems.

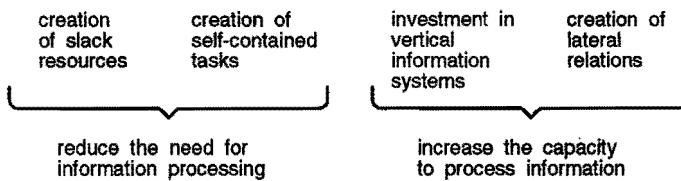


figure 6.8 organisation design strategies [Galbraith 73]

### 6.4.2 Creation of slack resources

The first strategy to reduce the need for information processing is the creation of slack resources. From an organisational viewpoint this entails the reduction of the number of exceptions that occur by reducing the required level of performance. The creation of slack resources will usually result in the weakening of integrity constraints and often also in the reduction of the number of integrity constraints. It was concluded in the previous chapter that the formal description and implementation of all relevant situations of an organisation over a longer period of time by means of conceptual modelling is very difficult, if not impossible. This holds especially for the description and implementation of integrity constraints, which is caused by the inherent complexity of these constraints. The introduction of slack may therefore facilitate the development of independent modules, especially in real-time software where the design and implementation of applications is drastically simplified by the weakening or elimination of time-related integrity constraints. In the model factory for example, physical i/o buffers are introduced between stations to reduce the need



for coordination between two modules. It would cause a complex adjustment procedure between the modules if these buffers were to be removed.

### **6.4.3 Creation of self-contained units**

The second strategy to reduce the need for information processing is the creation of self-contained units. From an organisational viewpoint this means a change from a functional task design to one in which each group has all resources necessary to execute its task. This was the reason in the model factory for the grouping of the test station and the repair station into one self-contained unit called 'test & repair'. If both stations were to operate independently, they would need a lot of mutual adjustment. In fact, their own domains should be completely public to each other. The combination of the modules resulted in a single module with a relatively high cohesion and low coupling. Hence, from a modular information system design viewpoint, the creation of self-contained units means a reduction of the coupling between modules and an increase of the cohesion of a module.

### **6.4.4 Investment in vertical information systems**

The first strategy to increase the capacity to process information is to invest in vertical information systems in order to bring the information to a central point of decision making. From an organisational viewpoint this means to collect information at all points of origin and direct it, at appropriate times, to the appropriate places in the control hierarchy. Thus, the organisation can react on unanticipated exceptions by generating adjustments to the original plans or target setting. This strategy has two negative consequences from the viewpoint of modular information system design. These consequences involve the complexity of the data referred upwards in the organisation, rather than the volume of data. First, this strategy increases the number and strength of integrity constraints between two modules at different levels in the control hierarchy. A stronger coupling between both modules will be the result. Second, this strategy requires a broadening of the view domain of the decision making module since it has to know more details of the operation of subordinate levels. This will increase the complexity of the communicated data. Also the coupling of the lower module increases, the cohesion of the higher module decreases, and the complexity of the decision making module increases as well.

Notably, this strategy is sometimes chosen to increase rationalism in decision making processes in organisations and to avoid behavioural control problems. Although these aims may be reached, this strategy does generally not contribute to more modular information systems.

### **6.4.5 Creation of lateral relations**

The second strategy to increase the capacity to process information is to create lateral relations. This strategy decentralises decisions by moving the level of decision making down to where the information exists rather than bringing it up to the points of decision. The consequence of this strategy from the viewpoint of modular information system design is the (partial) elimination of a supervising module. Information is exchanged laterally between two lower modules instead of sending information to a supervising module that forwards the information to the other lower module. As a result, the two modules can exchange information more effectively.

Lateral relations can be implemented through so called liaison officers. These liaison officers have two tasks. The first task is to 'filter' information from one module and pass it to the other. The second task is to provide cooperative activities. Examples of cooperative activities are the initiation of potentially complex series of actions involving the cooperation between modules, or the negotiation of shared data. Electronic Data Interchange (EDI) is an interesting research area in this respect. In EDI two autonomous organisations have to cooperate. The flexibility of EDI connections depends largely upon the possibility to negotiate the data to be exchanged. The question is how this negotiation can take place. This problem is addressed in literature, for example in [Heimbigner et al. 85], but not yet solved. Apart from simple cooperative activities and information filtering, it is expected that these 'automated liaison officers' will not be available soon.

Also other lateral relations than liaison officers are known in control architectures and information system design. Client-server or related principles as for example consumer-producer can also be used for the communication between autonomous modules. It should be noted that one has to consider whether the serving module is obliged to provide the service or not. The obligation to provide the service can result in dependencies between perceived autonomous modules. Further research is needed however for evaluating different types of these heterarchical relations and their impact on control and information system architectures.

## **6.5 Concluding remarks**

This chapter has considered the influence of implementation architectures on the conceptual design. In particular it has been investigated which implementation architectures avoid such influences, and therefore provide best conditions for the implementation of independent modules.

Three types of architectures for the implementation of modules were discussed in this chapter: database architectures, system architectures and organisational control architectures. This exposition illustrates that there is no straightforward way to implement modules in software and hardware. It is however illustrated that architectures of a distributed nature are more appropriate for the satisfaction of the modularity criteria defined in chapter 2.

In the area of database architectures, the requirement for decentralisation was identified about two decades ago. The first solution here was the creation of 'composite databases' that may be heterogeneous. These architectures are now known as distributed databases, and are characterised by a single global conceptual schema. The difficulty of integrating multiple existing databases in one distributed database resulted in the definition of federated database architectures [Heimbigner et al. 85]. A federated database architecture allows a collection of database systems to unite in a federation in order to share and exchange information. This federation may either be tightly or loosely coupled. It is illustrated in this chapter that a module architecture may be considered as a special case of a loosely coupled database architecture, and matches in particular with the architecture described in [Heimbigner et al. 85]. The latter however does not provide tools for realising the modularity criteria defined in chapter 2.

New technologies in the area of system architectures allow the implementation of distributed information systems. In these systems it is not obligatory that applications and data of a particular module actually reside in computer systems related to this module. Thus, independent modules can be implemented in a distributed information system with a common infrastructure. Infrastructure is defined as hardware and software (computers, networks, operating systems, applications, etc.) that is shared between different autonomous units. It is expected that infrastructures will play an important role in the future of modular design of information systems in manufacturing [Truijens et al. 90].

An important concept introduced in this chapter is the client-server architecture. The applications and the information base of a module can be implemented as client/servers on a computer network that communicate through standard messages. The access of multiple clients to the same servers can enhance reuse of software, since the servers do not have to be replicated and maintenance of the servers can be coordinated.

Finally, the correspondence between organisation design and information system design is indicated by the four organisation design strategies of Galbraith. It is argued that there is a strong relation between the organisational design of control architectures and the modular design of information systems. However, the organisational control architecture will not always provide best opportunities to design independent modules in the information system. In that case, information system designers could indicate opportunities for improvement, although it is not their responsibility to actually change the organisational control architecture. However, the method of modular design provides a technique for analysing the modularity of an organisational control architecture. The application of the method will therefore likely result in recommendations for improvements in the organisational control structure.

## Chapter 7

# Generic modules

### 7.1 Introduction

Reuse of software is necessary to increase the productivity of information system development. Van Genuchten argues that a change in the management of software development is necessary to bridge the gap between software demand and supply [Genuchten 91]. The present chapter aims at a contribution to the reuse of software from a complementary perspective, by describing a method for designing 'generic' modules. Generic modules should include properties of specific modules, and it should be relatively simple to specialise a generic module to a specific module. A generic module should include both a conceptual design and the implementation of this design in database systems and software applications.

Four steps can be identified in the reuse of components: finding components, understanding components, modifying components and composing components [Biggerstaff et al. 87]. The major benefit of generic modules will be that they can contribute to each step. The first step however depends largely on the availability of repositories and libraries. Data models can be used as a starting-point for the definition of these repositories and libraries. This thesis will however not discuss this issue of repositories and libraries in further detail.

This chapter includes a discussion of the contents of a generic module, the language for specifying a generic module, and the techniques to create generic or specific modules. Section 7.2 discusses reusability of software. The use of reference models to enhance reusability will then be discussed in section 7.3. Sections 7.4 and 7.5 describe how generic and specific modules are to be specified, and section 7.6 describes how to define and implement software for generic modules.

### 7.2 Reusability of software

The classical approach to reusability is to develop libraries of routines that implement well-defined operations. This approach however has a number of limitations [Meyer 88]:

- each operation should allow for a simple specification
- the individual operation should be clearly distinct from each other
- no complex data structures should be involved.

Meyer argues that these limitations are essentially the consequence of the classical top-down functional approach to information system development. This approach is useful for insuring that the design will meet the initial specifications, but it does not promote reusability [Meyer 88]. Components tend to be narrowly adapted to the subproblems that led to their development; they are not naturally general. It is for example fairly simple to develop a library of string manipulation routines. Each of these routines is relatively simple, and uses simple data structures. The advantage of this library for development time and cost savings is however small.

The development of software libraries for more complex software as for instance production scheduling software is much more difficult. It requires essentially the definition of a complex data model. For this specific data model it will then be possible to develop a software library. A commercial software company will therefore base its software libraries on one or a limited number of data models. A major problem would arise if the data model has to change. It would often imply the change of the entire software library as well.

Top-down design does not in itself force the components to be specific and non-reusable. Designers may always write elements that transcend particular needs. In fact, the structure obtained in a top-down design is not constrained to be a pure tree: it can be a more general directed graph with some elements shared by several refinements. However, such reusable components are not a natural result of the method [Meyer 88].

The very notion of top-down design is essentially the contrary of reusability; reusable software implies that systems are developed by combining existing components. This is the definition of bottom-up design. Furthermore, an organisation has to face the following three reuse dilemmas [Biggerstaff et al. 89]:

- generality versus payoff
- component size versus potential
- setup and costs of a components library

First, the more general the component is, the less payoff for a specific application will be. On the other hand, the more a software product is specified towards one particular application, the less applicable it will be for reuse. Second, the bigger the component, the higher the payoff will be if the component is reused. The probability that the component can be reused will however be reduced because it will become increasingly specific as it gets bigger. Third, considerable efforts and investments have to be put into a software library before it starts to pay off.

The conclusion of this section is therefore that reusable components should be developed at a sufficient high conceptual level, avoiding solutions for specific problems. A hint is given to the use of data models as the starting-point for the definition of reusable components. Also other sources propagate the use of data models as starting-point for software development [Scheer 92] [Bertrand et al. 91]. Reference models establish the first initiative in software reuse based on data models.

### 7.3 Reference models

Reference models established the first initiative in software reuse based on data models. It will be argued however that this approach does not provide an answer to all problems related to the reuse of software.

A distinction can be made between two types of reference models, normative models and non-normative models. Normative models describe the data model of a certain type of organisations or functions. Bertrand *et al.* describe for example reference models for make-to-stock, assemble-to-order, make-to-order, and engineer-to-order type production organisations [Bertrand *et al.* 90]. The reference model for MRP packages is depicted in figure 7.1. The type of organisation is analysed before these models are applied. Then the corresponding reference model is chosen. If necessary, the reference model is adapted to the specific requirements of the organisation. Based on the reference model, either a commercially available software package is selected, or an information system is built from scratch.

The non-normative reference model tries to encompass as many solutions as possible. The operation of this type of reference model means to select those parts of the model that are reflected in the specific situation. In contrast to the normative reference model, a non-normative reference model will be developed independently of the context in which it will be used.

The major benefits of both types of reference models are that they create a set of models that summarise the experience in a specific area. They also provide a technique for evaluating software packages, which improves the selection of software packages for specific situations considerably [Heij 91].

A drawback to both approaches is that they do not appear to bring much beyond the reuse of know-how and experience in the conceptual design and analysis phase since a reference model exists independently of corresponding implementations. There is a need to bridge this gap between design and implementation. The following requirements for increasing the reusability of software can be specified:

- reusable components at a sufficient high conceptual level
- a policy for reusability that ultimately produces reusable programs
- a technique to describe a complex hierarchy of reusable modules, with different levels of parametrisation.

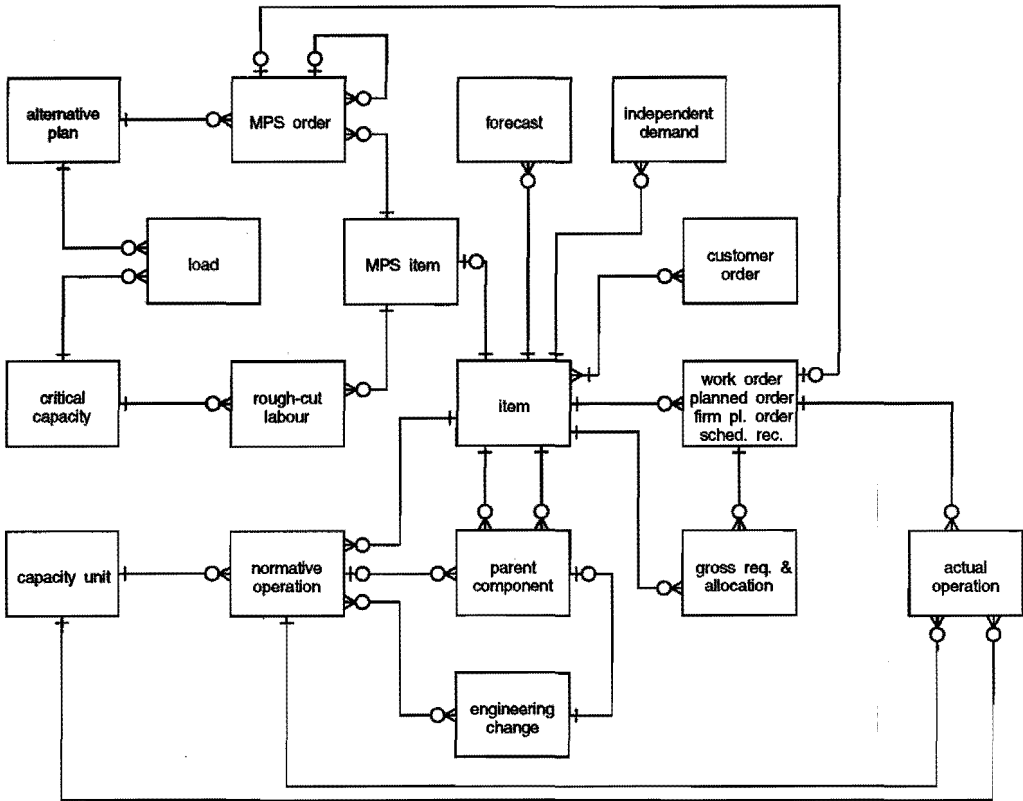


figure 7.1 reference model for MRP packages [Bertrand et al. 90]

## 7.4 Generic modelling

### 7.4.1 Requirements

The requirements for increasing the reusability of software, summarized in the previous section, imply a method that supports both the implementation and the modelling of an information system. Moreover, it should be possible to model the information system at various levels of abstraction. Modelling of information systems at various levels of abstraction is imperative to avoid the dilemma that specific solutions might not be reusable in other situations. A pragmatic solution to this problem is provided by object-orientation. One has to generalise the specifications of specific modules into a generalised (generic) module. Generalisation allows abstractions in terms of data

models. Application software can be associated to these generic modules. In this thesis, the abstraction of specific modules is referred to as generic modelling.

In the next subsection it will be discussed how object-orientation can contribute to the definition of generic modules. This will then be used in section 7.5 to describe a method for the definition of generic modules according to the method of modular design.

### **7.4.2 Object-oriented modelling**

Classes describing groups of objects are the basic building blocks in object-oriented modelling. The technique of inheritance is used as the constructor for specifying generalisation and specialisation. This is expressed in an object-oriented model by the supertype/subtype constructor. The generalisation operation is therefore restricted to a single object class.

For example, each class definition in the screenprinter module of the model factory is eligible for generalisation. The generalisation of the module would then consist of generalisations of the individual classes. It would also be possible to consider the whole screenprinter as one object to be generalised. In that case, the screenprinter would be a complex composed object where the attributes are the object class definitions in the screenprinter. It is then possible to generalise this complex screenprinter object into, for example, a generic station control object. Thus, it would be possible to describe a complex hierarchy of objects, with different levels of parametrisation. This was specified as one of the requirements for increasing the reusability of software. This approach will also be applied in the definition of a generic module. A generic module comprises a generalisation of the conceptual schema. Generalisation/specialisation has to be applied to both the structure of objects and their attributes, and the domain definitions in the module, which differs from object-orientation.

An important discussion pertains to the difficulties with defining generalised methods. Problems can occur concerning the independence of a module if one does not take care of the constraints involved with the generalised methods. In chapter 2 it is explained that applications should be developed as applications on a module. This means that the applications should comply with the domain rules and integrity constraints. This rule could be violated when methods are generalised or applied to generalised objects. In object-orientation it is difficult to validate this rule since most integrity constraints in object-orientation are described by the behaviour of the objects (by means of pre- and postconditions of the methods), rather than by the structure of the objects. Another means for generalisation of a module is to weaken the domain rules and/or the integrity constraints. The weakening of integrity constraints does not have a meaning in object-orientation since pre- and postconditions should be considered as specifications of a method, rather than a property of the object structure.

However, the weakening of integrity constraints on objects can cause errors in applications due



to unexpected values of retrieved data, as was illustrated in chapter 2. Furthermore, adding integrity constraints in specialised modules can cause errors when an intended transition is not allowed in combination with the current information base state, due to the added integrity constraint. A solution to this dilemma will be suggested in the following section.

## 7.5 Generic modules

This thesis proposes a data modelling oriented approach to generic modelling based on principles such as generalisation, specialisation, and inheritance. Generalisation in this approach is principally encapsulated in the structural aspects of the module, including integrity constraints and domain definitions. This differs from the object-oriented approach in the sense that generalisation in object-orientation is both applied to structure and behaviour.

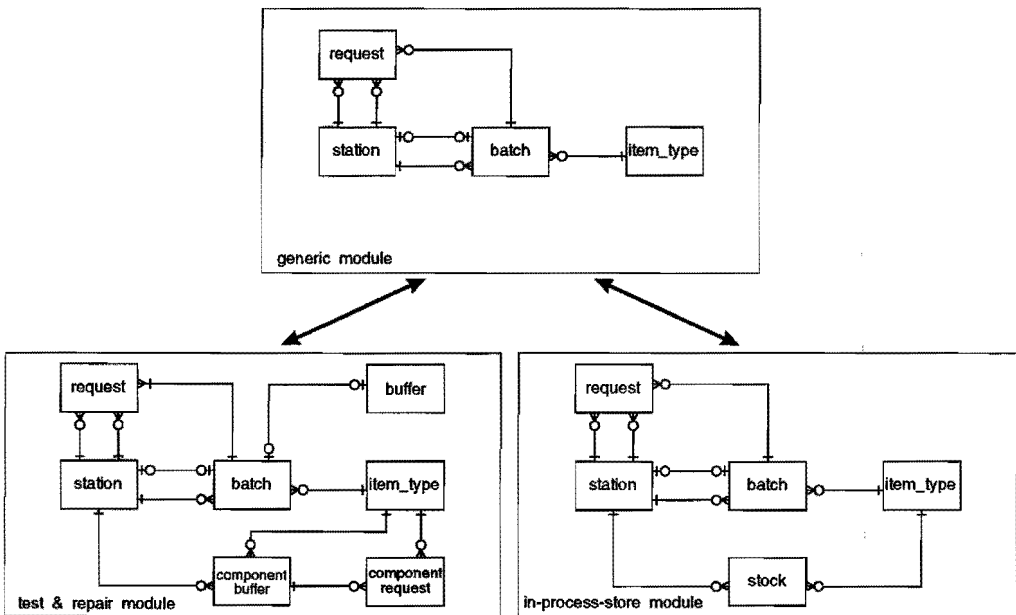


figure 7.2 abstraction of data structures

### 7.5.1 Schema abstraction

If two modules have similarities in their conceptual schema it is possible to abstract the modules into one generalised module specification. An example of such an abstraction is schematically given

in figure 7.2, where the data structure diagrams of the test & repair module and the in-process-store module are abstracted into one generic module.

The abstraction of a module concerns in general the removal of details. These details concern class specifications, attribute specifications, integrity constraints, or domain definitions. Generalisation of classes is implemented by supertyping. The classes in specialised modules inherit the specifications of classes in the generic module, as in object-oriented design. It is however also possible to reduce the number of class definitions or to weaken integrity constraints to obtain a generic module. Such a reduction of specifications will also affect the scope of the domain definitions of the module. These have to be adapted as well when the conceptual schema is generalised. Conversely, the conceptual schema will be extended when the generic module is applied in a specific situation. The domain definitions must be adapted after the conceptual schema is specialised.

### 7.5.2 Domain abstraction

The abstraction of domains is another method for abstracting modules. A manufacturing company with multiple sites might for example have the same conceptual schema at each site. Each site however will have different own and foreign domains. The abstraction of both modules will result in a generic module in which the conceptual schema is identical to the conceptual schema of each module. However, the domain definitions will be generalised.

Domain abstraction can be specified in either two ways. One way is to reduce the domain rules. Specialisation is then realised by adding domain specifications. Consider for example the following domain specifications, taken from section 2.7.3. This example expresses a generic module based on the screenprinter module for the handling of requests.

**own domain (i) =**  $\{t \in i.request \mid t.consumer.station\_name = 'screenprinter'\} \cup$   
 $\{t \in i.station \mid t.station\_name = 'screenprinter'\}$

**foreign domain(i) =**  $\{t \in i.item\_type\} \cup \{t \in i.batch\} \cup$   
 $\{t \in i.request \mid t.producer.station\_name = 'screenprinter'\} \cup$   
 $\{t \in i.station \mid t.station\_name = 'second-side' \vee$   
 $t.station\_name = 'component-placement\#1'\}$

These specifications can be generalised into:

**own domain (i) =**  $\{t \in i.request \mid t.consumer.station\_name = 'screenprinter'\}$

**foreign domain (i) =**  $\{t \in i.request \mid t.producer.station\_name = 'screenprinter'\}$

The second alternative is to parametrise the domain rules. A specific module can be derived from the generic module by filling in the parameters. An abstraction of the example above could be:

$$\begin{aligned} \text{public domain (i,x)} &= \{t \in i.\text{request} \mid t.\text{consumer.station\_name} = x\} \cup \\ &\quad \{t \in i.\text{station} \mid t.\text{station\_name} = x\} \\ \text{foreign domain (i,x,y,z)} &= \{t \in i.\text{item\_type}\} \cup \{t \in i.\text{batch}\} \cup \\ &\quad \{t \in i.\text{request} \mid t.\text{producer.station\_name} = x\} \cup \\ &\quad \{t \in i.\text{station} \mid t.\text{station\_name} = y \vee t.\text{station\_name} = z\} \end{aligned}$$

This generalisation can be used in each module that wants to implement request handling according to the principles set out in the model factory. The generic module would consist of not only the schema specifications and the domain specifications, but also the associated applications. Both ways of parametrisation will be used in the following section for the definition of the concept of generic software.

## 7.6 Generic software

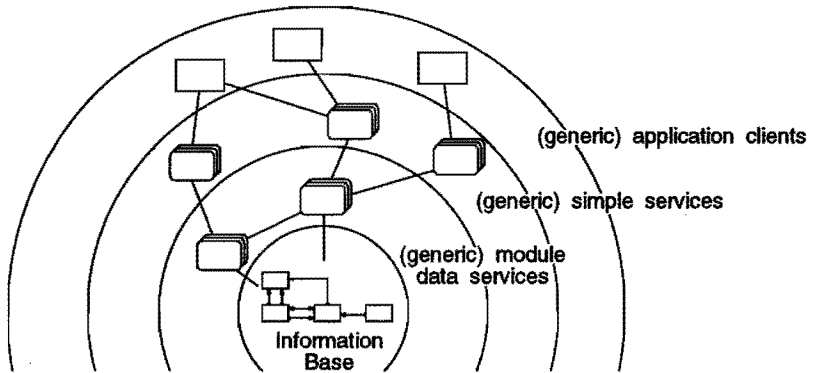
Generic modules provide the facility for specifying complex hierarchies of modules at different levels of parametrisation. They provide reusable components at a sufficient high conceptual level and genericness, not just solutions for a specific problem. However, the method should also specify reusable applications.

There are three conventional ways to develop applications from generic modules. These are: ready-made applications, parametrised applications, and the generation of applications by software generators. Ready-made applications can be used directly without modifications, but can only be developed for that part of a generic module that is less probable to change when the module is specialised into a particular module. The dilemmas here are the generality versus application payoff and application size versus reuse potential. The same dilemmas hold for parametrised applications. Parametrised applications are for instance software packages that are to be installed by setting a number of parameters. The third alternative for developing applications is to apply software generators. The disadvantage of software generators is that they are often geared towards a specific application area.

A modern, more promising way of application development for generic modules is based on the client-server principle. A generic module plus associated applications can be regarded as building blocks. These building blocks will be implemented as a number of clients/servers for applications and data management, based on the module definition. The size of each of these clients/servers can vary, and they can be layered according to basic services and application clients (figure 7.3). For example, an embedded SQL call to a foreign database for a specific object class

could be a service. Thus, the client/servers for the generic module comprise a subset of the module CSA discussed in section 6.3.5. The completion of this subset into a specific module requires the following activities:

- specialisation of the schema and domain abstractions
- setting of the parameters of parametrised clients/servers
- configuration of client applications by calling the appropriate servers
- design and implementation of clients and servers that are specific for the module



7.3 CSA of the building blocks

Not all services specified for the generic module need to be included in the specialised module. This is an important in view of the problem described in the previous section that inheritance of applications may cause problems concerning the independence of a module. To ensure the independence of a module, applications must respect the integrity constraints and domain rules of the module. When specialising a module however, integrity constraints may be added or strengthened. These integrity constraints can cause errors when they do not allow certain transitions upon the information base that used to be valid in the generic module. Removing the services that could violate the independence rule will solve this problem.

The following guidelines are given for the inheritance of services from a generic module. Services involving retrieval operations only can be always be inherited unless the view domain has been restricted concerning objects pertaining to the retrieval operations. Services involving update operations can be inherited if no applicable constraints to objects pertaining to the update operations are introduced, and if the update domains have not changed concerning the objects pertaining to the update operations.

Furthermore, new services can be designed and implemented for the specialised module using

new schema specifications. These services can be developed specifically for the particular module. Reusability of software is increased because of the possibility of redesigning only the required services.

The importance of generic software becomes even more apparent when it is combined with the technological possibilities discussed in the previous chapter. The combination of an infrastructure and generic software enables the extensive reuse of services. The infrastructure would create a software bus that allows the plug-and-play of services, and services would not be restricted to the hardware of one module in the 'integrated information system' discussed in section 6.3.1. Sharing services between two or more modules will be possible, provided that the services adhere to the domain rules of both modules.

## 7.7 Concluding remarks

The presented approach to generic modules differs considerably from the traditional view. Data models are used as the starting-point for reuse instead of focusing on libraries of routines. In this chapter it is indicated how a module definition, including the conceptual schema and domain definitions, can be used for the specification of generic modules. Successively, applications can be specified on these generic modules. Except for conventional techniques such as ready-made software, parametrised software and software generators, the modern, more promising approach based on the client-server principle is discussed. In this approach, a generic module plus applications are regarded as building blocks, consisting of a module and a number of services. The specialisation of this generic module to a specific module includes the implementation of the client applications. The complexity of the clients will be low.

The benefits of generic modules are summarised as:

- the reuse of modules at the conceptual level
- hierarchies of modules, at different levels of parametrisation
- reuse of implemented programs

An important point of attention when designing generic modules and implementing specialised modules is the validity of integrity constraints and domain rules. It may be difficult to maintain the consistency of parametrised specifications. Moreover, it may require a considerable effort to evaluate whether an application on a generic module does not violate the constraints of a specialised module. Therefore, further research is recommended in the specification and implementation of tools to ensure consistency.

---

## Chapter 8

# Discussion and conclusions

### 8.1 Summary and conclusions

#### 8.1.1 Modular conceptual design

The subject of this thesis is the modular design of information systems for shop floor control. The aim of modular design is to specify modules that can be designed, redesigned and implemented autonomously. I.e., it should be possible to design, redesign and implement a module by considering exclusively the interfaces to other modules. These interfaces should be clear, simple and easy to change.

A method for the conceptual design of a module has been introduced in chapter 2. The starting-point of this method is the conceptual schema. The interfaces between modules are defined by the public and foreign domains of a module, and it is described how the independence of a module can be determined by evaluating these domain specifications and the applicable integrity constraints.

The validity of the method has been demonstrated by means of a number of applications. The most important application is the design of a shop floor control system for a model factory. This design has been included in this thesis.

The most important learning from the model factory experiment is that the concepts of modular design provide adequate tools for reducing the complexity of the shop floor control system. The shop floor control system was split up into modules. Each of the modules was successively designed and implemented independently. There has been no need for an overall conceptual design.

It is however not possible to design and implement the information system of the model factory independently from the product structure, the primary process and the control architecture. For example, the absence of a buffer between the reflow and cleaning station and the in-process-store required that the station would only start an operation on a batch when the store was ready to receive this batch. This constraint increased the complexity of the interfaces of both modules considerably.

Succeedingly, other methods for the reduction of complexity in information system for manufacturing have been discussed. A common method is to define levels. It is argued that this

method does not provide sufficient criteria for a modular information system. In particular the criteria of modular protection, modular continuity and modular decomposability could be neglected.

### **8.1.2 Modular implementation**

Modular design is a necessary condition for a flexible information system. This condition, however, is not sufficient. Technology and organisation often either facilitate or inhibit the realisation of flexible information systems. Also complex organisational control structures require complex information system architectures, and conversely, simplified organisational control structures ease the relations between information system modules.

Three types of architectures concerning the implementation of modular information systems have been discussed: database architectures, system architectures and organisational control architectures. The importance of these architectures lies in the fact that they provide conditions for the implementation of modular information systems, as well as the fact that they determine the structure of the information system for a long period of time, which includes the future flexibility. Special attention has been given to distributed architectures since they provide more adequate conditions to satisfy the modularity criteria than for example centralistic architectures or hierarchical architectures.

### **8.1.3 Reuse of software**

There are a number of limitations to the use of routine libraries for the realisation of reusability. Reusability of software based on data models provides a good alternative. The module definition was proposed as a starting-point for the definition of reusable components. A module both allows the conceptual specification of a component and the implementation of this component. Furthermore, the method described in this thesis fulfils modularity criteria such as decomposability, composability, understandability, continuity and protection.

There are three conventional ways to develop applications for generic modules: ready-made applications, parametrised applications, and the generation of applications by software generators. A modern way of application development for generic modules is based on the client-server principle. A generic module plus applications can be regarded as building blocks. These building blocks consist of a module and a number of applications in the form of servers. The specialisation of this generic module into a specific module involves the implementation of the client applications, which is merely be the configuration of appropriate server calls.

An advantage of this approach is the possibility of sharing services between two or more

modules. A client application can call multiple services that are physically located at other modules. Furthermore, an application can both be a client and a server, thus allowing different levels of abstraction in application software. Reusability of software is increased because of the possibility of redesigning only those services needed.

#### **8.1.4 Generalisation of the results**

An interesting discussion item is the possibility of generalising the research results. In general, it is not allowed to claim the general applicability of a method based on a limited number of experiments. It should be noticed however that in the model factory as described in this thesis nine different modules were implemented. The interfaces between these modules are all distinct. The method provided independence for each of the modules. The same holds for the other experiments as well.

The main contribution of the research is however not to prove the generalisation of the results. The contribution from a scientific point of view is the description and argumentation of a method application in a complex manufacturing system. The goal of this design-oriented research is to illustrate how modular information systems are to be designed so that the main findings can be of use in other designs.

## **8.2 Recommendations for further research**

### **Typology of control architectures versus modular information systems**

The emphasis of this thesis lies on the modular design of information systems. It is indicated that there is a strong relationship between the modular design of information systems and the definition of control architectures. In particular distributed control architectures provide good conditions for the implementation of modular information systems. This thesis is however not exhaustive in describing the relation between different control architectures and the modularity of information systems. It is therefore recommended to develop a typology for control architectures, and to specify for each type its relation to modular information systems.

### **Heterarchical cooperation**

The method presented in this thesis allows the specification and implementation of federations of autonomous modules. Modules can be designed, redesigned and implemented only considering the interfaces with other modules. It will be more difficult when the interfaces themselves have to change. This requires the negotiation between different autonomous modules. Examples where these negotiations may take place can be found in Electronic Data Interchange (EDI), where two



autonomous organisations have to cooperate. An interesting research subject would be whether the negotiation in these heterarchical systems could be formalised. A particular important issue in this research would be the specification and negotiation of the pragmatics of the data. This issue is also indicated by the terms 'information resource integration' or 'schema trading' [Eliassen et al. 88].

### **Reusable modules**

Reuse of software is discussed in chapter 7. An interesting possibility for the reuse of software is the definition of reusable modules. A reusable module is defined by its conceptual schema and the applications associated to that schema. It would be an interesting research issue to implement software packages based on this principle.

### **Infrastructure and reuse**

The definition of reusable applications as services on a module is of great interest for the improvement of software reusability. A condition for the definition of these services is the availability of an appropriate infrastructure in terms of hardware and system software. The study of the infrastructural requirements for the implementation of reusable services would be a valid research subject.

---

## References

- [Aalst 92] Aalst, W.P.M. van de, *Timed coloured petri nets and their application to logistics*, Ph.D. dissertation, Eindhoven University of Technology, The Netherlands, 1992.
- [Amice 89] *Open System Architecture for CIM*, ESPRIT Consortium AMICE (eds.), Research Report Project 688, Volume 1, Springer-Verlag, Berlin, 1989.
- [Andany et al. 91] Andany, José, Michel Léonard, Carole Palisser, "Management of schema evolution in databases", *Proceedings of the 17th international conference on very large data bases*, Barcelona, September, 1991.
- [Baats 92] Baats, Erik, *Modular decomposition of the conceptual schema: applied in computer integrated manufacturing*, M.Sc. thesis, Eindhoven University of Technology, Digital Cooperative Engineering Centre, Amsterdam, June 1992.
- [Bakker 89] Bakker, J.J.A., *DFMS: architecture and implementation of a distributed control system for FMS*, Ph.D. dissertation, Delft University of Technology, 1989.
- [Bauer et al. 91] Bauer, A., R. Bowden, J. Browne, J. Duggan, G. Lyons, *Shop floor control systems: from design to implementation*, Chapman & Hall, London, 1991.
- [Bell et al. 92] Bell, David, Jane Grimson, *Distributed database systems*, Addison-Wesley Publishing Company, Wokingham, England, 1992.
- [Bemelmans 84] Bemelmans, T.M.A., *Bestuurlijke informatiesystemen en automatisering* (in Dutch), second revised edition, Stenfert Kroese bv, Leiden/Antwerpen, 1984.
- [Bertrand et al. 90] Bertrand, J.W.M., J. Wijngaard, J.C. Wortmann, *Production control systems: a structural and design-oriented approach*, Elsevier, Amsterdam, 1990.
- [Biemans 90] Biemans, Frank P.M., *Manufacturing planning and control*, Elsevier, Amsterdam, 1990.
- [Biggerstaff et al. 87] Biggerstaff, T.J., C. Richter, "Reusability framework, assessment and directions", *IEEE transaction on software*, March 1987.
- [Biggerstaff et al. 89] Biggerstaff, T.J., Perlis, A.J., "Introduction", *Software reusability, Volume 1, concepts and models*, ACM press, New York, 1989.

- [Bouzeghoub et al. 91] Bouzeghoub, Mokrane, Elisabeth Métais, "Semantic modelling of object oriented databases", *Proceedings of the 17th international conference on very large data bases*, Barcelona, September, 1991.
- [Bracchi et al. 79] Bracchi, G., A. Furtado, G. Pelagatti, "Constraint specification in evolutionary data base design", in [Schneider 79].
- [Brodie et al. 82] Brodie, M.L., E. Silva, "Active and passive component modelling: ACM/PCM", in [Olle et al. 82].
- [Bunce 88] Bunce, Peter, 'CAM-I intelligent manufacturing program: accomplishments and plans', *Proceedings of the production planning and control information exchange between CAM-I and ESPRIT projects*, Munich, Germany, May 1988.
- [Burbidge 89] Burbidge, *Production flow analysis*, Oxford, 1989.
- [CAM-I 84] CAM-I, Inc., *Conceptual information model for an advanced factory management system, Factory & Jobshop level final report R-84-FM-03*, August 1984.
- [Chen 76] Chen, P.P.S., "The entity-relationship model: towards a unified view of data", *ACM trans. database syst.*, March 1976, pp. 9-36.
- [CFT 87] *Reference for production systems*, version 1.0, also *CAM reference model*, Digital Equipment Corporation / Philips, CFT report 13/87.
- [Consilium 92] *Workstream overview*, Consilium, Inc., 1992.
- [Dilts et al. 91] Dilts, D.M., N.P. Boyd, H.H. Whorms, "The evolution of control architectures for automated manufacturing systems", *Journal of manufacturing systems*, vol.10, no.1, 1991.
- [Duggan 90] Duggan, James, *A design tool for production activity control*, Ph.D. thesis, University College Galway, Ireland, 1990.
- [Duggan et al. 91] Duggan, James, Jim Browne, "Production activity control: a practical approach to scheduling", *The international journal of flexible manufacturing systems*, no. 4, 1991.
- [Eliassen et al. 88] Eliassen, Frank, Jari Veijalainen, "A functional approach to information system interoperability", *Research into networks and distributed applications*, R. Speth (ed.), Elsevier, 1988.

- [Elmasri et al. 89] Elmasri, Ramez and Shamkant B. Navathe, *Fundamentals of database systems*, The Benjamin/Cummings Publishing Company, 1989.
- [Frissen 91] Frissen, Paul, *Flexibility in Shop Floor Management: Implementation of a Production Activity Control System*, M.Sc. thesis, Eindhoven University of Technology, Digital Cooperative Engineering Centre, Amsterdam, July 1991.
- [Galbraith 73] Galbraith, Jay, *Designing complex organizations*, Addison-Wesley, Reading Massachusetts, USA, 1973.
- [Genuchten 91] Genuchten, Michiel van, *Towards a software factory*, Ph.D. dissertation, Eindhoven University of Technology, 1991.
- [Geraerds et al. 89] Geraerds, Prof.ir. W.M.J. Geraerds, M. Igel (eds.), *Flexibiliteit in logistiek* (in Dutch), Eindhoven University of Technology, Samson/Nive, 1989.
- [Greveling 90] Greveling, N.J.W., *Informatieplanstudie: model voor strategie* (in Dutch), Ph.D. dissertation, Eindhoven University of Technology, Academic Service, 1990.
- [Griethuysen 82] Griethuysen, J.J. van, *Concepts and terminology for the conceptual schema and the information base*, ISO/TC97/SC5 N695, 1982.
- [Hakkesteeft 93] Hakkesteeft, Ruud, *Client-server computing in shop floor management*, M.Sc. thesis, Eindhoven University of Technology, Digital Cooperative Engineering Centre, Amsterdam, March 1993.
- [Heij 91] Heij, J.C.J. de, *Gegevensmodellen voor productiebesturing en voorraadbeheersing*, M.Sc. thesis, Eindhoven University of Technology, The Netherlands, 1991.
- [Heimbigner et al. 85] Heimbigner, D., D. McLeod, "A federated architecture for information management", *ACM trans. office inf. syst.*, July 1985, pp 253-278.
- [Hull 87] Hull, Richard, Roger King, "Semantic Database Modelling: Survey, Applications, and Research Issues", *ACM Computing Surveys*, Vol. 19, No. 3, September 1987.
- [Jackson 83] Jackson, Michael, *System development*, Prentice-Hall, 1983.
- [Jacques 90] Jacques, Elliott, "In praise of hierarchy", *Harvard business review*, jan-feb 1990.
- [Jones et al. 86] Jones, A.T., C.R. McLean, "A proposed hierarchical control model for automated manufacturing systems", *Journal of manufacturing systems*, vol.5, no.1, 1986.
- [Jones 88] Jones, Vincent C., *MAP/TOP networking*, McGraw-Hill, New York, 1988.

- [Kearns 90] Kearns, Chris, *Shop floor management project plan*, Digital Cooperative Engineering Centre, 1990.
- [Kent 89] Kent, William, "The leading edge of database technology", *Information systems concepts: an in-depth analysis*, E.D. Falkenberg and P. Lindgreen (eds.), Elsevier Science Publishers B.V. (North-Holland), IFIP, 1989.
- [Kim et al. 88] Kim, W., H.T. Chou, "Versions of schema for object-oriented databases", *Proceedings of the 14th international conference on very large data bases*, Los Angeles, August 1988.
- [Kim et al. 89] Kim, Won, Frederick H. Lochovsky (eds.), *Object-oriented concepts, databases, and applications*, ACM press, Addison-Wesley Publishing Company, 1989.
- [King 89] King, Roger, "My cat is object-oriented", in [Kim et al. 89].
- [Koopmans 92] Koopmans, Carlo, *Exchangeability of CIM components: implementation of an information system*, M.Sc. thesis, Eindhoven University of Technology, Digital Cooperative Engineering Centre, Amsterdam, June 1992.
- [Litwin et al. 90] Litwin, W., L. Mark, N. Roussopoulos, "Interoperability of multiple autonomous databases", *ACM computing surveys*, vol.22, no.3, 1990.
- [Martin et al. 92] Martin, James, James J. Odell, *Object-oriented analysis & design*, Prentice-Hall, Englewood Cliffs, 1992.
- [Meal 84] Meal, H.C., "Putting production decisions where they belong", *Harvard business review*, 1984.
- [Melkanoff 84] Melkanoff, Michel A., "The CIMS Database: Goals, Problems, Case Studies And Proposed Approaches Outlined", *Information Engineering*, nov. 1984.
- [Mesarovic et al. 70] Mesarovic, M.D., D. Macko, and Y. Takahara, *Theory of hierarchical, multilevel, systems*, Academic Press, New York, 1970.
- [Meyer 88] Meyer, Bertrand, *Object-oriented software construction*, Prentice Hall, 1988.
- [O'Grady et al. 88] O'Grady, Peter, Kwan H. Lee, 'An intelligent cell control system for automated manufacturing', *International journal of production research*, vol.26, no.5, 1988.
- [Olle et al. 82] Olle, T.E., H.G. Sol, C.J. Tully (eds.), *Information systems design methodologies: improving the practice*, North-Holland, Amsterdam, 1982.

- [Özsu et al. 91] Özsu, M. Tamer, Patrick Valduriez, *Principles of distributed database systems*, Prentice-Hall International, 1991.
- [Pels et al. 86] Pels, H.-J., G.J. Wegter, "Integration of databases for computer integrated manufacturing", *Proceedings of the 2nd international conference on computer applications in production and engineering*, K. Bø, L. Estensen, E.A. Warman (eds.), Copenhagen, 1986.
- [Pels 88] Pels, Henk-Jan, *Geïntegreerde informatiebanken* (in Dutch), Eindhoven University of Technology, The Netherlands, 1988.
- [Pels et al. 90] Pels, H.-J., J.C. Wortmann, "Modular design of integrated databases in production management systems", *Journal of Production Planning and Control*, 1990.
- [Pelusi 90] Pelusi, James A., 'RAMP: flexible workcell control', *IDSystems*, June, 1990.
- [Put 88] Put, Ferdi, *Introducing dynamic and temporal aspects in a conceptual (database) schema*, Ph.D. dissertation, Katholieke Universiteit Leuven, Belgium, 1988.
- [Rozendal 91] Rozendal, Rob, *JIT control on the shop floor: implementation of an information system*, M.Sc. thesis, Eindhoven University of Technology, Digital Cooperative Engineering Centre, Amsterdam, July 1991.
- [Rumbaugh et al. 91] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson, *Object-oriented modelling and design*, Prentice Hall, 1991.
- [Scheer et al. 92] Scheer, August-Wilhelm, Alexander Hars, "Extending data modelling to cover the whole enterprise", *Communications of the ACM*, vol.35, no.9, 1992.
- [Scheer 92] Scheer A.-W., *Architecture for integrated information systems*, Springer-Verlag, Berlin, 1992.
- [Schneider 79] Schneider, H.-J. (ed.), *Formal models and practical tools for information systems design*, North-Holland, Amsterdam, 1979.
- [Sheth et al. 90] Sheth, Amit P., James A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases", *ACM computing surveys*, vol.22, no.3, 1990.
- [Sol 89] Sol, E.J., "CIMphony pilot assemblagelijijn", *PT industriële automatisering*, no.1, Jan. 1989.

- [Sol 90] Sol, E.J., "Cell control on the VMEbus", *VITA congress proceedings VMEbus in factory automation*, Mainz, Germany, 1990.
- [Sol 92] Sol, E.J., "CIM: communication and information in manufacturing", *Integration in production management*, H.J. Pels and J.C. Wortmann (eds.), Elsevier, Amsterdam, 1992.
- [Spiby et al. 91] Spiby, Philip, Douglas A. Schenck, *EXPRESS language reference model*, ISO TC184/SC4/WG5 N14, 1991.
- [Stut 92] Stut jr., W.J.J., *Constructing large conceptual models with MOVIE*, Ph.D. dissertation, Leiden University, The Netherlands, 1992.
- [Tanenbaum 88] Tanenbaum, Andrew S., *Computer networks*, Prentice Hall, Englewood Cliffs, 1988.
- [Tanenbaum 92] Tanenbaum, Andrew S., *Modern operating systems*, Prentice Hall, Englewood Cliffs, 1992.
- [Tiemersma 92] Tiemersma, J.J., *Shop floor control in small batch part manufacturing*, Ph.D. dissertation, Twente University, The Netherlands, 1992.
- [Timmermans et al. 92] Timmermans, Patric, Chris Kearns, "Modular design of a shop floor control system", *Integration in production management*, H.J. Pels and J.C. Wortmann (eds.), Elsevier, Amsterdam, 1992.
- [Timmermans 92] Timmermans, Patric, *FAST - MPS*, note, Digital Cooperative Engineering Centre, Jan. 1992.
- [Timmermans et al. 93] Timmermans, Patric, Laszlo Szakal, Ronald van Riessen, "On the modular design of control systems", *Computer integrated manufacturing — proceedings of the ninth CIM-Europe annual conference*, Amsterdam, May 1993.
- [Timmermans 93] Timmermans, Patric, "Control architectures and modular information systems: a comparative experiment", *Proceedings of the international conference on advances in production management*, Athens, September 1993.
- [Truijens et al. 90] Truijens, J., A. Oosterhaven, R. Maes, H. Jägers, F. van Iersel, *Informatieinfrastructuur, een instrument voor het management*, Kluwer, 1990
- [Tsichritzis et al. 77] Tsichritzis, D., A. Klug (eds.), *The ANSI/X3/SPARC DBMS framework. report of study group on data base management systems*, AFIPS Press, Montvale NJ, 1977.

- [Veen 91] Veen, Eelco van, *Modelling product structures by generic bills-of-material*, Elsevier, Amsterdam, 1992.
- [Weber et al. 89] Weber, Detlef M., Colin L. Moodie, "From database systems to information management systems: a requirement for computer integrated manufacturing and assembly", *Proceedings of the international conference on advances in production management*, Barcelona, 1989.
- [Womack et al. 91] Womack, James P., Daniel T. Jones, Daniel Roos, *The machine that changed the world*, Harper Perennial, New York, 1991.
- [Wortmann 92] Wortmann, J.C., "Typology for one-of-a-kind production", *Conference proceedings advanced technologies in production management systems*, IFIP WG5.7 working conference, Beijing, China, 4-8 May 1992.
- [Yourdon et al. 79] Yourdon, E., L.L. Constantine, *Structured design: fundamentals of a discipline of computer program and systems design*, Prentice-Hall, Englewood Cliffs, 1979.





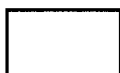
## Appendix A1

# Notational conventions

The notational conventions for diagrams in this thesis are based on [Martin et al. 92]. Notations will be used for classes, relationships, cardinalities, generalisation/specialisation, and composition.

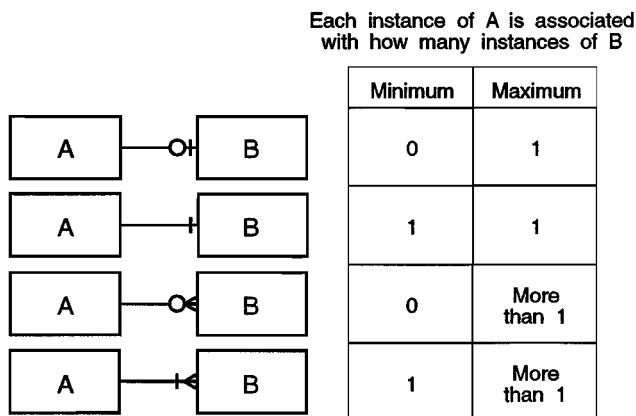
### Class

A class is drawn as a rectangle:

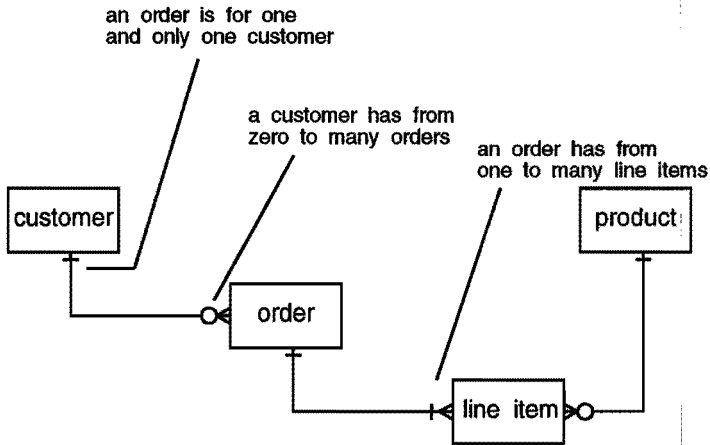


### Cardinality constraints

The term cardinality constraint refers to the restriction of how many of one item can be associated with another. A line represents an association between two classes. This line should always have cardinality symbols on both ends. The cardinality symbols express a maximum and minimum constraint. The following figure summarizes the representation of minimum and maximum cardinality constraints:

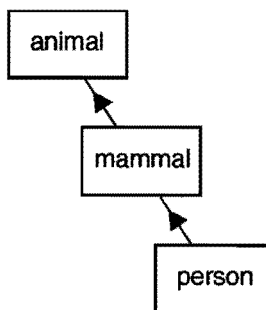


The following figure presents an example of a data structure diagram with cardinality constraints:



### Generalisation/specialisation

Classes can have more specialised types called subtypes and more general types called supertypes. Large filled arrows are sometimes used to indicate the direction of generalisation. The following figure can be read as 'Mammal is a subtype of Animal' or 'Animal is a supertype of Mammal':



## Appendix A2

# Modelling language

### Notation

Logical symbols:

$\forall$  for all (universal quantifier symbol)  
 $\exists$  exists at least one (existential quantifier symbol)  
 $\exists!$  exists exactly one (existential quantifier symbol)  
 $\wedge$  and (conjunction)  
 $\vee$  or (disjunction)  
 $\neg$  no (negation)  
 $\rightarrow$  implies (implication)  
 $=$  is equal (identify predicate) /equivalence

Extensions:

$\in$  is member (membership predicate)  
 $\subset$  is included ( inclusion predicate)  
 $\subseteq$  is included or equal  
 $\supset$  is subset  
 $\supseteq$  is subset or equal  
 $\cup$  union  
 $\cap$  intersection  
 $\emptyset$  empty set  
 $\Sigma$  summation  
 $\#$  cardinality  
 $\{x|\alpha\}$  set of x satisfying  $\alpha$

Symbols for arithmetic operations:

$+$ ,  $-$ ,  $\times$ ,  $/$ , etc.

Symbols for predicates:

$<$ ,  $\leq$ ,  $>$ ,  $\geq$

### Propositions

Propositions can take the following form:

1.  $(\forall x : A(x): P(x) )$
2.  $(\exists x : A(x): P(x) )$

where:

- $A(x)$  is a condition on  $x$
- $P(x)$  is a proposition on  $x$
- $(\forall x : A(x): P(x))$  is true if and only if for all  $x$ 's that satisfy condition  $A$  proposition  $P$  is true.
- $(\exists x : A(x): P(x))$  is true if there is at least one  $x$  that satisfies condition  $A$  for which proposition  $P$  is true.
- $(\exists! x : A(x): P(x))$  is true if there is exactly one  $x$  that satisfies condition  $A$  for which proposition  $P$  is true.

### Other notations

Other notations used are:

1.  $(\sum x : A(x): P(x))$
2.  $(\# x : A(x): R(x))$

where:

- $A(x)$  is a condition on  $x$
- $P(x)$  is a numerical function on  $x$
- $R(x)$  is a proposition on  $x$
- $S$  is a set
- $(\sum x : A(x): P(x))$  is the sum of all  $P(x)$  that satisfy condition  $A$ .
- $(\# x : A(x): P(x))$  is the number of  $x$ 's that satisfy condition  $A$  and for which proposition  $P$  is true.

### Examples

- $(\forall x : x \in \{1, 2, 3\}: x < 2.5) = 1 < 2.5 \wedge 2 < 2.5 \wedge 3 < 2.5 = \text{false}$
- $(\exists x : x \in \{1, 2, 3\}: x < 2.5) = 1 < 2.5 \vee 2 < 2.5 \vee 3 < 2.5 = \text{true}$
- $(\exists! x : x \in \{2, 3\}: x < 2.5) = (2 < 2.5 \wedge \neg(3 < 2.5)) \vee (\neg(2 < 2.5) \wedge 3 < 2.5) = \text{true}$
- $(\sum x : x \in \{1, 2, 3\}: x^2) = 1^2 + 2^2 + 3^2$
- $(\# x : x \in \{1, 2, 3\}: x < 2.5) = |\{x < 2.5 \mid x \in \{1, 2, 3\}\}| = 2$

## Appendix B

# Example: the model factory

### B.1 Introduction

The information system for the distributed control architecture is designed and implemented according to the method of modular design. An information system module is specified for each of the controllers in the control architecture (figure 3.3, section 3.3). The specification consists of a functional description of the module, a conceptual schema, and the domain definitions, and is based on the specifications in [Koopmans 92].

### B.2 Second-side controller

#### B.2.1 Functional description

The second-side controller receives requests from the screenprinter controller. The second-side controller distinguishes two types of requests: requests for batches of new empty boards and requests for batches of boards that need components on the second-side. In the latter case, the second-side controller places a request for a newly defined batch of half products at the reflow & cleaning controller.

The incoming and outgoing batches are dealt with as follows. If the second-side buffer contains a batch, then this batch is forwarded to the screenprinter station and the corresponding batch definition and request to the reflow & cleaning controller are removed from the database. If the second-side buffer does not contain a batch then the second-side controller considers the requests for a batch of empty boards. The eldest request will be fulfilled first. Fulfilment of a request involves the forwarding of a batch of empty boards from the raw material store to the screenprinter station. The reason why batches in the second-side buffer have a higher priority than batches from the raw material store is to avoid a dead-lock in the material flow of the loop that is caused by products that need a second side. A dead-lock can occur when all buffers and stations in the loop contain a batch and none of the batches can be moved to the next station or buffer. Furthermore, it should be noted that the maximum contents of the second-side buffer in the current implementation is one batch only, although physically it would be possible to contain more batches.

### B.2.2 Conceptual schema

A data structure diagram of the conceptual schema is given in figure B.1. The conceptual schema, the constraints and the domains of the second-side controller are defined as follows. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. The buffer has an optional relation to batch to indicate the batch it contains.

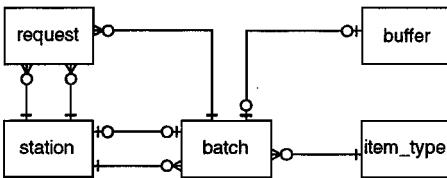


figure B.1 data structure diagram of the second-side controller

#### SCHEMA second-side

##### CLASSES

```
class station
  attributes
    station_name : string;
    produced_requests : SET OF request;
    received_requests : SET OF request;
    batch_available: {available,non-available};
    ready_to_receive : {Yes, No};
    batch_in_process : batch;
end; -- class station
```

```
class buffer
  attributes
    buffer_name : string;
    status : {full, empty};
    batch_in_buffer : batch;
end; -- class buffer
```

```
class batch
  attributes
    batch_id : integer;
    creator : station;
    item_type: item_type;
    size : {1..3};
end; -- class batch
```

```
class request
  attributes
    producer : station;
    -- the station that will produce the batch,
    i.e. receive the request
    consumer : station;
    -- the station that will consume the batch,
    i.e. create the request
    batch : batch;
    item_type: item_type;
end; -- class request
```

```
class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type
```

##### INTEGRITY CONSTRAINTS

-- for every information base state  $i$  must hold that for each request  $r$  produced by the controller there exists a request  $r'$  received by the controller for a batch of the same item type and size. for a second-sided board. Moreover,  $r'$  requires a second side ( $\text{second\_side} = \text{'Yes'}$ ), and  $r$  is not double sided ( $\text{second\_side} = \text{'No'}$ ).

C2(i) =  
 $(\forall r: r \in i.request \wedge r.consumer.station\_name =$   
 'second-side':  
 $(\exists r': r' \in i.request: r'.producer.station\_name =$   
 'second-side'  $\wedge r'.batch.item\_type.item\_type =$   
 $r.batch.item\_type.item\_type \wedge r'.batch.size =$   
 $r.batch.size \wedge r'.batch.item\_type.second\_side =$   
 'yes'  $\wedge r.batch.item\_type.second\_side =$  'no' ))

-- for every pair  $\langle i, j \rangle$  of information base states must hold that if in state  $i$  the second-side buffer is full and the screenprinter station does not contain a batch, it is not allowed that in state  $j$  the second-side buffer is still full and the screenprinter does contain a batch (which was received from the raw material stock). This constraint expresses the priority of the second-side buffer over the raw material store.

C9( $\langle i, j \rangle$ ) =  
 $(\forall b, b', s, s': b \in i.buffer \wedge b' \in j.buffer \wedge$   
 $s \in i.station \wedge s' \in j.station \wedge b = b' \wedge s = s'$   
 $\wedge b.buffer\_name =$  'second-side'  $\wedge$   
 $s.station\_name =$  'screenprinter':  
 $b.status =$  'full'  $\wedge s.batch\_in\_process =$  nil  
 $\rightarrow \neg (b'.status =$  'full'  $\wedge s'.batch\_in\_process$   
 $\neq$  nil ))

-- for every information base state  $i$  must hold that the second-side controller may not produce more than one request per batch.

C11(i) =  
 $(\forall b: b \in i.batch: (\# r: r \in i.request:$   
 $r.consumer.station\_name =$  'second-side'  $\wedge$   
 $r.batch = b) \leq 1)$

-- for every information base state  $i$  must hold that once a batch is in process in 'second-side', there may be no outstanding requests for that batch by 'second-side'

C12(i) =  
 $(\forall s, b: s \in i.station \wedge b \in i.batch \wedge$   
 $s.batch\_in\_process = b \wedge$   
 $s.station\_name =$  'second-side':  
 $\neg (\exists r: r \in i.request: r.batch = b \wedge$   
 $r.consumer = s) )$

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

C20(i) =  $(\forall s: s \in i.station: s.produced\_requests$   
 $= \{r \in i.request \mid r.consumer = s\})$

-- for every information base state  $i$  must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

C21(i) =  $(\forall s: s \in i.station: s.received\_requests =$   
 $\{r \in i.request \mid r.producer = s\})$

-- for every information base state  $i$  must hold that for each request there is one producer station

C37(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.station: p = l.producer))$

-- for every information base state  $i$  must hold that for each request there is one consumer station

C38(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.station: p = l.consumer))$

-- for every information base state  $i$  must hold that for each request there is one batch

C39(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.batch: p = l.batch))$

-- for every information base state  $i$  must hold that for each batch there is one creator station

C41(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.station: p = l.creator))$

-- for every information base state  $i$  must hold that for each batch there is one item\_type

C42(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.item\_type: p = l.item\_type))$

## DOMAIN RULES

-- the own domain of the module consists of the objects of the object types request that have 'second-side' as the name of the associated station, and the objects of the object types station and buffer with the name 'second-side'.



**own domain (i) =**

$\{t \in i.request \mid t.consumer.station\_name = 'second-side'\} \cup \{t \in i.buffer \mid t.buffer\_name = 'second-side'\} \cup \{t \in i.station \mid t.station\_name = 'second-side'\}$

-- the foreign domain of the module consists of all objects of the object types `item_type` and `batch`, the objects of the object type `request` with 'second-side' as the station name, and the objects of the object type `station` with 'screenprinter' or 'reflow&cleaning' as the station name.

**foreign domain (i) =**

$\{t \in i.item\_type\} \cup \{t \in i.batch\} \cup \{t \in i.request \mid t.producer.station\_name = 'second-side'\} \cup \{t \in i.station \mid t.station\_name = 'screenprinter' \vee t.station\_name = 'reflow\&cleaning'\}$

-- note: formally spoken is 'i.item\_type' identical to ' $\{t \in i.item\_type\}$ '. However, for reasons of uniformity and understandability, the latter notation is used (end of note).

END; -- schema second-side

### B.3 Screenprinter controller

#### B.3.1 Functional description

The screenprinter station has the most straightforward controller. The controller receives a request from the component placement 1 controller. This request is converted to a request for the second-side controller. In due time, the screenprinter station receives a batch from the raw material store or the second side buffer. The type of the products of the batch is then identified. A screen-printing mask is then selected dependent on the type of product, and a 'squeegee' operation is performed. The batch is forwarded to component placement station 1 when all operations have been performed on all products in a batch.

#### B.3.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.2. The central object classes in the diagram are `request` and `batch`. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the `item_type` it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process.

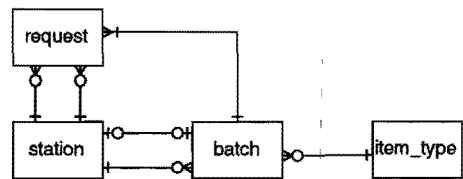


figure B.2 data structure diagram of the screenprinter controller

## SCHEMA screenprinter

## CLASSES

## class station

## attributes

```

station_name : string;
produced_requests : SET OF request;
received_requests : SET OF request;
batch_available: {available,non-available};
ready_to_receive : {Yes, No};
batch_in_process : batch;

```

end; -- class station

## class batch

## attributes

```

batch_id : integer;
creator : station;
item_type: item_type;
size : {1..3};

```

end; -- class batch

## class request

## attributes

```

producer : station;
consumer : station;
batch : batch;
item_type: item_type;

```

end; -- class request

## class item\_type

## attributes

```

item_type : string;
second_side : {yes, no};

```

end; -- class item\_type

## INTEGRITY CONSTRAINTS

-- for every outgoing request there must be an incoming request with the same item type and batch size: for every information base state  $i$  must hold that for every request that this station produces there must exist a request that it receives, and the item type and batch size of both requests should be identical.

C1(i) =

$$(\forall r: r \in i.request \wedge r.consumer.station\_name = 'screenprinter':$$

$$(\exists r': r' \in i.request: r'.producer.station\_name = 'screenprinter' \wedge r'.batch.item\_type = r.batch.item\_type \wedge r'.batch.size = r.batch.size))$$

-- for every information base state  $i$  must hold that the screenprinter may not produce more than one request per batch.

C11(i) =

$$(\forall b: b \in i.batch: (\# r: r \in i.request: r.consumer.station\_name = 'screenprinter' \wedge r.batch=b) \leq 1)$$

-- for every information base state  $i$  must hold that once a batch is in process in 'screenprinter', there may be no outstanding requests for that batch by 'screenprinter'

C12(i) =

$$(\forall s, b: s \in i.station \wedge b \in i.batch \wedge s.batch\_in\_process=b \wedge s.station\_name = 'screenprinter':$$

$$\neg (\exists r: r \in i.request: r.batch=b \wedge r.consumer=s))$$

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

C20(i) =  $(\forall s: s \in i.station: s.produced\_requests = \{r \in i.request \mid r.consumer = s\})$

-- for every information base state  $i$  must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

C21(i) =  $(\forall s: s \in i.station: s.received\_requests = \{r \in i.request \mid r.producer = s\})$

-- for every information base state  $i$  must hold that for each request there is one producer station

C37(i) =  $(\forall l: l \in i.request: (\exists! p: p \in i.station: p = l.producer))$

-- for every information base state  $i$  must hold that for each request there is one consumer station

C38(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.station: p = l.consumer))$

-- for every information base state i must hold that  
 for each request there is one batch

C39(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.batch: p = l.batch))$

-- for every information base state i must hold that  
 for each batch there is one creator station

C41(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.station: p = l.creator))$

-- for every information base state i must hold that  
 for each batch there is one item\_type

C42(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state i must hold that  
 for each batch there is at least one request

C47(i) =  $(\forall b: b \in i.batch:$   
 $(\exists r: r \in i.request: r.batch = b ))$

## DOMAIN RULES

-- the own domain of the module consists of the  
 objects of the object types request and station that  
 have 'screenprinter' as the name of the (associated)  
 station.

**own domain (i) =**  
 $\{t \in i.request \mid t.consumer.station\_name =$   
 $'screenprinter'\} \cup \{t \in i.station \mid$   
 $t.station\_name='screenprinter'\}$

-- the foreign domain of the screenprinter module  
 consists of all objects of the object types item\_type  
 and batch, the objects of the object type request with  
 'screenprinter' as the associated station name, and the  
 objects of the object type station with 'second-side'  
 or 'component-placement#1' as the station name.

**foreign domain(i) =**  
 $\{t \in i.item\_type\} \cup \{t \in i.batch\} \cup \{t \in i.request \mid$   
 $t.producer.station\_name = 'screenprinter'\} \cup$   
 $\{t \in i.station \mid t.station\_name = 'second-side' \vee$   
 $t.station\_name = 'component-placement\#1'\}$

**END;** -- schema screenprinter

## B.4 Component placement #1 controller

### B.4.1 Functional description

The component placement #1 controller receives a request from the component placement #2 controller. This request is converted to a request for the screenprinter controller. When a batch is received from the screenprinter the appropriate components are placed upon the boards. The batch is forwarded to component placement station #2 when all operations have been performed on all products in a batch. If the component placement #1 controller has a shortage of components the status of the component buffer is set to empty in order to trigger the material handler for replenishment.

### B.4.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.3. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. Finally, there is a component buffer which refers to the station and the item\_type it contains.

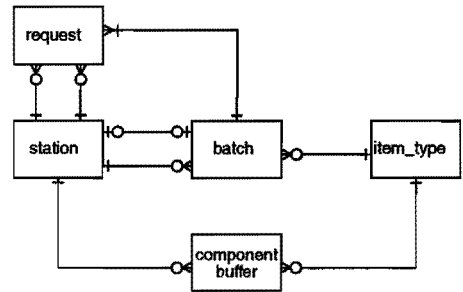


figure B.3 data structure diagram of the component placement #1 controller

### SCHEMA component-placement#1

#### CLASSES

##### class station

###### attributes

```

station_name : string;
produced_requests : SET OF request;
received_requests : SET OF request;
batch_available : {available,non-available};
ready_to_receive : {Yes, No};
batch_in_process : batch;

```

end; -- class station

##### class component\_buffer

###### attributes

```

buffer_name : string;
buffer_station : station;
status : {full, empty};
item_type : item_type;

```

end; -- class buffer

##### class batch

###### attributes

```

batch_id : integer;
creator : station;
item_type : item_type;
size : {1..3};

```

end; -- class batch

```

class request
  attributes
    producer : station;
    consumer : station;
    batch : batch;
    item_type : item_type;
end; -- class request

```

```

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type

```

### INTEGRITY CONSTRAINTS

-- for every outgoing request there must be an incoming request with the same item type and batch size: for every information base state  $i$  must hold that for every request that this station produces there must exist a request that it receives, and the item type and batch size of both requests should be identical.

```

C1(i) =
(∀ r: r ∈ i.request ∧ r.consumer.station_name =
'component-placement#1':
(∃ r': r' ∈ i.request: r'.producer.station_name =
'component-placement#1' ∧
r'.batch.item_type = r.batch.item_type ∧
r'.batch.size = r.batch.size))

```

-- for every information base state  $i$  must hold that the component placement #1 may not produce more than one request per batch.

```

C11(i) =
(∀ b: b ∈ i.batch: (# r: r ∈ i.request:
r.consumer.station_name= 'component-
placement#1' ∧ r.batch=b) ≤ 1)

```

-- for every information base state  $i$  must hold that once a batch is in process in 'component-placement#1', there may be no outstanding requests for that batch by 'component-placement#1'

```

C12(i) =
(∀ s, b: s ∈ i.station ∧ b ∈ i.batch ∧
s.batch_in_process=b ∧ s.station_name =

```

```

'component-placement#1': ¬ (∃ r: r ∈
i.request: r.batch=b ∧ r.consumer=s))

```

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

```

C20(i) = (∀ s: s ∈ i.station: s.produced_requests
= {r ∈ i.request | r.consumer = s})

```

-- for every information base state  $i$  must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

```

C21(i) = (∀ s: s ∈ i.station: s.received_requests =
{r ∈ i.request | r.producer = s})

```

-- for every information base state  $i$  must hold that for each component\_buffer there is one item\_type

```

C35(i) = (∀ l: l ∈ i.component_buffer:
(∃! p: p ∈ i.item_type: p = l.item_type))

```

-- for every information base state  $i$  must hold that for each component\_buffer there is one station

```

C36(i) = (∀ l: l ∈ i.component_buffer:
(∃! p: p ∈ i.station: p =
l.buffer_station))

```

-- for every information base state  $i$  must hold that for each request there is one producer station

```

C37(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.station: p = l.producer))

```

-- for every information base state  $i$  must hold that for each request there is one consumer station

```

C38(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.station: p = l.consumer))

```

-- for every information base state  $i$  must hold that for each request there is one batch

```

C39(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.batch: p = l.batch))

```

-- for every information base state  $i$  must hold that for each batch there is one creator station

```

C41(i) = (∀ l: l ∈ i.batch:
(∃! p: p ∈ i.station: p = l.creator))

```

-- for every information base state i must hold that  
for each batch there is one item\_type

C42(i) = ( $\forall l: l \in i.batch:$   
( $\exists! p: p \in i.item\_type: p = l.item\_type$ ))

-- for every information base state i must hold that  
for each batch there is at least one request

C47(i) = ( $\forall b: b \in i.batch:$   
( $\exists r: r \in i.request: r.batch = b$ ))

## DOMAIN RULES

-- the own domain of the module consists of the  
objects of the object types request, station and  
component\_buffer that have 'component-  
placement#1' as the name of the (associated) station.

**own domain (i) =**

{t  $\in$  i.request | t.consumer.station\_name =  
'component-placement#1'}  $\cup$  {t  $\in$  i.station |  
t.station\_name = 'component-placement#1'}  $\cup$  {t  $\in$   
i.component\_buffer | t.buffer\_station.station\_name =  
'component-placement#1'}

-- the foreign domain of the module consists of all  
objects of the object types item\_type and batch, the  
objects of the object type request with 'component-  
placement#1' as the producer station name, and the  
objects of the object type station with 'component-  
placement#2' or 'screenprinter' as the station name.

**foreign domain(i) =**

{t  $\in$  i.item\_type}  $\cup$  {t  $\in$  i.batch}  $\cup$  {t  $\in$  i.request |  
t.producer.station\_name = 'component-  
placement#1'}  $\cup$  {t  $\in$  i.station | t.station\_name =  
'component-placement#2'  $\vee$  t.station\_name =  
'screenprinter'}

**END;** -- schema component-placement#1

## B.5 Component placement #2 controller

This module is identical to component placement #1.

## B.6 Reflow & cleaning controller

### B.6.1 Functional description

The reflow & cleaning controller receives a request from the in-process store or from the second-side controller. In either case, the request is converted to a request for the component placement #2 controller. When a batch is received from component placement #2, the type of products in the batch is detected. In case the products need a second-side to be processed, the reflow & cleaning operation is performed, and the batch is forwarded to the second-side buffer. In case the products do not need a second-side, a slightly different series of actions have to be taken. This is a consequence of the fact that the in-process store does not have a physical input-buffer to store batches temporarily, and the fact that a batch should not stay in the reflow & cleaning operation area after the operation is performed. Therefore, the availability of a batch is broadcasted before an operation is performed upon the batch. The operations are delayed until the in-process store is ready to receive the batch. Then the operations are performed and the batch is forwarded.

### B.6.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.4. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process.

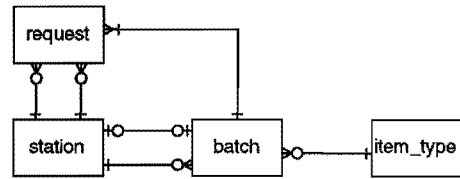


figure B.4 data structure diagram of the reflow & cleaning controller

### SCHEMA reflow&cleaning

#### CLASSES

##### class station

###### attributes

station\_name : string;  
 produced\_requests : SET OF request;  
 received\_requests : SET OF request;  
 batch\_available: {available,non-available};  
 ready\_to\_receive : {Yes, No};  
 batch\_in\_process : batch;

end; -- class station

##### class batch

###### attributes

batch\_id : integer;  
 creator : station;  
 item\_type: item\_type;  
 size : {1..3};

end; -- class batch

##### class request

###### attributes

producer : station;  
 consumer : station;  
 batch : batch;  
 item\_type : item\_type;

end; -- class request

```

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
  end; -- class item_type

```

### INTEGRITY CONSTRAINTS

-- for every outgoing request there must be an incoming request with the same item type and batch size: for every information base state  $i$  must hold that for every request that this station produces there must exist a request that it receives, and the item type and batch size of both requests should be identical.

```

C1(i) =
(∀ r: r ∈ i.request ∧ r.consumer.station_name =
'reflow&cleaning':
(∃ r': r' ∈ i.request: r'.producer.station_name =
'reflow&cleaning' ∧ r'.batch.item_type =
r.batch.item_type ∧ r'.batch.size = r.batch.size))

```

-- to avoid a deadlock in the flow of second-side products the following constraint is defined: for every information base state  $i$  must hold that if there are two or more requests produced by the reflow & cleaning controller for a batch that needs a second\_side then there may be no more than 4 requests produced in total.

```

C8(i) =
(# r: r ∈ i.request ∧ r.consumer='reflow&cleaning':
r.batch.item_type.second_side = 'yes') ≥ 2 → (#
r: r ∈ i.request ∧ r.consumer.station_name =
'reflow&cleaning': r) ≤ 4

```

-- for every information base state  $i$  must hold that the reflow & cleaning controller may not produce more than one request per batch.

```

C11(i) =
(∀ b: b ∈ i.batch: (# r: r ∈ i.request:
r.consumer.station_name='reflow&cleaning' ∧
r.batch=b) ≤ 1)

```

-- for every information base state  $i$  must hold that once a batch is in process in 'reflow&cleaning', there may be no outstanding requests for that batch by

'reflow&cleaning'

```

C12(i) =
(∀ s, b: s ∈ i.station ∧ b ∈ i.batch ∧
s.batch_in_process=b ∧ s.station_name =
'second-side': ¬ (∃ r: r ∈ i.request: r.batch=b
∧ r.consumer=s) )

```

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

```

C20(i) = (∀ s: s ∈ i.station: s.produced_requests
= {r ∈ i.request | r.consumer = s})

```

-- for every information base state  $i$  must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

```

C21(i) = (∀ s: s ∈ i.station: s.received_requests =
{r ∈ i.request | r.producer = s})

```

-- for every information base state  $i$  must hold that for each request there is one producer station

```

C37(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.station: p = l.producer))

```

-- for every information base state  $i$  must hold that for each request there is one consumer station

```

C38(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.station: p = l.consumer))

```

-- for every information base state  $i$  must hold that for each request there is one batch

```

C39(i) = (∀ l: l ∈ i.request:
(∃! p: p ∈ i.batch: p = l.batch))

```

-- for every information base state  $i$  must hold that for each batch there is one creator station

```

C41(i) = (∀ l: l ∈ i.batch:
(∃! p: p ∈ i.station: p = l.creator))

```

-- for every information base state  $i$  must hold that for each batch there is one item\_type

```

C42(i) = (∀ l: l ∈ i.batch:
(∃! p: p ∈ i.item_type: p = l.item_type))

```



-- for every information base state *i* must hold that  
for each batch there is at least one request

C47(i) =  $(\forall b: b \in i.batch:$   
 $(\exists r: r \in i.request: r.batch = b))$

#### DOMAIN RULES

-- the own domain of the module consists of the  
objects of the object types request and station that  
have 'reflow&cleaning' as the name of the  
(associated) station.

**own domain (i) =**

{t ∈ i.request | t.consumer.station\_name =  
'reflow&cleaning'} ∪ {t ∈ i.station |  
t.station\_name = 'reflow&cleaning'}

-- the foreign domain of the module consists of all  
objects of the object types item\_type and batch, the  
objects of the object type request with  
'reflow&cleaning' as the producer station name, and  
the objects of the object type station with 'in-process-  
store', 'second-side' or 'component-placement#2' as  
the station name.

**foreign domain(i) =**

{t ∈ i.item\_type} ∪ {t ∈ i.batch} ∪ {t ∈ i.request |  
t.producer.station\_name = 'reflow&cleaning'} ∪ {t  
∈ i.station | t.station\_name = 'in-process-store' ∨  
t.station\_name = 'second-side' ∨ t.station\_name =  
'component-placement#2'}

**END;** -- schema reflow&cleaning

## B.7 In-process store controller

### B.7.1 Functional description

The in-process store controller receives a request from the test & repair controller. The request will decrease the economic stock level of the in-process store. If possible, the in-process store controller will fulfil the request by taking the products from the stock and forwarding them to test & repair. A new batch is defined and a request is generated when the minimum stock level is reached. This request is then sent to the reflow & cleaning controller. The in-process store controller monitors continuously the availability of a batch in the reflow & cleaning controller. A batch received from the reflow & cleaning controller will be stored in the stocks and the stock level will be increased.

### B.7.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.5. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. Finally, the in-process store schema contains objects of the object type stock. Stocks refer to the item\_type they contain and the station they belong to.

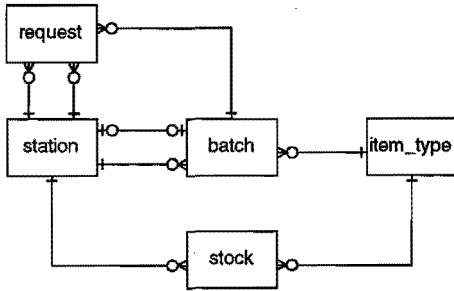


figure B.5 data structure diagram of the in-process-store controller

```

class request
  attributes
    producer : station;
    consumer : station;
    batch : batch;
    item_type : item_type;
end; -- class request
    
```

```

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type
    
```

**SCHEMA in-process-store**

**CLASSES**

```

class station
  attributes
    station_name : string;
    produced_requests : SET OF request;
    received_requests : SET OF request;
    batch_available: {available,non-available};
    ready_to_receive : {Yes, No};
    batch_in_process : batch;
end; -- class station
    
```

```

class stock
  attributes
    station : station
    item_type : item_type;
    actual_stock_level : integer;
    economic_stock_level : integer;
    maximum_stock_level : integer;
    minimum_stock_level : integer;
end; -- class stock
    
```

```

class batch
  attributes
    batch_id : integer;
    creator : station;
    item_type: item_type;
    size : {1..3};
end; -- class batch
    
```

**INTEGRITY CONSTRAINTS**

-- batches can only be received by the in-process store if the stock for that item type has sufficient space: for every information base state *i* must hold that if there is a batch available in reflow and cleaning, but the maximum stock level for that item type would be exceeded, then the in-process store is not ready to receive that batch.

```

C3(i) =
(∀ s, t, st: s,t ∈ i.station ∧ st ∈ i.stock:
  t.station_name = 'in-process-store' ∧
  s.station_name = 'reflow&cleaning' ∧
  s.batch_available = 'available' ∧
  st.station = t ∧
  st.item_type = s.batch_in_process.item_type ∧
  st.actual_stock_level + s.batch_in_process.size
  > st.maximum_stock_level →
  t.ready_to_receive = 'No')
    
```

-- for every information base state *i* must hold that the economic stock equals the actual stock plus outstanding requests minus incoming requests.

```

C10(i) =
(∀ s: s ∈ i.stock: s.economic_stock_level =
  s.actual_stock_level + (∑ r: r ∈ i.request ∧
  r.consumer.station_name = 'in-process-store':
  r.batch.size) - (∑ r: r ∈ i.request ∧
  r.producer.station_name = 'in-process-store':
  r.batch.size) )
    
```

-- for every information base state *i* must hold that the in-process-store may not produce more than one request per batch.

C11(i) =  
 $(\forall b: b \in i.\text{batch}: (\# r: r \in i.\text{request}: r.\text{consumer.station\_name} = \text{'in-process-store'} \wedge r.\text{batch} = b) \leq 1)$

-- for every information base state *i* must hold that once a batch is in process in 'in-process-store', there may be no outstanding requests for that batch by 'in-process-store'.

C12(i) =  
 $(\forall s, b: s \in i.\text{station} \wedge b \in i.\text{batch} \wedge s.\text{batch\_in\_process} = b \wedge s.\text{station\_name} = \text{'in-process-store'}: \neg (\exists r: r \in i.\text{request}: r.\text{batch} = b \wedge r.\text{consumer} = s))$

-- for every information base state *i* must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

C20(i) =  $(\forall s: s \in i.\text{station}: s.\text{produced\_requests} = \{r \in i.\text{request} \mid r.\text{consumer} = s\})$

-- for every information base state *i* must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

C21(i) =  $(\forall s: s \in i.\text{station}: s.\text{received\_requests} = \{r \in i.\text{request} \mid r.\text{producer} = s\})$

-- for every information base state *i* must hold that for each stock there is one item\_type

C33(i) =  $(\forall l: l \in i.\text{stock}: (\exists! p: p \in i.\text{item\_type}: p = l.\text{item\_type}))$

-- for every information base state *i* must hold that for each stock there is one station

C34(i) =  $(\forall l: l \in i.\text{stock}: (\exists! p: p \in i.\text{station}: p = l.\text{station}))$

-- for every information base state *i* must hold that for each request there is one producer station

C37(i) =  $(\forall l: l \in i.\text{request}: (\exists! p: p \in i.\text{station}: p = l.\text{producer}))$

-- for every information base state *i* must hold that for each request there is one consumer station

C38(i) =  $(\forall l: l \in i.\text{request}: (\exists! p: p \in i.\text{station}: p = l.\text{consumer}))$

-- for every information base state *i* must hold that for each request there is one batch

C39(i) =  $(\forall l: l \in i.\text{request}: (\exists! p: p \in i.\text{batch}: p = l.\text{batch}))$

-- for every information base state *i* must hold that for each batch there is one creator station

C41(i) =  $(\forall l: l \in i.\text{batch}: (\exists! p: p \in i.\text{station}: p = l.\text{creator}))$

-- for every information base state *i* must hold that for each batch there is one item\_type

C42(i) =  $(\forall l: l \in i.\text{batch}: (\exists! p: p \in i.\text{item\_type}: p = l.\text{item\_type}))$

## DOMAIN RULES

-- the own domain of the module consists of the objects of the object types request, station and stock that have 'in-process-store' as the name of the (associated) station.

**own domain (i) =**  
 $\{t \in i.\text{request} \mid t.\text{consumer.station\_name} = \text{'in-process-store'}\} \cup \{t \in i.\text{station} \mid t.\text{station\_name} = \text{'in-process-store'}\} \cup \{t \in i.\text{stock} \mid t.\text{station.station\_name} = \text{'in-process-store'}\}$

-- the foreign domain of the module consists of all objects of the object types item\_type and batch, the objects of the object type request with 'in-process-store' as the producer station name, and the objects of the object type station with 'test&repair' or 'reflow&cleaning' as the station name.

**foreign domain(i) =**  
 $\{t \in i.\text{item\_type}\} \cup \{t \in i.\text{batch}\} \cup \{t \in i.\text{request} \mid t.\text{producer.station\_name} = \text{'in-process-store'}\} \cup \{t \in i.\text{station} \mid t.\text{station\_name} = \text{'test&repair'} \vee t.\text{station\_name} = \text{'reflow&cleaning'}\}$

**END;** -- schema in-process-store

## B.8 Test & repair controller

### B.8.1 Functional description

Test & repair contains two stations, a test station and a repair station. A buffer is located between the repair station and the test station to store repaired batches. The test & repair controller receives a request from the final product store controller. This request is converted to a request for the in-process store controller.

The test station can accept batches from both the in-process store and the repair buffer. The batches in the repair buffer have priority to the batches coming from the in-process store. This constraint is created in order to avoid hardware deadlocks in the test-and-repair cycle.

When a batch is received from in-process store the corresponding request is deleted by the test-and-repair controller. Then the product type is determined, so the test operation can be performed. Depending on the result of the test, the batch is either forwarded to the finished product store, or sent to the repair station. Rejected batches are repaired manually at the repair station, and then put into the repair buffer. The operator of the repair station can request a tray of components at the material handler system by pushing the appropriate button.

### B.8.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.6. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that will consume the batch related to the request and to the station that will produce the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. The buffer has an optional relation to batch to

indicate the batch it contains. Furthermore, the component buffer refers to the station and the item\_type it contains, and the component request concerns one specific item type and one component buffer.

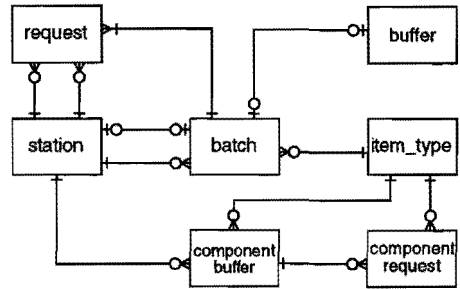


figure B.6 data structure diagram of the test & repair controller

### SCHEMA test&repair

#### CLASSES

##### class station

###### attributes

- station\_name : string;
- produced\_requests : SET OF request;
- received\_requests : SET OF request;
- batch\_available: {available,non-available};
- ready\_to\_receive : {Yes, No};
- batch\_in\_process : batch;

end; -- class station

##### class buffer

###### attributes

- buffer\_name : string;
- status : {full, empty};
- batch\_in\_buffer : batch;

end; -- class buffer

```

class component_buffer
  attributes
    buffer_name : string;
    buffer_station : station;
    status : {full, empty};
    item_type: item_type;
end; -- class component_buffer

```

```

class batch
  attributes
    batch_id : integer;
    creator : station;
    item_type: item_type;
    size : {1..3};
end; -- class batch

```

```

class request
  attributes
    producer : station;
    consumer : station;
    batch : batch;
    item_type : item_type;
end; -- class request

```

```

class component_request
  attributes
    item_type : item_type;
    buffer : component_buffer;
end; -- class request

```

```

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type

```

## INTEGRITY CONSTRAINTS

-- for every outgoing request there must be an incoming request with the same item type and batch size: for every information base state  $i$  must hold that for every request that this station produces there must exist a request that it receives, and the item type and batch size of both requests should be identical.

```

C1(i) =
(∀ r: r ∈ i.request ∧ r.consumer.station_name =
' test&repair':
(∃ r': r' ∈ i.request: r'.producer.station_name =
' final-product-store' ∧ r'.batch.item_type =
r.batch.item_type ∧ r'.batch.size = r.batch.size))

```

-- for every pair  $\langle i,j \rangle$  of information base states must hold that if in state  $i$  the repair buffer is full and there is a batch available in the in-process-store then the repair buffer has priority (in other words, it is not allowed to remove the batch from the in-process-store first)

```

C7((i,j)) =
(∀ b, s: b ∈ i.buffer ∧ b ∈ j.buffer ∧ s ∈ i.station
∧ s ∈ j.station ∧ b.buffer_name =
' repair_buffer' ∧ s.station_name = ' in-process-
store':
b.status = 'full' ∧ s.batch_available =
' available' → ¬ (b.status = 'full' ∧
s.batch_available = 'non-available' ) )

```

-- for every information base state  $i$  must hold that the test & repair controller may not produce more than one request per batch.

```

C11(i) =
(∀ b: b ∈ i.batch: (# r: r ∈ i.request:
r.consumer.station_name= ' test&repair' ∧
r.batch=b) ≤ 1)

```

-- for every information base state  $i$  must hold that once a batch is in process in 'test&repair', there may be no outstanding requests for that batch by 'test&repair'

```

C12(i) =
(∀ s, b: s ∈ i.station ∧ b ∈ i.batch ∧
s.batch_in_process=b ∧ s.station_name
' test&repair': ¬ (∃ r: r ∈ i.request: r.batch=b ∧
r.consumer=s) )

```

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

```

C20(i) = (∀ s: s ∈ i.station: s.produced_requests
= {r ∈ i.request | r.consumer = s})

```

-- for every information base state i must hold that  
the set of received\_requests of a station is identical to  
the set of requests with that station as producer

C21(i) =  $(\forall s: s \in i.station: s.received\_requests =$   
 $\{r \in i.request \mid r.producer = s\})$

-- for every information base state i must hold that  
for each component\_buffer there is one item\_type

C35(i) =  $(\forall l: l \in i.component\_buffer:$   
 $(\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state i must hold that  
for each component\_buffer there is one station

C36(i) =  $(\forall l: l \in i.component\_buffer:$   
 $(\exists! p: p \in i.station: p = l.buffer\_station))$

-- for every information base state i must hold that  
for each request there is one producer station

C37(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.station: p = l.producer))$

-- for every information base state i must hold that  
for each request there is one consumer station

C38(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.station: p = l.consumer))$

-- for every information base state i must hold that  
for each request there is one batch

C39(i) =  $(\forall l: l \in i.request:$   
 $(\exists! p: p \in i.batch: p = l.batch))$

-- for every information base state i must hold that  
for each batch there is one creator station

C41(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.station: p = l.creator))$

-- for every information base state i must hold that  
for each batch there is one item\_type

C42(i) =  $(\forall l: l \in i.batch:$   
 $(\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state i must hold that  
for each component\_request there is one item\_type

C45(i) =  $(\forall l: l \in i.component\_request:$   
 $(\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state i must hold that  
for each component\_request there is one  
component\_buffer

C46(i) =  $(\forall l: l \in i.component\_request:$   
 $(\exists! p: p \in i.component\_buffer:$   
 $p = l.buffer))$

-- for every information base state i must hold that  
for each batch there is at least one request

C47(i) =  $(\forall b: b \in i.batch:$   
 $(\exists r: r \in i.request: r.batch = b))$

## DOMAIN RULES

-- the own domain of the module consists of the  
objects of the object types request,  
component\_request, and component\_buffer that have  
'test&repair' as the name of the associated station,  
and the objects of the object types buffer and station  
with 'test&repair' as the name.

**own domain (i) =**

$\{t \in i.request \mid t.consumer.station\_name =$   
 $'test\&repair'\} \cup \{t \in i.station \mid t.station\_name =$   
 $'test\&repair'\} \cup \{t \in i.component\_request \mid$   
 $t.buffer.buffer\_station.station\_name = 'test\&repair'\}$   
 $\cup \{t \in i.buffer \mid t.buffer\_name = 'test\&repair'\} \cup$   
 $\{t \in i.component\_buffer \mid$   
 $t.buffer\_station.station\_name = 'test\&repair'\}$

-- the foreign domain of the module consists of all  
objects of the object types item\_type and batch, the  
objects of the object type request with 'test&repair'  
as the producer station name, and the objects of the  
object type station with 'final-product-store' or 'in-  
process-store' as the station name.

**foreign domain(i) =**

$\{t \in i.item\_type\} \cup \{t \in i.batch\} \cup \{t \in i.request \mid$   
 $t.producer.station\_name = 'test\&repair'\} \cup \{t \in$   
 $i.station \mid t.station\_name = 'final-product-store' \vee$   
 $t.station\_name = 'in-process-store'\}$

**END;** -- schema test&repair

## B.9 Final product store controller

### B.9.1 Functional description

The final product store controller receives production orders and related production order lines from an external source. On due date, the production orders are opened and decrease the economic stock level of the final product store. If possible, the final product store controller will fulfil the production orders by taking the products from the stock and forwarding them to the shipment area in the model factory. A new batch is defined and a request is generated when the minimum stock level is reached. This request is then sent to the test & repair controller. The final product store controller waits upon the arrival of a batch. A batch received from the test & repair controller will be stored in the stocks and the stock level will be increased.

### B.9.2 Conceptual schema

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.7. The central object classes in the diagram are request and batch. A request refers to the batch that is requested. Furthermore, a request refers to the station that consumes the batch related to the request and to the station that produces the batch related to the request. The batch refers to the item\_type it contains, and to the station that created the batch. From the station, there is an optional relation to the batch to indicate the batch-in-process. Finally, the final product store schema contains objects of the object type stock. Stocks refer to the item\_type they contain and the station they belong to.

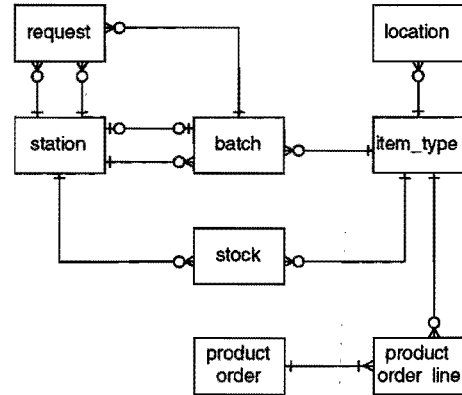


figure B.7 data structure diagram of the final product store controller

#### SCHEMA final-product-store

##### CLASSES

###### class station

###### attributes

```

station_name : string;
produced_requests : SET OF request;
received_requests : SET OF request;
batch_available: {available,non-available};
ready_to_receive : {Yes, No};
batch_in_process : batch;

```

end; -- class station

###### class location

###### attributes

```

location_id : {1..9};
current_amount : {0,1};
maximum amount : integer;
item_type : item_type;

```

end; -- class location

###### class stock

###### attributes

```

station : station
item_type : item_type;

```

```

    actual_stock_level : integer;
    economic_stock_level : integer;
    maximum_stock_level : integer;
    minimum_stock_level : integer;
end; -- class stock

class batch
  attributes
    batch_id : integer;
    creator : station;
    item_type : item_type;
    size : {1..3};
end; -- class batch

class request
  attributes
    producer : station;
    consumer : station;
    batch : batch;
    item_type : item_type;
end; -- class request

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type

class production_order
  attributes
    order_id : integer;
    order_lines : set of production_order_line;
    due_date : date;
    status : {hold, open, closed};
end; -- class production_order

class production_order_line
  attributes
    order : production_order;
    line_id : integer;
    item_type : item_type;
    status : {hold, open, closed};
    qty_requested : integer;
    qty_delivered : integer;
end; -- class production_order_line

```

## INTEGRITY CONSTRAINTS

-- for every information base state  $i$  must hold that if the status of a production order is hold, then the status of all its order lines must be hold.

C4(i) =  
 $(\forall o: o \in i.\text{production\_order}: o.\text{status} = \text{'hold'} \rightarrow$   
 $(\forall l: l \in o.\text{order\_lines}: l.\text{status} = \text{'hold'}))$

-- for every pair  $(i,j)$  of information base states must hold that if in state  $i$  the status of all order lines of a production order is closed, then the status of the production order must be closed in state  $j$ .

C5( $(i,j)$ ) =  
 $(\forall o: o \in i.\text{production\_order} \wedge o \in$   
 $j.\text{production\_order}: (\forall l: l \in i.o.\text{order\_lines}:$   
 $l.\text{status} = \text{'closed'}) \rightarrow i.o.\text{status} = \text{'closed'})$

-- for every information base state  $i$  must hold that if the status of one of the order lines of a production order is open, then the status of the production order must be open.

C6(i) =  
 $(\forall o: o \in i.\text{production\_order}: (\exists l: l \in$   
 $o.\text{order\_lines}: l.\text{status} = \text{'open'}) \rightarrow o.\text{status} =$   
 $\text{'open'})$

-- for every information base state  $i$  must hold that the economic stock equals the actual stock plus outstanding requests minus incoming requests.

C10(i) =  
 $(\forall s: s \in i.\text{stock}: s.\text{economic\_stock\_level} =$   
 $s.\text{actual\_stock\_level} + (\sum r: r \in i.\text{request} \wedge$   
 $r.\text{consumer.station\_name} = \text{'final-product-store':}$   
 $r.\text{batch.size}) - (\sum r: r \in i.\text{request} \wedge$   
 $r.\text{producer.station\_name} = \text{'final-product-store':}$   
 $r.\text{batch.size}) )$

-- for every information base state  $i$  must hold that the final-product-store may not produce more than one request per batch.

C11(i) =  
 $(\forall b: b \in i.\text{batch}: (\# r: r \in i.\text{request}:$   
 $r.\text{consumer.station\_name} = \text{'final-product-store'}$   
 $\wedge r.\text{batch} = b) \leq 1)$



-- for every information base state  $i$  must hold that once a batch is in process in 'final-product-store', there may be no outstanding requests for that batch by 'final-product-store'.

C12(i) =

$$(\forall s, b: s \in i.station \wedge b \in i.batch \wedge s.batch\_in\_process=b \wedge s.station\_name = 'final-product-store': \neg (\exists r: r \in i.request: r.batch=b \wedge r.consumer=s))$$

-- for every information base state  $i$  must hold that the set of produced\_requests of a station is identical to the set of requests with that station as consumer

C20(i) =  $(\forall s: s \in i.station: s.produced\_requests = \{r \in i.request \mid r.consumer = s\})$

-- for every information base state  $i$  must hold that the set of received\_requests of a station is identical to the set of requests with that station as producer

C21(i) =  $(\forall s: s \in i.station: s.received\_requests = \{r \in i.request \mid r.producer = s\})$

-- for every information base state  $i$  must hold that the order\_lines of a production\_order is identical to the set of order\_lines with that production\_order as the order

C22(i) =  $(\forall s: s \in i.production\_order: s.order\_lines = \{r \in i.order\_line \mid r.order = s\})$

-- for every information base state  $i$  must hold that for each production order there is at least one production orderline.

C30(i) =  $(\forall p: p \in i.production\_order: (\# l: l \in p.order\_lines: l) \geq 1)$

-- for every information base state  $i$  must hold that for each production orderline there is one production order.

C31(i) =  $(\forall l: l \in i.production\_order\_line: (\exists! p: p \in i.production\_order: p = l.production\_order))$

-- for every information base state  $i$  must hold that for each production orderline there is one item\_type.

C32(i) =  $(\forall l: l \in i.production\_order\_line: (\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state  $i$  must hold that for each stock there is one item\_type

C33(i) =  $(\forall l: l \in i.stock: (\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state  $i$  must hold that for each stock there is one station

C34(i) =  $(\forall l: l \in i.stock: (\exists! p: p \in i.station: p = l.station))$

-- for every information base state  $i$  must hold that for each request there is one producer station

C37(i) =  $(\forall l: l \in i.request: (\exists! p: p \in i.station: p = l.producer))$

-- for every information base state  $i$  must hold that for each request there is one consumer station

C38(i) =  $(\forall l: l \in i.request: (\exists! p: p \in i.station: p = l.consumer))$

-- for every information base state  $i$  must hold that for each request there is one batch

C39(i) =  $(\forall l: l \in i.request: (\exists! p: p \in i.batch: p = l.batch))$

-- for every information base state  $i$  must hold that for each batch there is one creator station

C41(i) =  $(\forall l: l \in i.batch: (\exists! p: p \in i.station: p = l.creator))$

-- for every information base state  $i$  must hold that for each batch there is one item\_type

C42(i) =  $(\forall l: l \in i.batch: (\exists! p: p \in i.item\_type: p = l.item\_type))$

-- for every information base state  $i$  must hold that for each location there is one item\_type

C44(i) =  $(\forall l: l \in i.location: (\exists! p: p \in i.item\_type: p = l.item\_type))$

**DOMAIN RULES**

-- the own domain of the module consists of the objects of the object types production\_order, production\_order\_line and location, and all objects of the object types request and station that have 'final-product-store' as the name of the (associated) station.

**own domain (i) =**  
 $\{t \in i.production\_order\} \cup$   
 $\{t \in i.production\_order\_line\} \cup \{t \in i.location\} \cup$   
 $\{t \in i.request \mid t.consumer.station\_name = 'final-$   
 $product-store'\} \cup \{t \in i.station \mid t.station\_name=$   
 $'final-product-store'\}$

-- the foreign domain of the module consists of all objects of the object types item\_type and batch, and the objects of the object type station with 'test&repair' as the station name.

**foreign domain(i) =**  $\{t \in i.item\_type\} \cup \{t \in$   
 $i.batch\} \cup \{t \in i.station \mid t.station\_name=$   
 $'test\&repair'\}$

END; -- schema final-product-store

**B.10 Material handler controller**

**B.10.1 Functional description**

The material handler controller monitors continuously the component buffers of the component placement stations. Simultaneously, the controller can receive component requests from the test & repair controller. A move order for a component tray is created if either a component buffer is empty or if a component request is placed. Each move order receives a priority number, and orders are executed according their priority.

**B.10.2 Conceptual schema**

The conceptual schema, the constraints and the domains are defined as follows. A data structure diagram of the conceptual schema is given in figure B.8. A move order refers to one component buffer which contains one item type. Furthermore, component requests are made for one item type and one component buffer.

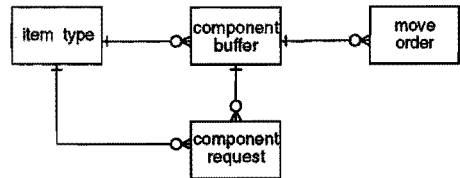


figure B.8 data structure diagram of the material handler controller

**SCHEMA material-handler**

**CLASSES**

```

class component_buffer
  attributes
    buffer_name : string;
    status : {full, empty};
    item_type: item_type;
end; -- class component_buffer
  
```

```

class move_order
  attributes
    order_id : integer;
    buffer : component_buffer;
    priority : integer;
end; -- class move_order

```

```

class component_request
  attributes
    item_type : item_type;
    buffer : component_buffer;
end; -- class request

```

```

class item_type
  attributes
    item_type : string;
    second_side : {yes, no};
end; -- class item_type

```

### INTEGRITY CONSTRAINTS

```

-- for every information base state i must hold that
for each component_buffer there is one item_type
C35(i) = (∀ l: l ∈ i.component_buffer:
          (∃! p: p ∈ i.item_type: p = l.item_type))

```

```

-- for every information base state i must hold that
for each move_order there is one component_buffer
C40(i) = (∀ l: l ∈ i.move_order:
          (∃! p: p ∈ i.component_buffer: p =
          l.buffer))

```

```

-- for every information base state i must hold that
for each component_request there is one item_type
C45(i) = (∀ l: l ∈ i.component_request:
          (∃! p: p ∈ i.item_type: p = l.item_type))

```

```

-- for every information base state i must hold that
for each component_request there is one
component_buffer
C46(i) = (∀ l: l ∈ i.component_request:
          (∃! p: p ∈ i.component_buffer: p =
          l.buffer))

```

### DOMAIN RULES

```

-- the own domain of the module consists of the
objects of the object types move_order

```

```

own domain (i) = {t ∈ i.move_order}

```

```

-- the foreign domain of the module consists of all
objects of the object types item_type and
component_buffer

```

```

foreign domain(i) =
{t ∈ i.item_type} ∪ {t ∈ i.component_request} ∪
{t ∈ i.component_buffer}

```

```

END; -- schema material-handler

```

## Appendix C

# Glossary

*Aggregation*: a special form of association, between a whole and its parts, in which the whole is composed of the parts [Rumbaugh et al. 91].

*Applicable constraint*: a constraint is applicable if there is a possibility that the constraint may be violated by an update operation that is defined for that module.

*Association*: a relationship among instances of two or more classes describing a group of links with common structure and common semantics [Rumbaugh et al. 91].

*Cartesian product*: given the collection of (not necessarily distinct) sets  $D_1, D_2, \dots, D_n$ , the Cartesian product  $D_1 \times D_2 \times \dots \times D_n$  is the set of all possible n-tuples  $\langle d_1, d_2, \dots, d_n \rangle$ , such that  $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$  [Put 88].

*Class*: a description of a group of objects with similar properties, common behaviour, common relationships, and common semantics [Rumbaugh et al. 91].

*Classification*: a form of abstraction in which a collection of things is considered as a higher level construct called type, class or set [Put 88]

*Client*: a system component that calls upon the services provided by another component. The component providing the service is the supplier [Rumbaugh et al. 91].

*Cohesion*: the insensitivity of a module for structural changes in its environment. Cohesion should be maximized to create more autonomous units. A measure for cohesion is defined as the ratio between the number of own specifications and the number of visible specification of a module [Pels 88].

*Conceptual schema*: a definition of the total information contents of the information system, both structure and semantics [Put 88].

*Conceptualisation principle*: a conceptual schema should only include conceptually relevant aspects, both static and dynamic, of the universe of discourse, thus excluding all aspects of (internal and external) data representation, physical data organisation and access as well as all aspects of particular external user representations such as message formats, data structures, etcetera [Griethuysen 82].

*Constraint*: a boolean function about some condition or relationship that must be maintained as true.

*Coupling*: a measure for the knowledge other modules have about that module. Coupling should be minimized to allow changes in a module to take place without interfering other modules. A measure for coupling of a module A is defined as the average of the number of other modules for which each own specification (object class, attribute or constraint) of A is visible.

*Database*: the concrete physical representation of the information contained in the data processing system to describe the universe of discourse.

*Data independence*: the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.

*Derived attribute*: an attribute that is computed from other attributes [Rumbaugh et al. 91].

*Distributed database*: a database which is distributed over multiple sites, while a single global conceptual schema is provided to the users [Bell et al. 92].

*Dynamic constraint*: a boolean function on the transition from the current information base state to a new information base state.

*External schema*: a description of the database view of one group of database users. Each view typically describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group [Elmasri et al. 89].

*Federated database system*: a collection of cooperating but autonomous component database systems [Sheth et al. 90].

*Flexibility*: the ability to adjust the primary process according to new requirements of the environment.

*Foreign domain*: the set of objects (in a specific instant or period of time) for which a module has retrieval authorization but no update authorization.

*Generalisation*: the relationship between a class and one or more refined or specialised versions of it [Rumbaugh et al. 91].

*Horizontal fragmentation*: (1) horizontal module fragmentation. Allotting different objects of one class to different domains or modules, (2) horizontal database fragmentation. distribution of different objects of one class over different local schemas.

*Information base*: a description of the actual objects consistent with the conceptual schema that are perceived to exist in the universe of discourse in a specific instant or period of time and their actual states of affairs.

*Information hiding*: all information about a module should be private to the module unless it is specifically declared public [Meyer 88].

*Infrastructure*: hardware and software (computers, networks, operating systems, applications, etc.) that is shared between different autonomous units.

*Inheritance*: an object oriented mechanism that permits classes to share attributes and operations based on a relationship, usually generalisation [Rumbaugh et al. 91].

*Instance*: an object described by a class [Rumbaugh et al. 91].

*Integrity constraint*: see Constraint.

*Internal schema*: a description of the physical storage structure of the database. An internal schema uses a physical data model and describes the complete details of data storage and access paths for the database [Elmasri et al. 89].

*Linguistic modular units*: this principle refers to the requirement that the language used to specify the design must support the view of modularity. I.e., the grammar of the language should support the notion of modularity.

*Logical data independence*: the capacity to change the conceptual schema without having to change external schemas or application programs.

*Memory independence*: corresponds to the possibility of recalling the past without being involved in its actual representation in the current state of the system [Put 88].

*Method*: the implementation of an operation for a specific class [Rumbaugh et al. 91].

*Modular composability*: a method should favour the production of software elements that may freely be combined with each other to produce new systems, possibly in an environment different from the one in which they were initially developed.

*Modular continuity*: a method should make it possible that a small change in a problem specification results in a change of just one module. Such changes should not affect the architecture of the system, that is to say the relations between modules.

*Modular decomposability*: a method should help in the decomposition of a new problem into several subproblems, whose solution may be pursued separately. In addition, the method should also help the decomposition of an existing system into separate components.

*Modular protection*: a method should yield architectures in which the effect of an abnormal condition occurring at run-time in a module will remain confined to this module, or at least will propagate to a few neighbouring modules only.

*Modular understandability*: a method should help to produce modules that can be separately understood by a human reader. At worst, the reader will have to look at a few neighbouring modules.

*Module*: a part of an information base that can be used separately [Pels 88]. Sometimes a weaker definition is used: a coherent subset of a system containing a tightly bound group of classes and their relationships [Rumbaugh et al. 91].

*Object*: a concept, abstraction, or thing with crisp boundaries and meanings for the problem at hand; an instance of a class [Rumbaugh et al. 91].

*Object-orientation*: a software development strategy that organises software as a collection of objects that contain both data structure and behaviour [Rumbaugh et al. 91].

*Own domain*: the set of objects (in a specific instant or period of time) for which a module has update authorization.

*Physical data independence*: the capacity to change the internal schema without having to change the conceptual (or external) schemas.

*Private domain*: the set of objects (in a specific instant or period of time) from the own domain that are not included in the view domain of any other module.

*Public domain*: the set of objects (in a specific instant or period of time) from the own domain of a module that are visible for one or more other modules.

*Specialisation*: the creation of subclasses from a superclass by refining the superclass [Rumbaugh et al. 91].

*Static constraint*: a boolean function that must be satisfied in every information base state.

*Temporal constraint*: generalisation of a dynamic constraint referring to information base states different from the current (or new) state; checking a temporal constraint requires information about one or more states in the past (or future).

*Transferability*: the property of an application that the application can be relocated to another module or a modified module. An application is transferable to a modified module if it satisfies all constraints of the modified module. An application is transferable to another module if it satisfies all constraints of that module.

*Universe of discourse*: that portion of the real world or postulated world that is being modelled [Griethuysen 82].

*User* (in a strict sense): anybody or anything that issues commands and messages to the information system and receives messages from the information system [Griethuysen 82].

*Vertical fragmentation*: (1) vertical module fragmentation. Allotting different attributes of one object to different domains or modules. Vertical module fragmentation can be avoided by the definition of subtypes. (2) vertical database fragmentation. distribution of different attributes of one object over different local schemas.

*View domain*: the set of objects (in a specific instant or period of time) that are visible for a module.

*Visible constraint*: a constraint is visible in a module if and only if a user of that module can determine the logic value of the constraint.

*100% principle*: all relevant general static and dynamic aspects, i.e. all rules, laws, etc., of the universe of discourse should be described in the conceptual schema; the information system cannot be held responsible for not meeting those described elsewhere, including those in application programs [Griethuysen 82].

## Summary

In the past, many information systems were built as complex integrated systems on centrally located computers. These systems were often built with old technologies that exhibit the characteristics of a monolith, i.e., a system that is increasingly difficult to enhance or modify.

New Information Technologies make it possible to design information systems as federations of more or less autonomous modules. According to this approach, an information system is extended by adding modules, and modified through step by step changes in the individual modules.

The research question of this thesis is to describe a method for the modular design of information systems for Computer Integrated Manufacturing (CIM). The focus of the research has been on the modular design and implementation of shop floor control systems. These systems contribute considerably to goals of industrial companies, such as greater flexibility and responsiveness, better use of resources, a reduction in inventory levels and faster delivery of customer orders in order to be competitive. However, in the research area of manufacturing relatively little attention is being paid to shop floor control systems.

Furthermore, the proposed method is validated, and extended where appropriate. Two areas of extension are the design of CIM architectures and the reuse of software. The research question concerning the extension towards CIM architectures is: what is the relation between modular design of information systems at the conceptual level and the modular implementation of information systems in different CIM architectures. The research question concerning the extension towards the reuse of software is: how can the method of modular design contribute to the reuse of software.

The characteristics of the proposed method differ in a number of ways from traditional methods for information systems design for CIM. Most notably, the method is based on the following starting-points, which are often lacking in traditional, waterfall based methods:

- information system design is an evolutionary process
- information systems have to be built from components
- information system requirements are continuously changing

The method described in this thesis is based on the design of the conceptual schema of the information base. An information system is defined as a collection of interfaced modules. Each module refers to a part of the information base that can be used separately, and is therefore specified by its own conceptual schema. Interfaces between modules are defined in terms of domains, which describe the update and retrieval authorisation of a module with respect to the information base. These concepts make it possible to validate the modularity of a design in terms of criteria such as



decomposability, composability, understandability, continuity, and protection.

In this thesis, it is shown that it is not sufficient to consider only the modular design of the information system if one wants to implement a modular information system. Therefore, three types of implementation architectures are discussed, viz. database architectures, system architectures, and organisational control architectures. It is indicated that distributed architectures provide best conditions for the implementation of independent modules in such a way that the modularity criteria are satisfied.

The proposed method has been validated in a laboratory that consists of a scale model factory of a Printed Circuit Board production line. In this laboratory it was possible to implement a modular information systems in a realistic environment. This experiment has been used as a vehicle for research, and is described extensively in this thesis.

Finally, further research is recommended in the following areas. First, it is worthwhile to study the relation between control architectures and modular information systems in further detail. This is likely to result in a more adequate application of Information Technology in manufacturing. Second, the formalisation of negotiation procedures between autonomous modules and the pragmatics of data are important research issues. Third, the reuse of modules and the design of standard software requires additional study. Fourth, more research is needed of the infrastructural requirements for the implementation of reusable software.

## Samenvatting

Tot voor kort werden informatiesystemen vaak ontworpen en geïmplementeerd als complexe, geïntegreerde systemen op centrale computersystemen. Deze systemen zijn vaak gebouwd met behulp van oude technologieën en hebben de karakteristieken van een monoliet. Dat wil zeggen, een systeem dat in toenemende mate moeilijk te onderhouden of uit te breiden is.

Nieuwe informatietechnologieën maken het mogelijk om informatiesystemen te ontwerpen als federaties van min of meer autonome modulen. Een informatiesysteem wordt dan uitgebreid door nieuwe modulen toe te voegen en aangepast door middel van stapsgewijze wijzigingen in de individuele modulen.

De onderzoeksvraag van dit proefschrift bestaat uit het beschrijven van een methode voor het modulaair ontwerpen van informatiesystemen voor Computer Integrated Manufacturing (CIM). Het accent van het onderzoek heeft daarbij gelegen op het ontwerpen en implementeren van informatiesystemen voor shop floor control. Deze systemen vormen een belangrijke bijdrage voor het realiseren van doelstellingen als het vergroten van flexibiliteit, het beter benutten van produktiemiddelen, het reduceren van voorraden en het verkorten van levertijden. Tot nu toe heeft er echter relatief weinig onderzoek plaatsgevonden naar het ontwerpen van informatiesystemen voor shop floor control.

Vervolgens is de methode gevalideerd en op een aantal punten uitgebreid. Twee gebieden van uitbreiding zijn het ontwerpen van CIM architecturen en het hergebruik van software. De vraagstelling met betrekking tot CIM architecturen luidt: wat is de relatie tussen het conceptueel modulaair ontwerpen van informatiesystemen en het implementeren van deze systemen in CIM architecturen. De onderzoeksvraag met betrekking tot het hergebruik van software luidt: hoe kan de methode van modulaair ontwerpen bijdragen aan het hergebruik van software.

De karakteristieken van de voorgestelde methode verschillen op een aantal punten van de meer traditionele methoden. In tegenstelling tot de traditionele methoden, die vaak gebaseerd zijn op het waterval-model, heeft de voorgestelde methode de volgende uitgangspunten:

- het ontwerpen van informatiesystemen is een evolutionair proces
- informatiesystemen dienen, indien mogelijk, opgebouwd te worden uit bestaande componenten
- informatiesysteemeisen zijn onderhevig aan continue veranderingen

De methode beschreven in dit proefschrift is gebaseerd op het ontwerpen van het conceptuele schema van de gegevensbank. Een informatiesysteem wordt daarbij gekarakteriseerd als een verzameling van modules die gekoppeld zijn door middel van hun interfaces. Iedere module wordt beschreven door middel van zijn eigen conceptuele schema. De interfaces tussen modules worden beschreven door middel van de raadpleeg- en wijzigingsbevoegdheden van de module met betrekking tot de gegevensbank. Deze bevoegdheden worden beschreven in zogenaamde 'domeinen'. Het eigen domein beschrijft welke gegevens de module mag wijzigen en het vreemde domein beschrijft welke gegevens de module mag raadplegen, maar niet wijzigen. Deze concepten maken het mogelijk om een ontwerp te valideren met betrekking tot de modulariteit ervan. Criteria voor modulariteit zijn onder andere, decomponeerbaarheid, componeerbaarheid, inzichtelijkheid van een module, continuïteit van een module, en bescherming van een module.

Een modulair ontwerp van een informatiesysteem is echter niet voldoende om een modulaire implementatie van een informatiesysteem te verkrijgen. In vele gevallen zullen implementatiezaken een beperking vormen van de wijze waarop een conceptueel ontwerp geïmplementeerd kan worden. Hiertoe zijn in dit proefschrift drie typen implementatiearchitecturen besproken, namelijk databasearchitecturen, systeemarchitecturen en besturingsarchitecturen. Hierbij wordt aangegeven dat gedistribueerde architecturen dusdanige condities kunnen creëren voor het implementeren van modules dat zij voldoen aan de hierboven gestelde criteria voor modulariteit.

De voorgestelde methode is vervolgens gevalideerd in een laboratorium bestaande uit een schaalmodel van een produktielijn voor printplaat-assemblage. Dit laboratorium biedt een realistische omgeving voor het implementeren van een modulair informatiesysteem voor shop floor control. Dit experiment is gebruikt voor een belangrijk deel van het onderzoek en is daarom uitvoerig beschreven in dit proefschrift.

Tenslotte worden de volgende aanbevelingen voor verder onderzoek gedaan. Ten eerste zou er verder onderzoek plaats moeten vinden naar de relatie tussen besturingsarchitecturen en modulaire informatiesystemen. Dit onderzoek zal waarschijnlijk leiden tot een effectiever gebruik van informatietechnologie. Ten tweede vormen het formaliseren van procedures tussen autonome modules en de pragmatiek van gegevens belangrijke onderwerpen voor het realiseren van gedistribueerde systemen. Ten derde zal er meer onderzoek plaats moeten vinden naar de mogelijkheid van hergebruik van modules en het ontwerpen van standaardsoftware ten behoeve van deze herbruikbaarheid. Ten vierde zal er meer onderzoek plaats moeten vinden naar de infrastructuur die nodig is voor het ontwerpen, implementeren en beheren van herbruikbare software.

## Curriculum Vitae

Patric Timmermans was born on 26 August 1966 in Gilze. In 1984 he finished University Preparatory Education (Gymnasium  $\beta$ ) at the Theresia Lyceum in Tilburg. He studied Computing Science at the Eindhoven University of Technology (TUE) with a major in Management Information Systems, and received his Master's in March 1989. In April 1989 he joined the School of Industrial Engineering & Management Science at TUE as a researcher to carry out a Ph.D. research in the area of modular design of information systems. This research was carried out in cooperation with the Digital Cooperative Engineering Centre in Amsterdam during a period of three years, partially on a research contract.

Additionally, he is employed since May 1989 as a scientific consultant of the Netherlands Organisation for Applied Scientific Research (TNO), Institute of Production and Logistics Research (IPL), Section Logistics Management. In this affiliation he has participated in the ESPRIT basic research action nr. 3143 'Factory of the Future', and was a guest researcher at the Technical University of Denmark in the System Science group of the Faculty of Electrical Power Engineering. At present, he holds a post-doc position at TUE, and is involved in several international research initiatives, such as the Esprit Working Groups Esprit-7400 'modelling of CIM systems' and Esprit-7401 'development of CIM systems'.

# Stellingen

behorende bij het proefschrift

Modular design of information systems  
for shop floor control

van

Patric Timmermans

9 juli 1993

Technische Universiteit Eindhoven

## I

Gedistribueerde architecturen, zowel op conceptueel als op implementatieniveau, creëren betere condities voor modulaire informatiesystemen dan bijvoorbeeld hiërarchische architecturen.

(Bron: dit proefschrift, hoofdstuk 6)

## II

Top-down functionele decompositie als methode voor het decomponeren van informatiesystemen is vaak ontoereikend vanwege het ontbreken van een top in zulke systemen.

(Bronnen: dit proefschrift, hoofdstuk 4.

Bertrand Meyer, *Object-oriented software construction*, Prentice Hall, 1988)

## III

Hergebruik van software wordt pas gemeengoed indien (tevens) gebruik gemaakt wordt van bottom-up ontwerpen in de levenscyclus van een informatiesysteem. Bovendien moeten herbruikbare componenten als zodanig ontworpen worden.

(Bron: Bertrand Meyer, *Object-oriented software construction*, Prentice Hall, 1988)

## IV

Een omvangrijke klasse van informatiesystemen voor de fabricage wordt beschreven door middel van de volgende karakteristieken: evolutionair ontwikkeld, continu veranderende systeemeisen, opgebouwd uit standaardcomponenten. Een methode voor systeemontwikkeling dient hier dan ook op gebaseerd te zijn.

(Bron: dit proefschrift, hoofdstuk 5)

## V

Bertrand *et al.* stellen: "Alhoewel computer-simulatiemodellen in vele gevallen voldoen, zal de realiteit van een productiebesturing in vele gevallen niet volledig beschreven kunnen worden door middel van een computer-simulatiemodel."

Het gebruik van een werkend schaalmodel als representatie van een produktiesysteem biedt een realistischer beeld.

(Bronnen: dit proefschrift, hoofdstuk 3.

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

## VI

Geen enkele modelleertaal is in staat louter neutrale en objectieve feiten weer te geven.  
(Bron: R.C. Kwant, *Fenomenologie van de taal*, Aula, 1963)

## VII

Gezien Kuhn's wetenschappelijk-filosofisch theorie valt er in de Bedrijfskunde een revolutie te verwachten die de overgang van pre-paradigmatisch naar 'normale' wetenschap aangeeft, waarna er een stroomversnelling in de vooruitgang van de wetenschap Bedrijfskunde zal plaatsvinden.

(Bron: Thomas S. Kuhn, *De structuur van wetenschappelijke revoluties*, Boom, Meppel/Amsterdam, 4e druk, 1987 (oorspronkelijke editie 1962))

## VIII

Gezien stelling 7 heeft falsificerend onderzoek in de Bedrijfskunde (nog) nauwelijks enige betekenis.

(Bronnen: Thomas S. Kuhn, *De structuur van wetenschappelijke revoluties*, Boom, Meppel/Amsterdam, 4e druk, 1987 (oorspronkelijke editie 1962))

## IX

De doorlooptijd van een bedrijfskundig promotieonderzoek dient ten minste 4 jaar te bedragen a) om zich in te werken in de wetenschappelijke arena, b) om voldoende diepgang te bereiken, en c) om voldoende terugkoppeling uit de wetenschappelijke wereld te garanderen.

## X

Internationalisering van onderzoekers is de hoofdvoorwaarde voor internationalisering van onderzoek. De meest serieuze methode hiertoe is langdurige uitwisseling van deze onderzoekers.

## XI

Europese onderzoeksprogramma's in Basic Research dienen niet zozeer beoordeeld te worden op hun directe output in de vorm van rapporten, artikelen, software, etc., als wel op de groei van kennis bij en uitwisseling van kennis tussen de deelnemende bedrijven, instituten, universiteiten en individuen in deze organisaties.

## XII

Het huidige imperialisme, in de vorm van het transponeren van culturele waarden en normen, gaat psychologisch verder en is niet minder verwerpelijk dan het imperialisme in strikte zin.

## XIII

Stijldansen is een sport en dient als zodanig erkend te worden, ook in Nederland.  
(Bron: Grote Winkler Prins Encyclopedie, Amsterdam, 1990)

## XIV

Het begrip dat men kan opbrengen voor het feit dat iemand weinig televisie kijkt, staat niet in verhouding met de verbazing bij het niet aantreffen van een televisietoestel in diens woonkamer.

## XV

Promoveren is 10% inspiratie en 90% transpiratie. Een verblijf in een ver warm land verandert deze verhouding niet, echter wel de intensiteit.