Mark de Weger

Structuring of Business Processes

An architectural approach to distributed systems development and its application to business processes

Mark de Weger

Structuring of Business Processes

An architectural approach to distributed systems development and its application to business processes

CTIT Ph. D-thesis series, no. 98-17
CTIT, P.O. Box 217, 7500 AE Enschede, The Netherlands

ISBN 90365-1052-X
ISSN 1381-3617, no. 98-17

Copyright © 1997 by M.K. de Weger, Enschede, The Netherlands

STRUCTURING OF BUSINESS PROCESSES

AN ARCHITECTURAL APPROACH TO DISTRIBUTED SYSTEMS DEVELOPMENT AND ITS APPLICATION TO BUSINESS PROCESSES

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit Twente, op gezag van
de Rector Magnificus, prof. dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen op
donderdag 22 januari 1998 te 15.00 uur.

door

MARTEN KLAAS DE WEGER,

geboren op 2 juni 1967
te Stadskanaal

Dit proefschrift is goedgekeurd door de promotor, prof. dr. ir. C.A. Vissers.

Preface

A change in the Dutch social security system has led to the commercial introduction of insurances that protect against the loss of income when becoming unfit for work. One insurance company has set up a business unit that has become very successful in selling these insurances. This success and the short time in which the business unit was set up have their drawback: the processes of the business unit are quite chaotic. For example, five "document seekers" continuously search desks and archives for documents required by other employees. Management wants more insight in, and control over the processes. What should they do?

The money transfer department of a bank has evolved over time and shows signs of ageing. For example, due to its decentralised organisation, with little contact between units, it is hard to give clients insight in the status of their orders. Past emphasis on security has led to many costly and labour-intensive controls. And the department is not equipped to support "electronic banking" on a large scale. Management decides the business process should be re-designed and supported by a new telematics system. What should they do?

In both examples management requires insight in their business processes, the activities carried out and the relations between these activities. The management of the insurance company needs insight in their current business processes. The management of the bank needs insight in potential new business processes.

But not only management requires insight. A reason the insurance company employs so many document seekers is that other employees do not know where to obtain the documents they need. If the management of the bank wants to convince its employees that they need to work in new ways, these employees need to be shown their role in the current process and why this is not optimal. If the business processes are re-designed, the employees need to know what their new tasks are and how these relate to other tasks. The designers of the new business process need insight in all aspects of the potential new business process, and in its alternatives, to design it such that it best conforms to the requirements. And the designers of the telematics system need insight in the new business processes to determine how the telematics system can offer the best support.

Business processes are often very complex. In this thesis we show how *structuring* can aid in obtaining insight in current and new business processes. The structuring of a system is the conception and subsequent representation of the system in a such a way that its complexity can be controlled. System structuring is sometimes called *architecting* [Rechtin, 1992].

Role of structuring in systems development

Structuring can be useful in any situation in which one wishes to get insight in an existing system. However, structuring is especially important in the development of new systems. This is because systems development requires good insight in many aspects of systems, and their relations. Examples of these aspects are: all requirements for the system under design, all parts of this system and their functions at many abstraction levels, possible alternatives for these parts and their functions, all systems that should interact with the system under design, and a system to be replaced by the system under design.

The following is a list of (not necessarily independent) situations in which business processes, and/or models thereof, have to be developed. The list is partly based on [Ould, 1995].

- Management wants more control over their current business, for example, if things often go wrong.
- Employees need information about their tasks and related tasks, for example, if new employees are introduced in the organisation.
- Business processes need to be documented or standardised, for example, if one introduces an ISO-9000 quality system.
- Business processes are to be improved in an evolutionary way, for example, if one introduces a system known as Total Quality Management.
- Business processes are to be replaced by other business processes, which is known as business process re-design.
- Business processes are to be supported by telematics systems, for example, workflow management systems or groupware.

Business processes as distributed systems

A part of this thesis deals with distributed systems development. Business processes *are* distributed systems and thus have many aspects in common with other distributed systems. By initially considering the larger class of distributed systems we try to reach the following objectives.

- *Enhancement of understanding*. Ferreira Pires [1994] speaks of a "methodology crisis" to denote the enormous difficulties in communication between people used to different development methods. By first treating the generic aspects of distributed systems, we try to show that many aspects of business processes are

systemic and independent of the specific application area of business processes. Because these generic aspects are (potentially) common to many development methods, we try to enhance in this way the understanding of development methods for both distributed systems and business processes.

- *General applicability*. By separating the generic aspects of distributed systems from the specific aspects of business processes, a part of this thesis is also applicable to telematics systems. This is important, because new telematics systems often enable business process re-design. The development of these supporting telematics systems is much eased if it can be carried out according to a similar method as the one used for the development of the business processes.

Topics of this thesis

Just as the architect of a house makes drawings of the house, the architect of a business process should make conceptions and specifications of the business process. Just as the architect of a house is provided with pre-defined building blocks for his drawings, which represent doors, windows, etc., the architect of a business process should be provided with a complete, consistent, and generally applicable set of architectural concepts, and their representations, that form the building blocks of business process models.

Basic architectural concepts are the elementary building blocks of business process models. We argue in this thesis that the basic architectural concepts should include: entities, to model objects that carry out behaviour; actions, to model activities; interactions, to model the joint activity of entities; and causality relations, to model the dynamic relations between activities.

We also argue that data is important for concisely modelling results of activities, relations between these results, and states of behaviours. We show, however, that the data concepts and the basic concepts are not orthogonal; instead, all data concepts can be defined in terms of the basic concepts and are thus not elementary building blocks. Data can be used as “shorthand” concepts for the basic concepts, enabling the development of insightful structures and efficient implementations. This insight allows us to describe in what cases data is useful in behaviour modelling, thus providing architects with development criteria.

Just as the architect of a house may simplify his design by considering different parts of the house separately, the architect of a business process needs to decompose the business process into different aspects to keep its complexity manageable.

We argue that modelling styles are appropriate to support architects in doing so. A modelling style is a set of rules for the structuring of a model to meet particular development objectives. The provision of a limited number of well-chosen and well-defined modelling styles does not only aid architects in structuring business process models, but also leads to more uniformly structured models, thus enhancing their comprehensibility.

Just as the architect of a house may first consider the main structure of the house in terms of, e.g., the foundation, the supporting walls and the roof, before considering the more ornamental aspects, the architect of a business process needs to be able to consider the business process at subsequent levels of detail.

We argue that abstraction levels are appropriate to support architects in doing so. Models of a business process at distinct abstraction levels are ordered on the basis of the amount of detail they possess. At a particular abstraction level, an architect models particular aspects of a business process. For example, at a high abstraction level the architect models the requirements for the business process that reflect the customer wishes. At lower abstraction levels, the architect models additional aspects, which define how the customer requirements are implemented, and the architect thus gradually introduces more detail in business process models.

Just as the architect of a house considers different aspects of the house in a particular order, the architect of a business process needs to determine the order in which he carries out his development steps.

A development strategy determines the order in which an architect carries out his development steps. Examples of development strategies are the waterfall model and the evolutionary strategy. We argue that no development strategy is appropriate for all development processes. We therefore construct criteria for the selection of development strategies by 1) identifying the main elements of development strategies, 2) identifying the development objectives that development strategies may support, and 3) identifying the relation between each identified element of a development strategy and its support for the development objectives. The selection of a development strategy can then be based on the need for the support of particular development objectives in a development process.

We finally argue that development methods for business processes need not be "methods of the cookbook" [Simon, 1996], which are almost solely based on the induction of practical experience. We show instead that concretely applicable development methods can be constructed by the synthesis and refinement of structuring techniques that are appropriate for particular development processes.

Structure of this thesis

Chapter 1, Introduction, introduces distributed systems, business processes, systems development, and systems development methods. It also outlines our research approach.

Chapter 2, Structuring, analyses aspects of complexity and introduces structuring techniques to control these aspects. It also treats quality criteria for structures and discusses the role of structuring in development processes.

Chapters 3 to 7 present structuring techniques, which can be used as elements of methods for distributed systems development.

Chapter 3, Basic architectural concepts, introduces basic concepts for distributed systems modelling. These include entities, which model things, actions, which model activities, and causality relations, which model dynamic relations between activities.

Chapter 4, Role of data in behaviour modelling, introduces additional concepts for distributed systems modelling. These include action values, which model results of activities, reference relations, which model relations between results, and state values, which model one or more states of a system. The chapter shows when data is useful in modelling behaviour comprehensibly, why data is useful in those cases, and that all data concepts can be defined in terms of the basic architectural concepts.

Chapter 5, Structuring of behaviour models, introduces some modelling styles, techniques for the structuring of behaviour models that support the pursuit of particular development objectives. The chapter shows how the defined styles aid in making comprehensible models of frequently encountered structures of business processes.

Chapter 6, Abstraction levels, introduces some abstraction levels, which are system perspectives, such that each model of a system at a particular abstraction level contains either more or less detail than a model of the system at any other abstraction level. The chapter defines the relations between the introduced abstraction levels and shows the use of each abstraction level in the pursuit of particular development objectives.

Chapter 7, Development strategies, introduces an approach to the selection and construction of development strategies, which prescribe the order of steps to be taken in a development process. The chapter analyses the structure of development strategies and shows how the different aspects of development strategies support the pursuit of particular development objectives.

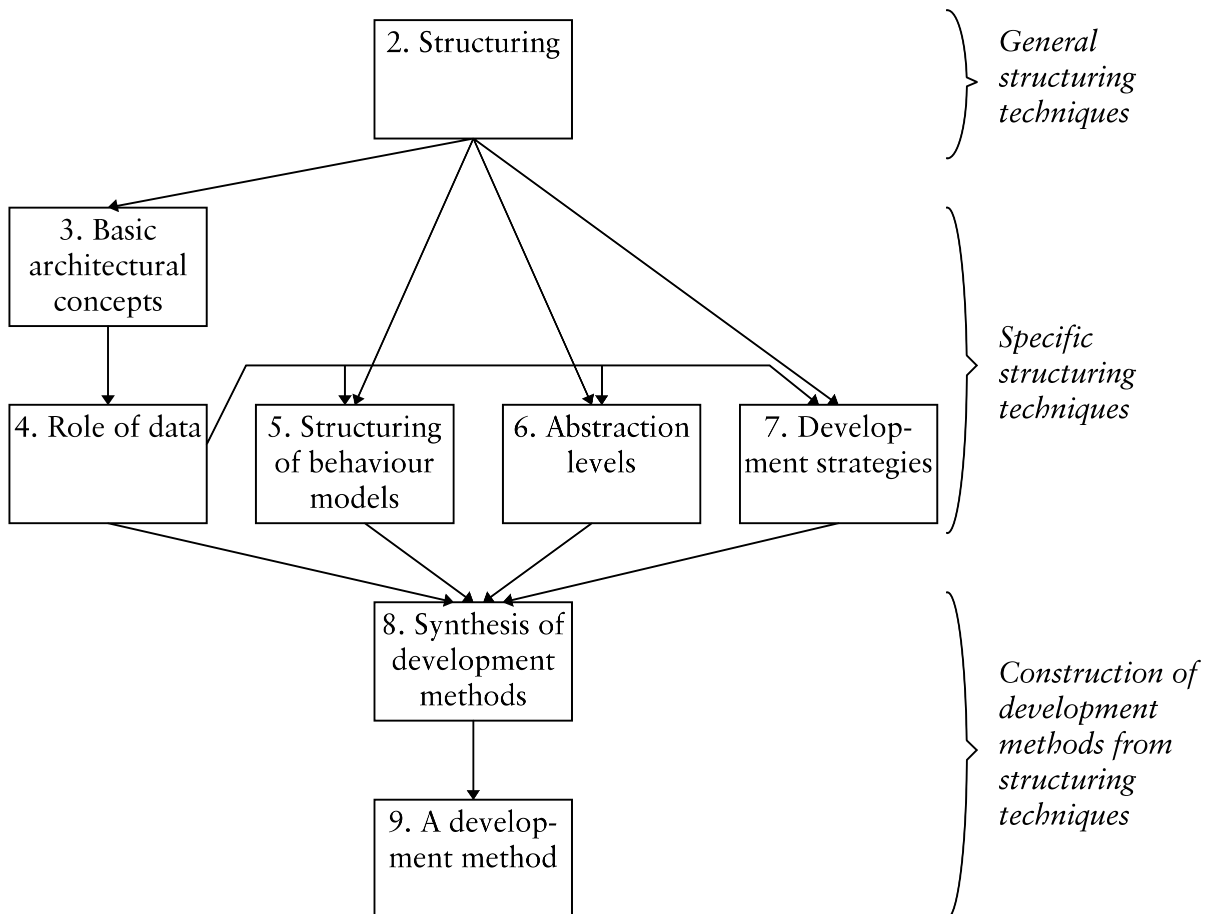
Chapters 8 and 9 treat the construction and application of methods for the development of business processes.

Chapter 8, Synthesis of development methods, introduces an approach for the construction of methods for the development of business processes. The approach comprises the synthesis and refinement of selected structuring techniques.

Chapter 9, A work-flow-oriented evolutionary development method, presents a development method for business processes to illustrate the use of the structuring techniques and to show that these abstract techniques can be synthesised and refined to a concretely applicable development method.

Chapter 10, Conclusions and further research, presents our conclusions and recommendations for further research.

The figure below illustrates the structure of the body of this thesis. Arrows represent dependencies between chapters, which are explained below, and thus indicate the preferred reading order. The general structuring techniques, discussed in chapter 2, form the basis of the more specific structuring techniques for distributed systems, discussed in chapters 3 to 7. The architectural concepts, discussed in chapters 3 and 4, are used in the definition or presentation of the other specific structuring techniques. The synthesis of development methods, discussed in chapter 8, takes place on the basis of all of the specific structuring techniques. The development method of chapter 9 forms an example of the synthesis of development methods.



Acknowledgements

First and foremost I thank Chris Vissers, who supervised and guided my research. His distinct approach to systems development, always stressing the needs of the architect, and always calling for clarity and preciseness, has profoundly influenced this thesis.

Henry Franken's and Luís Ferreira Pires' guidance was motivating and helped me in clarifying my ideas. The discussions with my (room)mate Dick Quartel had an important impact on this thesis. A part of chapter 3 is based on research carried out with Dick.

A large part of the research reported on in this thesis was carried out in the Testbed project [Franken, 1997]. In this project, in which methods and tools are developed and applied for the development of business processes, the following organisations co-operate: ABP, De Belastingdienst, IBM, ING Group, and TRC. I thank all of the employees I have co-operated with for sharing their insights.

The co-operation with my TRC colleagues was fruitful. I thank René Bal, Harmen van den Berg, Frank Biemans, Henk Eertink, Wil Janssen, Henk Jonkers, Marc Lankhorst, Ferial Moelaert, Johan Oldenkamp, Paul Oude Luttighuis, Petra van der Stappen, Wouter Teeuw, Erik Wierstra, and Dirk de Wit. René Bal and Wouter Teeuw provided some of the heuristics presented in chapter 9.

This thesis has further benefited from discussions with the following persons: Marten van Sinderen, Aiko Pras, Pim Kars, Harro Kremer, Sjaak Brinkkemper, Yair Wand, Leo Essink, Jeroen Kneppers, the distinguished mathematicians Lex Heerink and Karin Gerritsen, and the linguistic consultants Val Jones and Phil Chimento.

My fellow BOCOM members, Lex Heerink, Marcel Rijnders, Georgios Karagianos, and Vincent Berkhout were a great source of inspiration and showed me that at times social research is far more fun than engineering research.

Joke Lammerink, Ila Reinders, Wilma Hiddink, Marloes Castaneda, and Ellen Berkien were helpful in daily activities, particularly those requiring stress relief. Some of the attempts by Jeroen van de Lagemaat and Frans van der Avert to manage unmanageable organisations have made working life easier for me.

Finally, I thank my parents and Magdalis for their support.

Contents

Preface v

Acknowledgements xi

Contents xiii

1 Introduction 1

1.1 Introduction 1
1.1.1 Motivation 1
1.1.2 Structure 2
1.2 Systems, distributed systems and business processes 2
1.2.1 Systems: delimitation 2
1.2.2 Distributed systems 3
1.2.3 Business processes 3
1.3 System development 4
1.3.1 Implementation, conception and representation 4
1.3.2 Analysis and design 6
1.3.3 Development methods 7
1.4 Research objectives and approach 9
1.4.1 Research objectives 9
1.4.2 Research approach 10

2 Structuring 13

2.1 Introduction 13
2.2 Complexity 14
2.3 Structuring techniques 16
2.3.1 Distributed perspective 16
2.3.2 Integrated perspective 19
2.3.3 Forms of abstraction and refinement 21
2.4 Quality of systems 22
2.4.1 Cleanliness 22
2.4.2 Consistency 23

2.4.3 Orthogonality 23
2.4.4 Propriety 24
2.4.5 Generality 25
2.5 Role of structuring in development processes 26
2.5.1 Structuring as architecting 26
2.5.2 Structuring in this thesis 27
2.6 Conclusions 28

3 Basic architectural concepts 29

3.1 Introduction 29
3.1.1 Motivation 29
3.1.2 Structure 29
3.2 Entities and (inter)action points 30
3.2.1 Entities 30
3.2.2 Interaction points 31
3.2.3 Entity (de)composition 31
3.3 Actions and interactions 32
3.3.1 Actions 32
3.3.2 Interactions 35
3.3.3 Abstraction and refinement 35
3.3.4 Coupling of actions and entities 36
3.4 Behaviours containing two actions 37
3.4.1 Causality relations 38
3.4.2 Elementary causality relations 38
3.4.3 Disjunctions of elementary causality conditions 42
3.4.4 Reciprocal aspect of synchronisation and disabling conditions 44
3.5 Monolithic behaviours 45
3.5.1 Conjunction of elementary causality conditions 45
3.5.2 Disjunction of alternative causality conditions 47
3.5.3 Behaviour executions 49
3.5.4 Additional shortcut representations 52
3.6 Behaviour composition 52
3.6.1 Causality-oriented behaviour composition 53
3.6.2 Constraint-oriented behaviour composition 58
3.7 Conclusions 61

4 Role of data in behaviour modelling 63

4.1 Introduction 63
4.1.1 Motivation 63
4.1.2 Structure 63
4.1.3 Summary of concepts 64
4.2 Role of action values 65

4.2.1 Inconvenience of current concepts 65
4.2.2 Data concepts 66
4.2.3 Meaning of data concepts in terms of basic concepts 68
4.3 Role of reference relations 69
4.3.1 Inconvenience of current concepts 69
4.3.2 Data concepts 70
4.3.3 Meaning of data concepts in terms of basic concepts 76
4.4 Role of state values 78
4.4.1 Inconvenience of current concepts 78
4.4.2 Data concepts: formal definition and use 79
4.4.3 Data concepts: architectural semantics 85
4.4.4 Meaning of data concepts in terms of basic concepts 94
4.5 Extensions 95
4.5.1 Extensions for disabling and synchronisation relations 95
4.5.2 Extensions for composite behaviours 96
4.5.3 Value decomposition 100
4.6 Conclusions 100

5 Structuring of behaviour models 103

5.1 Introduction 103
5.1.1 Motivation 103
5.1.2 Structure 104
5.2 Some modelling styles 105
5.2.1 Example 105
5.2.2 Extensional versus intensional styles 105
5.2.3 Monolithic versus modular styles 108
5.2.4 Process-oriented versus state oriented styles 112
5.3 Structures of business processes 115
5.3.1 Business processes structured around work-flows or business functions 115
5.3.2 Structuring around work-flows or business functions 116
5.4 Application of styles to business processes 119
5.4.1 Example 119
5.4.2 Requirements modelling 120
5.4.3 Implementation modelling 124
5.5 Role of styles in development methods 127
5.6 Conclusions 128

6 Abstraction levels 131

6.1 Introduction 131
6.1.1 Motivation 131
6.1.2 Structure 132

6.2 Abstraction, refinement, and their role in abstraction levels 133
6.2.1 Abstraction levels 133
6.2.2 Refinement operations 134
6.2.3 Conformance 136
6.3 Generic abstraction levels 138
6.3.1 Distributed and integrated system perspectives 138
6.3.2 Interaction system perspective of parts 139
6.3.3 Summary 141
6.4 Specific abstraction levels 142
6.4.1 System embedded in its environment 142
6.4.2 Integrated system perspective 143
6.4.3 Logically distributed system perspective 144
6.4.4 Physically distributed system perspective 145
6.4.5 Local interface refined system perspective 146
6.5 Application example 149
6.5.1 System embedded in its environment 149
6.5.2 Integrated system perspective 149
6.5.3 Logically distributed system perspective 150
6.5.4 Physically distributed system perspective 152
6.5.5 Local interface refined system perspective 153
6.6 Conclusions 153

7 Development strategies 155

7.1 Introduction 155
7.1.1 Motivation 155
7.1.2 Structure 156
7.2 Some basic development strategies 156
7.2.1 Waterfall model 157
7.2.2 Evolutionary development strategy 157
7.2.3 Code-and-fix strategy 157
7.2.4 Rapid prototyping 158
7.3 Structure of development strategies 158
7.3.1 Separation of concerns 158
7.3.2 Order of development steps 160
7.4 Evaluation of basic development strategies 161
7.4.1 Criteria 161
7.4.2 Support for minimisation of re-work 162
7.4.3 Support for optimal resource use 169
7.4.4 Support for conformance to quality criteria 171
7.4.5 Summary of conclusions 174
7.5 Selection of basic development strategies 174
7.5.1 Risks of development processes 174

7.5.2 Methods for risk assessment 176
7.5.3 Selection of a basic development strategy 176
7.6 Composite development strategies 177
7.6.1 Use of different basic development strategies at different abstraction levels 177
7.6.2 Use of different basic development strategies for different increments or versions of the system 179
7.7 Development strategies for specific types of systems 180
7.8 Conclusions 181

8 Synthesis of development methods 183

8.1 Introduction 183
8.1.1 Motivation 183
8.1.2 Structure 183
8.2 Construction of development methods 184
8.2.1 Approach 184
8.2.2 Development processes comprising both analysis and design 186
8.2.3 Comparison to other approaches 188
8.3 Synthesis: some examples 189
8.3.1 Initial development step 189
8.3.2 Examples of development methods based on structuring around work-flows 189
8.3.3 Examples of development methods based on structuring around business functions 192
8.4 Conclusions 193

9 A work-flow-oriented evolutionary development method 195

9.1 Introduction 195
9.2 Overview of the method 195
9.3 The method 199

10 Conclusions and further research 223

10.1 Introduction 223
10.2 Research contributions 223
10.3 Practical application 225
10.3.1 Background 225
10.3.2 Architectural concepts and their representation 225
10.3.3 Modelling styles 227
10.3.4 Abstraction levels 228
10.3.5 Development strategies 228
10.3.6 Development methods 229
10.4 Further research 229
10.4.1 Topics of this thesis 229

10.4.2 Quantitative analysis 230
10.4.3 Automated support tools 230
10.4.4 Implementation 231
10.4.5 Telematics systems 232

References 233

Index 245

Summary 249

Samenvatting 253

1

Introduction

1.1 Introduction

1.1.1 Motivation

A business process is the set of related activities carried out by an organisation, or a part thereof, to deliver particular services to clients. A well-structured business process that effectively and efficiently meets the clients' requirements will give an organisation an important competitive advantage over other organisations. A badly structured business process is likely to yield poor service to clients, and, given the many alternatives clients have nowadays, they will quickly switch to other organisations. More and more organisations therefore realise that their business processes are their key resources [Leymann & Altenhuber, 1994].

Therefore, new business processes should be developed consciously, always keeping in mind the customer requirements they should fulfil. Current business processes should be evaluated and improved or re-designed when necessary. Business processes should not evolve in uncontrolled ways.

Practice shows this is hard to meet, because most business processes, and the processes of their development, are complex. Many organisations have difficulty coping with an environment in which customers have taken the upper hand, in which competition has intensified, in which new technology offers opportunities to work more effectively, and in which change is constant. Even if organisations realise they need to change their business processes, they often do not know how to do so [Hammer & Champy, 1993].

This thesis contributes to the field of business process development by providing methods and structuring techniques to control the complexity of business processes and of their development.

This chapter introduces the problems addressed in this thesis. It introduces business processes and distributed systems, a class of systems business processes form a part of. It introduces system development, the objectives of system development methods, and the main elements of such methods. Finally, it formulates the objectives of this thesis and discusses the way in which we have striven to reach these objectives.

1.1.2 Structure

Section 1.2 delimits the class of systems under concern. It defines distributed systems and business processes, and gives examples of these.

Section 1.3 defines system development and development methods, and discusses the main elements of development methods.

Section 1.4 defines our research objectives and discusses our research approach.

1.2 Systems, distributed systems, and business processes

W consider an organisation as a system, and a business process as the behaviour of this system.

1.2.1 Systems: delimitation

A *system* is a regularly interacting or interdependent group of items forming a unified whole [Webster's]. The class of systems thus includes Michelangelo's Mona Lisa, an insurance company handling claims, the word processor this thesis is written with, hurricanes, atoms, a swinging pendulum, elephants, and the tax office.

This thesis is confined to *artificial systems*, systems made by man. Thus, we do not consider hurricanes, atoms, and elephants, which are natural systems.

This thesis is further confined to *dynamic systems*, systems whose properties change over time. In chapter 3 we call the set of changing and related properties of the dynamic systems we consider their *behaviour*. Thus, we do not consider the Mona Lisa, which is a static system.

This thesis is further confined to *discrete systems*, systems whose behaviour can be properly described in terms of their related properties at selected moments in time.[1] In chapter 3 we call the behaviour properties of discrete systems considered at selected moments their *actions*. Thus, we do not consider a swinging pendulum, which is a continuous system whose behaviour (movement) can be precisely described by means of a differential equation.

---

1. A distinction is sometimes made between discrete time systems and discrete event systems. In discrete time systems the interval between the selected moments of time is constant. This need not be the case in discrete event systems [Mitrani, 1982].

1.2.2 Distributed systems

This thesis concentrates on distributed systems in general. *Distributed systems* are systems whose parts exhibit behaviour that is partly independent of other parts. Such parts are sometimes called *autonomous parts*. In chapter 3 we call the actions a part can execute independent of other parts *internal actions* of the part and the actions that the part can execute only in co-operation with other parts *interactions* of the part.

Examples of distributed systems are the tax office, a communication protocol, a zoo, a work-flow management system, and a university. The word processor this thesis is written on is not a distributed system, but a *centralised system*, since it carries out all its activities sequentially.

Since the parts of distributed systems may be centralised systems, our concentration on distributed systems does not exclude centralised systems.

1.2.3 Business processes

This thesis concentrates on business processes in particular. A *business process* is the set of related activities carried out by an organisation, or a part thereof, to deliver particular services to clients. Such services may include tangible products. The business processes we consider, and the entities that execute them (for example, employees, managers, computer programs, and cars), always comprise autonomous parts and are therefore distributed systems.

Examples of business processes are the tax office collecting tax, zoo keepers feeding their animals, an insurance company handling claims, and university lecturers teaching their students.

A business process can be viewed in terms of the services it provides. In that case it is viewed from the perspective of its environment, which includes its customers. For example, the handling of a claim by an insurance company can be described from the perspective of the environment as follows:

- a customer presents a claim by filling in a damage form and sending it to the insurance company;
- the customer receives the form back if the information on the form is incomplete or incorrect;
- otherwise, the customer is informed by mail of the validity of the claim;
- the customer receives a payment of the damage if the claim is valid.

A business process can also be viewed in terms of the related activities required to provide its services. In that case it is viewed from an internal perspective. For example, the handling of a claim by an insurance company can be described from an internal perspective of the insurance company as follows:

- a customer presents a claim by filling in a damage form and sending it to the insurance company;
- a junior employee judges the information on the form on completeness and correctness;
- the junior employee sends the form back to the customer if the information on the form is incomplete or incorrect;
- the junior employee passes a form with complete and correct information on to a manager;
- the manager decides which senior employee is to validate the claim;
- the manager passes the claim to the appointed senior employee;
- the senior employee validates the claim;
- the senior employee informs the customer by mail of his or her decision;
- the senior employee passes a valid claim on to the debit department of the insurance company;
- the debit department pays the damage to the customer.

1.3 System development

An artificial system is, by definition, created by man. It is developed to serve its environment; the environment should be able to function better with the system than without it. *System development* is the set of related activities with the purpose of creating such a system.

1.3.1 Implementation, conception, and representation

Alexander [1964] distinguishes three types of development processes.

In the first type of development processes, which Alexander calls the "unselfconscious" type, a developer directly reacts to "misfits" perceived in a current situation. Almost without thinking the developer makes changes to the situation, resulting in a system that allows its environment to function better than before. Alexander shows that this is the way in which for thousands of years the shapes of shelters and houses evolved. Modern examples of this type of development process are the fixing of a bicycle tire or the assignment of an additional cashier in a supermarket when the queues in front of the pay-desks get too long. Since the changes that can be made by this type of development process are small, it is sometimes referred to as adaptation or evolution.

In the second type of development processes, which Alexander calls the "selfconscious" type, a developer thinks about the desired system, creating concep-

tions or “mental pictures” of it, which model the aspects of concern to the developer. By doing so, the developer can analyse which aspects of the current situation need improvement and how a new system can provide this improvement. An example provided by Alexander of this type of development process is the way most houses were built in the Middle Ages and the way in which simple houses are built even today in many countries. Other examples are the planning of a delivery route by a mail man, or the writing of a simple computer program.

In the third type of development processes, developers do not only create mental pictures of the system under development, but also “real pictures”, representations of their conceptions. Such representations serve at least four purposes. First, they help developers in overcoming the limits of their mental capacity. “Two minutes on the back of an envelope lets us solve problems which we could not do in our heads if we tried for a hundred years” [Alexander, 1964, p. 6]. Second, they help developers in making their ideas precise, consistent, and complete. “The act of writing turns out to require hundreds of mini-decisions, and it is the existence of these that distinguishes clear, exact policies from fuzzy ones” [Brooks, 1995, p. 111]. Third, such representations allow the communication of ideas and decisions between developers. This is required in development processes that are too complex to be carried out by a single developer. Fourth, such representations allow the communication of requirements between developers and users. This is required in development processes in which the user of a system does not develop the system on his own. Examples of the third type of development processes are the development processes of sky scrapers, aeroplanes, telematics systems, and business processes.

It is the third type of development process we consider in this thesis, since it is the only one suitable for the development of complex systems.

Figure 1.1 depicts the elements of a development process discussed above. A system under development is to become an element of the real world. To distinguish a system in the real world from a conception or a representation of this system, we call a system in the real world a system *implementation*. By abstracting from a system implementation and considering only the aspects relevant to one’s purpose, one constructs a *conception* of the system. Conversely, the system in the real world is an implementation of this conception. A conception of a system can be represented, resulting in a representation or *specification* of the system. Conversely, the interpretation of a specification results in a conception of the system.

Above we argued that developers make conceptions and specifications of a system for partly the same purposes: both conceptions and specifications allow them to study aspects of the system and, to some extent, test their ideas without fiddling around with the system implementation. If it is irrelevant whether we refer to a conception or a specification of a system, we use the term system *model*. The term model is used here according to the (quite broad) definition of Rothenberg [1989]: the cost-effective use of something in place of something else.

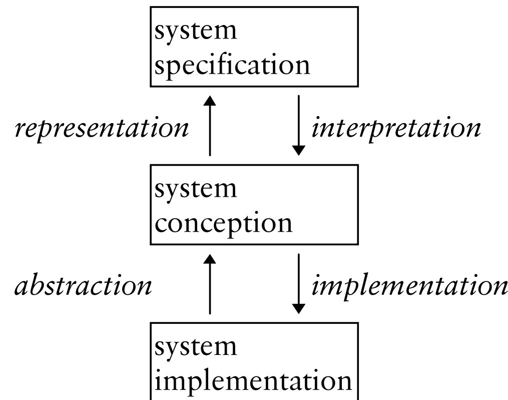


Figure 1.1 Implementation, conception, and representation of a system, and their relations

1.3.2 Analysis and design

A development process comprises two types of activities that are so different we distinguish them from each other: analysis and design.

Analysis is the detailed examination of an existing system in order to understand it. Analysis usually involves the breaking up of a system into its constituent parts. In a development process analysis is applied in order to identify the requirements for the system under development. It may involve the analysis of a current system that is to be replaced by the system under development, to identify the ways in which the new system can improve on the current system. It may involve the analysis of user or customer requirements, to make their wishes precise. And it may involve the analysis of the remainder of the environment of the system, to identify the constraints this environment puts on the system.

Since analysis describes existing things, whether it be a current system, or current user requirements, it is a *descriptive* activity[2]. Only when the results of the analysis turn into requirements for a new system, these results form a prescription for this new system.

Design or *synthesis* is the construction of a new system from parts. Ideally, design starts of with a set of requirements and a set of parts (or implementation components) from which a system that satisfies these requirements should be constructed. In practice, however, the requirements are usually incomplete and inconsistent, and

2. The description of a system should not be confused with the observation of a system. Observing a system is a means to identify elements of the system to be described. (Other means for identifying such elements are e.g. considering a specification of the system and interviewing a developer of the system.) Certain things that can be described cannot be observed. Causality relations are a well-known example of this.

the set of parts is usually (partly) undefined. Design therefore also involves choices regarding the requirements and the parts.

Since design results in a specification of how future things should be, rather than how current things are, it is a *prescriptive* activity. We call a prescriptive model of a system *a design* of the system.

In a development process analysis and design activities are usually intermingled.

1.3.3 Development methods

A method is a systematic way of working to reach an objective. A (system) *development method* is a systematic way of working to develop a system.

We argue in this thesis that the following elements should be part of a development method.

- *Architectural concepts*. Architectural concepts are abstractions of system elements and model required characteristics of systems. They are the building blocks available to developers for modelling systems. Basic architectural concepts are the most elementary architectural concepts, from which other, derived, architectural concepts can be constructed.

  A development method should provide a complete, consistent, and generally applicable set of basic architectural concepts [Vissers et al., 1995]. Moreover, if certain derived architectural concepts are frequently used in system conceptions, and if these architectural concepts are complex compositions of basic architectural concepts, a development method should also be capable to provide these derived architectural concepts.

- *Specification language*. Because system conceptions need to be represented, a development method should provide a specification language in which system conceptions can be represented.

  Ideally, each basic architectural concept provided by a development method is represented by one element of the specification language. In practice, there is not always a one-to-one relation. The relation should, however, always be clear. We call this relation between the elements of a specification language and the architectural concepts the *architectural semantics* of the specification language.

  In general, the *semantics* of something is its meaning in terms of something else. The architectural semantics of something is its meaning in terms of (abstractions of) real-world phenomena. In order to make a specification language more precise, this language is sometimes equipped with a *formal semantics*, which defines the meaning of the specification language in terms of a mathematical model.

  Figure 1.2, which is based on [Vissers et al., 1995], represents the relations between architectural concepts, a specification language, a system conception, and a specification.

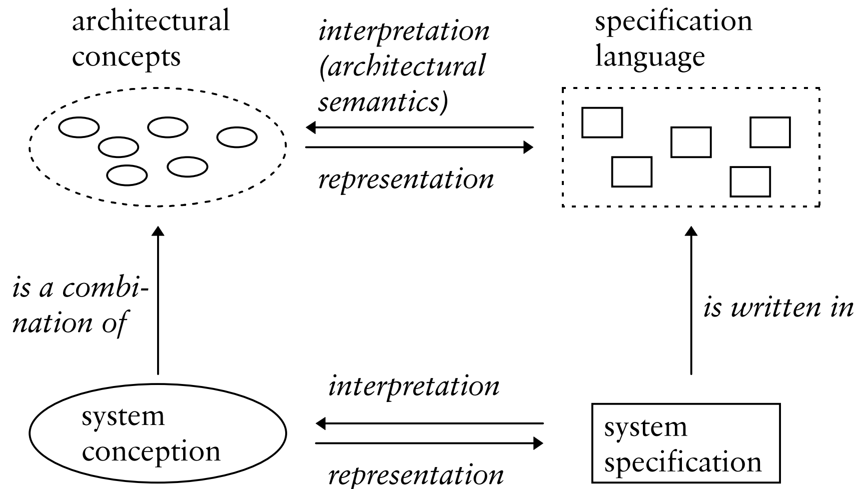


Figure 1.2 Relations between architectural concepts, a specification language, a system conception and a system specification.

- *Further structuring support*. Architectural concepts and a specification language allow developers to model systems. Such models may, however, be very complex. Further support is therefore required to aid developers in structuring systems. Such support may vary from general guidelines that are applicable to, e.g., most distributed systems, to specific guidelines that are applicable to, e.g., tax offices or tele-sales processes only. In this thesis we are mainly concerned with the more general support. We argue that in order to properly give this support, the following is required.
  - *Modelling styles*. A modelling style is a set of rules for the structuring of a model to meet particular development objectives. The provision of a limited number of well-chosen and well-defined modelling styles does not only aid individual developers in structuring system models, but should also lead to more uniformly structured models, thus enhancing their comprehensibility.
  - *Abstraction levels*. Models of a system at distinct abstraction levels are ordered on the basis of the amount of detail they posses. At a particular abstraction level a developer takes particular development decisions that regard particular development objectives. These decisions are then reflected in a system model at that abstraction level. At lower abstraction levels, a developer makes other development decisions that regard other development objectives, thus gradually introducing more detail in system models.
- *Development strategies*. The application of further structuring leads to related development steps in which development decisions are taken. A development strategy determines the order in which these steps are carried out. Examples of development strategies are the waterfall model and the evolutionary strategy. Current development methods usually provide only one development strategy. This thesis argues that different development processes may require different de-

velopment strategies. Developers should therefore be allowed to select an appropriate development strategy.

In chapter 2 it is shown that the above elements of development methods aid in structuring systems, and are therefore *structuring techniques*.

All of these structuring techniques should be applicable to both the analysis of a current system and the design of a new system. Since a system is developed to meet particular objectives, the structuring techniques should explicitly support the pursuit of certain objectives.

The above elements of development methods can be considered conceptual tools. Development methods may also provide *automated support tools*. These tools, usually computer programs, support developers in carrying out development steps by automating parts of them. Examples of support tools are editors, which support developers in specifying systems, and simulators, which support developers in simulating and thereby analysing certain aspects of systems. Automated support tools are not considered in this thesis.

1.4 Research objectives and approach

1.4.1 Research objectives

The objectives of this thesis are the following.

1. The definition of structuring techniques for business processes that aid in controlling their complexity, and the complexity of their development processes.

   The structuring techniques should include the ones mentioned in section 1.3.3.

   Wherever possible, the structuring techniques should be general and applicable to other distributed systems as well in order to 1) aid in identifying the fundamental aspects of business processes that are applicable to other systems, thereby enhancing the understanding of development methods for both business processes and distributed systems, and 2) be applicable to telematics systems, thereby easing the development of telematics systems and the business processes they support in a coherent framework.

2. The explication of the development objectives supported by each defined structuring technique.

We observed that the demand for generality leads to structuring techniques that are perceived by some developers as too abstract and thereby too difficult to understand for practical application. Therefore, we add a third objective.

3. To show how the defined structuring techniques can be synthesised and refined to concretely applicable development methods for business processes.

1.4.2 Research approach

In order to be able to provide structuring techniques of high quality that aid in controlling the complexity of business processes and their development processes, we have investigated what complexity is, how generic structuring techniques may aid in controlling aspects of complexity, and how structuring techniques should satisfy quality criteria. Chapter 2 presents an overview of this research, which has guided the development of structuring techniques for distributed systems and business processes in the remainder of this thesis.

Architectural concepts

In our approach to the definition of basic architectural concepts, we first make a distinction between the dynamic aspect of business processes, behaviour, and the carriers of behaviour, entities. A behaviour is a set of related activities. Thus, architectural concepts are required to model activities and their relations. An action is an abstraction of an activity and models the relevant result of this activity. A causality relation models the way in which the occurrence of an action depends on the occurrences or non-occurrences of other actions. Chapter 3 introduces and motivates these and related architectural concepts.

Data is a group of architectural concepts that can be used to concisely model results of activities, relations between these results, and states of behaviours. Since data is well-known in practice, we have focused on the relation between the data concepts and the basic architectural concepts. This research has resulted in the insight that the data concepts and the basic concepts are not orthogonal; instead, all data concepts can be defined in terms of the basic concepts and are thus derived concepts. This insight has allowed us to describe in exactly what cases data is useful in concisely modelling behaviours, thus providing developers with development criteria. Chapter 4 presents the results of this research.

Further structuring support

Some modelling styles for distributed systems were already defined by Vissers et al. [1988]. We have striven to improve these modelling styles to comply to our new insights regarding architectural concepts. This has resulted in a number of generic modelling styles for distributed systems. Next, our practical experience with the structuring of business processes has led to the distinction of some frequently occurring structures of business processes. By defining these structures, describing them in terms of the generic modelling styles, and identifying the development objectives they support, we have defined concrete modelling styles for business processes. Chapter 5 presents the results of this research.

Many system development methods support “perspectives”, abstractions of a system under development. Unfortunately, the relations between these perspectives are not always well-defined. Since a complete model of a system is formed by the composition of all correct abstractions of the system, it is useful to know how the abstrac-

tions relate to each other. We have therefore defined a group of abstraction levels, which are perspectives with clearly defined (refinement) relations between them. The definition of the development objectives to be dealt with at each abstraction level and the definition of the resulting structure of a model at each abstraction level has been guided by our practical experience with business process development. Chapter 6 presents the results of this research.

Development strategies

Experience has shown that there is no single development strategy that suits all development processes: different development processes may require different development strategies. Since it is not clear for which development processes a development strategy is most appropriate, we have investigated this relation. We started by analysing current development strategies, identifying which elements may influence development objectives. We then identified the development objectives that a development strategy may support. Finally, by investigating how an element of a development strategy supports the pursuit of a particular development objective, we were able to provide criteria for the selection of development strategies. Chapter 7 presents the results of this research.

Synthesis and refinement of development methods

Synthesis of a development method may take place by selecting appropriate modelling styles and abstraction levels. This results in groups of related development steps. By selecting a development strategy, the order of these groups of development steps is determined, and the various elements are synthesised in a development method. Chapter 8 presents this approach to the synthesis of development methods.

By providing further separation of concerns, heuristics, design criteria, and examples, the development method can be made more concrete. Chapter 9 gives an example of a concrete development method.

2

Structuring

2.1 Introduction

Motivation

The most important reason for the existence of methods for system development is the complexity of system development: if system development were a simple task, there would be little need for guidance on this task. It is argued below that the complexity of system development is caused by the complexity of the system under development.

The objective of this chapter is to analyse how structuring can aid in controlling this complexity. We therefore first analyse what complexity is and identify some important aspects of complexity. We then analyse how structuring techniques can aid in controlling the identified aspects of complexity. Since the application of structuring techniques does not guarantee the development of high quality structures, we also describe quality criteria for structures.

Structure

Section 2.2 analyses complexity and identifies some important aspects of complexity.

Section 2.3 defines some structuring techniques and shows how these techniques can aid in controlling the identified aspects of complexity.

Section 2.4 describes some quality criteria for structures.

Section 2.5 describes the role of structuring in system development and introduces the structuring techniques for business process development that are presented in the remainder of this thesis.

2.2 Complexity

A system is a group of interrelated elements forming a unified whole. When, in the following, we analyse systems in general, this analysis therefore applies both to a system under development and to the development process of this system.

One of the definitions of "complex" Webster's Dictionary [Webster's] gives is: "having many varied interrelated ... elements and consequently hard to understand fully". Since the difficulty of understanding a complex system is caused by the limited mental capabilities of humans, the perception of complexity is partly subjective.

The development of a system requires a full understanding of it. A complex system is therefore hard to develop. The objective of a development method is to make this complexity manageable (synonym: controllable). Therefore, it is important to have insight in the various aspects of complexity when developing a development method. The first part of the definition of "complex" helps us in gaining this insight. We distinguish the following aspects of complexity.

1. *Variety of elements*. As the variety of elements in a system increases, while other things remain unchanged, the system becomes more complex. This is because the different sorts of elements in the system have to be comprehended and treated differently. The more sorts of elements a system consists of, the more attention this requires and the harder this becomes.

   As an example, consider an organisation. If in this organisation everybody would carry out the same task, the organisation would be relatively easy to manage, even if the amount of employees were large, because every employee can be managed in the same way. However, if different tasks are introduced, for example purchasing, production, and sales, while the amount of employees remains the same, management of the organisation becomes harder: managers should have knowledge of three tasks, they should know which task each employee carries out, and they should adjust their management activities to this.

   Variety of elements does not only apply to the static element sorts that can be distinguished, such as purchasing and production employees. It also applies to the different states elements can be in. For example, if, at some moment, different production employees are involved in different production tasks, they may have to be managed differently, which increases the complexity of management.

2. *Variety of relations*. As the variety of relations in a system increases, while other things remain unchanged, the system becomes more complex. First, this is because the different sorts of relations have to comprehended and treated differently. The more sorts of relations a system consists of, the more attention this requires and the harder this becomes. Second, as the variety of relations increases, it becomes increasingly harder to decompose the system into (nearly) independent groups of related elements which can be comprehended separately.

Consider the task of making time-tables for a school. If all students and teachers could be scheduled independent of each other, this would be a simple task. However, for the purpose of scheduling they cannot be treated independently: students should share classes with other students, teachers should teach their own subjects to only a limited amount of classes, classes should be distributed over a limited amount of rooms, etc. Such relations make scheduling a complex task.

Courtois [1985] distinguishes variety (which he calls lack of homogeneity) of relations as the most important aspect of complexity. He gives many examples of dynamic relations, ranging from the complex ways in which termites interact when building a nest, to the difficulties in predicting failure of satellite communications due to collision of data packets. The area of science concerned with systems consisting of elements between which intricate relations exist is sometimes referred to as the "science of complex systems" [Complex Systems, 1987] or "sciences of complexity" [Stein, 1989]. Examples of such systems are the brains, evolutionary processes (and algorithms inspired by these, e.g. neural networks and genetic algorithms), and weather systems.

3. *Manyness*. As the number of elements or relations in a system increases, while other things remain unchanged, the system becomes more complex, provided it contains at least some variety in elements or relations. As the system contains more variety in elements or relations, the effect of a particular increase in the number of elements or relations on the complexity of the system becomes larger.

   This is because, as the number of elements or relations increases, more attention is required to comprehend and classify the sorts of the elements and relations and to treat each element or relation according to its sort (unless there is only no variety in elements and relations, in which case no attention is required to comprehend and classify the sort). The larger the number of sorts is, the more attention is required to comprehend and classify the sort of a single element or relation, so the greater the effect is of a particular increase in the number of elements or relations. We therefore say that manyness has a "multiplier" effect on complexity: the more element sorts and relation sorts a system comprises, the greater the effect of manyness is.

   We use two examples, based on [Simon, 1981], to illustrate this. The first example concerns the memorisation of a sequence of random numbers. It is very easy to quickly memorise a row of ten numbers, if only the number 1 occurs in this row (only one element sort). Manyness of numbers has hardly any effect on the complexity of memorisation, since it is nearly as easy to memorise a row of a hundred 1's. However, if the numbers 1 and 2 are allowed in the row (two element sorts), it may still be easy to quickly memorise a row of ten numbers, it will already be more difficult to quickly memorise a row of fifteen numbers, and it will be impossible to quickly memorise a row of a hundred numbers. And if any

number from 0 to 9 is allowed in the row (ten element sorts), most people will find it impossible to quickly memorise a row of ten numbers.

The second example concerns a library. To a librarian an increase in the number of books will hardly mean an increase in complexity, because the librarian views every book as just another element of the same sort (another "card in the catalogue"). However, to a reader who is interested in particular types of books, this increase in the number of books does mean an increase in complexity, since there are more books the reader has to evaluate to determine whether they are interesting or not.

Another, quite different, reason why a system may be hard to understand is the lack of knowledge by the conceiver of the purpose or elements of this system. For example, it is impossible to design a proper business process without knowledge of the customer requirements and of the people and technology that have to carry out the process. Even though the structuring techniques introduced in this chapter can aid in structuring this knowledge, their application is not a substitute for obtaining the required knowledge. We therefore expect all developers to know and understand the purpose of their development processes. We merely provide the conceptual tools to support these development processes.

2.3 Structuring techniques

A *structure* is a group of related elements. A system can thus be seen as a structure. As Simon [1981] notes, the amount to which the complexity of a system is controlled, critically depends upon the way the system is conceived and represented. *Structuring*, which is meant to make complexity manageable, is thus nothing more than attempting to conceive or represent a system in a manner that is as simple as possible for one's purpose. The structuring techniques presented in this section are meant to aid in doing so.

We first describe the structuring of systems from the distributed (or analytical) perspective, since we assume this perspective is the most familiar to most readers. We then describe the structuring of systems from the integrated (or synthetic) perspective, which is more appropriate in design processes. Finally, we discuss two structuring techniques, abstraction and refinement, in more detail.

2.3.1 Distributed perspective

When considering a system from the distributed perspective, one considers the system as a group of related elements. Depending on the type of system and the amount of detail these are, for example, molecules with Van der Waals-forces, or the related activities of people.

We distinguish two techniques to structure such related elements.

1. *Categorisation* (synonyms: classification, grouping, aggregation) is the grouping of elements and their relations. A resulting group of related elements is called a category. We consider a category to be a (composite) element again.

   Categorisation can aid in making the complexity of a system manageable, since it can be used to group elements of the same sort or strongly related elements. Thereby, elements of different sorts or weakly related elements are explicitly separated. In this way categorisation helps controlling the first and second aspects of complexity ("variety").

   An example of categorisation is the use of modules in a computer program. In a module a number of related program statements that together carry out a particular function are grouped. The use of modules makes a program more insightful, because program statements that are used to carry out different functions are explicitly separated.

   (In this example, categorisation of elements is applied on the basis of their contribution to a common function. As is shown in section 2.3.3, this form of categorisation forms the basis of composition. Another common reason for applying categorisation of elements is their possession of common properties. This form of categorisation forms the basis of generalisation.)

2. *Abstraction* is the omission of details that are irrelevant for a particular purpose.

   Abstraction can aid in making the complexity of a system manageable, since it can be used to reduce the number of elements and relations to be considered. In this way it helps controlling all of the identified aspects of complexity.

   An example of abstraction is the consideration of only the externally observable behaviour of a business process. From this viewpoint the detailed way in which, for example, a particular product is produced is omitted.

Categorisation and abstraction can be applied in combination: the application of categorisation, followed by abstraction is often very useful. This allows one first to categorise similar elements and their relations, and then to abstract from elements in each category. See Figure 2.1a, in which the relations between elements have been left out for the sake of simplicity.

For a proper understanding of abstraction it is important to realise that categorisation of elements, followed by abstraction from the elements in each category can also be viewed as abstraction *only*. The essential difference between the two forms of abstraction is *what one abstracts from*.

- As Figure 2.1a shows, we introduce categories of elements by means of categorisation. We can next abstract *from (the existence of) all elements* in each category. The result is the set of previously introduced categories, each of which is an element.

- As Figure 2.1b shows, by means abstraction *from the difference between certain elements*, we do not abstract from the existence of elements, but consider certain elements as indistinguishable. We thus consider a number of (lower-level) elements as a single (higher-level) element.

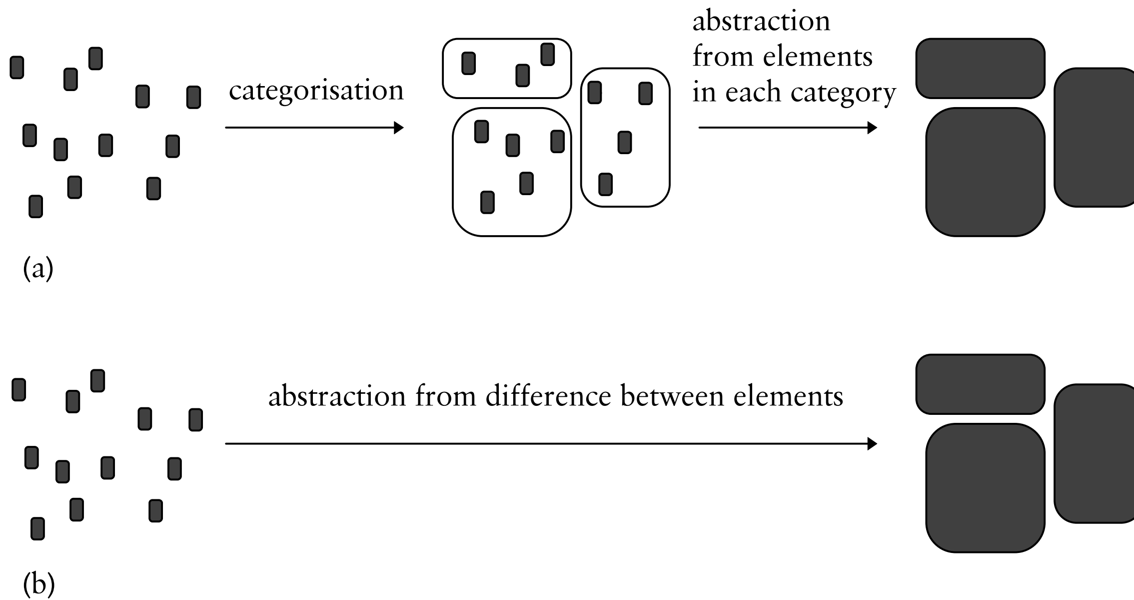


Figure 2.1 (a) Categorisation of elements, followed by abstraction from (existence of) elements in each category. (b) Abstraction from difference between elements. Each black rectangle represents an element; each open rectangle represents a category, which is also a (composite) element.

Abstraction from the difference between elements can be applied repeatedly, enabling further structuring. Take, for example, a company with a number of staff. We can abstract from the difference between all staff members involved in purchasing. We can than refer to "a" purchasing staff member. In the same way we can abstract from the difference between staff members involved in production and from the difference between staff members involved in sales.

Next we can abstract from the difference between the elements "purchasing staff member", "production staff member", and "sales staff member". We can then refer to "a" staff member of the company. Figure 2.2 depicts the resulting structure.

The structure resulting from repeated abstraction from the difference between elements is a hierarchical structure (a partial ordering). The tree structure of Figure 2.2, in which each element is part of only one category at the next higher abstraction level, is a special case of such a structure. If an element can be part of multiple categories at the next higher abstraction level (which would, for example, be the case if we introduced the categories "man" and "woman"), the resulting structure is not a tree anymore.

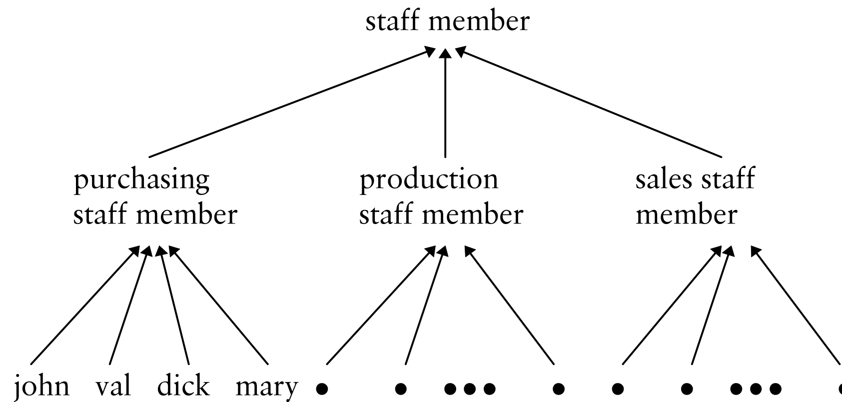


Figure 2.2 Tree structure resulting from repeated abstraction from difference between elements.

2.3.2 Integrated perspective

When considering a system from the integrated perspective, one considers the system as a unified whole. The objective of structuring is then the consideration of the system in more detail (in a development process: the prescription of how it should be implemented), while controlling its complexity. We distinguish two techniques for doing so.

1. *Refinement* (synonym: giving a concrete form) is the addition of details that are relevant for a particular purpose. It is the inverse of abstraction.

   An example is the transition of the consideration of a business process from the viewpoint of external observers (e.g. clients) to the viewpoint of internal observers (e.g. employees). In a consideration from the latter viewpoint e.g. the way in which a product is made becomes apparent.

   (In this example, refinement results in elements that are all part of a higher-level element. In section 2.3.3 this is called decomposition. Another form of refinement, in which the resulting elements possess all properties of the higher-level element, is called specialisation.)

2. *Individualisation* (synonyms: de-grouping, de-classification) is the consideration of elements and their relations without the category they are part of. It is the inverse of categorisation.

   An example of individualisation is a recent change in the Dutch pensions act. Formerly, when in a family only the husband had a paid job, only the husband built up pension claims. Nowadays, also the wife builds up pension claims. That means, for example, that after a divorce the ex-wife may claim a part of the pension. By the new pensions act women are thus treated more as an independent individual, rather than as a member of a group, the family.

In a development process individualisation is useful, because it allows us to consider and build the components of a system individually, without knowledge of the system as a whole. A car manufacturer may, for example, have the wheels and the chairs of a new car built by sub-contractors without giving away knowledge of the car's new engine.

Just as the application of categorisation followed by abstraction can be useful when considering a system from the distributed perspective, the application of refinement followed by individualisation can be useful when considering a system from the integrated perspective. This allows elements to be refined to categories with more detailed (possibly related) elements, after which these detailed elements can be considered without the categories they belong to. See Figure 2.3a in which the relations between elements have been left out for the sake of convenience.

Such refinement followed by individualisation can also be viewed as refinement *only*. The essential difference between the two forms of refinement is *what one refines to*.

- As Figure 2.3a shows, refinement of elements *to categories with (more detailed) elements* introduces elements that each belong to a category. By individualisation of these latter elements, they can next be considered without the categories they belong to.
- As Figure 2.3b shows, refinement of elements *to more detailed elements* directly introduces elements that do not belong to any category.

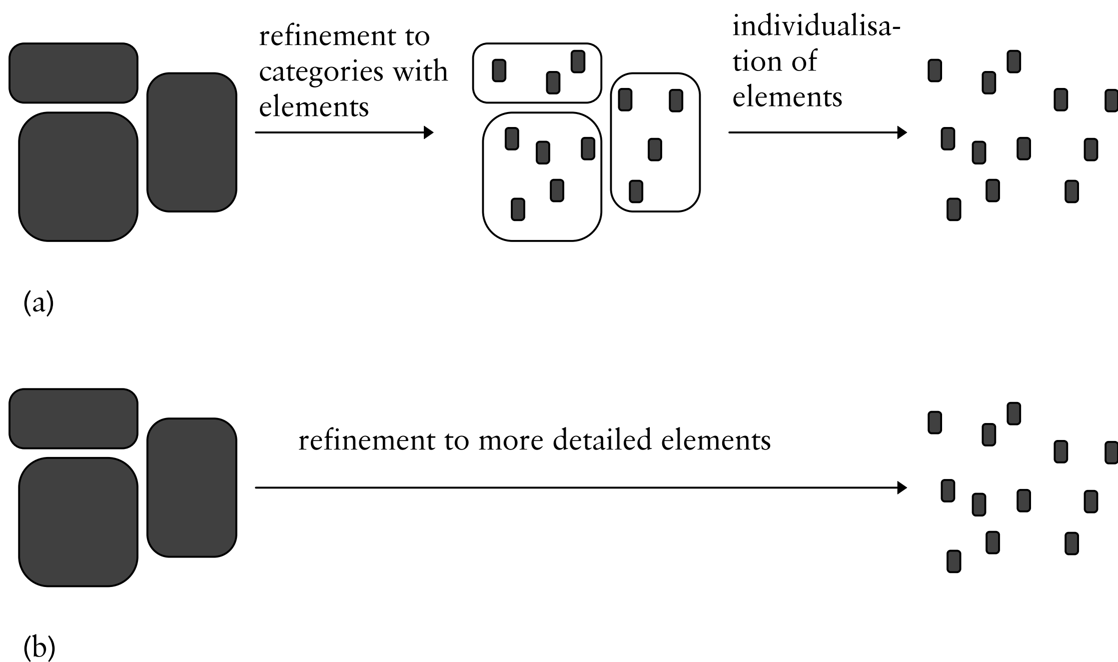


Figure 2.3 (a) Refinement of elements to categories with more detailed elements, followed by individualisation of these latter elements. (b) Refinement of elements to more detailed elements.

Refinement of elements to more detailed elements can be applied repeatedly as well. For example the element "staff member" of a company can be refined into the elements "purchasing staff member", "production staff member", and "sales staff member". Next, the element "purchasing staff member" can be refined to all individuals that perform this function. The same can be done for the elements "production staff member" and "sales staff member". The resulting structure is shown in Figure 2.2. This structure is, of course, the same as the one resulting from the last example of section 2.3.1, which dealt with structuring from a distributed perspective.

2.3.3 Forms of abstraction and refinement

Two forms of abstraction and refinement are so different in their application that we clearly distinguish them here. We make this distinction on the basis of the criterion for categorisation of elements.

1. *Composition* is the categorisation of elements on the basis of their contribution to common properties (such as a common function), followed by abstraction from the elements in the resulting categories. *Decomposition* is the inverse of composition.

   A bicycle, for example, can be viewed as the composition of the elements "frame", "handlebar", "front wheel", "back-wheel", etc. This is because these elements together enable the function "cycling".

   Each property of an element resulting from composition is thus a particular function of the properties of the constituent elements. Since a composition is only useful if it results in additional properties, not all properties of the composite element are present in the constituent elements. For example: one can cycle on a bicycle, but not on a handlebar.

   The relation between a constituent element and its corresponding composite element is sometimes called the "(is a) part of" relation (e.g. [Wirfs-Brock et al., 1990]). For example: a handlebar is a part of a bicycle.

2. *Generalisation* is the categorisation of elements on the basis of corresponding properties, followed by abstraction from the elements in the resulting categories. *Specialisation* is the inverse of generalisation.

   A tree, for example, can be viewed as the generalisation of the elements "oak", "beech", "spruce-fir", etc. This is not because oaks, beeches, and spruce-firs carry out a common function, but because they have so many common properties (such as the possession of a trunk, branches, and leafs or needles) that we wish to be able to abstract from the difference between the different kinds of trees.

   The properties of an element resulting from generalisation are an abstraction of the properties of the original elements. Thus, all properties of the resulting element are present in the original elements, whereas the opposite need not be true.

For example: an oak has all properties of a tree (such as the possession of a trunk and branches), but a tree does not have all properties of an oak (such as acorns).

The relation between a specific element and its corresponding generic element is sometimes called the “is a” relation or the “is of type” relation (e.g. [Wirfs-Brock et al., 1990]). For example: an oak is a tree.

2.4 Quality of systems

The application of structuring techniques does not guarantee high quality structures as a result. Although structuring techniques are meant to aid in making insightful structures, their application may equally well result in poor structures. In order to give developers more support in developing high quality structures, quality criteria are necessary for evaluating structures.

2.4.1 Cleanliness

An artificial system is developed to satisfy requirements set out by people such as principals and clients. Since developers never have unlimited funding, the satisfaction of requirements is subject to certain cost constraints. In practice, developers should try to optimise a complex (and usually implicit) balance between requirements and costs.

Therefore, it appears appropriate to evaluate the quality of a system on the basis of requirements satisfaction and costs [Van Sinderen, 1995]. This is indeed a common approach. For example, according to the ISO [1987b] quality is conformance to specified and “self-evident” requirements. Kerklaan and Knaapen [1981, p. 137] define quality as “the extent to which a product or service conforms to customer expectations [...] against minimal costs”.

Although such quality criteria can be highly useful when evaluating a particular system, they are of less use as generic quality criteria that should be applicable to the wide range of structures considered in this thesis (which include systems development processes, business processes, and designs thereof). This is because requirements and cost constraints may vary enormously over different structures. Moreover, it is often not possible to evaluate how well a structure satisfies requirements and cost constraints. For example, a design specifies a large class of implementations; the requirements satisfaction and costs of the various implementations are usually different. To keep the quality criteria applicable to all of these structures, we need more general criteria that concern the conceptual integrity of a structure. Cleanliness is such a criterion.

Blaauw & Brooks [1980] define a clean architecture as one that is straightforward to use. A clean architecture presents its functions to the user in a comprehensible man-

ner. Generalising this, a *clean* structure is one that is conceived or represented in a straightforward manner and that is thereby (as) easy (as possible) to understand.

The cleanliness criteria presented in sections 2.4.2 to 2.4.5 are basically aesthetic criteria. Therefore, they are to some extent subjective, not always precise, and in application sometimes even contradictive. Nevertheless, they can be very useful in evaluating structures. Blaauw & Brooks [1980] even argue that "good aesthetics yield good economics" and Van Sinderen [1995] and Roman [1985] stress the importance of cleanliness in meeting the requirements of various types of system users. We conclude that a clean structure is not only a goal in itself, but also provides insight that makes it far easier to satisfy requirements and cost constraints.

The cleanliness criteria presented here have, in different forms, already been applied to various types of structures. For example, in [Blaauw & Brooks, 1980] they are used as principles for good computer architectures, in [Vissers et al., 1988] they are used as criteria for the evaluation of specification styles (see chapter 5), and in [Van Sinderen, 1995] they are used as criteria for the evaluation of application protocols. Roman [1985] describes some quality criteria for user requirements. Auramäki et al. [1988] describe some quality criteria for office information system models.

2.4.2 Consistency

A *consistent* system possesses a regularity that enables the conceiver to anticipate aspects of the system once other aspects are known. A consistent system thus confirms (and thereby encourages) expectations time and time again when it is being conceived. A developer should pursue consistency by not making development decisions that conflict with earlier ones.

An example of inconsistency is formed by the different administrative coding schemes for objects, such as clients and products, in many large organisations. Autonomy of the different organisational units has led to each of them developing their own coding schemes. As long as these units exchange little information, this is not a great problem. However, if, for example, client and order information from the sales department, the marketing department, and the accounting department have to be combined for a promotion campaign, the inconsistencies quickly become apparent. In order to enforce a more consistent information storage and presentation, many companies have introduced so-called enterprise data models (see e.g. [Hennessy et al., 1994]).

2.4.3 Orthogonality

Orthogonality means that independent aspects of a system are kept separate. It allows for the separate conception of the independent aspects, thereby simplifying the understanding of the system as a whole. It also supports maintainability, since changes made to one aspect do not affect other aspects. In a development process, the pursuit of orthogonality requires separation of development concerns. The *com-*

plement of orthogonality, which says that related aspects should not be separated, is also a quality criterion.

The importance of orthogonality can be illustrated by the problems faced by some organisations that carry out legal procedures, such as pension funds or social security institutions. In order to improve their services, many of these organisations try to organise their business processes in business units that each provide all services to a particular group of clients. This should be beneficial for the clients, since they can obtain all services from a single access point. Unfortunately, business processes structured in this way are hard to maintain. Since every business unit carries out a large amount of legal procedures, a change in one legal procedure (often imposed by the government) usually requires changes in many business units, which is expensive and hard to accomplish. The business units are thus not orthogonal with respect to changes in legal procedures. A different structure, in which each legal procedure is carried out by a single business unit is far easier to maintain.

Since the related elements of a system form a unified whole, it is never possible to decompose a system into aspects that are fully independent with respect to all relations. For example, the business units that each carry out a their own distinct set of legal procedures may be mutually independent with respect to changes in these procedures, but they share the same customers.

The criterion of loose coupling is a criterion of practical importance that is more relaxed than orthogonality. A structure consists of *loosely coupled* categories if the relations between elements in different categories are weak and the relations between elements in the same category are strong. A structure whose elements can be categorised in loosely coupled categories is called near-decomposable [Simon, 1962].

In a system of loosely coupled parts, each part may operate largely independent of other parts, since only the interactions of the parts are relevant for the system as a whole. The internal aspects of a part are thus hidden from its environment, a quality that is called *encapsulation*. Encapsulation was first described as an important criterion in software engineering in [Parnas, 1972] and has become a key paradigm of object-oriented systems development [Booch, 1994].

2.4.4 Propriety

A system is *proper* to its purpose when 1) all of its aspects are relevant to this purpose and 2) all aspects relevant to the purpose are included in the system.

Propriety implies *parsimony*: aspects not relevant to the purpose of the system should not be included. This eases the understanding of a system, since superfluous aspects need not be comprehended. Propriety also implies c*ompleteness*: all aspects relevant to the purpose should be included in the system.

Lack of parsimony is an important reason why many large companies carry out "downsizing" operations, the shedding of business functions. Such companies realise

they suffer from at least two aspects of complexity: a large amount and a large variety of business functions [Beinhocker, 1997]. This makes these companies hard to manage. As a part of their restructuring process, they shed some of their business functions. An important criterion for keeping or shedding business functions is their contribution to the company's core competence, the processes the company wants to be good at. The core competence is thus the propriety criterion [Coyne et al., 1997].

Completeness is an important reason why many companies that pursue their core competence not only shed business functions, but also acquire other companies, or co-operate with them. For example, many telecommunications and airline companies nowadays work with other companies from the same sector in order to provide customers with an as large as possible coverage and an as good as possible service [Coyne et al., 1997].

2.4.5 Generality

A system is *general* when it can be used for many purposes. Generality eases the understanding of a system, since it separates the inherent properties of an aspect from the different ways it can be used. Generality also results in conciseness, which further enhances comprehensibility, since a representation of different instances of the system only requires the representation of the properties specific to each of the instances. Finally, generality supports flexibility and adaptability, since it allows systems to be used in different circumstances.

The so-called "year 2000 problem" is an illustration of the importance of generality. In the past computer programmers often used two digits to encode years, using the year 1900 as an offset. This encoding works fine in the 1900's, but does not allow the representation of the years 2000 and beyond. The adaptation required due to this lack of generality is estimated to cost the US Department of Defence alone between $358 million and $3 billion [Horn, 1996].

We distinguish between two types of generality[1]. We speak of *static generality* when an aspect of a system can be used for multiple purposes in the same system. An example is formed by procedures in a computer program, which can be called at different places in the program, possibly with different values. Static generality aids in making a system concise.

We speak of *dynamic generality* when an aspect of a system can be used for purposes for which it is not used in this system. Two types of dynamic generality may be distinguished. *Re-usability* is the property of an aspect of a system to be useable in other systems. Re-usability obviously aids in reducing the costs of development processes and is seen as an important factor in reducing the costs of software engineering

1 This distinction is inspired by Pirsig [1991], who distinguishes between static and dynamic quality.

[Wirfs-Brock et al., 1990]. *Open-endedness* is the property of an aspect of a system to be useable for new purposes in the same system. When an open-ended system is extended with extra aspects, no modifications are required to the existing aspects of the system. Open-endedness thus eases adaptability.

2.5 Role of structuring in development processes

2.5.1 Structuring as architecting

A system development process can be viewed as a process in which increasingly more functionality and structure is introduced in a design of this system, until it can finally be implemented. Once a certain structure has been introduced, it is beneficial to preserve this structure when the design is further refined. This has the following reasons.

- If the original structure is not preserved, an alternative structure has to be designed. This requires extra time and effort.
- If the refined design has a structure that deviates much from the original structure, it becomes more difficult to understand the relation between the original design and the refined design. This complicates, for example, the verification whether the refined design is a correct implementation of the original design.

This *principle of structure preservation* implies that the first development decisions are the most important for the structure of a system: these first development decisions determine the structure of the entire system, whereas later development decisions only have an impact on elements within this structure. The more development decisions have been taken, the less their impact is on the structure of the whole.

This is why “the” structure of a system is usually understood to be the *main structure* of the system and not the way in which all of its detailed elements relate to each other. For example, when one refers to the structure of a building, one refers to the way in which the building is sub-divided into, e.g., apartments and rooms, and the ways in which these are related, e.g., by passages, staircases, and elevators. One does usually not refer to the way in which a single room is furnished, although this can, strictly spoken, be considered as structure as well.

In this thesis we also use the term structure to refer to the main or underlying structure of systems. Rechtin [1992, p. 66] defines *architecture* as “the underlying structure of things”.[2] Our approach to systems development, in which we emphasise the

---

2 Different interpretations of the term “architecture” exist. For example, Blaauw & Brooks [1980, p. 1] define the architecture of a system as “the functional appearance of the system to its immediate user”. Blaauw & Brooks thus use the term to refer to a specific structure: the structure of the system as observed by the user.

structuring of systems, can also be characterised as an *architectural approach*. In order to emphasise this, we sometimes use the term architectural structuring.

2.5.2 Structuring in this thesis

Architectural concepts

The basic architectural concepts model elementary and common characteristics of system implementations. These concepts, together with their combination rules, determine the set of models that can be developed. The basic architectural concepts thus also determine which structures can be conceived and represented and are therefore the basis of the structuring of systems under development.

Therefore, the basic architectural concepts must satisfy the quality criteria set out in section 2.4. A clean set of basic architectural concepts (called a “design model” in [Van Sinderen et al., 1995]) should comprise a minimal number of generic and orthogonal architectural concepts that cover the entire domain of systems one aims at, thus enabling the development of structures that provide maximal insight.

Horizontal and vertical structuring of the development process

Given a system that consists of a hierarchy of sub-systems, Simon [1962] defines horizontal separation as the segregation of sub-systems at the same hierarchic level and vertical separation as the segregation of different hierarchic levels.

After Simon, we term the structuring of a single model, in which different aspects of the system at the same abstraction level are separated, *horizontal structuring* of the development process. The modelling styles presented in this thesis are thus structuring techniques for horizontal structuring.

We term the structuring of a development process in terms of related models made at distinct abstraction levels *vertical structuring*. The abstraction levels presented in this thesis are thus structuring techniques for vertical structuring.

Development strategies

The combination of horizontal and vertical structuring leads to the structuring of a development process as represented in Figure 2.4 (see also [Essink, 1986]). Each square in the figure represents a group of one or more development decisions that regard one aspect of the system at one abstraction level.

During a development process these development decisions have to be taken in a particular order. A development strategy prescribes this order.

aspect / abstraction level	1	2	...	m
n				
..				
2				
1				

Figure 2.4 Combination of horizontal and vertical structuring. Each square represent one or more development decisions that regard one aspect at one abstraction level.

2.6 Conclusions

Although the importance of structuring in system development is acknowledged in the literature, it appears that the role of structuring techniques in the control of complexity has been poorly addressed. Our analysis of complexity led to the identification of three important aspects of complexity: variety of elements, variety of relations, and manyness. We showed how the following general structuring techniques aid in the control of these aspects of complexity: categorisation and individualisation, abstraction and refinement, and the forms of abstraction and refinement: composition and decomposition, and generalisation and specialisation.

Since the application of structuring techniques does not guarantee the development of high quality structures, we presented some quality criteria for structures.

In order to clarify the role of the structuring techniques for distributed systems discussed in this thesis, we discussed the role of structuring in system development, and introduced the structuring techniques for distributed systems in terms of the discussed general structuring techniques.

3

Basic architectural concepts

3.1 Introduction

3.1.1 Motivation

This chapter introduces basic architectural concepts and their combination rules. *Architectural concepts* are abstractions of system elements and model required characteristics of systems. They are the building blocks available to developers for modelling systems. The *basic* architectural concepts form the elementary building blocks. The choice of basic architectural concepts thus determines the system models that can be developed. Since a model of a system under design serves as a prescription for the implementation of this system, the basic architectural concepts also determine the system implementations that can be designed.

Synonyms of "basic architectural concept" are "basic design concept" [Ferreira Pires, 1994] and "design construct" [Wand & Weber, 1993]. The complete set of basic architectural concepts and their combination rules offered by a development method is sometimes called an "architectural model" or a "design model" [Ferreira Pires, 1994].

3.1.2 Structure

Section 3.2 introduces entities, which are carriers of behaviour, and some related concepts. The remaining sections introduce concepts for the modelling of behaviours, which are sets of related actions. Section 3.3 introduces actions. Section 3.4 discusses the relations between two actions. Section 3.5 discusses the relations between multiple actions. Section 3.6, finally, discusses the structuring of behaviours as compositions of sub-behaviours.

Throughout the sections two notations are presented for the representation of the introduced concepts: a graphical notation and a textual notation. Since we concentrate on the concepts themselves, both notations are informal.

This chapter elaborates on [Ferreira Pires, 1994]. Part of the work reported on in this chapter was carried out together with Quartel. Some of the results of our joint work, on which sections 3.2 and 3.3, and a part of section 3.4 are based, have been published as [Franken et al., 1996a, chapter 5]. Some of the concepts introduced in this chapter have been described specifically for business processes in [Franken et al., 1996b] and [Franken & De Weger, 1997]. A formal semantics for the architectural concepts introduced in this chapter is given in [Quartel, 1998].

3.2 Entities and (inter)action points

3.2.1 Entities

An *entity* is a carrier of properties. Some synonyms of entity are “object”, “unit”, and “thing”.

A statue is an example of an entity. It is an entity made of a particular material, in a particular form, with a particular colour. The material, the form, and the colour are all properties of the statue. Another example is the sales department of a company. An important property of this sales department is its sales function. Without properties it would be meaningless to talk about a statue or a sales department.

This chapter argues that the relevant properties of a business process (including its parts, such as people, machines, and telematics systems) constitute its behaviour. For the purpose of this thesis we therefore consider an entity as *a carrier of behaviour*. An entity thus models a logical or physical system or system part that carries out behaviour.

The entity concept is important in systems development, because the concept makes it possible to structure systems and their models as compositions of entities, which facilitates insight. For example, the assembly process of cars can be quite complex, since many cars can be present on a multitude of assembly lines, while different people perform different operations on each car. This process can be made comprehensible by structuring it, for example, in work groups that each carry out a small part of all operations each car undergoes: one work group for assembling the wheels, another one for assembling the doors, etc.

When designing a new business process, structuring it in terms of entities is also important because the design serves as a prescription for implementation. Thus, if the activities specified in a detailed design are to be carried out by, for example, people or computers, the design should be structured in terms of such entities.

3.2.2 Interaction points

An artificial system is developed to perform a certain function in its environment. In order to perform such a function, the system should interact with its environment. We call a (logical or physical) location at which an entity can interact with its environment an *interaction point*. A synonym of "interaction point" is "access point" [Vissers et al., 1994]. Since an interaction point is a carrier of interactions, it can be viewed as an (abstract) entity.

Two or more entities can only interact at their shared interaction points. Therefore, interaction points allow us to model a system in terms of solely the locations where the environment can interact with the system, thus shielding all other (internal) locations at which activities of the system take place. Hence, an entity is delimited by its interaction points.

A computer user, for example, interacts with the computer via the keyboard (typing), the mouse (rolling and clicking) and the screen (watching), but has no direct access to the computer's processor. A client of a company may interact with the company at, for example, a counter (e.g. buying a product), but has no direct access to the credit administration. The latter is undesirable, since it would allow creditors to wipe out their debts.

Figure 3.1a represents two entities with their common interaction point. We represent an entity as a black rectangle with round corners. We represent an interaction point of a number of entities as an oval that overlaps the representations of these entities. Figure 3.1b gives an alternative representation.

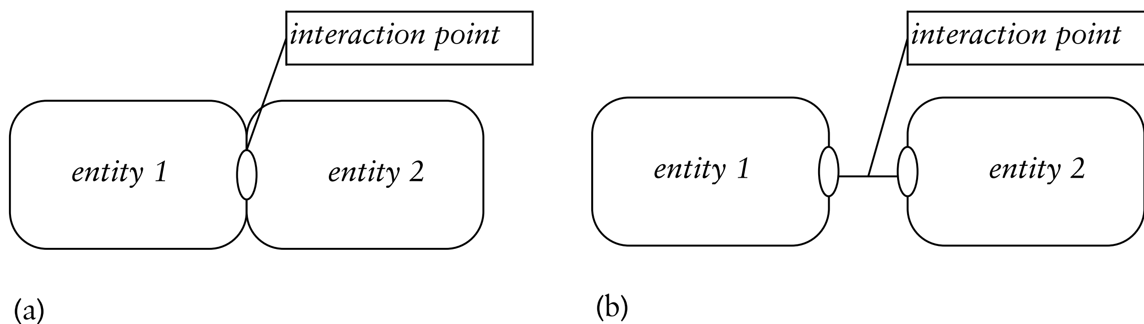


Figure 3.1 Two entities and their interaction point.

3.2.3 Entity (de)composition

An entity may consist of other, lower-level, entities. By abstracting from the difference between these lower-level entities, the higher-level entity is viewed as an integrated whole. In turn, the higher-level entity can (often repeatedly) be refined into lower-level entities.

When one abstracts from the difference between entities, one does not necessarily abstract from their interaction points. If one does not do so, the result of the abstrac-

tion is an entity with one or more internal interaction points, called action points. See Figure 3.2. An *action point* allows one to specify where internal activities of an entity take place, without specifying which sub-entities are involved. An example of an internal action point is a meeting room of a company: it is a location in a company at which interaction takes place between sub-entities of the company. It is, however, not necessary to specify which sub-entities interact (e.g. the board members, or the CEO and his secretary).

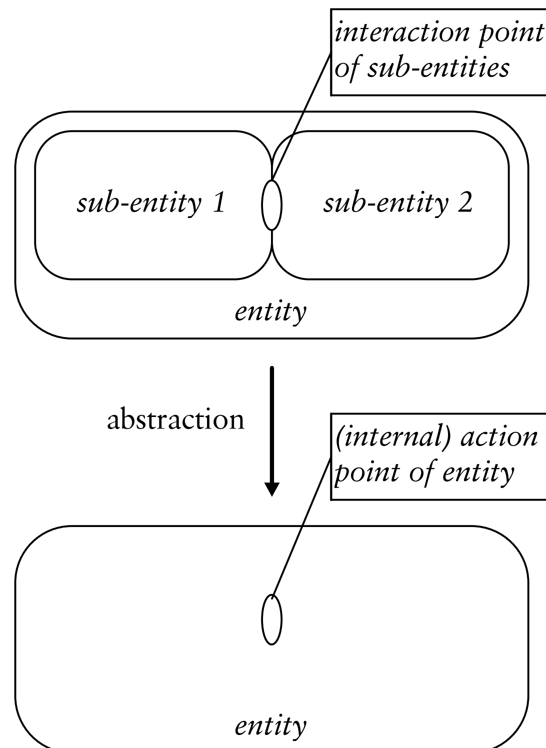


Figure 3.2 Abstraction from difference between entities, resulting in higher-level entity with (internal) action point.

We use the term *action point* as the generic term for both interaction points and internal action points.

3.3 Actions and interactions

3.3.1 Actions

Actions as abstractions of activities

An *activity* is something in the real world that happens (synonyms: occurs, takes place, is carried out). Examples of activities are the placement of an order, the administration of an insurance claim, and the writing of a thesis.

An *action* is an abstraction of an activity.

This definition of an action does not tell which aspects of an activity are ignored in an action, and consequently it does not tell which aspects of an activity an action models. Our choice of aspects of an activity that are ignored by an action is based on our use of abstraction levels in the development process: at each level we model *what* happens (including which actions may occur), whereas only at a lower abstraction level we model *how* this happens (including how each higher-level action occurs: in terms of one or more related lower-level actions—see below). Thus, an action models the contribution of an activity to a behaviour. We term this contribution the *result* of the activity.

Summarising, *an action models the result of an activity*. We say that an action *occurs* when the corresponding activity delivers its result. Since every activity is unique, an action may occur only once.

We give two examples:

- Consider an artist that creates a work of art for a client. The client gives the artist complete freedom, so from the point of view of the customer it is only relevant that the work is delivered. It is not relevant what the work looks like, when it is produced, etc. The action that is the abstraction of the activity of creating the work of art thus only models the delivery of the work of art. Once the action has occurred, the work of art has been created and delivered.
- Consider a car company that produces three types of cars. From the point of view of a customer it is not only relevant that the company produces a car, but also which type of car is produced. It is therefore not sufficient to model the production of "a" car as a single action. Three different actions need to be distinguished, where each of the actions represents the production of a distinct type of car. (This consideration may result into large models if, for example, the company produces hundreds of types of cars. Therefore, we introduce concepts to model such situations concisely in chapter 4.)

We give some examples of (aspects of) results that actions may model:

- some good, e.g., a work of art;
- information, e.g., the announcement of a new brand of washing powder;
- some service, e.g., the nursing of a patient;
- costs, e.g., the costs of a new car;
- the time at which an activity finishes, e.g., the time at which a person arrives at his job
- the location at which an activity occurs, e.g., the location at which a salesman sells his goods.

Atomicity of actions

An action is the most abstract model of an activity: it models only the result of an activity. As a consequence, an action cannot be considered as a composition of other actions at the abstraction level at which it is defined. This is called the *atomicity* property of actions.

As an example, consider the activity of buying a house. Obviously, this activity takes time and involves, e.g., visiting an estate-agent, examining the house, negotiating the price, etc. However, if it is only relevant that the house is bought, the action of buying the house only models this result. The action occurs at the moment the deed of purchase is finished, which is a single, indivisible point of time.

The atomicity property of actions is not to be understood as if actions cannot be decomposed into sub-actions. For example, it is perfectly allowed to decompose the action of buying the house into actions of visiting estate agents, examining the house, etc. However, in that case one considers the action at a lower abstraction level.

The atomicity of actions has the following consequences:

- An action either occurs completely, or does not occur at all. In contrast to activities, actions cannot occur partly.

  For example, consider the activity of applying for an insurance. If during this activity the client changes his mind and decides not to fill out the application form, it is possible that no relevant result is delivered. This implies that the corresponding application action does not occur.

- If an activity delivers multiple results at multiple points of time and the difference between the points of time is relevant (for example, because the first result is used before the other results are delivered), the activity has to be modelled as multiple actions.

  For example, consider the process of a postman who delivers his mail all day long. This may be considered a single activity. However, if it is relevant that people in one district get their mail early in the morning, whereas people in another district get their mail in the afternoon, the activity has to be modelled as multiple actions.

Representation

We represent an action graphically as a circle. In order to be able to refer to the action, we often augment it with a meaningful name, which we write inside or next to the circle that represents the action. See Figure 3.3. In the textual notation we represent an action by means of this name only. (By giving each action a unique name, it can be identified uniquely. This naming of actions, however, is not always necessary for identifying them: in the graphical notation actions can be uniquely identified by, e.g., their position on paper.)

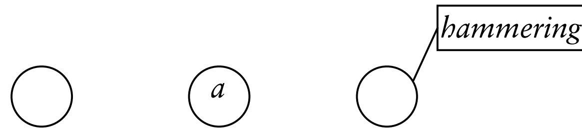


Figure 3.3 Some representations of actions.

3.3.2 Interactions

An action may be carried out by a single entity. It then takes place at an internal action point of this entity. An action may, however, also be jointly carried out by multiple entities. Such an action takes place at an interaction point of these entities.

The difference between actions carried out by a single entity and actions carried out by multiple entities is relevant, since in the latter case an entity is always dependent on the co-operation with other entities in carrying out the action. We therefore give a distinct name to actions of the latter type.

An *interaction* is an action carried out by multiple entities. It models a common activity of these entities. An interaction may only occur if all of the involved entities co-operate.

Examples of interactions are the transfer of a product (e.g., from a seller to a buyer), the shaking of hands (usually by two persons), and the election of a parliament (usually by the citizens of a country). Even the posting of a letter is an interaction, since it not only requires the co-operation of the person posting the letter, but also of the mail box (or, more abstractly, the mail company).

Representation

Each contribution to an interaction is represented graphically as a half circle. Half circles that represent contributions to the same interaction are connected and have the same names. These names may be underlined to distinguish them from names of actions. See Figure 3.4. In the textual notation we represent a contribution to an interaction by means of this name only.

3.3.3 Abstraction and refinement

Action (de)composition

An action only models the result of an activity. In order to define how this result is produced, the action should be modelled in more detail. Such a model can be obtained by decomposing the action into a number of related sub-actions.

As an example, consider the production of a car. At a high abstraction level, this may be viewed as a single action. However, at a lower abstraction level this may be viewed as a set of related actions, e.g., as the purchasing of parts, followed by the assembly of the parts, followed by the dyeing of the car. This is represented in Figure

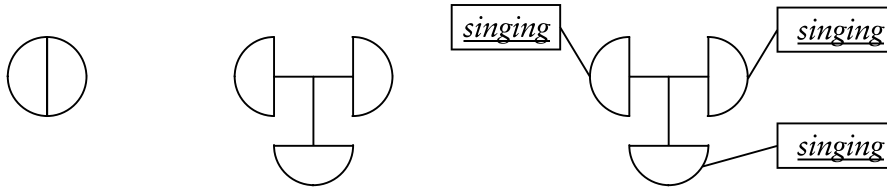


Figure 3.4 Some representations of interactions.

3.5, in which the arrows between actions represent an "enabling" relation, which is explained in more detail below.

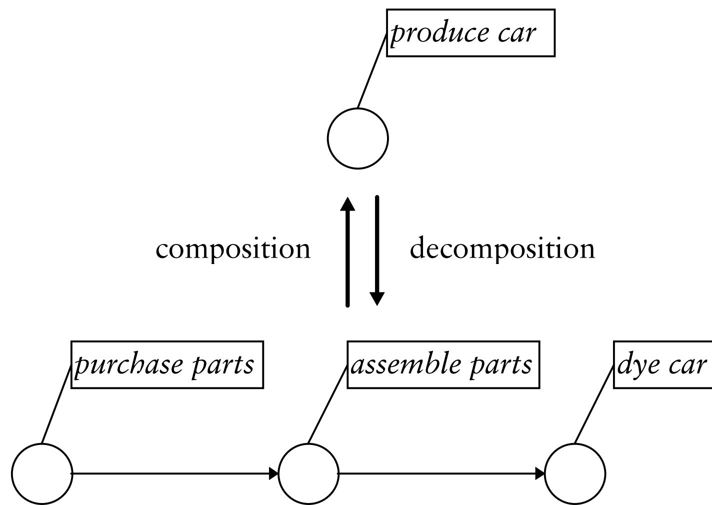


Figure 3.5 Example of action (de)composition.

Action distribution

A second type of action refinement is the replacement of an action by an interaction. This refinement is useful, since an action can be used to model an interaction from the integrated perspective, i.e. while abstracting from the different contributions of the involved entities. (Conversely, an interaction can be used to model an action from the distributed perspective.) We call the replacement of an action by an interaction *action distribution*, since the contributions to the action are distributed over the involved entities. Figure 3.6 illustrates action distribution.

Chapter 6 discusses action refinement in more detail.

3.3.4 Coupling of actions and entities

In order to specify which entities carry out which actions, the location of each action (which is an aspect of the result modelled by the action) should be defined as an action point in the entity domain. In this way each action is uniquely coupled to the entity(ies) that carry it out.

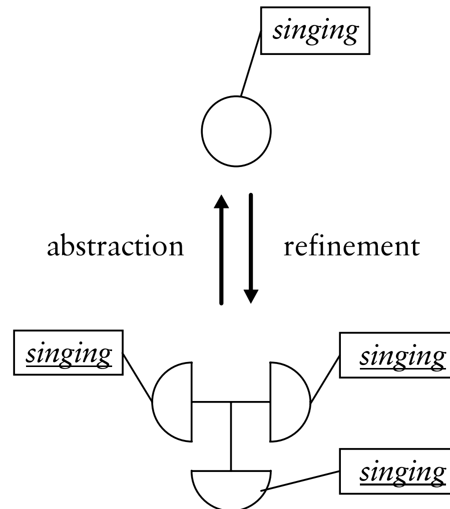


Figure 3.6 Example of action distribution.

Figure 3.7 illustrates the coupling of actions to entities. It models a system consisting of three entities: a *supplier*, a trading *company*, and a *client*. Three (inter)action points are recognised: one shared by the supplier and the company, one internal action point of the company, and one shared by the company and the client. The behaviour of the system consists of the interaction *purchasing* of the supplier and the company, followed by the internal action *storage* of the company, followed by the interaction *sales* of the company and the client. The coupling of actions and entities is carried out by specifying an (inter)action point for each action. This coupling is represented in the figure by the dotted arrows.

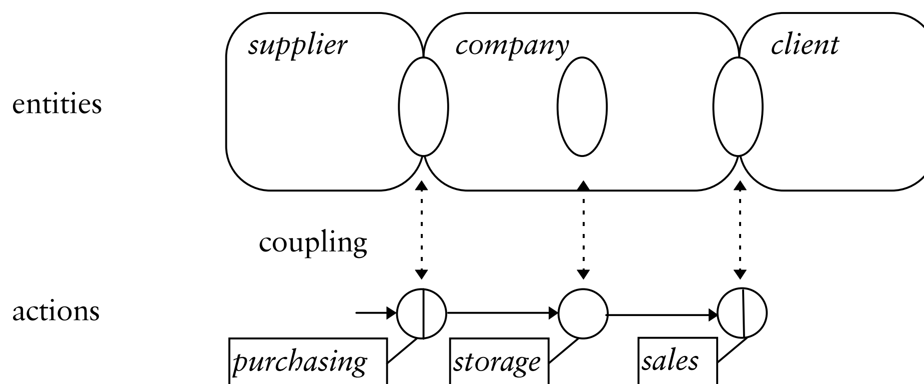


Figure 3.7 Coupling of actions and entities.

3.4 Behaviours containing two actions

A *behaviour* is a set of related actions. This section focuses on behaviours containing only two actions.

3.4.1 Causality relations

A causality relation of an action defines the conditions for this action to occur [Ferreira Pires, 1994]. A causality relation of an action consists of:

- this action, which we term the *result action*;
- a *causality condition*, which defines how the occurrence of the result action depends on the occurrences or non-occurrences of other actions in the behaviour;
- a *probability attribute*, which defines the probability of the occurrence of the result action when the causality condition is satisfied.

We say that the result action is enabled, or that it may occur, if and only if the causality condition is satisfied.

3.4.2 Elementary causality relations

Dependence and independence of action occurrences

We consider a behaviour of two actions *a* and *b*. The following alternatives can be identified:

1. the occurrence of *a* depends on the occurrence or non-occurrence of *b*;
2. the occurrence of *a* does not depend on the occurrence or non-occurrence of *b*.

In the first case there are again two alternatives:

1a. the occurrence of *a* depends on the occurrence of *b*, i.e. *b* must occur (at some time) for *a* to occur;

1b. the occurrence of *a* depends on the non-occurrence of *b*, i.e. *b* may never occur for *a* to occur.

In the second case we say that the occurrence of *a* is *independent* of the occurrence of *b*. Since the occurrence of *a* cannot depend on anything else than the occurrence of *b* in a behaviour of two actions, there are again two alternatives.

2a. *a* may never occur;

2b. *a* may occur at any time.

We do not know of any use in a development process for the definition of actions that may never occur. We therefore do not consider actions that may never occur and we dismiss the alternative 2a. Therefore, in case the occurrence of *a* is independent of the occurrence of *b*, *a* may occur at any time. We call actions that may occur at any time *initial actions* of a behaviour.

If the occurrence of *a* is independent of the occurrence of *b* and the occurrence of *b* is independent of the occurrence of *a*, we call *a* and *b (mutually) independent*.

If no confusion arises, we may, for conciseness reasons, say that an action depends on another action, rather than that the occurrence of an action depends on the occurrence of another action.

Temporal ordering

In case of alternative 1a, in which the occurrence of *a* depends on the occurrence of *b*, *b* must occur at some time for *a* to occur. We distinguish the following alternatives as a refinement of this dependence:

1ai. *b* must have occurred before *a* occurs;

1aii. *b* must occur at the same time as *a* occurs;

1aiii. *b* must occur after *a* occurs.

We dismiss alternative 1aiii, since our architectural concepts are abstractions of elements of system implementations. The dependence of the occurrence of an action on the occurrence of a future action cannot be implemented.

For a similar reason we do not allow either the modelling that the occurrence of an action is dependent on the *non*-occurrence of a future action. Thus, alternative 1b, which said that

> the occurrence of *a* depends on the non-occurrence of *b*, i.e. *b* may never occur for *a* to occur,

is replaced by

1b. the occurrence of *a* depends on the non-occurrence of *b* before or at the same time of *a*, i.e. *b* may neither occur before *a* nor at the same time as *a* for *a* to occur (but *b* may occur after *a* has occurred).

Elementary causality conditions

The above reasoning leads us to distinguish the following elementary causality conditions which should be satisfied for action *a* to occur:

- A condition that is always satisfied during the considered time span.
  This is called a *start condition*, which is the condition of any initial action. A start condition is represented as *start*.
  The following causality relation represents that the start condition is the causality condition for a:

  $start \rightarrow a.$

  Figure 3.8a represents this graphically.

- *b* occurs before *a*.
  This is called an *enabling condition*, since the occurrence of *b* enables the occurrence of *a*. This condition is represented as *b*. *b* is called an *enabling action* of *a*. The following causality relation represents that *b* is the enabling condition of *a*:

$b \rightarrow a.$

Figure 3.8b represents this graphically.

- *b* occurs at same time as *a*.
  This is called a *synchronisation condition*, since *a* may only occur if it synchronises in time with *b*. This condition is represented as $=b$. *b* is called a *synchronisation action* of *a*.
  The following causality relation represents that *b* is the synchronisation condition for *a*:

  $=b \rightarrow a.$

  Figure 3.8c represents this graphically.

- *b* has not occurred before *a*, nor occurs at the same time as *a*.
  This is called a *disabling condition*, since the occurrence of *b* disables the occurrence of *a* if *a* has not yet occurred when *b* occurs. This condition is represented as $\neg b$. *b* is called a *disabling action* of *a*.
  The following causality relation represents that *b* is the disabling condition for *a*:

  $\neg b \rightarrow a.$

  Figure 3.8d represents this graphically.

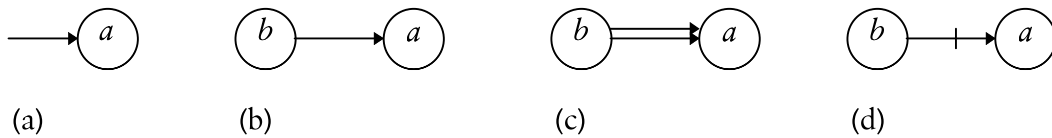


Figure 3.8 Graphical representations of (a) *start* $\rightarrow a$, (b) $b \rightarrow a$, (c) $=b \rightarrow a$, (d) $\neg b \rightarrow a$.

These elementary causality relations allow us to model some simple behaviours.

Figure 3.9a specifies a behaviour with two independent actions. In the figure action *a* can, for example, represent the arrival of an order from one client, and action *b* the incoming of an order from another client.

Figure 3.9b specifies a behaviour with two actions which may occur in sequence. Action *a* can, for example, represent the purchasing of raw materials, and action *b* the processing of these.

Figure 3.9c specifies a behaviour with two actions that disable each other (“choice”). Action *a* can, for example, represent the travelling to work by car, and action *b* the travelling to work by bicycle.

Figure 3.9d specifies a behaviour with two synchronising actions. Action *a* can, for example, represent the output of some video fragment by a multi-media device, and action *b* the output of a corresponding audio fragment.

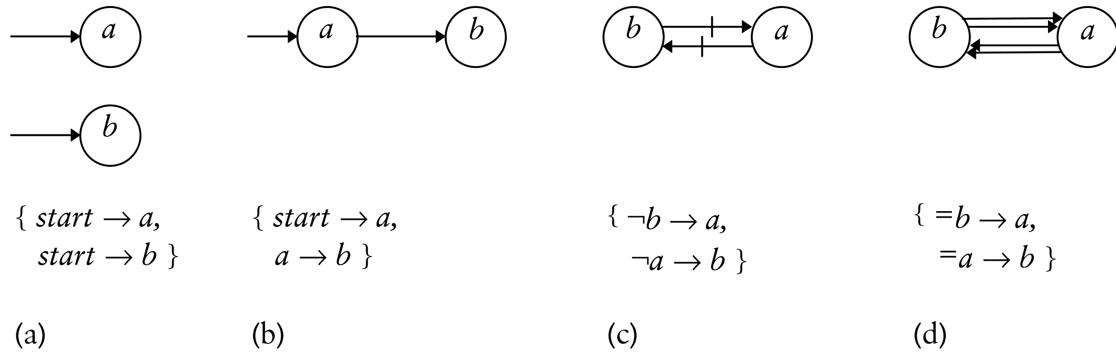


Figure 3.9 Some simple behaviours.

Uncertainty

Once the causality condition of a result action is satisfied, this action may occur. A probability attribute defines the probability of its occurrence.

In this chapter we consider an abstract form of probability: uncertainty. In the case of *uncertainty* we only distinguish between two probability values (must and may). Other types of probability are integral probability, and stochastic probability [Quartel, 1998]. In the case of *integral probability*, the probability of action occurrences is expressed as a real number in the range from 0 to 1. S*tochastic probability* considers the distribution of the probability of action occurrences over time as well.

The *uncertainty* of an action with respect to one of its elementary causality conditions defines that either the action must occur or the action may occur if the elementary causality condition is satisfied.

- If the action *must* occur, it always occurs (i.e., the action occurs with an integral probability of 1) if the elementary causality condition is satisfied.
  This is represented by an exclamation mark in subscript after the elementary causality condition. For example,

  $$b_! \rightarrow a$$

  represents that *a* must occur if *b* has occurred. Figure 3.10a represents this graphically.

- If the action *may* occur, it does not always occur (i.e. the action occurs with an integral probability smaller than 1) if the elementary causality condition is satisfied.
  This is represented by a question mark in subscript after the elementary causality condition. For example,

  $$\neg b_? \rightarrow a$$

  represents that *a* may occur if *b* has neither occurred before, nor occurs at the same time as *a*. Figure 3.10b represents this graphically.

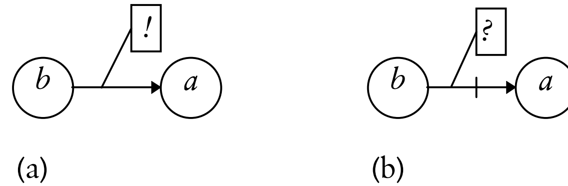


Figure 3.10 Graphical representations of causality relations.

A default interpretation relieves us from the burden of representing the uncertainty value of every elementary causality condition. If no uncertainty value is represented, the intended uncertainty value is *must*. The choice for must as the default interpretation is a pragmatic one, based on our experience that in practice must uncertainty values are more common than may values.

3.4.3 Disjunctions of elementary causality conditions

The elementary causality relations of section 3.4.2 only allow one to define that the occurrence of an action depends on a single elementary causality condition. In practice the occurrence of an action may depend on a disjunction of elementary causality conditions. We therefore allow for the specification of causality relations that model this.

Disjunctions of enabling, synchronisation, and disabling conditions

We use the $\vee$ (or) symbol to represent disjunctions of elementary causality conditions. A disjunction of the elementary causality conditions enabling, synchronisation, and disabling is satisfied if and only if at least one of these elementary causality conditions is satisfied.

This definition allows us to define the meaning of the various disjunctions of enabling, synchronisation, and/or disabling conditions. For example,

$$b \vee \neg b \rightarrow a$$

means that a may occur if (1) b has occurred, or if (2) b has not occurred before a and b does not occur at the same time as a. By writing the time a occurs as t_a and the time b occurs as t_b, this specification implies $t_a \neq t_b$.

Table 3.11 specifies the temporal order of a and b for other causality relations as well.

The interleaving of two actions is an example of a behaviour in which disjunctions of enabling and disabling conditions are used. Interleaving means that the actions may occur in any order, except simultaneously. This can be represented as follows:

$$\{\ a \vee \neg a \rightarrow b,\ b \vee \neg b \rightarrow a\ \}.$$

Causality relation	*Temporal order if both a and b occur*
$b \vee {=}b \rightarrow a$	$t_b \leq t_a$
$b \vee \neg b \rightarrow a$	$t_b \neq t_a$
$\neg b \vee {=}b \rightarrow a$	$t_b \geq t_a$
$b \vee {=}b \vee \neg b \rightarrow a$	any order

Table 3.11 Temporal order of actions for various disjunctions of enabling, synchronisation, and/or disabling conditions

The disjunction of elementary causality conditions is represented graphically by an open square. Figure 3.12 gives an example.

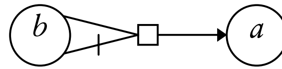


Figure 3.12 Graphical representation of $b \vee \neg b \rightarrow a$.

Disjunctions involving start condition

Causality relations may have conditions that consist of a disjunction of the start condition and one or more other elementary causality conditions, e.g.

$start \vee b \rightarrow a.$

Such a causality relation specifies that the result action, *a* in the example, is either independent of the other action, *b* in the example, or dependent on this other action. This is to be interpreted as follows: when the behaviour is executed, in some executions the occurrence of *a* is independent of the occurrence of *b*, whereas in other executions the occurrence of *a* depends on the occurrence of *b*.

Since a further explication of the disjunction of elementary causality conditions requires knowledge of the concept of behaviour execution, this explication is deferred to section 3.5.3, in which behaviour executions are addressed.

Uncertainty

In case an action is enabled by a disjunction of elementary causality conditions, the probability of its occurrence depends on the particular elementary causality condition that enables it. For example, in case of the causality relation

$b \vee {=}b \rightarrow a,$

it is possible that *a* must occur if *b* has occurred before, whereas *a* may occur at the time *b* occurs.

In order to specify this, a different probability of occurrence of the result action may be specified for each elementary causality condition that can enable it. Thus, the above requirement can be specified as:

$b_! \vee =b_? \rightarrow a$.

3.4.4 Reciprocal aspect of synchronisation and disabling conditions

Synchronisation conditions

The synchronisation condition $=b$ in the causality condition of action a defines that a may occur if b occurs at the same time as a. Thus, if a occurs due to the satisfaction of the synchronisation condition, b occurs simultaneously with a. Therefore, b should be allowed to occur simultaneously with a, which implies that $=a$ should be a part of the causality condition of b.

For example, given the causality relation

$=b \rightarrow a$,

some allowed causality relations of b are

$=a \rightarrow b$,
$a \vee =a \rightarrow b$, and
$start \vee =a \rightarrow b$,

but not

$a \rightarrow b$, and
$start \rightarrow b$.

Because the synchronisation of actions a and b requires both actions to occur, the uncertainty that a synchronises with b should be equal to the uncertainty that b synchronises with a. So, for example, the behaviour specification

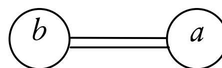
$\{ =b_! \rightarrow a,$
$=a_! \rightarrow b \}$

is allowed, but the behaviour specification

$\{ =b_? \rightarrow a,$
$=a_! \rightarrow b \}$

is not allowed.

Figure 3.13 shows a shortcut representation of the synchronisation of two actions.



$=b \rightarrow a$,
$=a \rightarrow b$

Figure 3.13 Shortcut representation of synchronisation.

Disabling conditions

The disabling condition $\neg b$ in the causality condition of action a defines that a may occur if neither b has occurred, nor b occurs at the same time as a. Thus, if a occurs due to the satisfaction of the disabling condition, b does not occur simultaneously with a. Therefore, b should not be allowed to occur simultaneously with a in that case. Therefore, the causality condition of b should prevent this.

For example, if

$$\neg b \rightarrow a,$$

then some allowed causality conditions of b are

$$\neg a \rightarrow b, \text{ and}$$
$$a \vee \neg a \rightarrow b,$$

but not

$$=a \rightarrow b.$$

Figure 3.14 shows shortcut representations of two frequently occurring pairs of reciprocal causality relations.

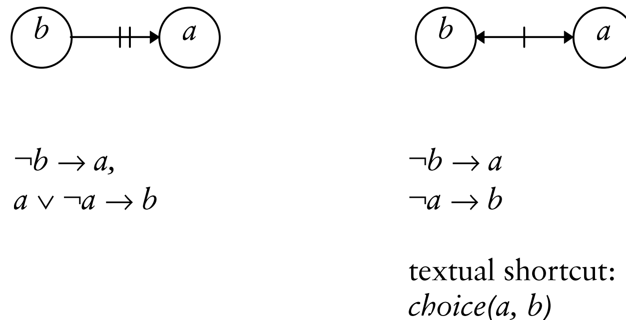


Figure 3.14 Shortcut representations of two frequently occurring pairs of reciprocal causality relations

3.5 Monolithic behaviours

This section considers behaviours that may contain more than two actions. In such behaviours the occurrence of an action may depend on a combination of elementary causality conditions that involve multiple other actions.

3.5.1 Conjunction of elementary causality conditions

We use the $\wedge$ (and) symbol to represent the conjunction of elementary causality conditions. A conjunction of two or more elementary causality conditions is satisfied if and only if all of these conditions are satisfied.

For example, the causality relation

$$a \wedge =b \wedge \neg c \rightarrow d.$$

specifies that an action *d* may occur if and only if (1) it occurs after *a*, (2) it occurs at the same time as *b*, and (3) *c* has not occurred yet and does not occur at the same time as *d*.

The elementary causality conditions in a conjunction of elementary causality conditions should all involve different actions. This is because a conjunction of two or more different elementary causality conditions that involve the same action can never be satisfied. For example, the causality relation

$$a \wedge =a \rightarrow b$$

specifies that action *b* may occur only if it occurs after *a* and at the same time as *a*, which is a condition that is never satisfied.

The conjunction of elementary causality conditions is represented graphically by a black square. Figure 3.15 gives two examples of conjunctions of elementary causality conditions.

In the behaviour of Figure 3.15a action *d* may only occur if all of the actions *a*, *b*, and *c* have occurred. Action *d* could represent the assembly of a car, and actions *a*, *b*, and *c* the arrival of the chassis, the arrival of the engine, and the arrival of the other parts, respectively.

In the behaviour of Figure 3.15b action *c* may only occur if action *a* has occurred and action *b* has not occurred yet, nor occurs at the same time as *c*. Action *a* could represent the loading of some freight in a truck, action *b* the breaking down of the truck, and action *c* the timely arrival of the freight at its destination.

Alternative causality conditions

An *alternative causality condition* of an action is an elementary causality condition or a conjunction of elementary causality conditions, such that the satisfaction of the elementary causality condition or of the conjunction of elementary causality conditions, respectively, is sufficient for the action to be enabled. For example, in the behaviour of Figure 3.15a $a \wedge b \wedge c$ is an alternative causality condition of *d*, but $a \wedge b$ is not.

Let $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ be an alternative causality condition of an action *a*, with $c_1, c_2, \ldots, c_n$ elementary causality conditions. This alternative causality condition is *minimal* if and only if there is no subset $S \subset \{c_1, c_2, \ldots, c_n\}$, such that satisfaction of all of the elementary causality conditions in *S* is sufficient for *a* to be enabled. In this thesis we assume that all alternative causality conditions in behaviour specifications are minimal.

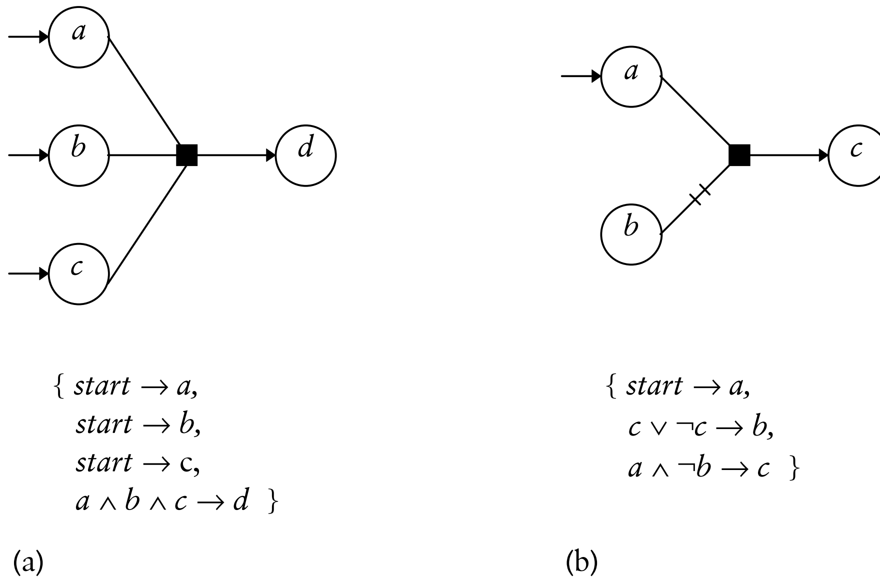


Figure 3.15 Some behaviours containing a conjunction of elementary causality conditions.

Uncertainty

In case of a causality relation that contains a conjunction of elementary causality conditions, the occurrence of the result action depends on the satisfaction of all of these elementary causality conditions. Therefore, it is not meaningful to specify the uncertainty of the occurrence of the result action if only one of these elementary causality conditions is satisfied. Instead, the uncertainty of the occurrence of the result action should be specified for the case in which all of the elementary causality conditions are satisfied. This is represented by an exclamation mark or question mark in subscript placed after the representation of the conjunction of elementary causality conditions.

For example, the causality relation

$$(a \wedge b \wedge c)_{!} \rightarrow d$$

specifies that *d* must occur if *a*, *b*, and *c* have occurred. And the causality relation

$$(a \wedge \neg b)_{?} \rightarrow c$$

specifies that *c* may occur if *a* has occurred and *b* has not occurred and *b* does not occur at the same time as *c*.

3.5.2 Disjunction of alternative causality conditions

We use the ∨ symbol to represent the disjunction of alternative causality conditions. A disjunction of two or more alternative causality conditions is satisfied if at least one of these alternative causality conditions is satisfied.

For example, the causality relation

$$a \vee b \vee c \rightarrow d$$

specifies that d may occur as soon as a, b, or c has occurred. And the causality relation

$$(a \wedge b \wedge c) \vee (a \wedge {=}d) \vee e \rightarrow f$$

specifies that f may occur if (1) a, b, and c have occurred, or (2) a has occurred and d occurs at the same time as f, or (3) e has occurred.

The disjunction of alternative causality conditions is a generalisation of the disjunction of elementary causality conditions (discussed in section 3.4.3), since each elementary causality condition that enables the result action is, by definition, an alternative causality condition.

We represent the disjunction of alternative conditions graphically as an open square. Figure 3.16 gives some examples of behaviours containing disjunctions of alternative causality conditions.

In the behaviour of Figure 3.16a action d may occur if a, b, or c has occurred. Actions a, b, and c can, for example, represent the arrival of different customers, and action d the serving of a customer.

In the behaviour of Figure 3.16b action d may occur if (1) a and b have occurred, or (2) c has occurred. Action a can, for example, represent the incoming of an order without payment, action b the incoming of a cheque, action c the incoming of a credit card order, and action d the shipment of the ordered good.

In the behaviour of Figure 3.16c action b may occur if a has occurred and c does not occur at the same time; action c may occur if a has occurred and b does not occur at the same time. Action a can, for example, represent the incoming of a batch of two orders, action b the processing of one order by a person, and action c the processing of the other order by the same person.

Uncertainty

In case the occurrence of an action depends on a disjunction of alternative causality conditions, the probability of its occurrence may depend on the alternative causality condition that enables this action. Therefore, a different probability of occurrence of the result action may be specified for each alternative causality condition that can enable this action.

For example, the causality relation

$$a_{!} \vee (b \wedge c)_{?} \rightarrow d$$

specifies that d must occur if a has occurred, and that d may occur if b and c have occurred.

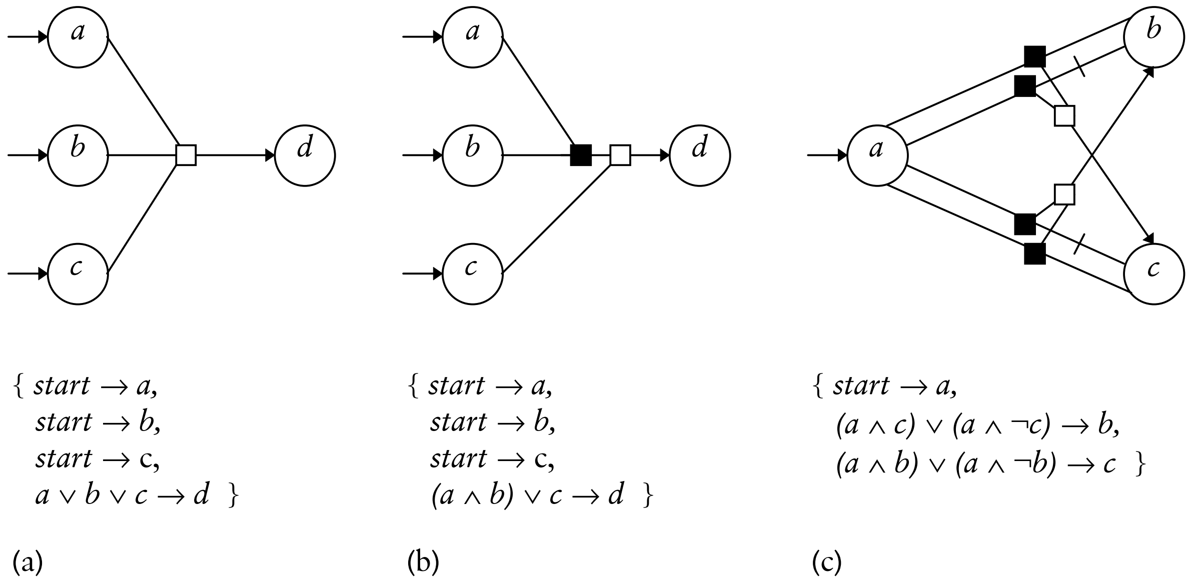


Figure 3.16 Some behaviours containing disjunctions of alternative causality conditions.

3.5.3 Behaviour executions

Since a behaviour specification may specify actions that need not always occur when the behaviour is executed, a behaviour specification generally models multiple behaviour executions. The concept of *behaviour execution* models what happens when a behaviour is executed. A behaviour execution specifies:

- the actions that occur when the behaviour is executed;
- the relations between these actions that hold during the execution.

As an example, consider the behaviour

$$\{\ start_{?} \rightarrow a,\quad a_{?} \rightarrow b\ \}.$$

This defines the following behaviour executions:

- the execution in which no action occurs;
- the execution in which only the initially enabled action *a* occurs;
- the execution in which actions *a* and *b* occur, where *a* is initially enabled and *b* is caused by *a*.

As another example, consider the behaviour

$$\{\ \neg b_{!} \rightarrow a,\quad \neg a_{!} \rightarrow b\ \}.$$

This defines the following behaviour executions:

- the execution in which only *a* occurs;
- the execution in which only *b* occurs.

In [Quartel et al., 1996; Quartel, 1998], behaviour executions are used to formally define the semantics of some of the concepts introduced in this chapter. Such a formal specification is out of the scope of this thesis. We only use behaviour executions here to explain the meanings of concepts that are hard to understand without considering behaviour executions, such as the disjunction of alternative causality conditions.

Meaning of disjunction of alternative causality conditions

We already defined that a disjunction of alternative causality conditions in a causality relation of an action *a* means that *a* may occur if at least one of the alternative causality conditions is satisfied. This definition does, however, not consider which of the alternative causality conditions effectively causes the occurrence of *a* in a behaviour execution.

It is useful to consider these causes, since different causes may lead to different implementations of the causality relation. Moreover, in chapter 4 we consider values that are established in actions and to which other actions may refer; the cause of an action determines to which other actions it may refer.

Given a disjunction of alternative causality conditions $c_1 \vee c_2 \vee ... \vee c_n$ of an action *a*, the following options seem useful.

1. Action *a* may be caused by either c_1, c_2, ..., or c_n but not by two or more of these alternative causality conditions.
2. Action *a* may be caused by c_1, c_2, ..., or c_n, or by two or more of these alternative causality conditions.

We select option 1, because it offers more power of expression. If we would select option 2, it would become impossible to express that action *a* may not be caused by two or more of the alternative causality conditions. However, if we select option 1, it is still possible to express that action *a* may also be caused by two or more of the alternative causality conditions, by specifying this explicitly as the conjunction of elementary causality conditions. For example, we could specify

$$a \vee b \vee (a \wedge b) \rightarrow c$$

to express that c may be caused by *a* alone, or by *b* alone, or by both *a* and *b*.

The selection of the first alternative has the following consequence for the definition of the meaning of a disjunction of alternative causality conditions of an action *a*: in a behaviour execution action *a* is caused by only one of these alternative causality conditions and independent of the other alternative causality conditions.

Thus, for example, the behaviour

$$\{ \ start \rightarrow a, \\ start \rightarrow b, \\ a \vee b \rightarrow c \ \}$$

has the following behaviour executions:

- the execution in which the initially enabled actions *a* and *b* occur, as well as action *c*, which is caused by *a*;
- the execution in which the initially enabled actions *a* and *b* occur, as well as action *c*, which is caused by *b*;

but not

- the execution in which the initially enabled actions *a* and *b* occur, as well as action *c*, which is caused by both *a* and *b*;

As a practical example of this behaviour, consider the case in which *a* represents a request to borrow a book from a library by one customer, *b* represents a request to borrow the same book from the same library by another customer, and *c* represents the granting of the request for a customer. Of course, the request is only granted for a single customer. The granting of the request is thus related to the request of this customer and occurs independent of the other request.

Other representations of causality relations

We adopted the convention that causality conditions that contain both disjunctions and conjunctions are specified as disjunctions of conjunctions of elementary causality conditions. We say such causality conditions are in *disjunctive normal form*. Our rules for the construction of causality relations prevent causality conditions that are formed as conjunctions of disjunctions of elementary causality conditions. Thus, for example, the specification of the causality condition

$$a \wedge (b \vee c \vee d \vee e \vee f)$$

is not allowed. Instead, what is meant by this condition (see below) should be specified as the more lengthy causality condition

$$(a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (a \wedge e) \vee (a \wedge f).$$

Since causality conditions in disjunctive normal form may be lengthy, we allow them to be specified in conjunctive normal form (i.e. as conjunctions of disjunctions of elementary causality conditions) from here on, by defining the conjunction of disjunctions of elementary causality conditions as follows. Given a causality relation

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n \rightarrow a$$

in which C_1, C_2, ..., C_n are disjunctions of one or more elementary causality conditions, *a* may be caused in a behaviour execution by any conjunction of one elemen-

tary causality condition from C_1, one elementary causality condition from C_2, ..., and one elementary causality condition from C_n, and not by any other condition.

Now the conjunction distributes over the disjunction, i.e. the causality condition $a \wedge (b \vee c)$ means the same as $(a \wedge b) \vee (a \wedge c)$. This is because

- $a \wedge (b \vee c) \rightarrow d$ means that d may be caused by either both a and b or by both a and c, and
- $(a \wedge b) \vee (a \wedge c) \rightarrow d$ means d may be caused by either both a and b or by both a and c.

When interpreting causality conditions, one should be aware that the disjunction does not distribute over the conjunction, i.e. that $a \vee (b \wedge c)$ does not mean the same as $(a \vee b) \wedge (a \vee c)$. This is because

- $a \vee (b \wedge c) \rightarrow d$ means that d may be caused either by a or by both b and c, whereas
- $(a \vee b) \wedge (a \vee c) \rightarrow d$ means that d may be caused either by a, or by both b and c, or by both a and b, or by both a and c.

3.5.4 Additional shortcut representations

We introduce a few more shortcut representations to allow for more concise behaviour representations.

In the graphical representation, multiple arrows pointing to an action are to be interpreted as the conjunction of the relations represented by the arrows. This allows e.g. the more concise representation in Figure 3.17c of the behaviour represented in Figure 3.17b.

A dotted line around a group of actions is to be interpreted as follows:

- An arrow pointing to the dotted line is a shortcut for copies of this arrow pointing to every action in the group.
- An arrow in a box that is connected to the dotted line is a shortcut for copies of this arrow pointing from every action in the group to every other action in this group.

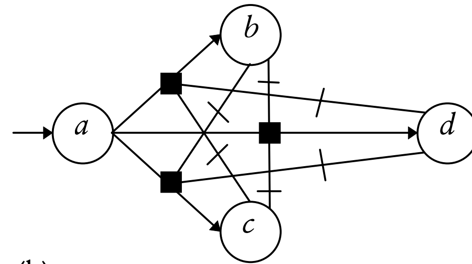
This allows the even more concise representation in Figure 3.17d of the behaviour represented in Figure 3.17b.

3.6 Behaviour composition

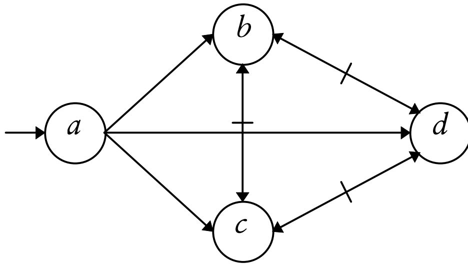
This section discusses the structuring of behaviours as compositions of (sub-)behaviours. We distinguish between causality-oriented behaviour compositions and constraint-oriented behaviour compositions.

$\{\, start \rightarrow a,$
$a \wedge \neg c \wedge \neg d \rightarrow b,$
$a \wedge \neg b \wedge \neg d \rightarrow d,$
$a \wedge \neg b \wedge \neg c \rightarrow d \,\}$

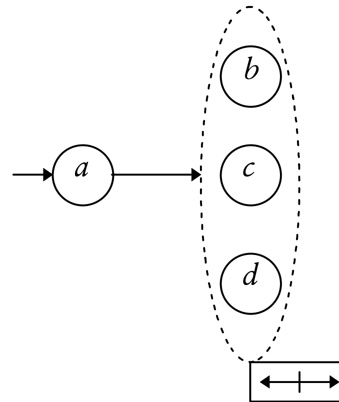
(a)



(b)



(c)



(d)

Figure 3.17 Four representations of same behaviour. (a) textual, (b) graphical, without shortcuts, (c) graphical, with shortcuts, (d) graphical, with further shortcuts

3.6.1 Causality-oriented behaviour composition

In a *causality-oriented behaviour composition* conditions in one behaviour enable actions in another behaviour.

Entry points and exit points make this type of behaviour composition possible. A causality relation whose result action depends on conditions defined in other behaviours contains an *entry point* in its causality condition. A causality relation that defines conditions for one or more result actions in other behaviours contains an *exit point* instead of a result action. Entry points and exit points are thus only syntactical constructs that represent conditions and result actions, respectively.

A relation between two or more behaviours can be defined by coupling entry points to exit points. Each entry point is coupled to exactly one exit point, but an exit point can be coupled to multiple entry points. The meaning of such a composition of behaviours in terms of a monolithic behaviour is as follows: each entry point in a causality condition represents the causality condition of the exit point coupled to this entry point.

We represent entry points and exit points in text by the words *entry* and *exit*, respectively, followed by suffixes to make them uniquely identifiable, if necessary. The

coupling of an entry point *entry* of a behaviour B_2 to an exit point *exit* of a behaviour B_1 is represented as

$B_1(exit) \rightarrow B_2(entry)$.

Examples

The following behaviour B is structured as a composition of behaviours B_1 and B_2.

$B=$
$\{ start \rightarrow B_1(entry),$
$B_1(exit) \rightarrow B_2(entry) \}$

$B_1=$
$\{ entry \rightarrow a,$
$a \rightarrow b,$
$b \rightarrow exit \}$

$B_2=$
$\{ entry \rightarrow c,$
$c \rightarrow d \}$.

Figure 3.18a represents this behaviour graphically. We represent a behaviour border as a grey rectangle with round corners, and we represent entry points and exit points as triangles.

This behaviour specification represents the same behaviour as the unstructured (monolithic) behaviour:

$B'=$
$\{ start \rightarrow a,$
$a \rightarrow b,$
$b \rightarrow c,$
$c \rightarrow d \}$

Figure 3.18b represents this behaviour graphically. Figure 3.18 illustrates that a causality-oriented behaviour composition looks as if the causality relations of a monolithic behaviour have been "cut", generating sub-behaviours in this way.

Figure 3.19 gives an example of two sub-behaviours coupled by multiple entry and exit points. A textual representation of this behaviour is:

$\{ start \rightarrow B_1(entry),$
$B_1(exit_1) \rightarrow B_2(entry_1),$
$B_1(exit_2) \rightarrow B_2(entry_2) \}$

$B_1=$
$\{ entry \rightarrow a,$
$a \rightarrow exit_1$

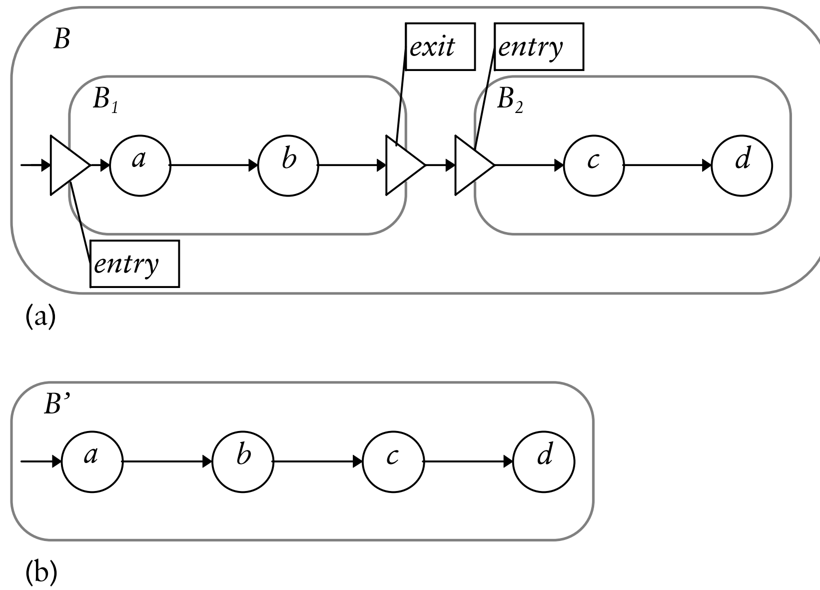


Figure 3.18 Example behaviour represented (a) as causality-oriented composition of sub-behaviours, and (b) monolithically.

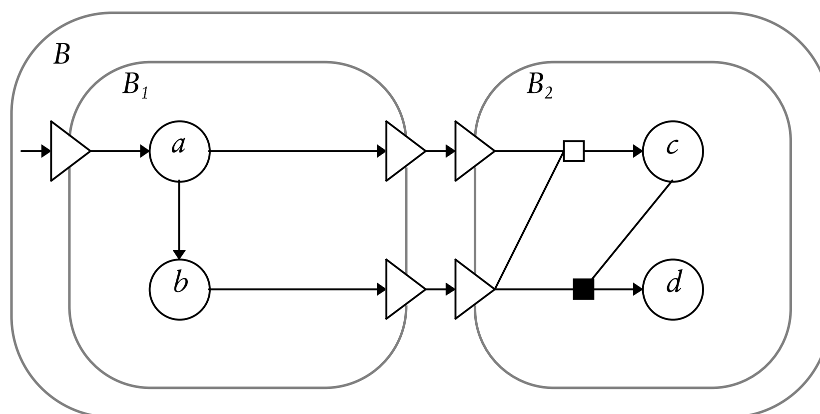


Figure 3.19 Example of sub-behaviours coupled by multiple entry and exit points.

$a \rightarrow b,$
$b \rightarrow exit_2 \}$

$B_2 =$
$\{ entry_1 \vee entry_2 \rightarrow c,$
$entry_2 \wedge c \rightarrow d \}$

More examples of causality-oriented behaviour composition can be found in chapter 5, which discusses the role of this composition technique in development processes.

Representation shortcuts

As a shortcut to the specification of the coupling of exit and entry points as

$B_1(exit) \rightarrow B_2(entry)$

and the enabling of the exit point of B_1 by an action *a* as

$a \rightarrow exit,$

we allow for the specification that *a* directly enables the entry point of B_2, i.e.

$a \rightarrow B_2(entry).$

The behaviour of Figure 3.18a can now be specified slightly more concisely as:

$B=$
$\{\ start \rightarrow B_1(entry)\ \}$

$B_1=$
$\{\ entry \rightarrow a,$
$a \rightarrow b,$
$b \rightarrow B_2(entry)\ \}$

$B_2=$
$\{\ entry \rightarrow c,$
$c \rightarrow d\ \}.$

Figure 3.20 represents this graphically.

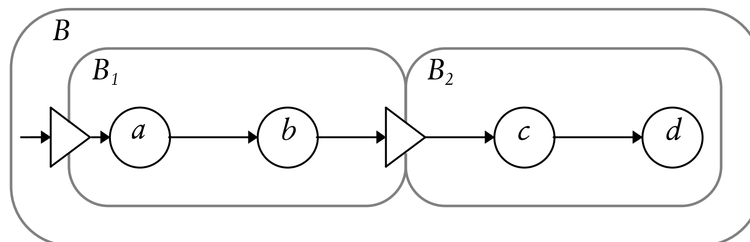


Figure 3.20 Shortcut representation of behaviour of Figure 3.18a.

Similarly, we allow for the shortcut specification of an action that becomes directly enabled by the exit point of another behaviour, e.g.

$B(exit) \rightarrow a.$

Multiple behaviour copies

Since each action or entry point has only one causality condition, multiple copies (sometimes called instances) of a behaviour can be defined by repeatedly enabling its entry point(s). For example, the behaviour

$\{$ *start* $\rightarrow$ *B(entry),*
start $\rightarrow$ *B(entry),*
start $\rightarrow$ *B(entry)* $\}$

B=
$\{$ *entry* $\rightarrow$ *a* $\}$

defines three independent copies of behaviour *B*. These copies are three different behaviours that have the same structure as behaviour *B*.

This also allows us to define repetitive behaviours. For example, the behaviour

$\{$ *start* $\rightarrow$ *B(entry)* $\}$

B=
$\{$ *entry* $\rightarrow$ *a,*
a $\rightarrow$ *B(entry)* $\}$

defines an infinite sequence of copies of an action *a*. Figure 3.21 represents this behaviour graphically.

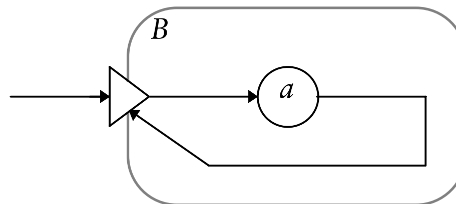


Figure 3.21 Graphical representation of example behaviour.

If it is necessary in a specification to distinguish the different copies of a behaviour, or their actions, from each other, one may do so by appending additional behaviour copy identifiers to their names. We use natural numbers for this purpose, which we represent in superscript after the behaviour names. For example, in the specification

$\{$ *start* $\rightarrow$ B^1*(entry)* $\}$

B^i=
$\{$ *entry* $\rightarrow$ *a,*
a $\rightarrow$ B^{i+1}*(entry)* $\}$,

one can refer to the copies of behaviour *B* as B^1, B^2, etc., where B^1 is the behaviour copy that is first executed, followed by B^2, etc. The copies of action *a* can be referred to as a^1, a^2, etc. Figure 3.22 represents this behaviour graphically.

In order to define multiple copies of a behaviour, a constraint on the behaviour copy identifiers can be specified before the causality relation of the entry points of the behaviour copies. For example, the specification

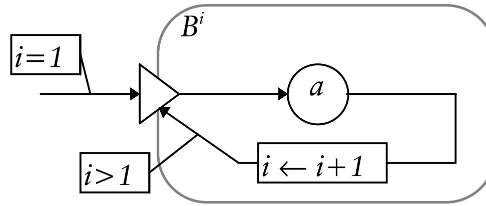


Figure 3.22 Graphical representation of example behaviour.

$\{ \ i=1, 2, 3: start \rightarrow B^i(entry) \ \}$

is a shortcut representation of the following specification:

$\{ \ start \rightarrow B^1(entry),$
$\quad start \rightarrow B^2(entry),$
$\quad start \rightarrow B^3(entry) \ \}.$

In case a behaviour has multiple entry points, we use the convention that the enabling of an entry point *e* only results in a new copy of the behaviour if in all current copies of the behaviour *e* is enabled. Otherwise, *e* becomes enabled in an previously created copy of the behaviour. Thus, for example, the behaviour specification

$\{ \ start \rightarrow B(entry_1),$
$\quad start \rightarrow B(entry_2),$
$\quad start \rightarrow B(entry_1),$
$\quad start \rightarrow B(entry_2) \ \}$

$B=$
$\{ \ entry_1 \rightarrow a,$
$\quad entry_2 \rightarrow b \ \}$

specifies that two copies of *B* are created, rather than four.

3.6.2 Constraint-oriented behaviour composition

In a *constraint-oriented behaviour composition* actions are, together with the conditions for their occurrence, distributed over multiple (sub-)behaviours.

A behaviour structured according to the constraint-oriented composition technique contains one or more interactions shared by its sub-behaviours. Since interactions may only occur if the conditions in all of the involved sub-behaviours are satisfied, a constraint-oriented behaviour composition may be viewed as a conjunction of constraints on actions, where the constraints are described in separate sub-behaviours.

The constraint-oriented composition technique requires that the conjunctions of constraints on interactions can be satisfied; otherwise one would specify interactions that can never occur. For example, if *a* and *b* are interactions of behaviours B_1 and B_2, the following behaviour specification is not allowed:

$B_1 = \{ \ldots, a \rightarrow \underline{b}, \ldots \}$
$B_2 = \{ \ldots, b \rightarrow \underline{a}, \ldots \}$.

We specify that interaction point *ip* is an interaction point of the behaviours B_1, B_2, ..., B_n, on which interactions a_1, a_2, ..., a_m take place as follows:

interactions $B_1, B_2, \ldots, B_n$ on *ip*: $a_1, a_2, \ldots, a_m$

If all actions of the behaviours B_1, B_2, ..., B_n that take place on *ip* are interactions of these behaviours, one may refrain from specifying the list of interaction names, provided one specifies the interaction point *ip*. If one is not interested in the name of the interaction point, one may refrain from specifying this name, provided one specifies the list of interaction names. One may not refrain from specifying both the list of interaction names and the name of the interaction point.

If interactions of behaviours that have multiple copies are specified, the behaviour identifiers may be followed by a constraint on the behaviour copy identifiers, represented between square brackets. For example,

interactions D^i, E^j $[i=j, i=1,2, j=1,2]$: a

represents that a copy of a is an interaction of D^1 and E^1 and that another copy of a is an interaction of D^2 and E^2. If it follows from the remainder of the behaviour specification that D^1, D^2, E^1, and E^2 are the only copies of D and E, the constraint $i=1,2$, $j=1,2$ may be left out. If the constraint on the behaviour copy identifiers is completely left out, i.e. as in

interactions D, E: a

this represents that each copy of D may interact with one copy of E, and vice versa. It does not specify which copy of D interacts with which copy of E.

Examples

Figure 3.23a gives an example of a constraint-oriented behaviour composition. The figure can, for example, represent the interaction of a system user (B_1) and a system (B_2). Both are prepared to perform an initial interaction a, which might represent the posing of a question. B_1 is initially also prepared to carry out interaction b, e.g. the receiving of an answer, but B_2 is only prepared to carry out b after a. Figure 3.23b models this behaviour in a monolithic form. The constraint-oriented behaviour composition is represented in text as:

{ interactions B_1, B_2: a,b }

$B_1 =$
$\{ start \rightarrow \underline{a},$
$start \rightarrow \underline{b} \}$

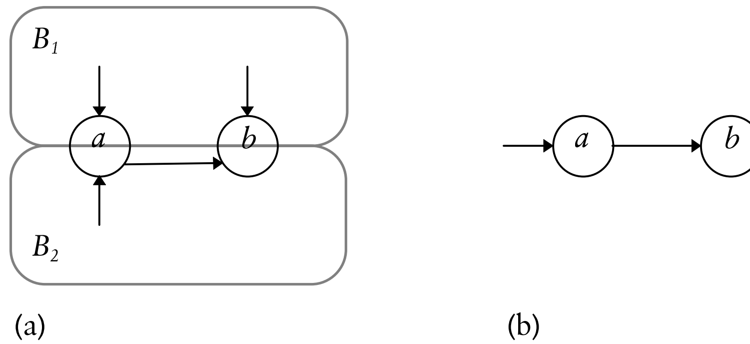


Figure 3.23 Example behaviour represented (a) as constraint-oriented composition of sub-behaviours, and (b) in monolithic form.

B_2=
{ $start \rightarrow \underline{a}$,
$a \rightarrow \underline{b}$ }

A more complex example of constraint-oriented behaviour composition is shown in Figure 3.24a. Figure 3.24b shows the monolithic representation of the behaviour. The constraint-oriented behaviour composition is represented in text as:

{ interactions $B_1, B_2: b$,
$B_1, B_3: f$,
$B_2, B_3: d$,
$B_1, B_2, B_3: c$ }

B_1=
{ $start \rightarrow a$,
$a \rightarrow \underline{b}$,
$b \rightarrow \underline{c}$,
$c \rightarrow \underline{f}$,
$f \rightarrow e$ }

B_2=
{ $start \rightarrow \underline{b}$,
$start \rightarrow \underline{c}$,
$b \rightarrow \underline{d}$ }

B_3=
{ $start \rightarrow \underline{d}$,
$d \rightarrow \underline{c}$,
$d \rightarrow \underline{g}$,
$g \rightarrow \underline{f}$ }

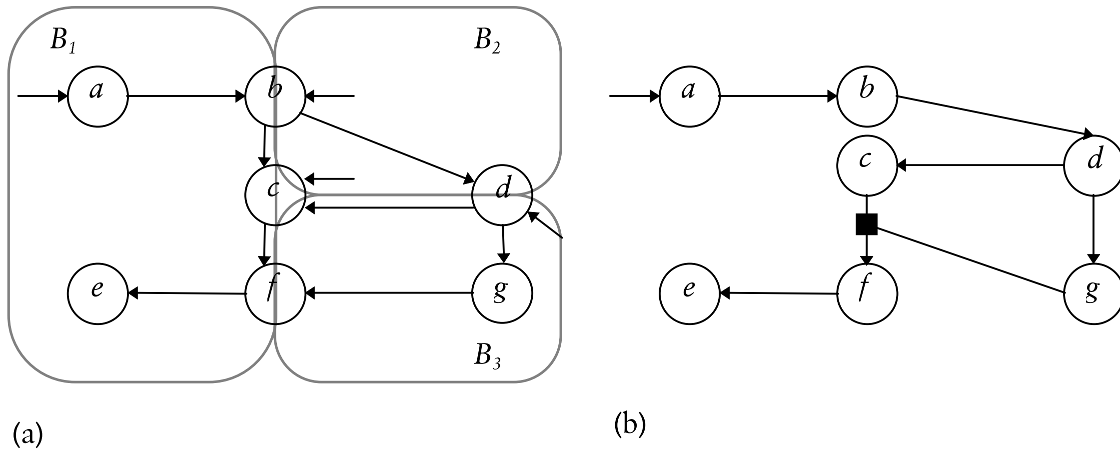


Figure 3.24 Example behaviour represented (a) as constraint-oriented composition of sub-behaviours, and (b) monolithically.

These examples illustrate that a constraint-oriented behaviour composition looks as if the actions of a monolithic behaviour have been "cut", generating sub-behaviours in this way.

When decomposing a behaviour into a constraint-oriented composition of sub-behaviours, one obviously has a large amount of decomposition options. Figure 3.25 shows some decompositions of the behaviour

$$\{\ start \rightarrow a,\ start \rightarrow b,\ a \wedge b \rightarrow c\ \}$$

into sub-behaviours B_1 and B_2.

More examples of causality-oriented behaviour composition can be found in chapter 5, which discusses the role of this composition technique in development processes.

3.7 Conclusions

The provision of a well-defined, complete, and parsimonious set of generally applicable basic architectural concepts is crucial for the proper support of developers [Vissers et al., 1995]. We provided such a set of architectural concepts for distributed systems modelling, and gave examples of their use in business process modelling. We showed that these concepts are applicable at many abstraction levels and argued that these concepts cover the relevant aspects of distributed systems.

Important basic architectural concepts include: entities, which are carriers of behaviours; actions, which model activities; interactions, which model common activities of two or more entities; and causality relations, which model dependency relations between activities.

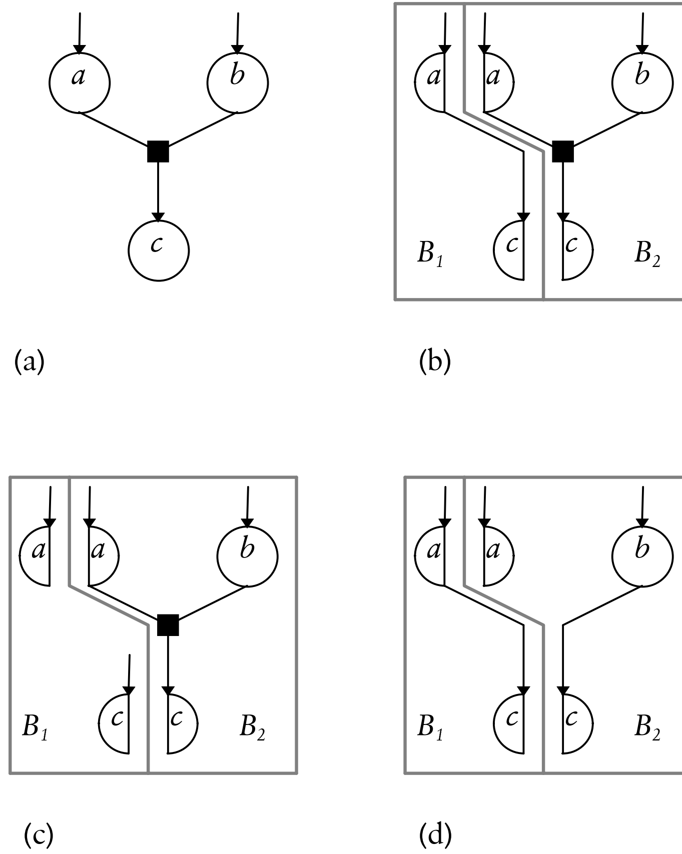


Figure 3.25 (b)-(d) Some constraint-oriented decompositions of behaviour of (a).

To keep behaviours comprehensible, it should be possible to construct behaviours as compositions of sub-behaviours. We therefore showed how a behaviour can be composed as a causality-oriented composition of sub-behaviours (in which conditions in one sub-behaviour enable actions in another sub-behaviour) and as a constraint-oriented composition of sub-behaviours (in which actions are, together with the conditions for their occurrence, distributed over multiple sub-behaviours).

4

Role of data in behaviour modelling

4.1 Introduction

4.1.1 Motivation

This chapter introduces *data*, which is considered as a collection of architectural concepts that can be used to model behaviours more concisely than with the previously discussed basic architectural concepts only.

Most of the data concepts introduced here, or similar concepts, are used in many specification and implementation languages. The mere definition of the data concepts is therefore not the main contribution of this chapter. Rather, it is our objective to show:

- when data is indispensable in behaviour modelling;
- why data is indispensable in those cases: that data can be used to model behaviours comprehensibly and facilitate economical implementation;
- that all data concepts can be defined in terms of the previously discussed basic concepts, and that, in that sense, data concepts are not basic concepts.

In short, it is our objective to show the role of data in behaviour modelling.

4.1.2 Structure

Sections 4.2, 4.3, and 4.4 introduce the data concepts. For clarity, these sections only introduce data concepts for monolithic behaviours in which actions are only related by (combinations of) enabling relations.

Each of the sections 4.2 to 4.4 is sub-divided in the following sub-sections:

- a section “Inconvenience of current concepts” that describes when a behaviour model comprising only basic concepts and the data concepts introduced thus far becomes lengthy, and thereby hard to comprehend and expensive to implement;
- a section “Data concepts” that introduces data concepts, defines their architectural semantics and their representation, and shows that they can be used to resolve the identified inconvenience (i.e. that these data concepts can be used to model behaviours more concisely and facilitate economical implementation);
- a section “Meaning of data concepts in terms of basic concepts” that defines the data concepts in terms of basic concepts.

In these sections we do not give objective measures of the complexity of a specification, since we argued in chapter 2 that complexity is, to some extent, subjective. Instead, we show the usefulness of data by means of example specifications in which the use of data clearly increases their comprehensibility.

Section 4.5 introduces extensions for composite behaviours and disabling and synchronisation relations.

4.1.3 Summary of concepts

Since this chapter introduces quite a number of concepts, we summarise them here for the convenience of the reader.

Section 4.2 introduces action values and data types. An *action value*, which can be established in an action, models the result of an activity. An action value is an element of a *data type*, which contains all action values that can be established in an action.

Action values can be used to concisely model a set of actions from which only one may occur. For example, the behaviour

$$\{\ \neg drink_coffee \wedge \neg drink_cola \rightarrow drink_tea,\\ \neg drink_tea \wedge \neg drink_cola \rightarrow drink_coffee,\\ \neg drink_tea \wedge \neg drink_coffee \rightarrow drink_cola\ \}$$

can be represented more concisely as

$$\{\ start \rightarrow drink\ (\{coffee,\ tea,\ cola\})\ \},$$

in which the set $\{coffee, tea, cola\}$ is a data type of action values.

Section 4.3 discusses action variables and reference relations. When an action value is established in an action, it is assigned to an *action variable*. The action variable can then be used to refer to the action value. A *reference relation* is a relation that makes an action, and possibly the value established in this action, dependent on values established in other actions.

Consider, for example, the behaviour specification:

$\{$ *start* $\rightarrow a\ (v_a:\{1,2\}),$
$a \rightarrow b\ (v_b:\{1,2\})\ [v_b=v_a]\ \}.$

In this specification, $a\ (v_a:\{1,2\})$ represents that one value from the data type $\{1,2\}$ is established in action a, which is assigned to variable v_a. $b\ (v_b:\{1,2\})$ represents that one value from the data type $\{1,2\}$ is established in action b, which is assigned to variable v_b. The reference relation $[v_b=v_a]$ represents that the value established in b is equal to the value established in a.

Section 4.4 discusses state values, state variables, and state functions. A *state function* is a function of action variables or other state functions. A *state value* is the value of a state function, which is assigned to a *state variable*. We show that a state value models one or more states of a behaviour, where the state concept conforms to a customary notion of state, e.g., as in finite-state machines.

Consider, for example, the behaviour specification:

$\{$ *start* $\rightarrow a\ (v_a:\{1,2\}),$
start $\rightarrow b\ (v_b:\{1,2\}) < vs=v_a+v_b >,$
$a \wedge b \wedge (vs=3) \rightarrow c\ \}.$

In this specification vs is a state variable that is assigned the value of the state function v_a+v_b. The reference relation $(vs=3)$ represents that c may only occur when vs, or equivalently v_a+v_b, is equal to 3.

The difference between action values and state values is important for a proper understanding of these concepts.

- An action value belongs to a single action in the sense that it is established in a particular action. An action value models the result of the activity modelled by the action.

  A state value does not necessarily belong to a single action in the sense that it is the value of a function of multiple action values. A state value does not model an action or a result of an action.

- An action value is a not necessarily a function of other action values: action values may be established non-deterministically.

  A state value is always a function of action values.

4.2 Role of action values

4.2.1 Inconvenience of current concepts

Inconvenience. If an activity delivers one out of many possible results, a model of this activity contains many actions that disable each other (since each action models

one possible result of the activity) when the model is defined using the basic concepts only. Such models are large and thereby hard to survey and comprehend.

Example. Consider a bank employee who drafts mortgage quotations. The drafting of "a" mortgage quotation may be viewed as one activity. However, the amount of the quotation is usually relevant. Each amount is a possible result of the activity. Since a bank employee may usually quote many amounts, a large number of actions are required to model this. (If the bank employee may quote amounts in whole dollars from $50,000 to $100,000, 50,000 actions are required.) See Figure 4.1 for the model.

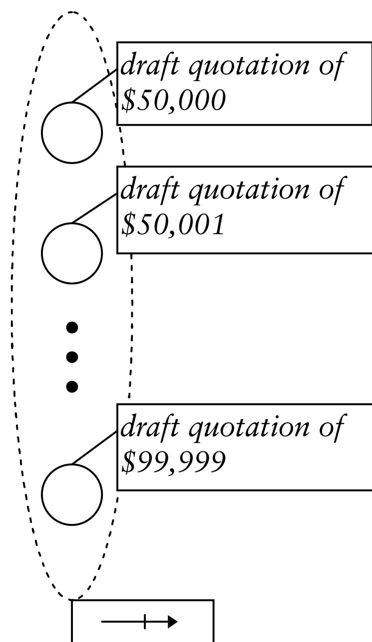


Figure 4.1 Modelling of multiple possible results of a single activity requires multiple actions that disable each other.

We call a set of actions, such that the occurrence of each action in the set disables the occurrences of all other actions in the set, a *set of mutually disabling actions*.

4.2.2 Data concepts

Concepts

Here we refine the action concept. An action, possibly associated with a set of action values, models an activity as follows.

- The *action* models the activity which delivers one result out of one or more possible results. The occurrence of the action models that the activity delivers a result.
- Each *action value* from the set of action values models one possible result of the activity. The *set of action values*, also called the *data type* of the action values,

models all possible results. The *establishment* in the action of a value *c* models that the activity delivers the result modelled by *c*. The value *c* should be an element of the set of action values.

From here on we use the term “action” to denote actions in which values may be established. In order to denote actions in which no values may be established, i.e. actions as defined in chapter 3, we use the term *basic action*.

Representation

We assume the existence of a suitable language to define and represent data types precisely. In the current work we use an informal notation that is explained only when its meaning may be unclear.

An action with its associated data type is textually represented by a unique action identifier, followed by the representation of the data type between round brackets. E.g.,

a ({1, 2, 3})

represents an action *a* associated with the data type {*1, 2, 3*}. Figure 4.2 represents this graphically.

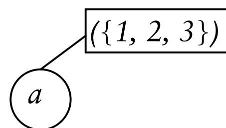


Figure 4.2 Graphical representation of *a ({1, 2, 3})*.

An action in which a value is established is textually represented by an action identifier followed by the representation of the value between round brackets. E.g.,

a (1)

represents and action *a* in which the value *1* is established.

Examples

The drafting of a mortgage quotation in the example discussed above can now be specified as:

draft quotation ({50000, 50001, ..., 99999}),

or, in case the data type representing the results has been predefined (e.g. as “the set of natural numbers from 50,000 to 100,000”) and been named *mortgage range*, as:

draft quotation (mortgage range).

The establishment of the value *75,000* in the action *draft quotation* can be specified as:

draft quotation (75000).

The behaviour of the employee who drafts a quotation can be specified as:

{ *start* → *draft quotation (mortgage range)* }.

Values can be used to represent any type of result of an activity, including its time of occurrence and its location. Thus, for example,

arrive at work({8.45, 9.00, 9.15})

may represent the arrival at work of a person at one of three possible points of time, and

visit city({Amsterdam, Rotterdam, The Hague})

may represent a visit to either Amsterdam, Rotterdam, or The Hague.

4.2.3 Meaning of data concepts in terms of basic concepts

- An action without an associated data type models a basic action.
- Each combination of an action a and a value from its associated data type D_a, i.e. $a(c_a)$, $c_a \in D_a$, models one basic action.

  The combination of an action a and its associated data type D_a, i.e. $a(D_a)$, models the basic actions modelled by $a(c_a)$, $c_a \in D_a$.
- Consider an action a with its causality condition C_a and an associated data type D_a. The causality condition of each of the basic actions modelled by $a(c_a)$, $c_a \in D_a$, is the conjunction of C_a and the partial causality condition modelled by $\bigwedge_{x \in D_a \setminus \{c_a\}} \neg a(x)$, i.e.

  $$C_a \rightarrow a\ (\{c_1, c_2, ..., c_n\}),$$

  in which a is an action, models the causality relations

  $$C_a \wedge \neg a_2 \wedge \neg a_3 \wedge ... \wedge \neg a_n \rightarrow a_1,$$
  $$C_a \wedge \neg a_1 \wedge \neg a_3 \wedge ... \wedge \neg a_n \rightarrow a_2,$$
  ...
  $$C_a \wedge \neg a_1 \wedge \neg a_2 \wedge ... \wedge \neg a_{n-1} \rightarrow a_n,$$

  in which $a_1, a_2, ..., a_n$ are the basic actions modelled by $a(c_a)$, $c_a \in D_a$. (For convenience we name the basic actions modelled by $a(c_a)$, $c_a \in D_a$, $a_1, a_2, ..., a_n$. However, since the purpose of naming actions is the establishment of their unique identity, we could have given the basic actions any unique name.)

  Example. The causality relation of an action a

  $$b \vee c \rightarrow a\ (\{1, 2, 3\})$$

  models the following causality relations of basic actions a_1, a_2, and a_3:

$(b \vee c) \wedge \neg a_2 \wedge \neg a_3 \rightarrow a_1,$
$(b \vee c) \wedge \neg a_1 \wedge \neg a_3 \rightarrow a_2,$
$(b \vee c) \wedge \neg a_1 \wedge \neg a_2 \rightarrow a_3.$

- Consider an action *a* with its associated data type that models a number of basic actions $a_1, a_2, ..., a_n$. If *a* is an enabling action of an action *b*, this models the following partial causality condition of *b*: $a_1 \vee a_2 \vee ... \vee a_n$.

  Example. The behaviour

  { *start* $\rightarrow$ *a* ({1, 2}),
  start $\rightarrow$ *b*,
  $a \wedge b \rightarrow c$ },

  in which *a* is an action, models the behaviour

  { *start* $\rightarrow a_1$,
  start $\rightarrow a_2$,
  start $\rightarrow$ *b*,
  $(a_1 \vee a_2) \wedge b \rightarrow c$ },

  in which a_1 and a_2 are basic actions.

Data types of action values allow for concise modelling and economical implementation of sets of mutually disabling actions because of the following reasons.

- By modelling a set of mutually disabling actions as an action with an associated data type, the disabling relations between all of the basic actions need not be represented and implemented.
- Data types can, unlike sets of mutually disabling actions, be defined by enumeration of all of their elements, e.g. by means of induction. (For example, the set of natural numbers **N** can be defined by specifying that *0* is an element of **N**, and that if *n* is an element of **N**, then the successor of *n* is an element of **N**.)
- Once defined, data types can be re-used in the modelling of multiple sets of mutually disabling actions. (For example, a data type *mortgage amount* can be associated with both the actions *draft quotation* and *effect mortgage*.)

4.3 Role of reference relations

4.3.1 Inconvenience of current concepts

Inconvenience

1. If the occurrence of an action *b* depends on the value established in another action *a*, this cannot be modelled concisely with the previously introduced architectural concepts.

Instead, *a* should be modelled using the basic concepts as a set of mutually disabling actions comprising basic actions $a_1, a_2, ..., a_n$. The dependence of *b* should then be modelled as a (possibly complex) causality relation of *b*, whose causality condition comprises one or more of the actions $a_1, a_2, ..., a_n$.

2. If the value established in an action *b* depends on the value established in another action *a*, this cannot be modelled concisely with the previously introduced architectural concepts.

   Instead, *a* should be modelled using the basic concepts as a set of mutually disabling actions comprising basic actions $a_1, a_2, ..., a_m$, and *b* should be modelled as a set of mutually disabling actions comprising basic actions $b_1, b_2, ..., b_n$. The dependence of the value established in *b* should then be modelled as (possibly complex) causality relations of $b_1, b_2, ..., b_n$, whose causality conditions comprise one or more of the actions $a_1, a_2, ..., a_m$.

Thus, with out current concepts, in both cases

- sets of mutually disabling actions cannot be modelled concisely as actions in which values may be established, and
- the modelling of the dependence of an action or the value established in an action on values established in other actions may result in lengthy causality relations.

Example of case 1. Assume we wish to specify that

- in an action *a* one of the values *1* to *4* is established;
- an action *b* is enabled by the establishment of a value larger than *1* in *a*.

To specify this, *a({1, 2, 3, 4})* should be specified as a set of four mutually disabling basic actions and *b* should be enabled by the disjunction of the basic actions modelled by *a(2)*, *a(3)*, and *a(4)*. See Figure 4.3.

Example of case 2. Assume we wish to specify that:

- in an action *a* one of the values *1* to *3* is established;
- action *b* becomes enabled by action *a*; in *b* the same value is established as in *a*.

To specify this, *a({1, 2, 3})* and *b({1, 2, 3})* should both be modelled as sets of three mutually disabling basic actions. Each of the basic actions from the set of mutually disabling actions modelled by *a({1, 2, 3})* should enable one of the basic actions from the set of mutually disabling actions modelled by *b({1, 2, 3})*. See Figure 4.4.

4.3.2 Data concepts

Concepts

An *action variable* is the carrier of an action value. It is used enable reference to this action value as follows.

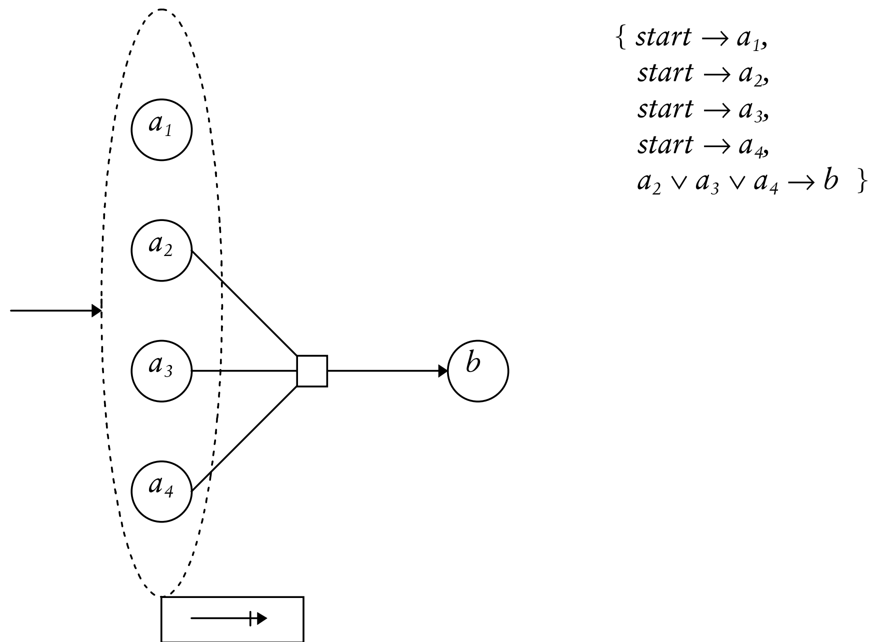


Figure 4.3 Example of inconvenience of case 1.

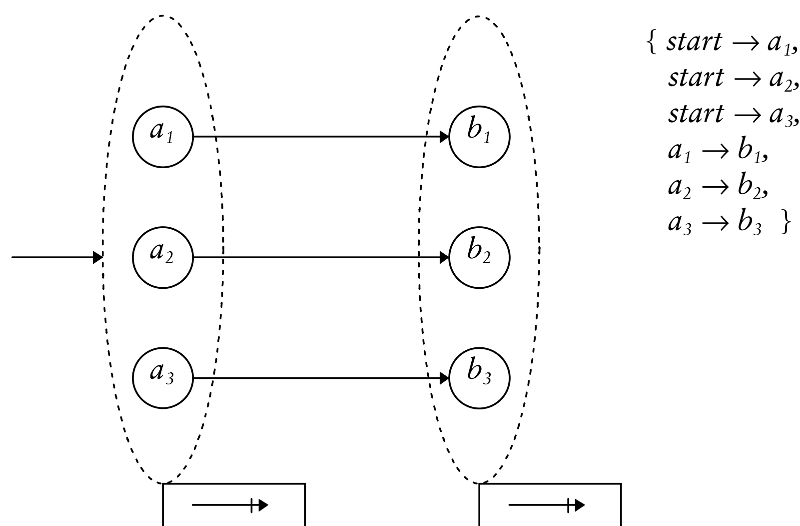


Figure 4.4 Example of inconvenience of case 2.

- A uniquely identified action variable is associated with each combination of an action and a data type. The action variable has the data type it is associated with as its universe of discourse, or, as we call it, it has the data type as its *type*.
- When, during the execution of a behaviour, a value is established in an action, we say the value is *assigned* to the action variable associated with the action.
- An action value can be referred to via the action variable assigned to this value.

A *reference relation* of an action *a* makes the occurrence of *a*, and possibly the value established in *a*, dependent on values established in other actions. A reference rela-

tion of an action a is a part of the causality relation of a. A causality relation may contain multiple reference relations.

Since an action value models the result of an activity, a reference relation models the dependence of (the result of) an activity on results of other activities.

As a shorthand for saying that "the occurrence of an action a, and possibly the values established in a, are dependent on values established in other actions b_1, b_2, ..., and b_n", we say that action a *refers* to actions b_1, b_2, ..., and b_n, or to the values established in these actions, or to the variables to which these values are assigned.

The actions an action can refer to are restricted by the following *reference rule*, which is motivated below: in a behaviour execution an action can only refer to actions by which it is directly or indirectly caused. An action a is directly caused by another action b if b is part of the alternative causality condition that causes the occurrence of a during a behaviour execution. An action a is indirectly caused by another action b if b directly causes another action c that directly causes a, or if b directly causes another action c that indirectly enables a.

We give some examples of the consequences of this reference rule.

- *Direct enabling*. In the behaviour

  $$\{ \quad start \rightarrow a, \\ a \rightarrow b \ \},$$

  b may refer to a, but a may not refer to b.

- *Indirect enabling*. In the behaviour

  $$\{ \quad start \rightarrow a, \\ a \rightarrow b, \\ b \rightarrow c \ \},$$

  c may refer to a.

- *Independence*. In the behaviour

  $$\{ \quad start \rightarrow a, \\ start \rightarrow b \ \},$$

  b may not refer to a and a may not refer to b.

- *Alternative causality conditions*. In the behaviour

  $$\{ \quad start \rightarrow a, \\ start \rightarrow b \\ a \vee b \rightarrow c \ \},$$

  c may either refer to a (in case a causes c during a behaviour execution), or c may refer to b (in case b causes c during a behaviour execution), but c may not refer to both a and b.

The motivation for the reference rule is that we do not wish to introduce other concepts than those of chapter 3 for making the occurrence of an action dependent on the occurrence of another action. (If, for example, two actions *a* and *b* are independent and we would allow *a* to refer to *b*, the occurrence of *a* would (implicitly) be made dependent on the occurrence of *b*.) Instead, we treat the concept of reference relation as a refinement of the concept of causality relation as defined in chapter 3.

We do allow an action to refer to actions that enable it indirectly, because of the transitivity of the enabling relation: if the occurrence of an action *a* depends on the occurrence of an action *b* that depends on the occurrence of an action *c*, the occurrence of *a* implicitly depends on the occurrence of *c*.

Since the actions that may be referred to by an action *a* may vary for the different alternative causality conditions that can cause *a*, we allow for the definition of a distinct reference relation for each alternative causality condition.

From here on we use the term "causality relation" to refer to causality relations that may contain reference relations. In order to denote causality relations that may not contain reference relations, i.e. causality relations as defined in chapter 3, we use the term *basic causality relation*.

Representation

Action variables

We represent the combination of an action *a*, an action variable v_a and its data type D_a as

$$a\ (v_a\!: D_a).$$

An action variable may have any name as long as it is unique in a behaviour.

Reference relations

We represent a reference relation of an action *a* that makes the occurrence of *a* dependent on the action values assigned to the variables $v_1, v_2, \ldots, v_n$ by means of a predicate over $v_1, v_2, \ldots, v_n$. For example, the predicate

$$v_1 + v_2 > 3$$

in the causality relation of an action *a* can be used to represent that *a* may only occur if the sum of two values that have been assigned to v_1 and v_2, is larger than 3. We assume the existence of a suitable language to define and represent predicates.

We represent a reference relation of an action *a* that makes the value established in *a*, which is assigned to an variable v_a, dependent on the action values assigned to the variables $v_1, v_2, \ldots, v_n$ by means of a predicate over $v_a, v_1, v_2, \ldots, v_n$. For example, the predicate

$$v_a = v_1 + v_2$$

in the causality relation of an action a can be used to represent that the value established in a, which is assigned to the variable v_a, is equal to the sum of the values assigned to v_1 and v_2. If no such value exists for v_a, action a may not occur.

When a reference relation R_a of an action a is defined for the alternative causality condition C_a of a, we represent this as the conjunction $C_a \wedge R_a$. Thus, for example, in the behaviour

$$\{ \; start \rightarrow a\ (v_a\!: \{1, 2\}),$$
$$start \rightarrow b\ (v_b\!: \{1, 2\}),$$
$$(a \wedge (v_a{=}1)) \vee (b \wedge (v_b{=}2)) \rightarrow c \; \},$$

the last causality relation represents that if a causes c, c may only occur if $v_a{=}1$, and that if b causes c, c may only occur if $v_b{=}2$.

In the behaviour

$$\{ \; start \rightarrow a\ (v_a\!: \{1, 2\}),$$
$$a \wedge (v_b{=}v_a) \rightarrow b\ (v_b\!:\{1, 2\}) \; \},$$

the last causality relation represents that b is enabled by a and that the value established in b is equal to the value established in a.

Alternative representations of reference relations

Since a reference relation represents a combination of one or more enabling relations (see section 4.3.3), the rules of chapter 3 for representing causality conditions in conjunctive normal form also apply to a conjunction involving reference relations. Thus, for example, the causality relation

$$(a \wedge b \wedge (v_a{=}1)) \vee (a \wedge c \wedge (v_a{=}1)) \rightarrow c,$$

where the value established in a has been assigned to v_a, may be represented alternatively as

$$((a \wedge b) \vee (a \wedge c)) \wedge (v_a{=}1) \rightarrow c.$$

In case a reference relation makes the value established in an action dependent on values established in other actions, we allow for an alternative representation in which reference relations are represented in square brackets behind the result action. For example, the causality relation

$$a \wedge (v_b{=}v_a) \rightarrow b\ (v_b\!:\{1, 2\}),$$

can be represented alternatively as:

$$a \rightarrow b\ (v_b\!:\{1, 2\})\ [v_b{=}v_a].$$

If a causality condition contains multiple alternative causality conditions, we use a notation with *if* to represent for which causality condition a reference relation is defined. For example, the causality relation

$a \vee b \rightarrow c\ (v_c{:}\{1, 2\})\ [\text{if } a{:}\ v_c=v_a+1;\ \text{if } b{:}\ v_c=v_b+1],$

where the values established in *a* and *b* have been assigned to v_a and v_b, respectively, represents that if *a* causes *c*, $v_c=v_a+1$, and that if *b* causes *c*, $v_c=v_b+1$.

Figure 4.5 gives two examples of the graphical representation of reference relations.

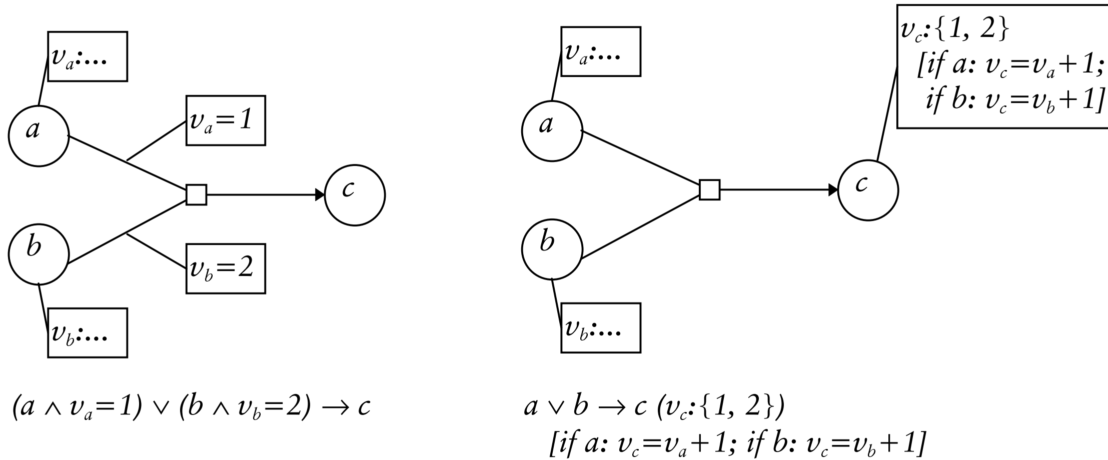


Figure 4.5 Examples of graphical representation of reference relations.

Examples

The example of case 1 of section 4.3.1 can now be represented more concisely as:

$\{\ start \rightarrow a\ (v_a{:}\ \{1, 2, 3, 4\}),$
$\quad a \wedge (v_a > 1) \rightarrow b\ \ \}.$

The example of case 2 of section 4.3.1 can now be represented more concisely as:

$\{\ start \rightarrow a\ (v_a{:}\ \{1, 2, 3\}),$
$\quad a \rightarrow b\ (v_b{:}\ \{1, 2, 3\})\ [v_b=v_a]\ \ \}.$

The value established in an action need not be a function of other values: values can also be established non-deterministically. For example, in the following behaviour, the value established in *b* should only be larger than the value established in *a*:

$\{\ start \rightarrow a\ (v_a{:}\ \{1, 2, 3\}),$
$\quad a \rightarrow b\ (v_b{:}\ \{1, 2, 3\})\ [v_b > v_a]\ \ \}.$

Figure 4.6 represents a slightly more practical behaviour: the drafting of a mortgage quotation by a bank. A person may apply for a mortgage, stating his or her name and income (*apply*). If the applicant is a client of a bank, the maximum mortgage amount is established at four times the applicant's income (*establish max_c*). Otherwise, this is established at three times the applicant's income (*establish max_nc*). Then the interest rate for the mortgage is established at the current day rate (*establish interest*). Finally, the applicant is sent a quotation with the maximum mortgage amount and the interest rate (*send quotation_c* and *send quotation_nc*).

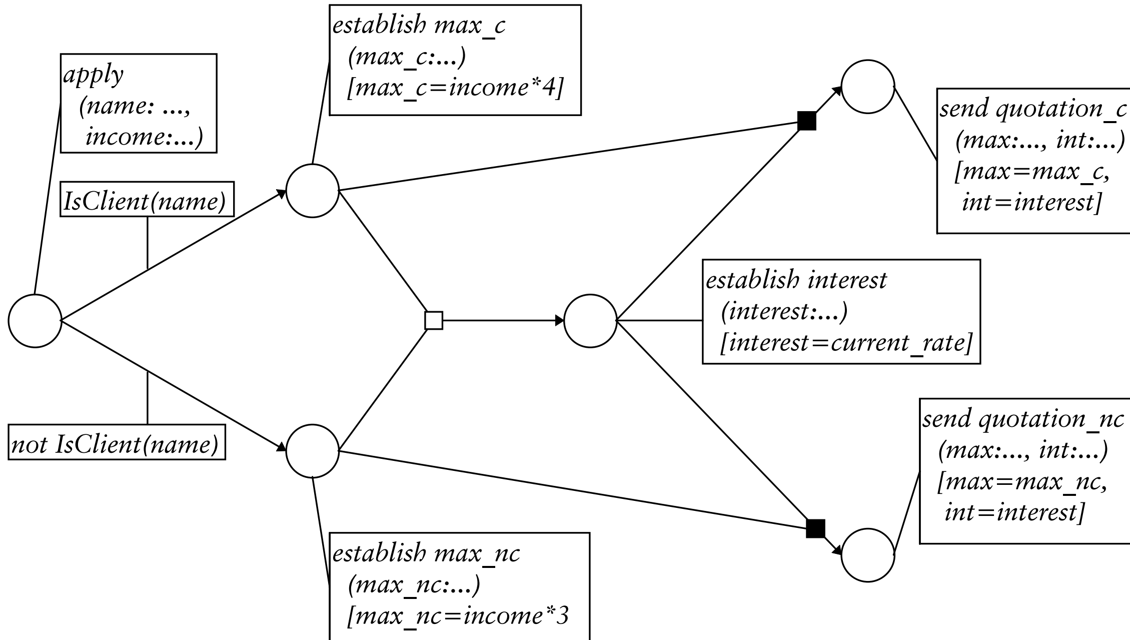


Figure 4.6 Example: drafting of mortgage quotation.

4.3.3 Meaning of data concepts in terms of basic concepts

This section presents a formal procedure for translating causality relations that contain reference relations into basic causality relations.

Reference relations that make an action, but not the value established in this action, dependent on values established in other actions

Let $P(v_1, v_2, ..., v_n)$ be a predicate representing a reference relation that is part of a causality condition C_b of an action b. Assume that the action variables $v_1, v_2, ..., v_n$ are of type $D_1, D_2 ..., D_n$, respectively, and associated with actions $a_1, a_2 ..., a_n$, respectively. This reference relation models a part of the basic causality condition of b. The basic causality condition of b is C_b to which the following modifications are made.

1. If there exist values $c_1 \in D_1, c_2 \in D_2, ..., c_n \in D_n$ such that $P(c_1, c_2, ..., c_n)$ is true, then:

   a. for each combination of values $c_1 \in D_1, c_2 \in D_2, ..., c_n \in D_n$, such that $P(c_1, c_2, ..., c_n)$ is true, a partial causality condition is defined which consists of the conjunction of all of the basic actions modelled by $a_i(c_i)$, $1 \leq i \leq n$;

   b. replace $P(v_1, v_2, ..., v_n)$ in C_b by the disjunction of all of the partial causality conditions defined in step 1.

2. Otherwise, remove the alternative causality condition that contains $P(v_1, v_2, ..., v_n)$ from C_b.

If case 2 results in an empty causality condition, action *b* may never occur and should therefore be removed from the behaviour. The removal of actions may have to be carried out recursively, since *b* may be an enabling action of other actions.

We give an example of the first case, in which values exist for which the predicate is true. In the behaviour specification

$$\{\ start \rightarrow a\ (v_a\text{: } \{1,2,3\}),\\ start \rightarrow b\ (v_b\text{: } \{1,2,3\}),\\ start \rightarrow c,\\ a \wedge b \wedge c \wedge (v_a + v_b > 2) \rightarrow d\ \},$$

the last causality relation concisely models the causality relation

$$c \wedge ((a(1) \wedge b(2)) \vee (a(1) \wedge b(3)) \vee (a(2) \wedge b(1)) \vee (a(2) \wedge b(2)) \vee\\ (a(2) \wedge b(3)) \vee (a(3) \wedge b(1)) \vee (a(3) \wedge b(2)) \vee (a(3) \wedge b(3))) \rightarrow d.$$

We next give an example of the second case, in which no values exist for which the predicate is true. In the behaviour

$$\{\ start \rightarrow a\ (v_a\text{: } \{1,2\}),\\ start \rightarrow b,\\ (a \wedge (v_a{=}3)) \vee b \rightarrow c\ \},$$

the last causality relation models the causality relation

$$b \rightarrow c,$$

since v_a is always either 1 or 2, so that action *c* can never be enabled by action *a* with $v_a{=}3$.

Reference relations that make an action, and the value established in this action, dependent on values established in other actions

Let $P(v_b, v_1, v_2, ..., v_n)$ be a predicate representing a reference relation that is part of a causality relation C_b of an action *b*. Assume that the action variables $v_1, v_2, ..., v_n$ are of type $D_1, D_2 ..., D_n$, respectively, and associated with actions $a_1, a_2 ..., a_n$, respectively, and where v_b is of type D_b and associated with action *b*. This reference relation models a part of the basic causality condition of every basic action modelled by $b(c_b)$, $c_b \in D_b$. The basic causality condition of every basic action modelled by $b(c_b)$, $c_b{\in}D_b$, is the conjunction of C_b and the partial causality condition modelled by $\bigwedge_{x \in D_b \setminus \{c_b\}} \neg b(x)$, to which the following modifications are made.

1. If there exist values $c_1 \in D_1$, $c_2 \in D_2$, ..., $c_n \in D_n$ such that $P(c_b, c_1, c_2, ..., c_n)$ is true, then:

   a. for each combination of values $c_1 \in D_1$, $c_2 \in D_2$, ..., $c_n \in D_n$, such that $P(c_b, c_1, c_2, ..., c_n)$ is true, a partial causality condition is defined which consists of the conjunction of all of the basic actions modelled by $a_i(c_i)$, $1 \leq i \leq n$;

b. replace $P(v_b, v_1, v_2, ..., v_n)$ in C_b by the disjunction of all of the partial causality conditions defined in step 1.

2. Otherwise, remove the alternative causality condition that contains $P(v_b, v_1, v_2, ..., v_n)$ from C_b.

If step 2 results in an empty causality condition, the basic action modelled by $b(c_b)$ may never occur and should therefore be removed from the behaviour. The removal of actions may have to be carried out recursively, since the basic action modelled by $b(c_b)$ may be an enabling action of other actions.

For example, in the behaviour

$$\{ \; start \rightarrow a\ (v_a\text{: } \{1, 2, 3\}), \\ start \rightarrow b\ (v_b\text{: } \{1, 2, 3\}, \\ a \wedge b \wedge (v_a + v_b > v_c) \rightarrow c\ (v_c\text{:}\{2, 3\}) \; \},$$

the last causality relation is a more concise specification of the following causality relations:

$$\neg\, c(3) \wedge ((a(1) \wedge b(2)) \vee (a(1) \wedge b(3)) \vee (a(2) \wedge b(1)) \vee (a(2) \wedge b(2)) \vee \\ (a(2) \wedge b(3)) \vee (a(3) \wedge b(1)) \vee (a(3) \wedge b(2)) \vee (a(3) \wedge b(3))) \rightarrow c(2),$$

and

$$\neg\, c(2) \wedge ((a(1) \wedge b(3)) \vee (a(2) \wedge b(2)) \vee (a(2) \wedge b(3)) \vee (a(3) \wedge b(1)) \vee \\ (a(3) \wedge b(2)) \vee (a(3) \wedge b(3))) \rightarrow c(3).$$

4.4 Role of state values

4.4.1 Inconvenience of current concepts

Inconvenience. If there are many actions whose reference relations all comprise the same function of action values, this function has to be repeated in every reference relation. This results in a lengthy behaviour specification, especially if the specification of the function is lengthy.

Example. Figure 4.7 represents the examination of a student. The student sequentially answers three questions (q_1, q_2, and q_3). If the student answers a question correctly, the value 1 is established in the respective action; if the student answers a question incorrectly, the value 0 is established. The values established in q_1, q_2, and q_3 are assigned to v_{q1}, v_{q2}, and v_{q3}, respectively. Depending on the number of questions answered correctly, the student is awarded a particular grade: an A (*grade A*) if all three questions were answered correctly, a B (*grade B*) if only two questions were answered correctly and a C (*grade C*) if only one question was answered correctly.

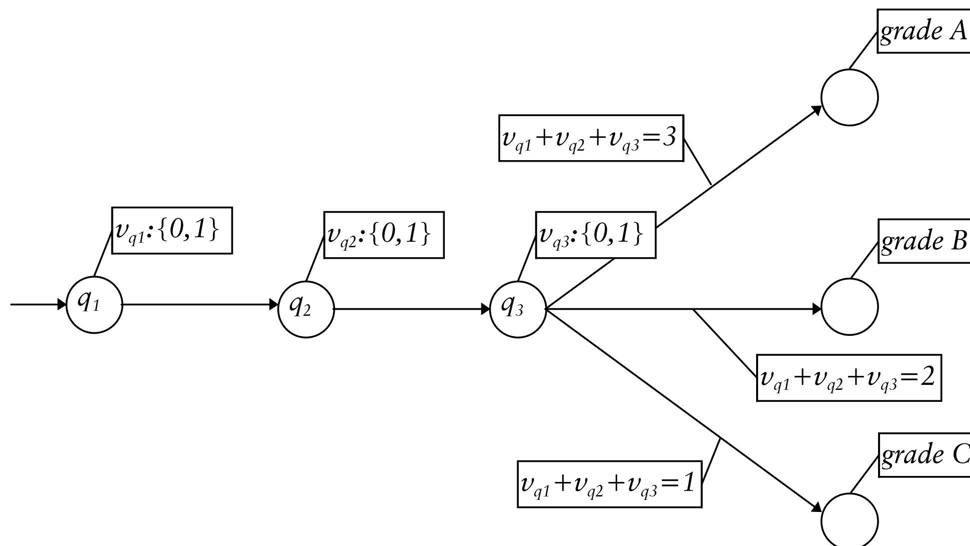


Figure 4.7 Example of function of action values repeated in multiple causality relations.

In this behaviour the function of action values $v_{q1}+v_{q2}+v_{q3}$ is not very long and it is repeated in only three causality relations. However, if the behaviour were modified as follows:

- a student answers 100 questions, rather than 3, and
- a student is awarded one of 100 grades, rather than one of 3,

it would become very inconvenient to repeat a lengthy function of action values in the causality relation of every *grade* action.

4.4.2 Data concepts: formal definition and use

Since the discussion of the architectural semantics of the data concepts defined here requires the definition of some additional notions (history and state), we separate the formal definition of data concepts and the discussion of their architectural interpretation. The latter is done in section 4.4.3.

Concepts

A *state function* is a function of action variables that is associated with a *state variable*. This state variable is assigned the value of the state function; actions may refer to the state variable in the same way as they may refer to an action variable. A *state value* is the value of a state function.

A state function can be defined as a function of other state functions, since a function of functions is a function as well. In that case the state function can thus be rewritten as a function of action variables only. We say that the values of these action variables *determine* the value of the state function.

The reference to state values in a behaviour execution is subject to the same constraints as the reference to action values. Thus, in a behaviour execution, an action *a* can only refer to a state value:

- if all of the action values that determine this state value have been established in actions at the moment *a* occurs;
- if the state value is determined by values established in actions that directly or indirectly cause *a*.

For clarity, we do not allow the definition of state functions whose values cannot be referred to by any action.

Representation

We assume the existence of a suitable language to define and represent state functions. We represent the association of a state variable *vs* and a state function $fs(v_1, v_2, \ldots, v_n)$, in which $v_1, v_2, \ldots, v_n$ are action variables and/or state variables, as

$$< vs = fs(v_1, v_2, \ldots, v_n) >.$$

The triangular brackets are used to distinguish state variables from action variables. Unless clarity demands so, we do not represent the data type of the state variable, since this data type is the co-domain of its associated state function.

Since state variables are, unlike action variables, not coupled to a specific action, they may be represented anywhere in the behaviour. However, for clarity we use the following conventions. (An action is the final action of a set of actions if it always occurs after all other actions from the set have occurred.)

- If one of the actions whose action values determine a state value is the final action of these actions, the state variable to which this state value is assigned is represented directly after the causality relation of this final action.

  For example, in the behaviour

  $\{\ start \rightarrow a\ (v_a{:}\{1,2\}),$
  $a \rightarrow b\ (v_b{:}\{1,2\}),$
  $b \rightarrow c\ (v_c{:}\{1,2\}) < vs{=}v_a{+}v_b{+}v_c >,$
  $c \wedge (vs > 4) \rightarrow d\ \},$

  the state variable *vs* and its associated state function $v_a+v_b+v_c$ are represented directly after the causality relation of action *c*, since *c* always occurs as the last action of the actions *a*, *b*, and *c*. Figure 4.8 represents this behaviour graphically.

- If none of the actions whose action values determine a state value is a final action of these actions, the state variable to which this state value is assigned is represented:

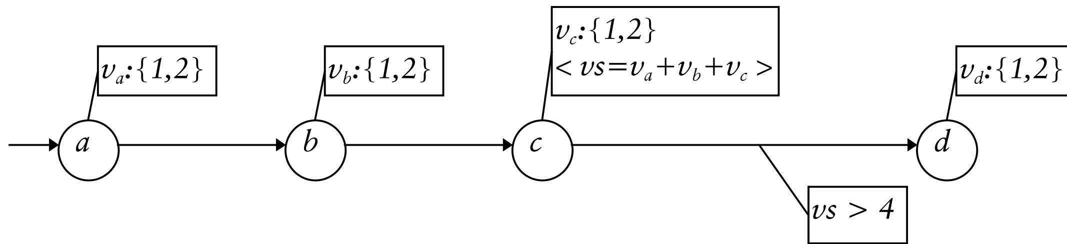


Figure 4.8 Example of graphical representation of state variable and state function in case a final action exists.

- in the textual notation "somewhere" after the causality relations of these actions;
- in the graphical notation in a box connected to a dotted line around the actions whose action values determine the state value.

For example, in the behaviour

$$
\begin{aligned}
\{\ \ & start \rightarrow a\ (v_a{:}\{1,2\}), \\
& start \rightarrow b\ (v_b{:}\{1,2\}), \\
& < vs=v_a+v_b >, \\
& a \wedge b \rightarrow c, \\
& c \wedge (vs=4) \rightarrow d\ \ \},
\end{aligned}
$$

the state variable *vs* and its associated state function v_a+v_b are represented after the causality relations of actions *a* and *b*, since none of these actions is a final action. Figure 4.9 represents this behaviour graphically.

Examples

Example 1: student examination

The grade assignment example of section 4.4.1 can now be represented more concisely as in Figure 4.10. The behaviour of Figure 4.10 is only slightly more concise than the behaviour of section 4.4.1, because the state variable *vs* is used in only 3 reference relations, and because the state function associated with *vs* is not very long. However, the modification of this behaviour mentioned in section 4.4.1, with 100 questions and 100 grade actions, would become considerably more concise by the use of state variables and state functions.

Example 2: calculation of income tax

Another example concerns the calculation of the income tax to be paid by a person, e.g. by the tax office. This tax is calculated as follows:

- The tax is levied over a person's taxable income (*tax_inc*). This taxable income consists of the person's income minus the person's deduction sum. The income consists of income out of labour (*lab_inc*) plus income out of debit interest

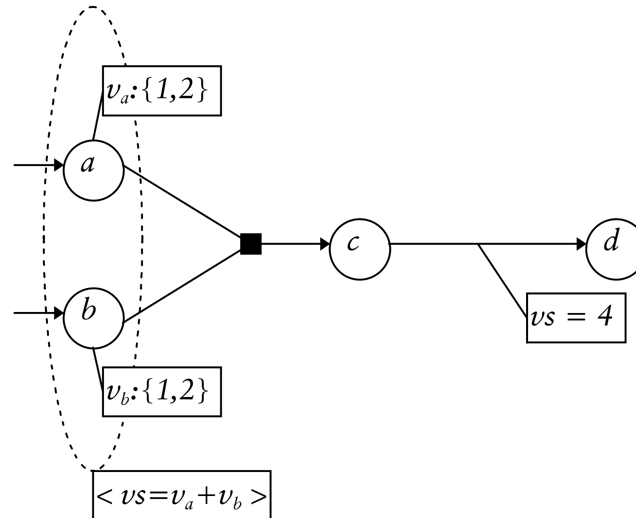


Figure 4.9 Example of graphical representation of state variable and state function in case no final action exists.

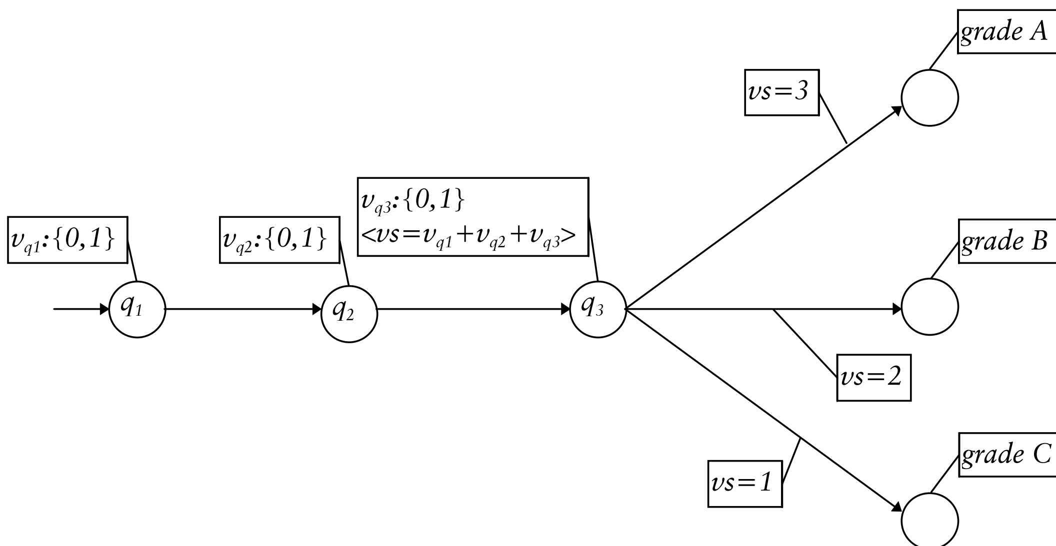


Figure 4.10 More concise specification of grade assignment example of section 4.4.1.

(*deb_inc*) plus other income (*oth_inc*). The deduction sum consists of credit interest (*cred_ded*) plus professional expenses (*prof_ded*).

- No tax is paid over the first \$5,000 of the taxable income. A person not earning more than \$5,000 is liable to tax class 0.
- The next \$30,000 of the taxable income are charged at a rate of 20%. A person earning more than \$5,000, but not more than \$35,000, is liable to tax class 1.
- The remainder of the income is taxed at a rate of 40%. A person earning more than \$35,000, is liable to tax class 2.

With the use of state values the calculation of a person's income tax can be specified concisely as follows.

{ *start* → *get_lab_income (lab_inc:$),*
get_lab_income → *get_deb_income (deb_inc:$),*
get_deb_income → *get_oth_income (oth_inc:$),*

get_oth_income → *get_cred_ded (cred_ded:$),*
get_cred_ded → *get_prof_ded (prof_ded:$)*
< *tax_inc=(lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)* >,

get_prof_ded ∧ *(tax_inc* ≤ *5,000)* → *tax_class0* (tax_0*:$)*
[tax_0*=0*],

get_deduction ∧ *(tax_inc > 5,000)* ∧ *(tax_inc* ≤ *35,000)* → *tax_class1* (tax_1*:$)*
[tax_1*=20%*(tax_inc-5,000)*],

get_deduction ∧ *(tax_inc > 35,000)* → *tax_class2* (tax_2*:$)*
[tax_2*=20%*(tax_inc-5,000)+40%*(tax-inc-35,000)*] }.

The behaviour can be specified, less comprehensibly, without state variables and state functions:

{ *start* → *get_lab_income (lab_inc:$),*
get_lab_income → *get_deb_income (deb_inc:$),*
get_deb_income → *get_oth_income (oth_inc:$),*

get_oth_income → *get_cred_ded (cred_ded:$),*
get_cred_ded → *get_prof_ded (prof_ded:$),*

get_prof_ded ∧
(((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)) ≤ *5,000)*
→ *tax_class0* (tax_0*:$)*
[tax_0*=0*],

get_deduction ∧
(((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)) > 5,000) ∧
(((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)) ≤ *35,000)*
→ *tax_class1* (tax_1*:$)*
[tax_1*=20%*((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)*
-5,000)],

get_deduction ∧
(((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)) > 35,000)
→ *tax_class2* (tax_2*:$)*
[tax_2*=20%*((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)*
-5,000)+

40%((lab_inc + deb_inc + oth_inc) - (cred_ded + prof_ded)-5,000)] }.*

Example 3: operations on a set

A third example shows the use of state variables and state functions in recursive behaviours. This example concerns the operations *insert*, *delete*, and *list* on a set that conforms to the following requirements:

- Initially the set is empty.
- None of the actions *insert*, *delete*, and *list* may take place at the same time.
- The action *insert* in which a value is established inserts this value into the set; the action may only take place if this value is not already in the set.
- The action *delete* in which a value is established removes this value from the set; the operation may only take place if this value is already in the set.
- In the action *list* a value is established that is the set.

This behaviour can be specified concisely with the use of state variables and state functions. In the specification below E contains the elements the set may contain and the symbol $\wp$ denotes the power set function. (The specification uses coupled parametrised exit and entry points to pass values from one copy of the behaviour to another. Parametrised exit points and entry points are discussed in section 4.5.2; here we assume their meaning to be understood.)

$\{ \ start \rightarrow B(entry(\emptyset)), B(exit(S'\!: \wp(E))) \rightarrow B(entry(S\!: \wp(E))) \ \}$

$B =$
$\{ \ choice(insert, delete, list),$
$\quad entry(S) \rightarrow insert\ (v_i\!:\!E)\ [v_i \notin S] \ < S = S \cup \{v_i\} >,$
$\quad entry(S) \rightarrow delete\ (v_d\!:\!E)\ [v_d \in S] \ < S = S \setminus \{v_d\} >,$
$\quad entry(S) \rightarrow list\ (v_l\!: \wp(E))\ [v_l = S],$
$\quad insert \vee delete \vee list \rightarrow exit(S)$
$\}.$

In this example the variable S is assigned a new value after an *insert* or *delete* action. Such re-assignment of values to variables is allowed, provided it does not result in ambiguous value references.

We do not know of a comprehensible specification of this behaviour without the use state variables and state functions. However, by restricting the elements of the set S to 1 and 2, the behaviour can be specified (albeit not very comprehensibly) without state variables and state functions as follows.

$\{ \ start \rightarrow B(entry_{\emptyset}), \quad B(exit_{\emptyset}) \rightarrow B(entry_{\emptyset}), B(exit_1) \rightarrow B(entry_1),$
$\qquad B(exit_2) \rightarrow B(entry_2), B(exit_{12}) \rightarrow B(entry_{12}), \ \}$

$B=$
$\{$ *choice(insert, delete, list),*
  $entry_{\emptyset} \vee entry_1 \vee entry_2 \rightarrow insert\ (v_i{:}\{1, 2\})$
    $[\quad if\ entry_1{:}\ v_i=2;$
      $if\ entry_2{:}\ v_i=1 \quad],$
  $entry_1 \vee entry_2 \vee entry_{12} \rightarrow delete\ v_d{:}\{1, 2\}$
    $[\quad if\ entry_1{:}\ v_d=1;$
      $if\ entry_2{:}\ v_d=2 \quad]$
  $entry_{\emptyset} \vee entry_1 \vee entry_2 \vee entry_{12} \rightarrow list\ (v_l{:}\{\emptyset, \{1\}, \{2\}, \{1, 2\}\})$
    $[\quad if\ entry_{\emptyset}:\ v_l=\emptyset;$
      $if\ entry_1{:}\ v_l=\{1\};$
      $if\ entry_2{:}\ v_l=\{2\};$
      $if\ entry_{12}{:}\ v_l=\{1, 2\} \quad]$
  $(entry_{\emptyset} \wedge list) \vee (entry_1 \wedge delete \wedge (v_d=1)) \vee (entry_2 \wedge delete \wedge (v_d=2)) \rightarrow exit_{\emptyset},$
  $(entry_1 \wedge list) \vee (entry_{\emptyset} \wedge insert \wedge (v_i=1)) \vee (entry_{12} \wedge delete \wedge (v_d=2)) \rightarrow exit_1,$
  $(entry_2 \wedge list) \vee (entry_{\emptyset} \wedge insert \wedge (v_i=2)) \vee (entry_{12} \wedge delete \wedge (v_d=1)) \rightarrow exit_2,$
  $(entry_{12} \wedge list) \vee (entry_1 \wedge insert \wedge (v_i=2)) \vee (entry_2 \wedge insert \wedge (v_i=1)) \rightarrow exit_{12}$
$\}$.

In this specification each possible state of the behaviour (see below) is represented by an entry point. Since the number of states of the behaviour is 2^n, where n is the maximum cardinality of the set S, this specification clearly does not scale up well for larger values of n.

4.4.3 Data concepts: architectural semantics

The main objectives of this section are to show:

- that the value of a state variable in a behaviour models one or more states of the behaviour, where the state concept conforms to a customary notion of state, as in e.g. finite-state machines;
- under what conditions state variables and state functions are useful in modelling behaviours concisely.

Preliminary notions: histories, equivalence classes of histories, and states

We first introduce some notions, which are based on [Minsky, 1967], which allow us to describe the architectural semantics of state values concisely.

Histories

The *history* of a behaviour that is executed at a time t is the complete set of basic actions that have occurred in the behaviour up till and including time t. If H is a history of a behaviour that is executed at a time t, we say that this behaviour *has* history H at that time.

(The difference between a history of a behaviour at a time t and the behaviour execution of the behaviour—which was defined in chapter 3—at time t is that the relations between actions that have occurred at time t are not part of the history, whereas they are part of the behaviour execution. We need not consider these relations between actions to explain how certain causality conditions can be modelled more concisely, since the order in which past actions have occurred is irrelevant for the occurrence of future actions.)

Figure 4.11 shows a behaviour that is being executed at different points in time. In this figure, grey circles represent actions that have occurred at a particular time and white circles represent actions that have not occurred at that time. The initial history of the behaviour, when no action has occurred, is the empty set (Figure 4.11a). If action a_1 then occurs, the history of the behaviour is $\{a_1\}$ (Figure 4.11b). If next action b occurs, the history is $\{a_1, b\}$ (Figure 4.11c). If finally action d occurs, the history is $\{a_1, b, d\}$ (Figure 4.11d). The other possible histories of the behaviour are $\{a_2\}$, $\{a_2, c\}$, and $\{a_2, c, d\}$. The set $\{b, d\}$, for example, cannot be a history of the behaviour at any time, since it is impossible for only the actions b and d to have occurred in the behaviour.

If action values are used in a behaviour specification, actions in which values have been established can be used to model the history of a behaviour at a particular time, since an action in which a value is established models a basic action. Take, for example, the following behaviour:

$\{\ start \rightarrow a\ (v_a{:}\{1,2\}),$
$\quad a \rightarrow b\ (v_b{:}\{1,2\})\ \ \}.$

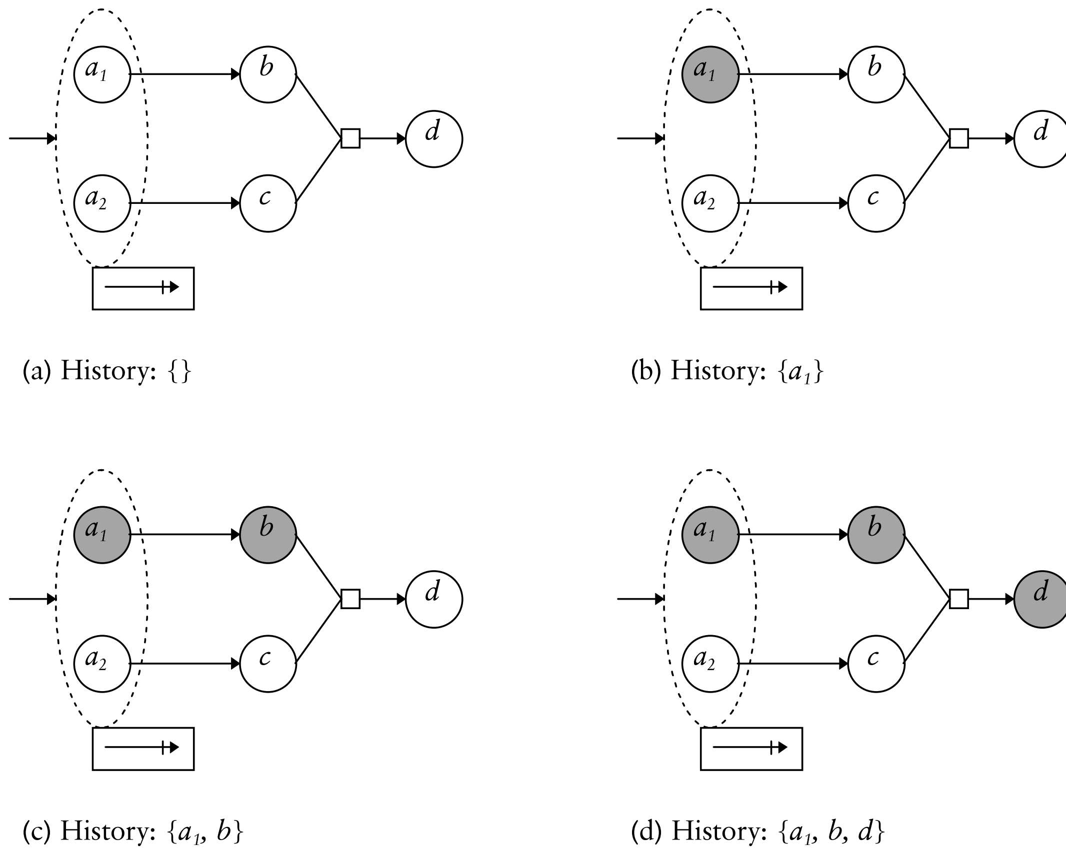
The history of the behaviour after only a has occurred, in which the value *1* was established, can be represented as $\{a(1)\}$. The history of the behaviour after b has also occurred, in which the value *2* was established, can be represented as $\{a(1), b(2)\}$.

States: equivalence classes of histories with respect to the future of a behaviour

The *future* of a behaviour B that is executed at a time t is the complete set of behaviour executions of B that are possible from time t.

We use the behaviour of Figure 4.11 as an example.

- When no action has occurred yet (Figure 4.11a), the future consists of two behaviour executions: one in which a_1, b, and d occur, and one in which a_2, c, and d occur.
- When action a_1 has occurred (Figure 4.11b), the future consists of the behaviour execution in which b and d occur.
- When also b has occurred (Figure 4.11c), the future consists of the behaviour execution in which the initially enabled action d may occur.



(a) History: {}

(b) History: $\{a_1\}$

(c) History: $\{a_1, b\}$

(d) History: $\{a_1, b, d\}$

Figure 4.11 Histories of a behaviour that is being executed at different points in time.

- When also action d has occurred (Figure 4.11d), the future consists of the empty behaviour execution, since no actions may occur anymore.

When a behaviour has a particular future, it has one out of a group of possible histories. For example, when the behaviour of Figure 4.11 has the future that consists of the behaviour execution in which the initially enabled action d may occur, it has one of the two possible histories $\{a_1, b\}$ and $\{a_2, c\}$. The difference between these histories is irrelevant with respect to the future of the behaviour. We introduce an equivalence notion to formalise this.

Two histories H_1 and H_2 of a behaviour are *equivalent with respect to the future of the behaviour* if and only if history H_1 determines the same future of the behaviour as history H_2.

An equivalence relation over the set of histories of a behaviour defines a number of *equivalence classes* of the histories of this behaviour. An equivalence class in general is a set of elements are equivalent with respect to an equivalence relation.

A *state* is an equivalence class of histories under equivalence with respect to the future of a behaviour. This notion of state conforms to the notion of state in finite state machines [Minsky, 1967].

The example behaviour of Figure 4.11 has five states:

- {}, corresponding to the future consisting of two behaviour executions: one in which a_1, b, and d occur, and one in which a_2, c, and d occur;
- $\{\{a_1\}\}$, corresponding to the future consisting of the behaviour execution in which b and d occur;
- $\{\{a_2\}\}$, corresponding to the future consisting of the behaviour execution in which c and d occur;
- $\{\{a_1, b\}, \{a_2, c\}\}$, corresponding to the future consisting of the behaviour execution in which d occurs;
- $\{\{a_1, b, d\}, \{a_2, c, d\}\}$, corresponding to the future consisting of the empty behaviour execution.

When we say a behaviour *is in* a particular state, we mean that at a particular time the behaviour has one of the histories from the state.

The relation between a set of elements and the equivalence classes of these elements is a (surjective) function from the former to the latter [Stanat & McAllister, 1977]. Therefore, the relation between the histories of a behaviour and its states is a *function*.

Equivalence classes of histories with respect to actions

Two histories H_1 and H_2 of a behaviour are *equivalent with respect to (the occurrence of) an action a* in the behaviour if and only if a is initially enabled when the behaviour has history H_1 or history H_2.

For example, in the behaviour of Figure 4.11, the histories $\{a_1, b\}$ and $\{a_2, c\}$ are equivalent with respect to action d, since d may occur either when actions a_1 and b have occurred, or when actions a_2 and c have occurred.

Under equivalence with respect to an action a, there are only two equivalence classes:

- the equivalence class $[H]_{a^+}$ of histories, such that for every $H \in [H]_{a^+}$: if the behaviour has history H, a is initially enabled;
- the equivalence class $[H]_{a^-}$ of histories, such that for every $H \in [H]_{a^-}$: if the behaviour has history H, a is not initially enabled.

For example, for the behaviour of Figure 4.11, the equivalence class $[H]_{d^+}$ is: $\{\{a_1, b\}, \{a_2, c\}\}$. $[H]_{d^-}$ is the set of all other histories of the behaviour.

We next consider the relation between on the one hand the two equivalence classes of histories of a behaviour under equivalence with respect to an action a, $[H]_{a^+}$ and $[H]_{a^-}$, and on the other hand the states $S_1, S_2, ..., S_n$ of this behaviour.

- For a particular behaviour, $[H]_{a^+}$ is the union of all of the states S_i, $i=1..n$, such that a is initially enabled when the behaviour is in state S_i. This is proven as follows:

  1. each S_i, $i=1..n$, such that a is initially enabled when the behaviour is in state S_i, only contains histories, such that a is initially enabled if the behaviour has any of these histories;

  2. therefore, the union of every S_i, $i=1..n$, such that a is initially enabled when the behaviour is in state S_i, contains all histories, such that a is initially enabled if the behaviour has any of these histories, and it contains no other histories;

  3. $[H]_{a^+}$ also contains all histories, such that a is initially enabled if the behaviour has any of these histories, and it contains no other histories.

  Therefore, $[H]_{a^+}$ is the union of all of the states S_i, $i=1..n$, such that a is initially enabled when the behaviour is in state S_i.

- For a particular behaviour, $[H]_{a^-}$ is the union of all of the states S_i, $i=1..n$, such that a is not initially enabled when the behaviour is in state S_i. The proof of this is similar to the proof above.

Consider for example the behaviour of Figure 4.12. The equivalence class of histories $[H]_{b^+}$ is $\{\{a_1\}, \{a_2\}\}$. This is the union of the states $\{\{a_1\}\}$ and $\{\{a_2\}\}$. (Although the histories $\{a_1\}$ and $\{a_2\}$ are equivalent with respect to the occurrence of action b, they are not equivalent with respect to future behaviour, since the occurrence of a_1 is necessary for the occurrence of c, whereas the occurrence of a_2 is necessary for the occurrence of d.)

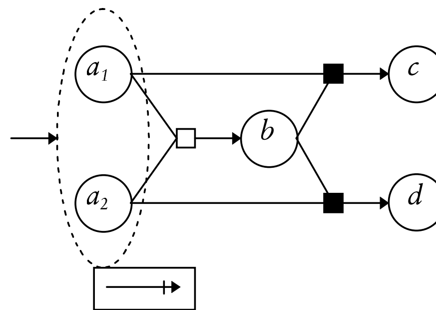


Figure 4.12 Example behaviour.

Architectural semantics of state values

A state variable is useful because the occurrence of an action, and possibly the value established in this action, can be made dependent on its value. Here we show what such a state value models.

Let a be an action whose occurrence depends on the value cs of a state variable vs. By definition, every state function can be re-written as a function of action variables

only. Let $fs(v_1, v_2, ..., v_m)$ be the state function associated with vs, in which $v_1, v_2, ...,$ v_m are action variables of data types $D_1, D_2, ..., D_m$, respectively.

1. Let C_i be a tuple of action values $<c_{i1}, c_{i2}, ..., c_{im}>$, $c_{i1} \in D_1$, $c_{i2} \in D_2$, ..., $c_{im} \in D_m$, such that $fs(c_{i1}, c_{i2}, ..., c_{im})=cs$. Let $C_1, C_2, ..., C_n$ be all of the tuples with this property.

   Then the dependence of the occurrence of *a* on the value *cs* of *vs* models that:

   > the occurrence of *a* depends on the establishment of all values from one of the sets C_i, $i=1..n$.

2. The establishment of an action value models a basic action. Let each A_i, $i=1..n$, be the set of basic actions modelled by the establishment of all action values from C_i.

   Then the dependence of the occurrence of *a* on the value *cs* of *vs* also models that:

   > the occurrence of *a* depends on the occurrence of all basic actions from one of the sets A_i, $i=1..n$.

3. Let H_j be a history, such that there exists an A_i, $i=1..n$, $A_i \subset H_j$ and $a \notin H_j$. Let H_1, H_2, ..., H_p be all histories with this property.

   The statement that

   > the occurrence of an action *a* depends on the occurrences of all actions from a set *A*

   is equivalent to the statement that

   > *a* may only occur when the behaviour has a history *H*, such that $A \subset H$ and $a \notin H$.

   Therefore, the dependence of the occurrence of *a* on the value *cs* of *vs* also models that:

   > *a* may only occur when the behaviour has one of the histories H_1,. H_2, ..., H_p.

4. Assume that the occurrence of *a* depends only on the value *cs* of *vs*. Then $\{H_1,. H_2, ..., H_p\}$ is the equivalence class of histories $[H]_{a^+}$, since *a* may only occur when the behaviour has one of the histories H_1,. H_2, ..., H_p. Let $S_1, S_2, ..., S_q$ be all states, such that *a* is initially enabled when the behaviour is in state S_k, $k=1..q$.

   It was shown above that $[H]_{a^+}= S_1 \cup S_2, ..., S_q$. Therefore, the dependence of the occurrence of *a* on the value *cs* of *vs* also models that:

   > *a* may only occur when the behaviour is in one of the states $S_1, S_2, ..., S_q$.

For brevity, we say that the value *cs* of *vs models the states* $S_1, S_2, ..., S_q$.

As an example, we consider the behaviour of Figure 4.13.

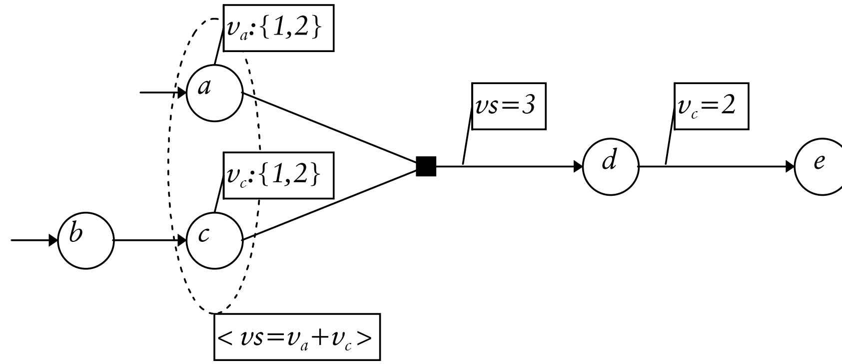


Figure 4.13 Example behaviour.

1. The dependence of *d* on the value *3* of state variable *vs* models that the occurrence of *d* depends on either:
   - the establishment of the value *1* in *a*, i.e. *a(1)*, and the establishment of the value *2* in *c*, i.e. *c(2)*, or
   - the establishment of the value *2* in *a*, i.e. *a(2)*, and the establishment of the value *1* in *c*, i.e. *c(1)*, or
2. Let a_1, a_2, c_1, and c_2 be the basic actions modelled by *a(1)*, *a(2)*, *c(1)*, and *c(2)*, respectively. Then the statement of (1) is equivalent to the following statement.

   The dependence of *d* on the value *3* of *vs* models that the occurrence of *d* depends on either:
   - the occurrence of all of the basic actions in $A_1=\{a_1, c_2\}$, or
   - the occurrence of all of the basic actions in $A_2=\{a_2, c_1\}$.
3. $H_1=\{a_1, b, c_2\}$ is the only history, such that $A_1 \subset H_1$ and $d \notin H_1$. $H_2=\{a_2, b, c_1\}$ is the only history, such that $A_2 \subset H_2$ and $d \notin H_2$. Then the statement of (2) is equivalent to the following statement.

   The dependence of *d* on the value *3* of *vs* models that *d* may only occur when the behaviour has the history $\{a_1, b, c_2\}$ or the history $\{a_2, b, c_1\}$.
4. The equivalence class of histories $[H]_{d^+}$ is $\{\{a_1, b, c_2\}, \{a_2, b, c_1\}\}$. $[H]_{d^+}$ is the union of the states $\{\{a_1, b, c_2\}\}$ and $\{\{a_2, b, c_1\}\}$. These two states are all states of the behaviour, such that *d* is initially enabled when the behaviour is in one of them. Therefore, the statement of (3) is equivalent to the following statement.

   The dependence of *d* on the value *3* of *vs* models that *d* may only occur when the behaviour is in one of the states $\{\{a_1, b, c_2\}\}$ and $\{\{a_2, b, c_1\}\}$.

Thus, we say that the value 3 of *vs* models the states $\{\{a_1, b, c_2\}\}$ and $\{\{a_2, b, c_1\}\}$.

Further analysis: usefulness of state variables and state functions

Let *vs* be a state variable which is assigned the value of the state function $fs(v_1, v_2, ..., v_m)$. We distinguish the following condition for *vs* and $fs(v_1, v_2, ..., v_m)$ to be useful in making a behaviour specification more concise, and thereby more comprehensible and more economical to implement.

> *The behaviour must contain multiple actions whose reference relations comprise* $fs(v_1, v_2, ..., v_m)$.

We motivate this condition in two steps.

1. The behaviour must contain at least one action whose reference relation comprises $fs(v_1, v_2, ..., v_m)$.

This is a necessary condition for the following reason. The state variable *vs* is only useful if it can be used to replace $fs(v_1, v_2, ..., v_m)$ in the reference relation of an action. Therefore, if there is no action whose reference relation comprises $fs(v_1, v_2, ..., v_m)$, it is not useful to define *vs*.

For example, in the behaviour of section 4.4.1 (the examination of a student), the reference relation of action *grade A* comprises the function $v_1+v_2+v_3$. In the more concise specification of this behaviour in section 4.4.2, this function was replaced by the state variable *vs*, which is assigned the value of the state function $v_1+v_2+v_3$.

2. The behaviour must contain multiple actions whose reference relations comprise $fs(v_1, v_2, ..., v_m)$.

Condition 2 is a necessary condition for the following reason. If there were only one action whose reference relation comprised $fs(v_1, v_2, ..., v_m)$, it would be more concise to directly use $fs(v_1, v_2, ..., v_m)$ in this reference relation, rather than to first assign the value of $fs(v_1, v_2, ..., v_m)$ to *vs* and then use *vs* in the reference relation.

Condition 2 is also a sufficient condition, since the only two options for specifying reference relations that contain $fs(v_1, v_2, ..., v_m)$ are either: 1) to directly use $fs(v_1, v_2, ..., v_m)$ in these reference relations, which is lengthy, or 2) to first assign the value of $fs(v_1, v_2, ..., v_m)$ to a state variable and then use this state variable in the reference relation, which is more concise.

For example, in the behaviour of section 4.4.1, not only the reference relation of action *grade A* comprises the function $v_1+v_2+v_3$, but the reference relations of actions *grade B* and *grade C* do so as well. In the more concise specification of this behaviour in section 4.4.2, the state variable *vs* is used in the reference relations of all of these actions. By doing so, the behaviour becomes more concise.

> *The more actions whose reference relations comprise* $fs(v_1, v_2, ..., v_m)$ *the behaviour contains, and the longer the definition of* $fs(v_1, v_2, ..., v_m)$ *is, the more useful the use of the state variable vs becomes.*

This is because the more often one uses *vs* in a reference relation instead of $fs(v_1, v_2, ..., v_m)$, and the more space is gained each time by doing so, the more concise the behaviour becomes.

This clarifies why recursively defined state functions, such as in example 2 of section 4.4.2, are often especially useful. In the example, the occurrences of the actions *insert* and *delete* depend, in every but the first copy of the behaviour, on the value of a state function that is a function of an action variable and a state function from the previous copy of the behaviour.

- For example, in the 100th copy of the behaviour, the value of the state function is a function of action values established in the previous 99 copies of the behaviour. The state function is thus a function of many action variables and its non-recursive definition in terms of all of these action variables would be extremely lengthy.
- The value of the state function in the 100th copy of the behaviour is used in all following copies of the behaviour. If no recursively defined state function were used, this very lengthy function would have to be explicitly defined as a condition for the actions *insert* and *delete* in all following copies of the behaviour. This would result in an extremely lengthy behaviour.

We finally illustrate that the use of state variables may result in a more lengthy behaviour, rather than a more concise one, if the above condition is not satisfied. Consider the behaviour

{ *start* → *a*,
a → *b*,
b → *c*,
c → *d* }.

This behaviour can be specified less comprehensibly with the use of a state variable as follows:

{ *start* → *B(entry("start"))* }

B=
{ *entry(vs: {"start", "a", "b", "c", "d"})*
∧ *(vs≠"d")* → *action*(v_{action}:{"start", "a", "b", "c", "d"}),
[*if vs="start" then* v_{action}="a",
if vs="a" then v_{action}="b",
if vs="b" then v_{action}="c",
if vs="c" then v_{action}="d"]
action → *B(entry*(v_{action})) }.

4.4.4 Meaning of data concepts in terms of basic concepts

Since every state function can be re-written as a function of action values, we can re-write a behaviour specification that contains state variables and state functions as a behaviour that does not contain either of these. This can be done as follows.

1. Re-write every state function in the behaviour as a function of action values only. This can be done by, possibly repeated, substitution.
2. Replace every state variable in every reference relation by its associated state function.
3. Remove all remaining state variables and their associated state functions from the behaviour.

After the application of these steps, the resulting behaviour does not contain any state variables anymore. Therefore, the procedures of section 4.3.3 can be applied for re-writing a behaviour in terms of basic concepts only.

As an example, we consider the following behaviour

$\{\ start \rightarrow a\ (v_a{:}\{1,2\}),$
$\quad start \rightarrow b\ (v_b{:}\{1,2\}),$
$\quad < vs_1=v_a+v_b >,$
$\quad a \wedge b \rightarrow c\ (v_c{:}\{1,2\}) < vs_2=vs_1 * v_c >,$
$\quad c \rightarrow d\ (v_d{:}\{1,2\}) < vs_3=vs_2 - v_d >,$
$\quad d \wedge (vs_3=1) \rightarrow e\ \ \}.$

The state function $vs_1 * v_c$, whose value is assigned to vs_2, can be re-written as

$(v_a+v_b) * v_c.$

The state function $vs_2 - v_d$, whose value is assigned to vs_3, can be rewritten as

$((v_a+v_b) * v_c) - v_d.$

By replacing vs_3 in the reference relation of action e by $((v_a+v_b) * v_c) - v_d$ and removing all remaining state variables and their associated state functions from the behaviour, the following behaviour is obtained:

$\{\ start \rightarrow a\ (v_a{:}\{1,2\}),$
$\quad start \rightarrow b\ (v_b{:}\{1,2\}),$
$\quad a \wedge b \rightarrow c\ (v_c{:}\{1,2\}),$
$\quad c \rightarrow d\ (v_d{:}\{1,2\}),$
$\quad d \wedge (((v_a+v_b) * v_c) - v_d) =1) \rightarrow e\ \ \}.$

This behaviour can be re-written as a behaviour in terms of basic concepts only by means of the procedures of section 4.3.3.

4.5 Extensions

4.5.1 Extensions for disabling and synchronisation relations

Disabling relations

Let, in a behaviour, *b* be a disabling action, and not an enabling action or a synchronisation action, of an action *a*. The following possibilities exist in the execution of this behaviour for the occurrences of *a* and *b* and their relation.

- *b* occurs before *a* has occurred. Then *a* cannot occur anymore. Therefore, *a* cannot refer to refer to the value established in *b*.
- *b* occurs after *a* has occurred. In chapter 3 we argued that we do not allow the occurrence of an action to depend on the occurrences of future actions. Since a reference relation of an action makes the occurrence of this action dependent on other actions, *a* cannot refer to a value established in *b*. (However, *b* can refer to *a*, if *a* has occurred.)
- *b* does not occur. If *a* occurs, it cannot refer to a value established in *b*, since no value is established in *b*. If *a* does not occur, it cannot refer to any value at all.

We conclude that an action can never refer to the value established in one of its disabling actions which is not also one of its enabling actions or synchronisation actions.

Thus, for example, the behaviour specification

$$\{\ a \vee \neg a \rightarrow b\ (v_b{:}\{1,2\}),$$
$$\neg b \rightarrow a\ (v_a{:}\{1,2\})\ [v_b{=}v_a]\ \ \}$$

is meaningless, since *a* cannot refer to *b*. However, the behaviour specification

$$\{\ a \vee \neg a \rightarrow b\ (v_b{:}\{1,2\}),$$
$$\neg b \vee b \rightarrow a\ (v_a{:}\{1,2\})\ [\mathit{if}\ b{:}\ v_a{=}v_b]\ \ \}$$

is allowed, since *a* can refer to *b* if it is caused by *b* in a behaviour execution.

Synchronisation relations

Let *b* be a synchronisation action of an action *a*. If *a* and *b* synchronise in a behaviour execution, the occurrence of *a* depends on the occurrence of *b*. Therefore, *a* can refer to the value established in *b*.

This requires the *reference rule* of section 4.3.2 to be extended as follows. In a behaviour execution an action can only refer to:

- actions that directly cause this action,
- actions that indirectly cause this action, and
- actions that synchronise with this action.

An action *a* is indirectly caused by an action *b* if:

- b directly causes another action c that directly or indirectly causes a, or that a synchronises with, or
- b synchronises with another action c that directly or indirectly causes a, or that a synchronises with.

Thus, for example, the following behaviour specifications are allowed:

$$\{ \ {=}a \rightarrow b\ (v_b{:}\{1,2\})\ [v_b{=}v_a],$$
$$\ {=}b \rightarrow a\ (v_a{:}\{1,2\})\ [v_a{=}v_b]\ \ \},$$

and

$$\{ \ {=}a \rightarrow b\ (v_b{:}\{1,2\})\ [v_b{=}v_a],$$
$$\ {=}b \rightarrow a\ (v_a{:}\{1,2\})\ [v_a{=}v_b],$$
$$a \rightarrow c\ (v_c{:}\{1,2\})\ [v_c{=}v_b]\ \ \}.$$

4.5.2 Extensions for composite behaviours

Constraint-oriented behaviour compositions

Value establishment in interactions

The addition of data concepts has the following consequences for interactions and the sub-behaviours involved in these interactions.

1. Values can be established in interactions.

   This is because an interaction is a specific type of action: it models the common activity of multiple sub-behaviours. Since values can be established in actions, they can also be established in interactions.

2. a. If a value is to be established in an interaction, each involved sub-behaviour should define the data type of the value. Each involved sub-behaviour may or may not contain a reference relation that makes the interaction, and possibly the value established in it, dependent on other values.

   This is because an interaction models a common activity of sub-behaviours. Each sub-behaviour may therefore constrain the value established in the interaction.

   b. The value established in an interaction is an element of the intersection of the data types defined in the contributions of the involved sub-behaviours and conforms to all reference relations defined in the contributions of these sub-behaviours. If there is no value that meets these criteria, the interaction does not take place.

   This is because the condition for the occurrence of an interaction is the conjunction of the conditions imposed on the occurrence of the interaction by each of the sub-behaviours involved in the interaction.

3. If a value is established in an interaction, actions in all of the involved sub-behaviours can refer to this value, if they are allowed to do so according to the reference rule.

   This is because an interaction in which a value is established models, in terms of our basic concepts, a set of mutually disabling basic interactions. Since actions in all of the involved sub-behaviours can depend on the occurrence of a basic interaction, they can also refer to the value established in an interaction.

4. An (inter)action *a* in a sub-behaviour *A* can only refer to a value established in another sub-behaviour *B* if this value was established in an interaction between *A* and *B* to which *a* can refer.

   This is because interactions are the *only* way in which sub-behaviours are related in a constraint-oriented composition of sub-behaviours. A sub-behaviour *A* has therefore only access to a value established in another sub-behaviour *B* if this value is passed from *A* to *B* in an interaction.

For example, the monolithic behaviour

$\{ \; start \rightarrow a \; (v_a{:}\{1,2,3,4\} \; [1 \leq v_a \leq 3],$
$\quad a \rightarrow b \; (v_b{:}\{1,2,3,4\}) \; [v_b{=}v_a],$
$\quad a \rightarrow c \; (v_c{:}\{1,2,3,4\}) \; [v_c{=}v_a] \quad \}$

can now be structured as a constraint-oriented composition of sub-behaviours as:

$\{$ interactions $B_1, B_2{:}a \;\; \}$

$B_1 =$
$\{ \; start \rightarrow \underline{a} \; (v_a{:}\{1,2,3,4\} \; [v_a \geq \; 1],$
$\quad a \rightarrow b \; (v_b{:}\{1,2,3,4\}) \; [v_b{=}v_a] \quad \}$

$B_2 =$
$\{ \; start \rightarrow \underline{a} \; (v_a{:}\{1,2,3,4\} \; [v_a \leq 3],$
$\quad a \rightarrow c \; (v_c{:}\{1,2,3,4\}) \; [v_c{=}v_a] \;\; \}.$

Figure 4.14 represents this behaviour graphically.

Types of interaction

Vissers [1983] distinguishes three types of interactions, based on the constraints each involved sub-behaviour puts on the value established in an interaction. These types of interaction can all be expressed using our concepts. Let $B_1, B_2, \ldots, B_n$ be the sub-behaviours involved in an interaction *i*.

- *Value checking*: each of the sub-behaviours $B_1, B_2, \ldots, B_n$ requires that one particular value is established in *i*. For example:

  $B_1 {=} \{ \; \ldots, \; a \rightarrow \underline{i} \; (v_i{:}\{1\}), \; \ldots\}$
  $B_2 {=} \{ \; \ldots, \; b \rightarrow \underline{i} \; (v_i{:}\{1\}), \ldots\}.$

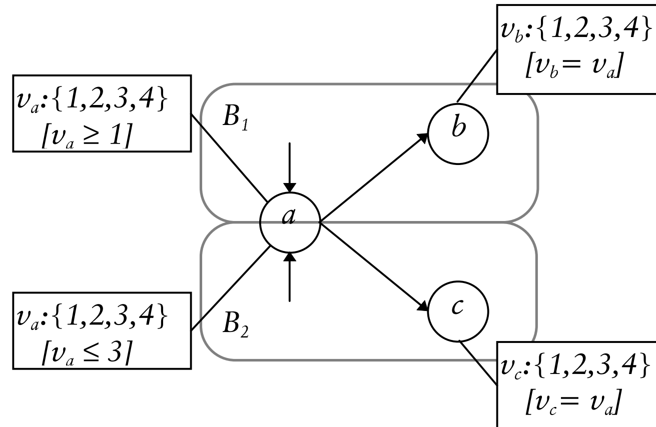


Figure 4.14 Example behaviour represented as constraint-oriented composition of sub-behaviours.

- *Value passing*: one of the sub-behaviours B_1, B_2, ..., B_n requires that one particular value is established in i; each of the other sub-behaviours offers some freedom for the value to be established. For example:

  B_1={ ..., $a \rightarrow \underline{i}$ (v_i:{1}), ...}
  B_2={ ..., $b \rightarrow \underline{i}$ (v_i:N),...}.

- *Value generation*: each of the sub-behaviours B_1, B_2, ..., B_n offers some freedom for the value to be established. For example:

  B_1={ ..., $a \rightarrow \underline{i}$ (v_i: N), ...}
  B_2={ ..., $b \rightarrow \underline{i}$ (v_i:N),...}.

Causality-oriented behaviour compositions

In a causality-oriented composition of sub-behaviours, the sub-behaviours are related by means of coupled exit and entry points. Since exit points and entry points are merely syntactical constructs, actions in one sub-behaviour should in principle be able to refer to values established in other sub-behaviours without extra notation being necessary. From an analytical point of view, considering the decomposition of a behaviour into a causality-oriented composition of sub-behaviours, this is correct.

However, from a synthetic point of view, considering the construction of a causality-oriented composition of sub-behaviours from separately defined sub-behaviours, this is unfortunate. In a causality-oriented composition of sub-behaviours an action in a sub-behaviour *B* should only be allowed to refer to values established in other sub-behaviours, if these values are indeed established in these other sub-behaviours. However, it need not be immediately clear which values from other sub-behaviours are referred to in *B*. In that case a developer has to analyse *B* in order to track down these values. For similar reasons the developer may have to analyse all other sub-behaviours.

In order to prevent such unnecessary analysis, we make the following syntactical extensions.

- Each exit point is parametrised with 1) the variables that are assigned the values made available for reference by other sub-behaviours and 2) the data types of these variables.
- Each entry point is parametrised with 1) the variables used to refer to values established in other sub-behaviours and 2) the data types of these variables.
- The coupling of an exit point parametrised with the variables $vx_1, vx_2, ..., vx_m$ to an entry point parametrised with the variables $vn_1, vn_2, ..., vn_m$ represents that the values assigned to $vx_1, vx_2, ..., vx_m$ can be referred to via the variables $vn_1, vn_2, ..., vn_m$, respectively. An exit point parametrised with m variables can only be coupled to an entry point parametrised with m variables of the same respective data types.

We represent an entry point or an exit point e that is parametrised with the variables $v_1, v_2, ..., v_n$ of the data types $D_1, D_2, ..., D_n$, respectively, as: $e\ (v_1{:}D_1,\ v_2{:}D_2,\ ...,\ v_n{:}D_n)$.

For example, the monolithic behaviour

$\{\ start \rightarrow a\ (v_a{:}\{1,2\}),$
$\quad a \rightarrow b\ (v_b{:}\{1,2\})\ [v_b{=}v_a]\ \ \}$

can be structured as a causality-oriented composition of sub-behaviours as:

$\{\ start \rightarrow A(entry),\ A(exit(v_a{:}\{1,2\})) \rightarrow B(entry(v_e{:}\{1,2\}))\ \ \},$

$A=$
$\{\ entry \rightarrow a\ (v_a{:}\{1,\ 2\}),$
$\quad a \rightarrow exit(v_a)\ \ \},$

$B=$
$\{\ entry(v_e) \rightarrow b\ (v_b{:}\{1,\ 2\})\ [v_b{=}v_e]\ \ \}.$

Figure 4.15 represents this behaviour graphically.

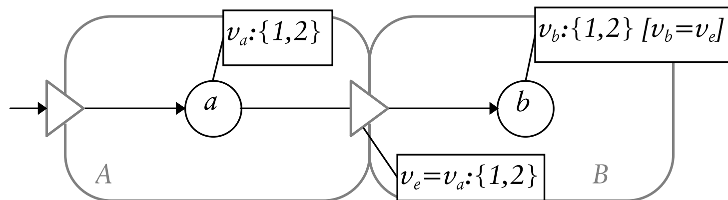


Figure 4.15 Example of coupled parametrised exit and entry points.

4.5.3 Value decomposition

This section discusses a technique for value structuring called value decomposition. A comprehensive treatment of data modelling, the area concerned with data structuring, is out of the scope of this thesis.

Value decomposition is the decomposition of a value into multiple values. It allows multiple values to be established in an action, rather than just one value. Since an action value models the result of an activity, each of the values resulting from its decomposition models an aspect of this result.

For example, let *arrive* be an action in which the value *at work at 8.45 by car* is established. This value can be decomposed into the value *at work*, the value *at 8.45* and the value *by car*, which all represent different aspects (location, time, and means of transport) of the result of the activity of arriving.

In order to allow each value established in an action to be referred to individually, each value is assigned to a different variable. The variables to which action values are assigned with their respective data types are represented between round brackets after the action name. For example,

arrive (location:L, time:T, means_of_transport:M)

represents an action in which three values are established; these values are assigned to the variables *location*, *time*, and *means_of_transport* of the data types *L*, *T*, and *M*, respectively.

For example, the following behaviour can be specified using value decomposition:

{ *start → book_airplain_ticket (booked_ticket:T, costs:$, booked_destination:D),*
book_airplain_ticket → pay_ticket (amount:$) [amount=costs],
pay_ticket → pickup_ticket (ticket:T) [ticket=booked_ticket],
pickup_ticket → fly (destination:D) [destination=booked_destination] }.

4.6 Conclusions

Data is often viewed as orthogonal to concepts for modelling system dynamics. We showed this view is incorrect: data concepts can be defined in terms of the previously discussed basic architectural concepts. This insight allowed us to show in what circumstances data is useful for the concise modelling of behaviours, thus providing guidance to developers in the development of comprehensible behaviour models that can be economically implemented.

We distinguished the following data concepts:

- An action value models the result of an activity. An action value is an element of a data type, which contains all action values that can be established in an action.

Data types of action values can be used to concisely model a set of actions from which only one may occur.

- A reference relation makes the occurrence of an action, and possibly the value established in this action, dependent on values established in other actions. A reference relation can be used to concisely model multiple basic causality relations.
- A state value is the value of a state function, which is a function of action values. A state value models one or more states of a behaviour, where the state concept conforms to a customary notion of state, e.g., as in finite-state machines. State values and state functions can be used to concisely model multiple reference relations.

5

Structuring of behaviour models

5.1 Introduction

5.1.1 Motivation

In this chapter we discuss behaviour structuring by means of the structuring of behaviour models. Since the objective of structuring is the control of complexity, we discuss the way in which the structuring of behaviour models contributes to achieving this objective. In particular, we consider the following development objectives:

- control of the complexity of comprehending a system: structuring should provide insight in the characteristics of the system;
- control of the complexity of building a system: structuring should facilitate implementation of the system.

The approach we follow is based on the introduction of a limited number of modelling styles. This approach was first described in [Vissers et al., 1988], and was later elaborated in among others [Vissers et al., 1991] and [Ferreira Pires, 1992].

A *modelling style* is a set of rules for the structuring of a model to meet particular development objectives. It is thus a structuring technique that explicitly supports particular development objectives.

A structure is determined by its elements and the way in which these elements are related. We therefore define each modelling style in terms of the (basic and derived) concepts used in a model. A modelling style thus defines the characteristics a number of structures have in common.

The adoption of a limited number of well-chosen and well-defined modelling styles can aid in the reduction of complexity in, among others, the following ways (after [Minsky, 1985] and [Vissers et al., 1991]).

- *Uniformity*. Styles enforce uniformity, thereby easing comprehension. For example, many companies adopt house styles to make their public expressions more uniform in e.g. colour and form.

  This principle also applies to models made in a development process. The use of a limited number of styles enforces some uniformity of structure of these models. It thus forms a protection against the application of personal traits by developers. Personal styles are undesirable properties of such models, because they may lead to different developers expressing the same concepts in different ways. This may result in confusion about the meaning of a model, which complicates communication about the system and complicates the implementation of the system.

- *Ease of decision making*. Styles allow considerations and decisions to be pruned to the ones that fit a style. For example, a person who has furnished his or her house according to a modern style will usually not even consider the purchase of a Victorian chair.

  This also applies to the development of system models. The adoption of a limited number of pre-defined styles directs development decisions to the ones that fit the styles. In our case the making of development decisions is further eased, because each modelling style is explicitly related to particular development objectives.

5.1.2 Structure

In section 5.2 we present three orthogonal criteria and define styles based on the (non-)satisfaction of these criteria:

- the extensional and intensional styles are defined in section 5.2.2 based on the absence or presence of internal (inter)actions in a behaviour, respectively;
- the monolithic and modular styles are defined in section 5.2.3 based on the absence or presence of sub-behaviours in a behaviour, respectively;

  the modular style is refined in the constraint-oriented style and the causality-oriented style based on the relations between sub-behaviours: shared causality relations or shared actions (interactions), respectively;
- the process-oriented and state-oriented styles are defined in section 5.2.4 based on the absence or presence of state variables and state functions in a behaviour, respectively.

In section 5.3 we describe two common structures of business processes: business processes structured around work-flows and business processes structured around business functions.

In section 5.4 we apply the styles identified in section 5.2 to business processes structured in these ways.

- In section 5.4.2 we apply the styles to the modelling of requirements for business processes. We show how an extensional style combined with either a constraint-oriented or a causality-oriented style is appropriate in structuring requirements for business processes organised around work-flows or business functions, respectively.
- In section 5.4.3 we apply the styles to the modelling of the internal behaviour of business processes. We show how an intensional style combined with a constraint-oriented style is appropriate in structuring implementations of business processes organised around work-flows or business functions.

In section 5.5 we discuss the role of styles in development methods.

Part of the work described in this chapter is based on [De Weger, 1996].

5.2 Some modelling styles

5.2.1 Example

We use the behaviour represented in Figure 5.1 as an example. The example concerns a postal company that offers receipt confirmation.

A client (*client 1*) may send a package (*send_package*), which is then received (*receive_package*) by the other client (*client 2*). The latter client then sends a confirmation (*send_confirmation*), which is received (*receive_confirmation*) by the former client.

The internal aspects of the postal company are depicted in the figure in grey. The postal company consists of three units: two geographically remote offices and a transport unit. If a client sends a package or a confirmation, the receiving office delivers the package or the confirmation at the transport unit (interactions *sp* and *sc*). The transport unit then delivers the package or the confirmation at the other office (interactions *rp* and *rc*), after which the client receives it.

5.2.2 Extensional versus intensional styles

Extensional style

A behaviour structured according to an *extensional style* does not contain any internal actions or internal interactions.[1]

---

1 The notions of "extension" and "intension" are customary in some areas of the study of logic. Carnap [1956] defines the extension of a predicate as its corresponding class. The intension of the predicate is defined as its corresponding property. These notions have been applied to systems development in similar ways as in this thesis in [Alexander, 1964], [Simon, 1981], and [Vissers et al., 1988].

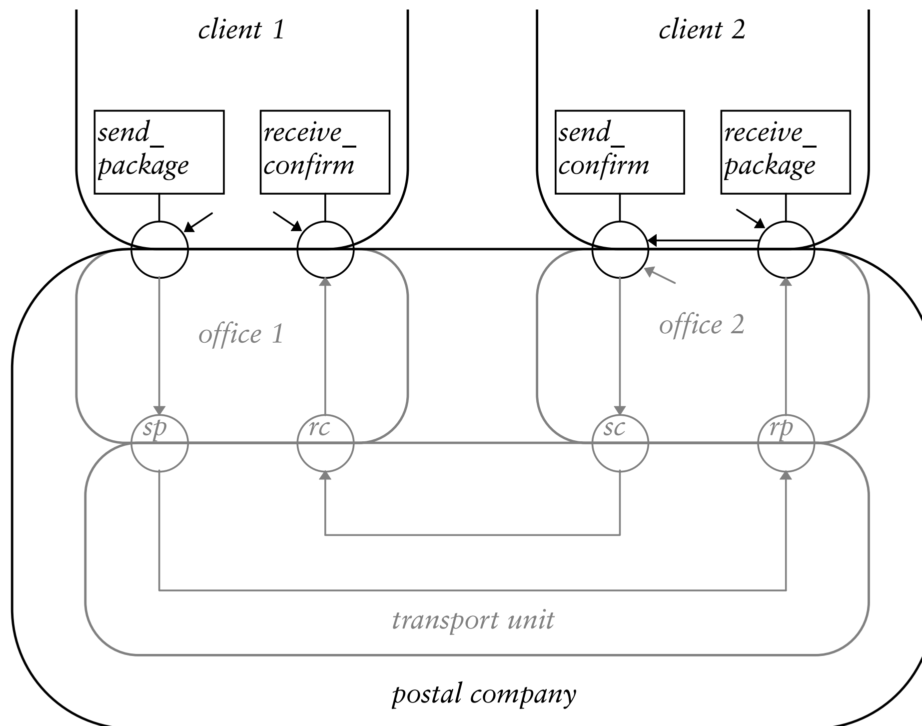


Figure 5.1 Example of postal company. The external structure of the company and each of the users is depicted in black. The internal structure of the company is depicted in grey.

The behaviour of the example postal company in an extensional style is the following:

{ *start* → *send_package*,
send_package → *receive_package*,
receive_package → *send_confirm*,
send_confirm → *receive_confirm* }

An extensional style thus describes the behaviour of a system only in terms of interactions of the system with other systems, values established in these interactions, and their relations. It abstracts from the internal aspects of the behaviour of the system. This is sometimes called a “black box” specification.

Therefore an extensional style is well suited for the specification of a system in the initial stages of a development process, in which one is only concerned with the requirements of the system. These requirements usually only apply to what a system should do and not to how the system should operate. An extensional style allows this external behaviour of a system to be considered by users and developers, without distracting them with irrelevant internal aspects of the system.

Intensional style

A behaviour structured according to an *intensional style* contains one or more internal actions or internal interactions.

A behaviour can be more or less intensional depending on the amount of internal (inter)actions it contains.

The behaviour of the example postal company in an intensional style is the following:

{ *start* → *send_package*,
send_package → *sp*,
sp → *rp*,
rp → *receive_package*,
receive_package → *send_confirm*,
send_confirm → *sc*,
sc → *rc*,
rc → *receive_confirm* }

An intensional style thus describes a behaviour in terms of interactions of the system with other systems, internal (inter)actions of the system, values established in these (inter)actions, and their relations. It thus describes internal aspects of the behaviour of the system. This is sometimes called a "white box" specification.

Therefore an intensional style is well suited for the specification of a system in later stages of a development process, in which one is concerned with the implementation of the system. An implementation describes not only what the system should do, but also, to some extent, how the system should operate. An intensional style allows (parts of) the internal behaviour of a system to be defined explicitly.

Discussion

Even though an intensional style is appropriate for the specification of the internal behaviour of a system, this is not the only possible application of the style.

An intensional style defines the external behaviour of a system implicitly (it "generates" the external behaviour). An example is the description of the service of a communication system in terms of the protocol that implements this service.

An intensional style is sometimes used to give an external specification of a system if the relations between interactions of the system with its environment are complex. The introduction of internal constructs can then be used to decompose these complex relations. Another possible reason for the use of an intensional style for external specification is a lack of expressiveness of the specification language: if the specification language does not provide concepts that are sufficiently abstract to allow for extensional specification, the user may be forced to use an intensional style.

If an intensional style is used to give an external specification of a system, and if the specification is a prescription for the implementation, the developer should make it clear which elements of the specification are prescriptive and which elements were only introduced to complete the external specification.

An extensional style cannot be used to give an internal description of a system, because the extensional style prohibits the description of internal aspects of behaviour.

Table 5.2 gives an overview of the use of extensional and intensional styles for the specification of external and internal behaviour.

	external specification	**internal specification**
extensional style	preferred	impossible
intensional style	implicit	preferred

Table 5.2 Use of styles for behaviour specification.

5.2.3 Monolithic versus modular styles

Monolithic style

A behaviour "structured" according to a *monolithic style* does not contain any sub-behaviours. (Actually, a monolithic style implies a lack of structure.)

Both specifications of the behaviour of the example postal company in section 5.2.2 are structured according to a monolithic style.

A monolithic style is characterised by a lack of structure, because it prohibits the structuring of a behaviour in terms of related sub-behaviours.

Therefore a monolithic style is well suited for the specification of very simple behaviours (like our example behaviour). In such behaviours the introduction of (extra) structure may only be superfluous and not enhance their comprehensibility. However, for larger and more complex behaviours the monolithic style is unsuitable, because these behaviours can only be represented in a comprehensible way if they are structured as a composition of sub-behaviours.

A monolithic style can be combined with both an intensional and an extensional style, so it is not specifically oriented towards the specification of requirements or the specification of an implementation. However, because implementations are usually more complex than requirements (as implementations contain more detail), a monolithic style will in practice be used more often for the extensional specification of systems or system components.

Modular style

A behaviour structured according to a *modular style* contains sub-behaviours.

A behaviour can be more or less modular depending on the amount of sub-behaviours it contains.

Examples of a modular style are given below, where two specialisations of the modular style are discussed.

Because a modular style requires the use of sub-behaviours, it is well suited for the specification of complex behaviours that have to be structured as compositions of sub-behaviours in order to remain comprehensible and implementable. For simple behaviours the modular style may be less suited, because it introduces additional structure.

A modular style can be combined with an extensional or intensional style and is thus not specifically oriented towards the specification of requirements or the specification of an implementation.

Causality-oriented style

A behaviour structured according to a *causality-oriented style* contains sub-behaviours that are exclusively related by means of shared causality relations.

The behaviour of the postal company of section 5.2.1 can be specified according to an (intensional) causality-oriented style by the introducing two sub-behaviours, in which one contains the actions dealing with a package and the other contains the actions dealing with a confirmation.

{ *start* → *package(entry)*, *package(exit)* → *confirmation(entry)* },

package=
{ *entry* → *send_package*,
send_package → *sp*,
sp → *rp*,
rp → *receive_package*,
receive_package → *exit* }

confirmation=
{ *entry* → *send_confirm*,
send_confirm → *sc*,
sc → *rc*,
rc → *receive_confirm* }

A causality-oriented style thus requires a behaviour to be structured as a composition of sub-behaviours in a specific way: each action is completely defined in one sub-behaviour, but the actions that constitute its causality condition may be located in a different sub-behaviour.

Therefore, a causality-oriented style is well suited for the structuring of behaviours in terms of sub-behaviours with this property. Behaviours structured according to non-overlapping phases are important examples of this. (The causality-oriented behaviour specification above is such an example.) A phase is a particular time period in which certain actions may occur. Therefore, if the phases of a behaviour are non-overlapping, each of its actions belongs to a single phase.

As any modular specification, a causality-oriented specification can be used to specify both external behaviour (facilitating the construction of requirements models) and internal behaviour (facilitating the construction of implementation models). However, a causality-oriented specification cannot be used to specify a behaviour that is structured as a composition of sub-behaviours, where each sub-behaviour is executed by an entity that interacts with other entities.

Constraint-oriented style

A behaviour structured according to a *constraint-oriented style* contains sub-behaviours that are exclusively related by means of interactions.

In the constraint-oriented style each sub-behaviour is considered as a constraint on the total behaviour. The total behaviour, the composition of the sub-behaviours, then satisfies the conjunction of all of the constraints imposed by each of the sub-behaviours.

The behaviour of the example postal company can be specified according to an (intensional) constraint-oriented style by defining three sub-behaviours: two representing the behaviours of the local offices and one representing the behaviour of the transport unit.

{ interactions *office1, transport_unit: sp, rc,*
office2, transport_unit: sc, rp },

office1 =
{ *start* → *send_package,*
send_package → *sp,*
sp → *rc,*
rc → *receive_confirm* },

office2 =
{ *start* → *rp,*
rp → *receive_package,*
receive_package → *send_confirm,*
send_confirm → *sc* },

transport_unit =
{ *start* → *sp,*
sp → *rp,*

start → → <u>*sc*</u>,
sc → <u>*rc*</u> }

The constraint-oriented style can also be applied in combination with the extensional style. In that case each of the sub-behaviours represents an aspect of the external behaviour:

{ interactions *at_office1, between_offices: send_package, receive_confirm,*
at_office2, between_offices: receive_package, send_confirm }

at_office1=
{ *start* → <u>*send_package*</u>,
send_package → <u>*receive_confirm*</u> },

at_office2=
{ *start* → <u>*receive_package*</u>,
receive_package → <u>*send_confirm*</u> }

between_offices=
{ *start* → <u>*send_package*</u>,
send_package → <u>*receive_package*</u>,
start → <u>*send_confirm*</u>,
send_confirm → <u>*receive_confirm*</u> }

A constraint-oriented style thus requires a behaviour to be structured as a composition of sub-behaviours in a specific way: certain actions (the interactions) are distributed over multiple sub-behaviours, together with the conditions for their occurrence.

Therefore, a causality-oriented style is well suited for the structuring of behaviours in terms of sub-behaviours with this property. This is the case if overlapping aspects can be identified in the behaviour: actions exist that belong to multiple aspects of the behaviour. In the extensional constraint-oriented specification above, three aspects can be recognised: the constraints at the interaction point of office 1 and its client (represented by behaviour *at_office1*), the constraints at the interaction point of office 2 and its client (represented by behaviour *at_office2*), and the constraints that apply to both interaction points (represented by behaviour *between_offices*). The first two aspects are independent of each other, but the third aspect overlaps with the first two: the activity of sending a package, for example, belongs both to the first and the third aspect.

Discussion

A difference between the styles defined here and those of [Vissers et al., 1988], is that the definition of our styles is based on three orthogonal criteria.

In behaviours structured according to the monolithic style of [Vissers et al., 1988] only interactions between the behaviour and the behaviour of its environment are

presented, which are ordered as a collection of alternative sequences in time. This differs from our definition in two respects.

- According to our definition a monolithically structured behaviour may contain internal actions. We thus do not restrict a monolithic style to extensional modelling.
- According to our definition the (inter)actions in a monolithically structured behaviour need not be ordered as a collection of alternative sequences in time.

  This difference can be clarified by a consideration of the modelling concepts used here and in [Vissers et al., 1988]. Vissers et al. illustrate their styles with specifications in the specification language LOTOS [ISO, 1987a]. The interleaving operator of LOTOS (|||), which is often used to express independence as well, operates on (sub-)behaviours. Therefore, the use of the interleaving operator introduces structure in a specification in terms of sub-behaviours. This operator can therefore not be used in a monolithic style; instead a developer is forced to specify all possible alternative sequences of interactions.

  With our concepts, the independence of actions in a behaviour can be modelled without the introduction of additional structure in terms of explicitly represented sub-behaviours. Therefore, a monolithically structured behaviour may contain independence.

In the constraint-oriented style of [Vissers et al., 1988] only interactions between the behaviour and the behaviour of its environment are presented. It is thus, unlike our constraint-oriented style, restricted to the extensional modelling of a system or system parts.

5.2.4 Process-oriented versus state oriented styles

Process-oriented style

A behaviour structured according to a *process-oriented style* does not contain any state variables and state functions.

All specifications of the behaviour of the example postal company given before are process-oriented.

The process-oriented style derives its name from literature in which "process-oriented" languages, and development methods based on these, are viewed in contrast to "data-oriented" languages, and development methods based on these (e.g. [Maddison, 1983; Essink, 1986]). Process-oriented languages focus on the modelling of activities and their relations. Data-oriented languages focus on the modelling of state values, state variables, and state functions.

A process-oriented style is well suited for the specification of a behaviour in which state variables and state functions are not useful for making the specification concise. Chapter 4 showed that these are behaviours in which the occurrences of actions do

not depend on complex functions of action values. The behaviour of the postal company is an example of this.

A process-oriented style can be combined with both an extensional and an intensional style, and can thus be used for both the specification of requirements and the specification of implementations.

State-oriented style

A behaviour structured according to a *state-oriented style* contains one or more state variables and state functions.

The behaviour of the example postal company in an (extensional) state-oriented style is the following:

{ *start* → *B(entry(initial))* }

B=
{ *choice(send_package, receive_package, send_confirm, receive_confirm),*
entry(s:State) ∧ *(s=initial)* → *send_package* <*s=package_sent*>,
entry(s:State) ∧ *(s=package_sent)* → *receive_package* <*s=package_received*>,
entry(s:State) ∧ *(s=package_received)* → *send_confirm* <*s=confirm_sent*>,
entry(s:State) ∧ *(s=confirm_sent)* → *receive_confirm* <*s=confirm_received*>,
send_package ∨ *receive_package* ∨ *send_confirm* → *B(entry(s))*
}

A behaviour can be more or less state-oriented, depending on the amount of actions whose occurrences depend on values of state variables. If, in a behaviour, every value of a state variable models a single state, and the occurrence of every action depends only on the value of this state variable (as in the above state-oriented specification), we call the behaviour fully state-oriented.

A state-oriented style is well suited for the specification of a behaviour in which state variables and state functions are useful for making the specification concise. It was shown in chapter 4 that these are behaviours in which the occurrences of actions depend on complex functions of action values. The behaviour of database systems is a prominent example of this.

A state-oriented style can be combined with both an extensional and an intensional style, and can thus be used for both the specification of requirements and the specification of implementations.

Discussion

Why a state-oriented style is not necessarily intensional

Simon [1981] calls states “internal states”. In [Vissers et al., 1988] a state-oriented style is classified as an intensional style. The intensionality of a state-oriented style

can be defended by stating that a state is a construct in a behaviour that cannot be observed by its environment.

This is indeed true for behaviour implementations, where the value of a state variable is stored in e.g. a memory location that cannot be accessed by external entities. It is also true if, as is common in practice, an implementer preserves the structure of a specification in the implementation.

However, a state value is not always necessarily intensional. In chapter 4 we showed that a state value is nothing more than a (possibly concisely defined) function of action values. One may leave an implementer freedom on how to implement a state variable (i.e. by not prescribing that a state variable should be implemented as, e.g., a memory location). In that case, if a state value is function of action values that were all established in interactions between the system and its environment, it is an extensional construct. If a state value is a function of action values that were also established in internal actions, it is an intensional construct.

Aspects of behaviour

In some works on systems development (e.g. [Sowa & Zachman, 1992; Essink, 1986; Ramackers, 1992]), several aspects of the real-world behaviour of systems are distinguished[2], such as:

- the dynamic or process aspect: the causality relations or temporal relations between actions;
- the data aspect: the structure of action values and (predominantly) state values;
- the functional aspect: the relations between action values and state values.

Our work on the role of data has shown that, given a certain real-world behaviour, “the” dynamic aspect and “the” functional aspect of this behaviour do not exist: when modelling a certain behaviour, a developer has the choice to do this according to a process-oriented style or according to a state-oriented style. So, to a certain extent, a developer has the choice to perceive certain relations as causal or functional. Only after these choices have been made and a model has been developed, one can unambiguously refer to “the” dynamic aspect or “the” functional aspect of the model.

2 Even though the same or similar aspects are identified by these authors, these aspects are named and classified differently. See [De Weger et al., 1995b] for an overview.

5.3 Structures of business processes

5.3.1 Business processes structured around work-flows or business functions

Two basic types of structures of business process are generally recognised (see e.g. [Mintzberg, 1979; Hammer, 1990]): business processes structured around work-flows and business processes structured around business functions. A business process is structured "around" work-flows or business functions at a certain abstraction level if the behaviours considered at this level are work-flows or business functions, respectively.

A *work-flow* is the complete set of related activities required to deliver a particular product, or to serve a particular customer. It has one or more initial actions (which start the work-flow) and one or more final actions (which end the work-flow). In general, a business process cannot be specified completely by only specifying all of its independent work-flows. This is because work-flows are usually dependent on each other, e.g. due to the fact that they share resources.

An example of a work-flow of a production company is the purchasing of some raw materials (the initial action), followed by their transformation into the final product, followed by the sales of this product (the final action). An example of a work-flow of an insurance company is the application of a client for an insurance (the initial action), followed by the judgement of the application, followed by the admittance or rejection of the application (the final actions). Examples of organisational units responsible for work-flows are "case workers", "customer service representatives" [Hammer, 1990] and work-flow management systems.

A core business function, or more briefly a *business function*, is the complete set of related activities required to bring about a part of every product, or to partially serve every customer. A business process can be specified completely by the specification of all of its business functions. All business functions can be sequentially ordered in a so-called "value chain" [Porter, 1980]. This is the order in which business functions are carried out in order to deliver a particular product or service. A business function is thus a phase in the process of delivering a product or service.

An example of business functions of a production company are purchasing, production, and sales. Whereas a work-flow to bring about a product contains, for example, only one or a few purchasing activities, the business function "purchasing" contains all purchasing activities of the business process. In order to make one product, the purchasing for this product is carried out before its production and the production of this product is carried before its sales. Therefore, this is the order of the business functions in the value chain.

Work-flows and business functions are orthogonal aspects of business processes. Figure 5.3 depicts this graphically.

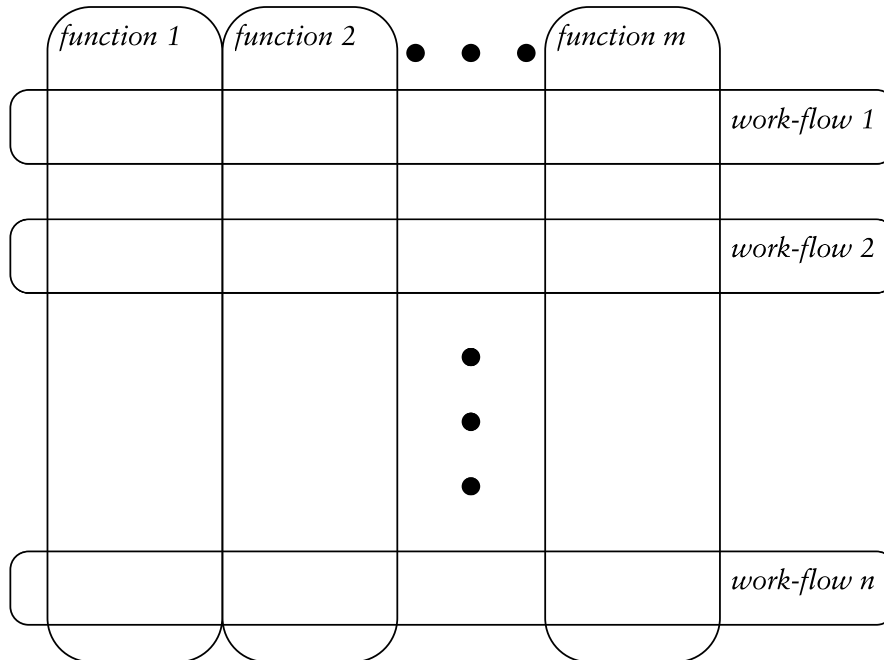


Figure 5.3 Work-flows and business functions as orthogonal aspects of business processes.

5.3.2 Structuring around work-flows or business functions

History

The traditional way in which business processes were structured until the 1960s, is around business functions [Hammer & Champy, 1993]. Business processes were decomposed into business functions that were each carried out by a separate unit: purchasing was carried out by a purchasing department, production by a production department, etc.

More recently, structuring around work-flows has got much attention. It has become one of the principles of business process re-design. In fact, many authors use the term (business) process solely to refer to a work-flow or a business process structured around work-flows. For example, Pall [1987], quoted in [Davenport & Short, 1990, p. 12], defines a business process as "the logical organisation of people, materials, energy, equipment, and procedures into work activities designed to produce a specified end result (work product)". Buitelaar & Groen [1994, p. 389] state that organisations should work "from customer to customer". And Hammer [1990, p. 108] states his first re-design principle as "organize around outcomes, not tasks"[3].

3. In order to prevent confusion, some attention to terminology is required. Many authors, including Hammer, use the term "task" as a synonym for a (small) business function. Other synonyms used for "business function" are "function", and "process" [Mintzberg,

Advantages of structuring around work-flows

Two main advantages of structuring business processes around work-flows are the following.

- *Better serving of customers*. In a business process organised around work-flows most required communication and co-ordination for the production of particular product or the serving of a particular customer is centralised in one unit. Since co-ordination and communication between organisational units is far more difficult and error-prone than communication and co-ordination within an organisational unit [Scharpf, 1977], structuring a business process around work-flows eases the production of products and serving of customers.

  An example of better being able to serve customers in an organisation structured around work-flows than in an organisation structured around functions is given in [Chapple & Sayles, 1961], quoted in [Mintzberg, 1979]. In two cases a credit department of a manufacturing firm cancelled orders made by the sales department, just after the sales department had thanked the customers for their confidence. These communication problems were solved by organising the business process around units that were each responsible for all activities for a group of clients, including credit administration and sales.

- *Faster reaction to changing customer requirements*. Customer requirements with respect to a product or a service are often subject to change. In an organisation structured around work-flows such changing customer requirements have an impact on only one unit (the unit responsible for this product or service). Therefore, the required changes are often easier and faster to implement than in an organisation structured around functions, in which the required changes may have an impact on many units. Moreover, since a unit is completely responsible for a product or service, it is often more sensitive to the customer requirements for this product or service. Therefore, changed customer requirements are often perceived more quickly in organisation structured around work-flows.

  An example of faster reaction to changing customer requirements is given in [Earl, 1994]. Texas Instruments' wafer production process was geared towards the production of generic wafers and was organised around business functions. The production time of customised wafers, increasingly required by customers, exceeded their life-cycle time. After reorganising the wafer production process around work-flows, customised wafers could be produced in 30 percent of the original time.

1979]. Synonyms used for "work-flow" are, apart from "process", "cross-functional process" and "interfunctional process" [Davenport & Short, 1990].

Advantages of structuring around business functions

Two main advantages of structuring business processes around business functions are the following.

- *Optimal use of scarce resources*. Certain resources in business processes may be scarce and relatively expensive compared to other resources. Examples are highly skilled employees and expensive machinery. In order to make the business process as economical as possible, these scarce resources may need to be used optimally. Since these resources are usually only involved in a part of a production process, it may be preferable to organise the business process around functions.

  A hospital forms an example of the case in which an organisation is able to use scarce resources better when it is structured around functions than when it is structured around work-flows. Specialised knowledge, possessed by only a few expensive specialists, is required in the treatment of patients. Therefore specialists only carry out specific activities, such as diagnostics, prescription of medicines, and surgery. It would be too expensive to make a specialist carry out all activities in a work-flow of a patient, such as nursing and administration.

- *Cheaper implementation of changes due to changing non-customer requirements*. Organisations are not only subject to customer requirements, but also to other requirements, such as government regulations. Some of these regulations involve many work-flows, but only a few functions. It would be very hard and expensive to re-design all involved work-flows every time the regulations change if these regulations often change significantly. In such cases it may be better to assign the business functions subject to these changing regulations to separate units.

  A pension fund forms an example in which an organisation may be able to implement changes due to changing non-customer requirements more easily when it is structured around functions than when it is structured around work-flows. Government regulations that often change, such as laws with regard to the financial consequences of divorces or of lasting ailments have an impact on only small parts of the total business process of a pension fund. Therefore, it may be more economical to centralise these parts into functional units than to distribute them over multiple work-flows.

Conclusions

We conclude the following, based on [Mintzberg, 1979]. If it is possible to structure a business process around relatively independent work-flows that are not subject to often changing non-customer requirements, this structure is usually preferable. This is because the structure eases the proper serving of customers, and because it makes it easier to quickly react to changing customer requirements.

However, if there are strong interdependencies between work-flows due to scarce resources required by many work-flows, or if many work-flows are subject to often changing non-customer requirements, it may be more economical to structure a business process around business functions.

In practice, most advantages are often obtained by a combination of work-flows and business functions as the basis for structuring. For example, an insurance company may well have all of its contacts with customers (insurance selling, collecting contributions, paying damages, etc.) carried out by agents who each serve their own group of customers. However, the investment of contributions is usually centralised in a single business function carried out by an investment department. This is due to the specialised knowledge and large sums of money required for investments.

Another common combination of work-flows and functions can be found in organisations using work-flow management systems. Many of these organisations are still organised around business functions, but use their work-flow management systems for the co-ordination of work-flows and the communication between business functions.

5.4 Application of styles to business processes

5.4.1 Example

We illustrate the application of the modelling styles of section 5.2 to business processes organised around work-flows or business functions with an example of a mail-order company that delivers a single good from its store. Figure 5.4 depicts a work-flow of activities carried out for a single customer. The customer places an order (*order*), which is followed by an internal order to get the good from the store and forward it to the customer (*int_order*). If the ordered good is available, it is picked up and made available for shipment (*int_ship*) and then shipped to the customer (*ship*). If the ordered good is not available, an internal notice is made (*int_unavailable*) and the customer is then notified of the good's unavailability (*unavailable*).

In this example three customers are served (resulting in three work-flows) and there is a stock of two goods. Therefore, two of the three customers get their ordered good shipped, whereas the third one receives a notice of unavailability.

The business functions identified in the example are the order function, which takes orders and then makes internal store orders, the store function, which accepts internal store orders and then makes internal shipments or internal unavailability notices, depending on availability of the good, and the shipment function, which accepts internal shipments and internal unavailability notices and then sends customers their shipments or unavailability notices, respectively.

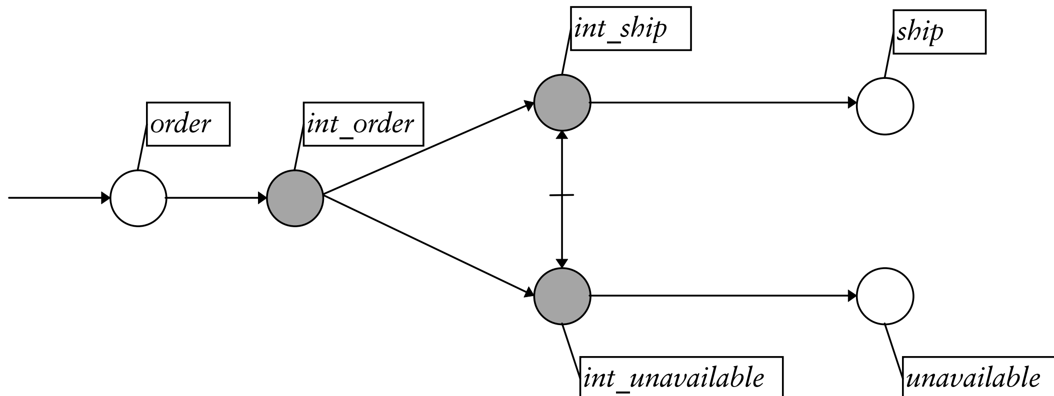


Figure 5.4 Single work-flow of mail-order company. Interactions with client depicted as white circles, internal actions depicted as grey circles.

5.4.2 Requirements modelling

Business processes organised around work-flows

An extensional constraint-oriented style is well-suited for the specification of requirements for business processes organised around work-flows.

The extensional aspects of a work-flow are termed a *use-case* in [Jacobson et al., 1994]. A use-case of a business process is therefore the complete set of related interactions between the business process and its environment required to bring about a particular product, or to serve a particular customer.

The application of the constraint-oriented style is useful, because it enables the following separation of concerns:

- constraints that determine the order of interactions of the business process with its environment in a single use-case;
- constraints that determine the order of interactions of the business process with its environment between use-cases.

This separation of concerns allows the constraints to be re-used in a specification of the implementation of a business process that is organised around work-flows.

We term the former constraints *local* with respect to a use-case and the latter constraints *remote* with respect to multiple use-cases. The usefulness of the distinction between local and remote constraints was already noted in [Vissers et al., 1988]. In that work the criterion for locality (and thus for remoteness) is locality with respect to a geographical or logical location. Here we generalise the meaning of locality: as Mintzberg [1977] notes, geographical location is one of the criteria that can be used for the identification of work-flows, and thus for use-cases.

We call an extensional constraint-oriented style in which the sub-behaviours represent either local constraints with respect to a use-case or remote constraints with respect to multiple use-cases a *use-case-oriented style*.

The requirements for the behaviour of the example mail-order company can be specified in a use-case-oriented style as follows.

$\{$ interactions $\mathit{local_customer}_1, \mathit{remote_between_customers}: \mathit{ship}_1,$
$\mathit{local_customer}_2, \mathit{remote_between_customers}: \mathit{ship}_2,$
$\mathit{local_customer}_3, \mathit{remote_between_customers}: \mathit{ship}_3 \}$

$\mathit{local_customer}_1 =$
$\{ \mathit{start} \rightarrow \mathit{order}_1,$
$\mathit{order}_1 \wedge \neg \mathit{unavailable}_1 \rightarrow \underline{\mathit{ship}_1},$
$\mathit{order}_1 \wedge \neg \mathit{ship}_1 \rightarrow \mathit{unavailable}_1 \}$

$\mathit{local_customer}_2 =$
(* *similar to* $\mathit{local_customer}_2$ *)

$\mathit{local_customer}_3 =$
(* *similar to* $\mathit{local_customer}_3$ *)

$\mathit{remote_between_customers} =$
$\{ \neg \mathit{ship}_2 \vee \neg \mathit{ship}_3 \rightarrow \underline{\mathit{ship}_1},$
$\neg \mathit{ship}_1 \vee \neg \mathit{ship}_3 \rightarrow \underline{\mathit{ship}_2},$
$\neg \mathit{ship}_1 \vee \neg \mathit{ship}_2 \rightarrow \underline{\mathit{ship}_3} \}$

In this specification each behaviour $\mathit{local_customer}_i$ ($i = 1, 2, 3$) represents the local constraints with respect to a particular use-case. Each use-case consists of the related interactions between the business process and a particular customer. The behaviour *remote_between_customers* represents the remote constraints with respect to all use-cases.

The behaviour is represented graphically in Figure 5.5.

Business processes organised around business functions

An extensional causality-oriented style is well-suited for the specification of requirements for business processes organised around business functions.

The causality-oriented style is suitable in this case, because a causality-oriented style is suitable to model behaviours structured according to phases and because business functions are phases in the process of delivering a product or service. The application of a causality-oriented style thus allows concerns that relate to different phases of a business process to be separated, where a phase comprises one or more business functions. This separation of concerns can be re-used in a specification of the implementation of a business process organised around business functions.

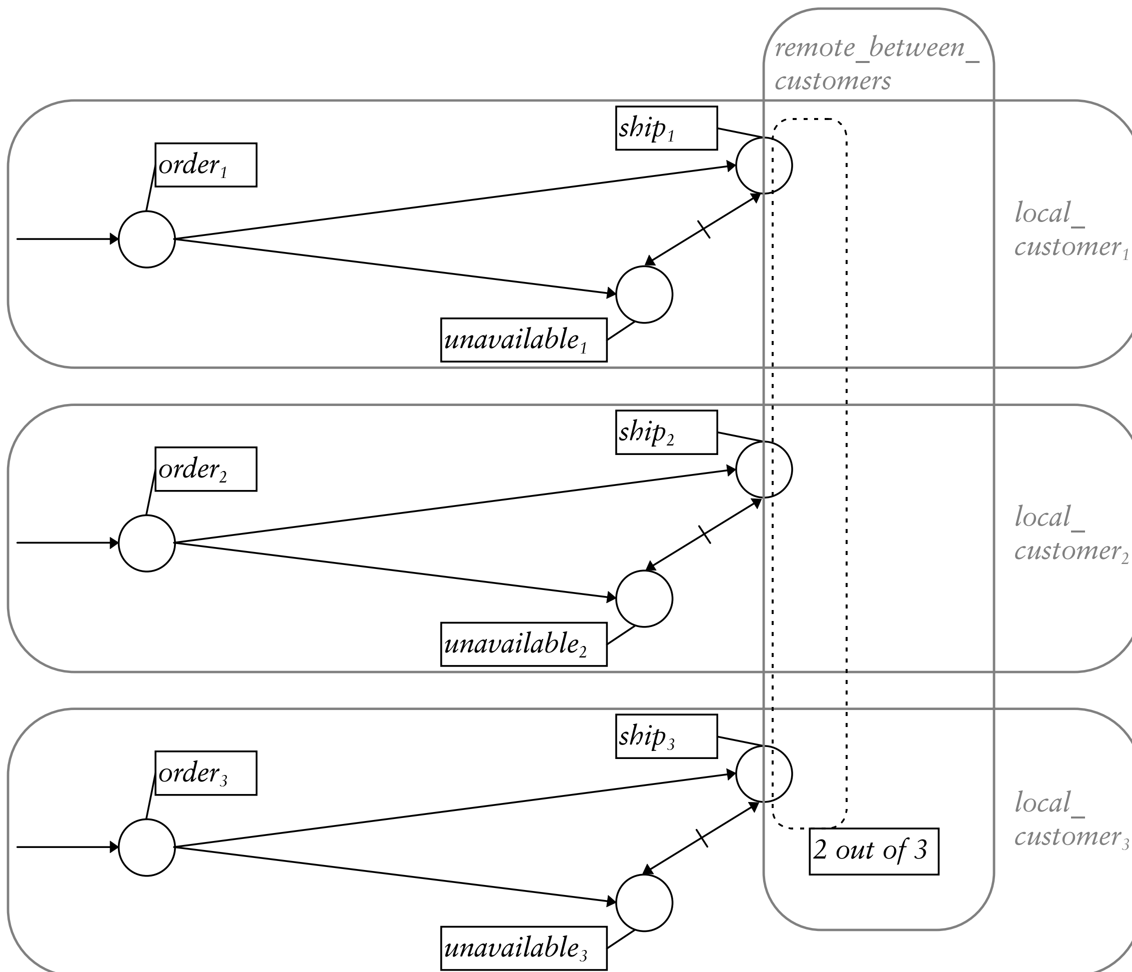


Figure 5.5 Requirements for example behaviour according to extensional constraint-oriented style.

We call an extensional causality-oriented style in which the sub-behaviours represent business functions an extensional *business function-oriented style*.

The requirements for the behaviour of the example mail-order company can be specified in an extensional business function-oriented style as follows.

$order_function =$
$\{ \; start \rightarrow order_1,$
$\quad start \rightarrow order_2,$
$\quad start \rightarrow order_3,$
$\quad order_1 \rightarrow store_and_ship_function(entry_1),$
$\quad order_2 \rightarrow store_and_ship_function(entry_2),$
$\quad order_3 \rightarrow store_and_ship_function(entry_3) \quad \}$

$store_and_ship_function =$
$\{ \; entry_1 \wedge \neg unavailable_1 \wedge (\neg ship_2 \vee \neg ship_3) \rightarrow ship_1,$

$entry_2 \wedge \neg unavailable_2 \wedge (\neg ship_1 \vee \neg ship_3) \rightarrow ship_2,$
$entry_3 \wedge \neg unavailable_3 \wedge (\neg ship_1 \vee \neg ship_2) \rightarrow ship_3$
$entry_1 \wedge \neg ship_1 \rightarrow unavailable_1,$
$entry_2 \wedge \neg ship_2 \rightarrow unavailable_2,$
$entry_2 \wedge \neg ship_2 \rightarrow unavailable_3 \quad \}$

In this specification the behaviour *order_function* contains all interactions of the order function with the company's customers and their relations (no relations in this case). The behaviour s*tore_and_ship_function* contains all interactions of the shipment function with the company's customers and the relations between these interactions imposed by the store function and the shipment function.

The behaviour is represented graphically in Figure 5.6.

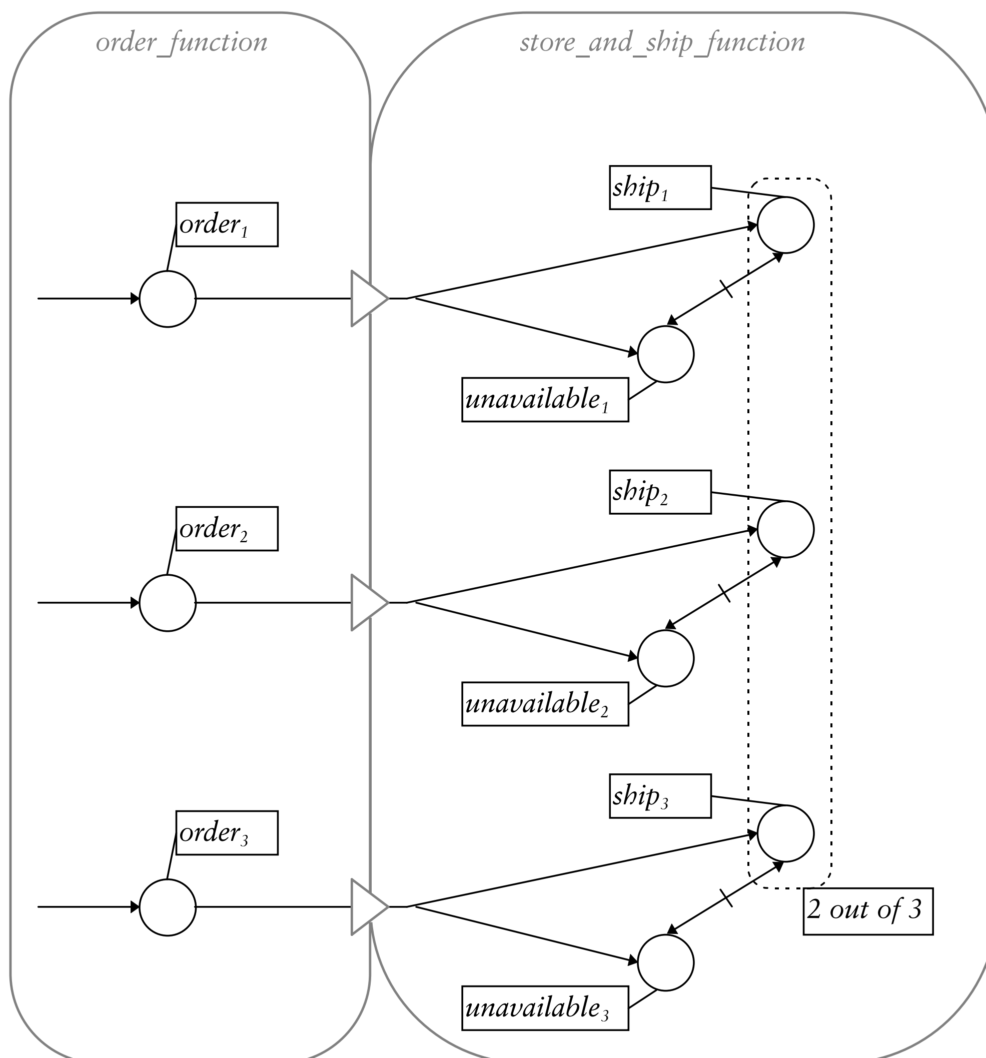


Figure 5.6 Requirements for example behaviour according to extensional causality-oriented style.

5.4.3 Implementation modelling

Business processes organised around work-flows

An intensional constraint-oriented style is well-suited for the specification of the implementation of business processes organised around work-flows.

This is because it enables the modelling of the behaviours of each of the units that carry out a work-flow, which interact with the behaviours of each of the resources shared by the work-flows.

We call an intensional constraint-oriented style in which the sub-behaviours represent either local constraints with respect to a work-flow or remote constraints with respect to multiple work-flows a *work-flow-oriented style*.

The requirements for the behaviour of the example mail-order company can be specified in a work-flow-oriented style as follows.

{ interactions $workflow_1$, $store$: int_ship_1,
$workflow_2$, $store$: int_ship_2,
$workflow_3$, $store$: int_ship_3 }

$workflow_1$=
{ $start \rightarrow order_1$,
$order_1 \rightarrow int_order_1$,
$int_order_1 \wedge \neg int_unavailable_1 \rightarrow \underline{int_ship_1}$,
$int_order_1 \wedge \neg int_ship_1 \rightarrow int_unavailable_1$,
$int_ship_1 \rightarrow ship_1$,
$int_unavailable_1 \rightarrow unavailable_1$ }

$workflow_2$=
(* similar to $workflow_1$ *)

$workflow_3$=
(* similar to $workflow_1$ *)

$store$=
{ $\neg int_ship_2 \vee \neg int_ship_3 \rightarrow \underline{int_ship_1}$,
$\neg int_ship_1 \vee \neg int_ship_3 \rightarrow \underline{int_ship_2}$,
$\neg int_ship_1 \vee \neg int_ship_2 \rightarrow \underline{int_ship_3}$ }

In this specification each behaviour $workflow_i$ ($i= 1, 2, 3$) represents the behaviour of each of the units that carries out a work-flow. The behaviour *store* represents the behaviour of a resource shared by the work-flows: the store with the stock.

The structure of the behaviour is represented graphically in Figure 5.7. This structure can be considered as a “parallel” structure: consisting of a number of independent sub-behaviours that synchronise with one or more other (central) behaviours.

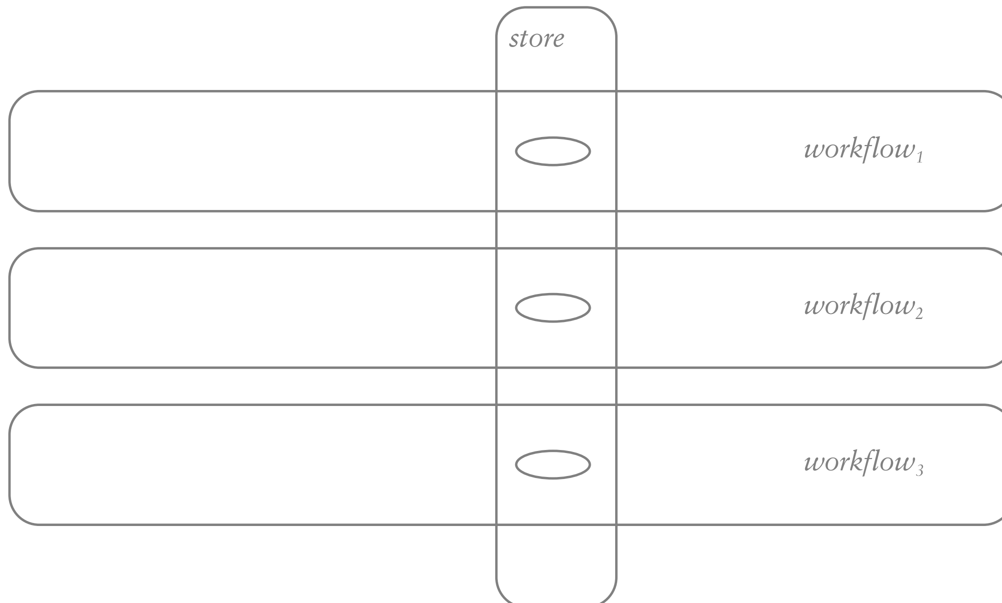


Figure 5.7 Structure of implementation of example behaviour organised around work-flows

Business processes organised around business functions

An intensional constraint-oriented style is also well-suited for the specification of the implementation of business processes organised around business functions.

This is because it enables the modelling of the interacting behaviours of the units that each carry out a business function.

We call an intensional constraint-oriented style in which the sub-behaviours represent business functions an intensional *business function-oriented style*.

The behaviour of the example mail-order company can be specified in an intensional business function-oriented style as follows.

{ interactions $order_function, store_function: int_order_1, int_order_2, int_order_3,$
$store_function, ship_function: int_ship_1, int_ship_2, int_ship_3,$
$int_unavailable_1, int_unavailable_2, int_unavailable_3$ }

$order_function =$
{ $start \rightarrow order_1,$
$start \rightarrow order_2,$
$start \rightarrow order_3$
$order_1 \rightarrow \underline{int_order_1},$
$order_2 \rightarrow \underline{int_order_2},$
$order_3 \rightarrow \underline{int_order_3}$ }

$store_function =$
{ $start \rightarrow \underline{int_order_1},$

$start \rightarrow \underline{int_order_2}$,
$start \rightarrow \underline{int_order_3}$,
$int_order_1 \wedge \neg int_unavailable_1 \wedge (\neg int_ship_2 \vee \neg int_ship_3) \rightarrow \underline{int_ship_1}$
$int_order_2 \wedge \neg int_unavailable_1 \wedge (\neg int_ship_1 \vee \neg int_ship_3) \rightarrow \underline{int_ship_2}$
$int_order_3 \wedge \neg int_unavailable_1 \wedge (\neg int_ship_1 \vee \neg int_ship_2) \rightarrow \underline{int_ship_3}$
$int_order_1 \wedge \neg int_ship_1 \rightarrow \underline{int_unavailable_1}$,
$int_order_2 \wedge \neg int_ship_2 \rightarrow \underline{int_unavailable_2}$,
$int_order_3 \wedge \neg int_ship_3 \rightarrow \underline{int_unavailable_3}$ }

ship_function=
{ $start \rightarrow \underline{int_ship_1}$,
$start \rightarrow \underline{int_ship_2}$,
$start \rightarrow \underline{int_ship_3}$,
$start \rightarrow \underline{int_unavailable_1}$,
$start \rightarrow \underline{int_unavailable_2}$,
$start \rightarrow \underline{int_unavailable_3}$,
$int_ship_1 \rightarrow ship_1$,
$int_ship_2 \rightarrow ship_2$,
$int_ship_3 \rightarrow ship_3$,
$int_unavailable_1 \rightarrow unavailable_1$,
$int_unavailable_2 \rightarrow unavailable_2$,
$int_unavailable_3 \rightarrow unavailable_3$ }

In this specification each behaviour *order_function*, *store_function* and *ship_function* represents the behaviour of a unit that carries out a business function.

The structure of the behaviour is represented graphically in Figure 5.8. This structure can be considered as a "cascading" structure: consisting of a number of sub-behaviours, such that each sub-behaviour interacts with at most two other sub-behaviours.

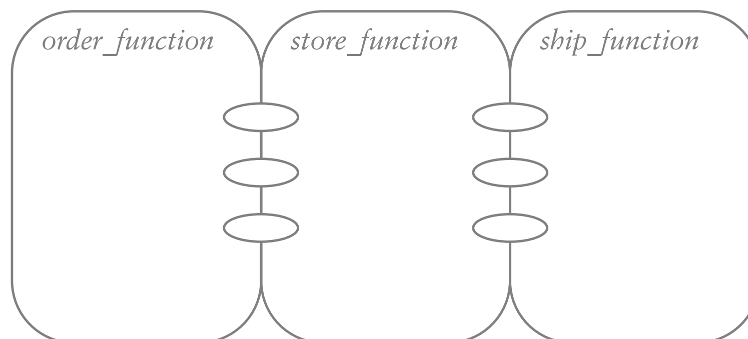


Figure 5.8 Structure of implementation of example behaviour organised around business functions

5.5 Role of styles in development methods

The extensional style supports the specification of system requirements, whereas the intensional style supports the specification of the implementation of a system. Therefore the extensional style is more suitable in the initial phases of the development of the system, whereas the intensional style is more suitable in the later phases.

Extensional and intensional styles can be applied iteratively in a development method in the following way. The behaviour of a system is initially specified extensionally, after which it is decomposed into sub-behaviours that are assigned to entities. The behaviour of each entity is then specified extensionally, so that the behaviour of the entire system is specified intensionally. This process is repeated until the entities need not be decomposed any further.

The usefulness of the other identified styles depends more on the type of system under development than on their place in the development process. The process-oriented style is suitable for the specification of behaviours in which the occurrences of actions do not depend on complex functions of action values. The state-oriented style is suitable for the specification of behaviours in which the occurrences of actions do depend on complex functions of action values. Similarly, the monolithic style is suitable for the specification of very simple behaviours, whereas the modular style is almost mandatory for more complex behaviours.

The modular styles, the causality-oriented style and the constraint-oriented style, do not provide much guidance for the selection of sub-behaviours. In business process modelling these sub-behaviours can represent work-flows or business functions.

One of the objectives of system structuring is to facilitate the implementation of the system. If a business process is carried out by units that are each responsible for a work-flow, this business process is best extensionally modelled using a constraint-oriented style, in which each sub-behaviour either represents the local constraints with respect to a use-case (the extensional aspects of a work-flow) or the remote constraints with respect to multiple use-cases. This form of separation of concerns can then be re-used in an intensional constraint-oriented model of the business process: the local constraints with respect to a use-case are decomposed and implemented by a work-flow, whereas the remote constraints are decomposed and implemented partly by work-flows and partly by resources shared by the work-flows. The intensional model can then be mapped onto the final implementation of the business process organised around work-flows, again with preservation of structure.

If a business process is carried out by units that are each responsible for a business function, this business process is best extensionally modelled using a causality-oriented style, in which each sub-behaviour represents a phase comprising one or more business functions. This form of separation of concerns can then be re-used in an intensional constraint-oriented model of the business process: the extensionally

identified sub-behaviours are decomposed and implemented by interacting business functions. The intensional model can then be mapped onto the final implementation of the business process organised around business functions, again with preservation of structure.

These alternatives for structuring business processes in a development process are shown in Figure 5.9.

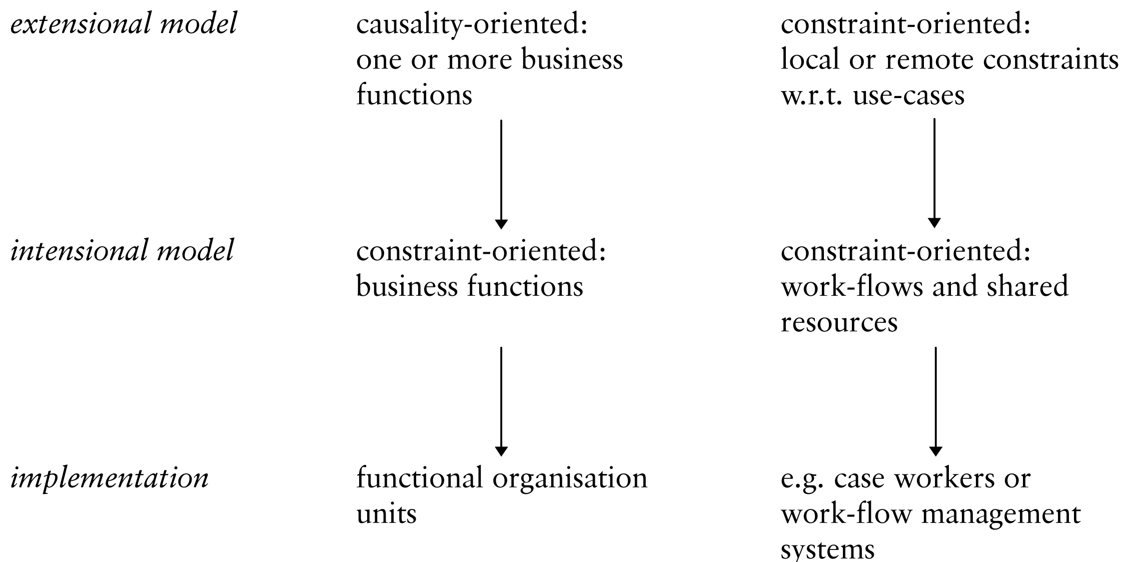


Figure 5.9 Structuring of business processes in a development process

5.6 Conclusions

Modelling styles ease the structuring of behaviour models and should lead to more uniformly structured models. Our new insights regarding the basic architectural concepts enabled us to extend the modelling styles for distributed systems of Vissers et al. [1988] and to define them in such a way that they are more orthogonal. We defined the following styles and the development objectives they support:

- the extensional and intensional styles, based on the absence or presence of internal (inter)actions in a behaviour, respectively;
- the monolithic and modular styles, based on the absence or presence of sub-behaviours in a behaviour, respectively; the modular style was refined in the constraint-oriented style and the causality-oriented style, based on the relations between sub-behaviours: shared causality relations or shared actions (interactions), respectively;
- the process-oriented and state-oriented styles, based on the absence or presence of state variables and state functions in a behaviour, respectively.

Our distinction of frequently occurring structures of business processes enabled us to define concrete modelling styles for business processes that aid developers in efficiently modelling business processes in an insightful way, both intensionally and extensionally.

- The use-case-oriented style is an extensional constraint-oriented style in which local and remote constraints are defined with respect to use-cases. The work-flow-oriented style is an intensional constraint-oriented style in which local and remote constraints are defined with respect to work-flows.
- The extensional business function-oriented style is an extensional causality-oriented style in which the sub-behaviours represent business functions. The intensional business function-oriented style is an intensional constraint-oriented style in which the sub-behaviours represent business functions.

6

Abstraction levels

6.1 Introduction

6.1.1 Motivation

During a system development process a large number of decisions are taken regarding the system under development. By abstracting from the properties of the system that are irrelevant for a particular development decision, a developer concentrates on that decision, without being bothered by other concerns. Therefore, many abstractions of a system are conceived during a development process. Such abstractions are sometimes called perspectives.

The complete model of the system is formed by the composition of all correct abstractions of the system resulting from the development process. Therefore, it is important to know how the abstractions relate to each other: if these relations are unclear, the meaning of the composition of these abstractions is undefined.

Models of a system at distinct abstraction *levels* are ordered on the basis of the amount of detail they possess. A model *A* of a system at a *higher* abstraction level than another model *B* of the same system is not only an abstraction of the system, but also of *B*. Model *B* is then at a *lower* abstraction level and thus contains more detail than *A*, while also expressing all properties expressed in *A*.

The use of abstraction levels in a development process is beneficial for the following reasons. First, it supports a separation between development concerns regarding the requirements of the system and development concerns regarding the implementation of the system, since the former can be properly represented at higher abstraction levels and the latter at lower ones. Second, models at distinct abstraction levels may serve as milestones in a development process, i.e. as models whose development is planned and controlled, which are reviewed, and which then serve as a basis for further development. Third, the use of abstraction levels eases validation of models

at intermediate levels, since the "distance" between models at adjacent abstraction levels is usually kept short.

Abstraction levels are beneficial in a development process, if they satisfy the following conditions (based on [Van Sinderen et al., 1995; Essink, 1986]):

- *Proper support for development objectives*. In different stages of a development process developers have different objectives. The definition of abstraction levels should ease the pursuit of these development objectives by 1) making the objectives pursued at each level explicit and 2) describing the model structure that is appropriate in the pursuit of each objective.
- *Well-defined relations between the abstraction levels*. The relations between the abstraction levels should be well-defined, so that 1) it is possible to validate whether a model at a particular abstraction level is a proper refinement of a model at a higher abstraction level, and 2) developers are guided in the type of transformations they have to carry out to proceed from each abstraction level to the next lower one.
- *General applicability*. The number of abstractions conceived in a development process is large; the amount of abstractions conceived in multiple development processes is even larger. For the abstraction levels to be useful in most development processes and to be used as milestones, a limited number of generally applicable abstraction levels should be defined. (A survey of development methods that support abstraction levels, e.g. those of [Aue & Breu, 1994; Yourdon, 1989; MacDonald, 1986; Essink, 1986; Lundeberg, 1982], shows that all of these methods describe three to five general abstraction levels.)

In this chapter we elaborate these conditions and develop a group of related abstraction levels that satisfy them.

6.1.2 Structure

This chapter is structured as follows.

Section 6.2 defines abstraction and refinement, defines the concept of abstraction levels and discusses some properties of abstraction levels.

Section 6.3 introduces some generic abstraction levels. These are based on different perspectives of systems and their parts. The section discusses the integrated system perspective, the distributed system perspective, and the interaction system perspective of the system parts.

Section 6.4 introduces five specific abstraction levels. These abstraction levels are, apart from the different perspectives of systems and their parts of section 6.3, based on the entities one should take into account during a development process. They thus give more specific support to the pursuit of development objectives.

Section 6.5 applies these abstraction levels to the development of an example system.

6.2 Abstraction, refinement, and their role in abstraction levels

6.2.1 Abstraction levels

Given two models *A* and *B* of a system, we say that *A* is defined at a higher abstraction level than *B* (or, equivalently, that *B* is defined at lower abstraction level than *A*) if and only if:

1. *A* expresses less properties than *B*, and
2. all properties of the system expressed in *A* are also expressed in *B*.

In that case we call *B* a *refinement* of *A*. Multiple models of a system are defined at distinct abstraction levels if each model is defined either at a higher or at a lower abstraction level than any other model.

The ordering of models on the basis of the amount of detail they contain, as in the first requirement, is important for the identification of abstraction levels, but insufficient to ensure well-defined relations between all models: a model *A* of a system may express more properties than another model *B*, but not express all properties expressed in *B*. The ordering of models on the basis of the amount of detail they posses is thus a partial ordering, so that developers may produce models that are unrelated.

The second requirement ensures for models ordered according to distinct abstraction levels that if a model *A* is defined at a higher abstraction level than a model *B*, *A* is not only an abstraction of the system, but of *B* as well. The ordering of models on the basis of abstraction levels is thus a linear ordering. Therefore each model is related to every other model, by being either at a higher or at a lower abstraction level.

Figure 6.1 shows models at distinct abstraction levels and the abstraction and refinement relations between them.

A model of a system at a high abstraction level is appropriate for expressing the system requirements, since it allows one to abstract from the way in which these requirements are realised. The model at the lowest abstraction level should contain all detail required for the implementation of the system.

This implies that the type of separation of concerns supported by the use of abstraction levels in a development process, the separation between requirements concerns and implementation concerns, is best exploited by considering (each aspect of) the system under development at consecutively lower abstraction levels. This is the case when one uses, e.g., the waterfall model or the evolutionary development strategy. See chapter 7 for a more detailed discussion of development strategies.

Abstraction levels should not be confused with viewpoints, which are sometimes used in system development (see e.g. [Bowen, 1991; Linington, 1992]). Such view-

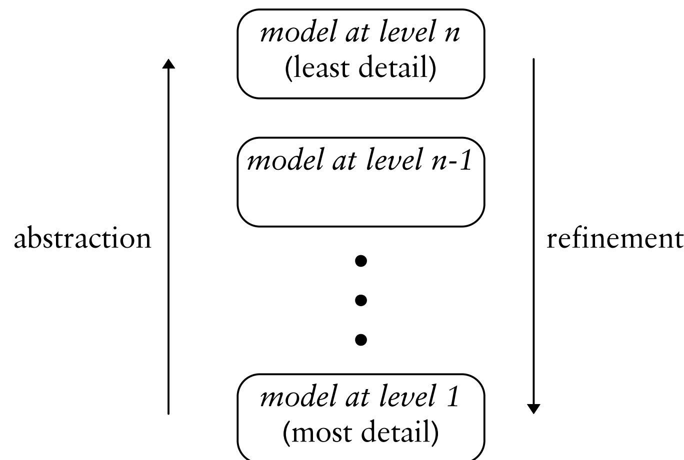


Figure 6.1 Models at distinct abstraction levels and abstraction and refinement relations.

points are abstractions of a system under development, but the relations between these abstractions are not well-defined.

6.2.2 Refinement operations

Given a model at a particular abstraction level, the development of a lower-level model requires refinement. Here we discuss, based on [Quartel et al., 1996; Ferreira Pires, 1994], some operations to be used in such refinement activity. In the discussion we term the model/behaviour to be refined the *abstract* model/behaviour and the model/behaviour resulting from the refinement the *concrete* model/behaviour.

Behaviour domain

Action refinement is the replacement of an abstract action by one or more related concrete actions. We distinguish between:

- *Action decomposition*. This is the replacement of an abstract action by a set of related concrete actions. Figure 6.2a illustrates action decomposition.

  As an example, consider the activity of building a house. At a high abstraction level it may only be relevant that the house is built, so the activity is modelled as a single action. At a lower abstraction level it may also become relevant how the house is built, requiring the original single action to be decomposed into a set of related actions such as laying bricks, doing carpentry, etc.

- *Action distribution*. This is the replacement of an abstract action by an interaction. The action is thus distributed over sub-behaviours. Figure 6.2b illustrates action distribution.

  As an example, consider the activity of evaluating an insurance application on multiple aspects. At a high abstraction level it may only be relevant that the

application is evaluated and not by whom this is done. At a lower abstraction level this may become relevant, requiring the action to be replaced by an interaction to which multiple experts contribute.

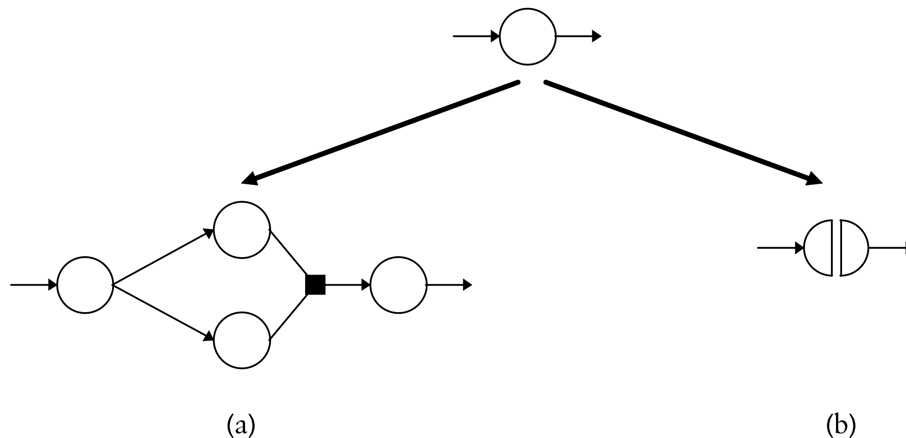


Figure 6.2 Examples of two types of action refinement. (a) action decomposition, (b) action distribution.

Entity domain

Action point refinement is the replacement of an abstract action point by multiple concrete action points or a concrete interaction point. We distinguish between:

- *Action point decomposition*. This is the replacement of an abstract action point by multiple concrete action points.

  Action decomposition in the behaviour domain requires action point decomposition in the entity domain, and vice versa, if any of the concrete actions that result from the refinement take place at a different location than the abstract action.

- *Action point distribution*. This is the replacement of an abstract action point by a concrete interaction point.

  Action distribution in the behaviour domain requires action point distribution in the entity domain, and vice versa.

Entity decomposition is the replacement of an abstract entity by a structure of concrete entities.

Action point distribution implies entity decomposition, since an interaction point is always shared by two or more entities. Entity decomposition requires action point distribution and action distribution if the resulting concrete entities interact.

6.2.3 Conformance

Consider models defined at consecutive abstraction levels. A model at a lower abstraction level should preserve all properties of a model at a higher abstraction level. This requirement is called a *conformance* requirement.

Quartel et al. [1996] distinguish two types of conformance requirements:

- *Preservation of relations*. The relations between actions in the abstract behaviour should be preserved by the relations between their corresponding actions in the concrete behaviour.
- *Preservation of action values*. The values established in actions in the abstract behaviour should be preserved by the values established in their corresponding actions in the concrete behaviour.

Quartel et al. give a method to determine whether a concrete behaviour conforms to an abstract behaviour. For the convenience of the reader we summarise this method here. The method is based on the recognition that the refinement of an abstract behaviour can result in many alternative concrete behaviours, whereas the abstraction of a concrete behaviour is unique. Therefore, the method requires the assessment of conformance by comparing the abstraction of the concrete behaviour to the original abstract behaviour. It comprises the following steps:

1. *Identification of reference actions and inserted actions in the concrete behaviour*. Reference actions are actions in the concrete behaviour that correspond to actions in the abstract behaviour. Inserted actions are actions in the concrete behaviour that do not correspond to actions in the abstract behaviour, i.e. they are actions that have been inserted during refinement.
2. *Abstraction from all of the inserted actions*. This requires for each inserted action *i*:
   - the replacement of each causality condition in which *i* occurs by an equivalent causality condition in which *i* does not occur;
   - the replacement of each reference relation defined in terms of values established in *i* by an equivalent reference relation in which these values do not occur.
3. *Replacement of each group of reference actions by an abstract action*. Each group of reference actions whose occurrence corresponds to the occurrence of a single abstract action should be replaced by the abstract action.
4. *Comparison of the abstraction of the concrete behaviour to the original abstract behaviour*. The concrete behaviour only conforms to the abstract behaviour if both behaviours comply to a certain correctness relation. Depending on the conformance requirement, this correctness relation can be:

- an equivalence relation which defines that the concrete behaviour should preserve all properties of the abstract behaviour (e.g., for action values this means that all action values possible for an abstract action are also possible for the corresponding concrete actions), or
- a partial order relation which defines that the concrete behaviour should preserve a subset of the properties of the abstract behaviour (e.g., for action values this means that some action values possible for an abstract action are not possible for the corresponding concrete actions).

Consider the behaviours of Figure 6.3 as an example. We wish to assess whether the concrete behaviour of Figure 6.3b conforms to the abstract behaviour of Figure 6.3a, assuming that *e'* is an inserted action and that c_1' and c_2' are reference actions, such that the occurrence of both c_1' and c_2' corresponds to the occurrence of *c* in the abstract behaviour. Step 2 requires the abstraction from the inserted action *e'*, which results in the behaviour of Figure 6.3c. Step 3 requires the replacement of c_1' and c_2' by a single action, which results in the behaviour of Figure 6.3d. This behaviour is considered equivalent to the behaviour of Figure 6.3a in case the different action identifiers are abstracted from. The concrete behaviour thus conforms to the abstract behaviour.

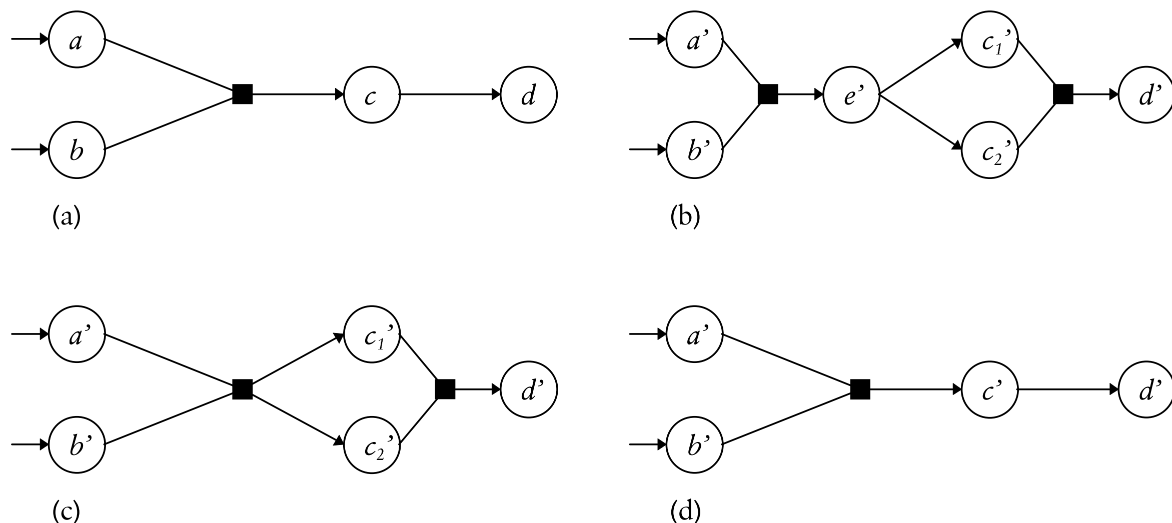


Figure 6.3 Behaviour abstraction. (a) abstract behaviour, (b) concrete behaviour, (c) concrete behaviour after step 2, (d) concrete behaviour after step 3.

6.3 Generic abstraction levels

6.3.1 Distributed and integrated system perspectives

A system is a group of interacting parts forming a unified whole. Since a model of a system may serve as a prescription for its realisation, it is necessary to be able to represent a system as a group of interacting parts in a model. This is termed a representation of the system from the *distributed perspective*. Figure 6.4a illustrates the distributed system perspective.

Many different compositions of entities may implement the same externally observable aspects of a system. Therefore, it is useful to be able to represent as a system as a whole, independent of any specific implementation in terms of a group of interacting parts. This is termed a representation from the *integrated perspective*. Figure 6.4b illustrates the integrated system perspective.

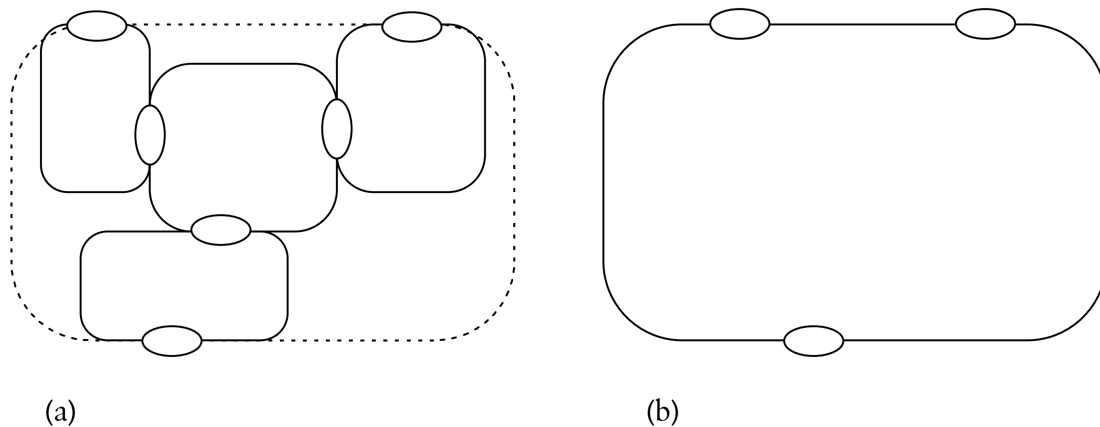


Figure 6.4 Example of system represented from (a) distributed perspective, (b) integrated perspective. Figure shows entity domain only.

The integrated and distributed system perspectives are two consecutive abstraction levels: the integrated system perspective is a higher abstraction level than the distributed system perspective. Entities can be viewed as compositions of other entities. (For example, an organisation can be viewed as a composition of business units, which in turn can be viewed as compositions of work units, which in turn can be viewed as compositions of people, computer systems, etc.) Therefore, the distributed and integrated perspectives can be applied repeatedly as follows. At the highest abstraction level a system is represented from the integrated perspective. At the next lower level it is represented from the distributed perspective, showing the system parts, such that each part is represented from the integrated perspective. At the next lower level each part is represented from the distributed perspective, and so on. Figure 6.5 shows the recursive application of entity decomposition graphically.

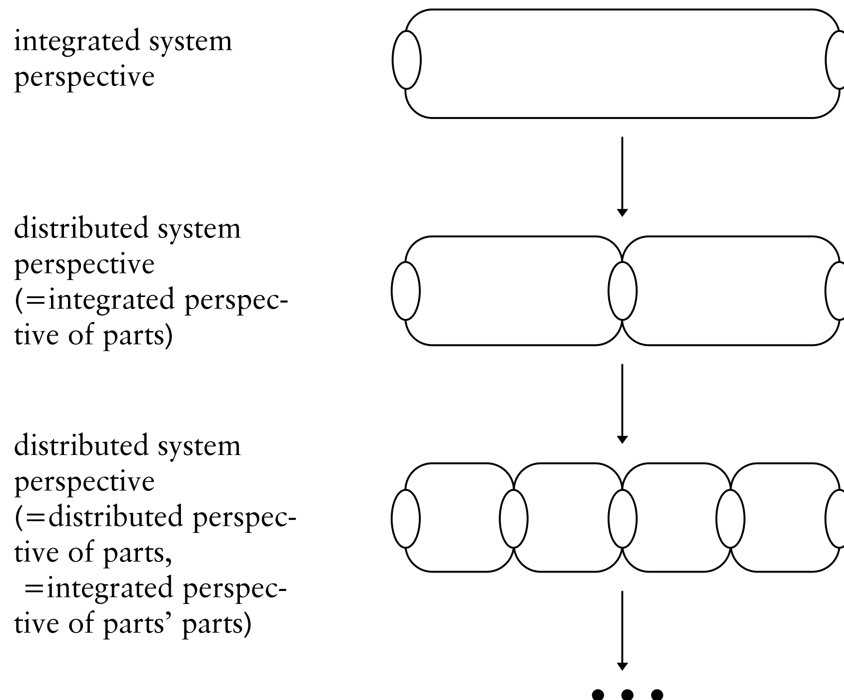


Figure 6.5 Example of repeated integrated and distributed perspectives of system and parts as abstraction levels. Arrows represent decomposition.

Methods requiring the use of data flow diagrams or similar modelling techniques (e.g. [De Marco, 1979; MacDonald, 1986; Yourdon, 1989]) support the integrated and distributed system perspectives. At every abstraction level the "processes" in a data flow diagram are represented from the integrated perspective, in terms their inputs, their outputs, and (informally) the relations between the inputs and outputs. At a lower abstraction level, the processes are decomposed into related sub-processes, which are again described from the integrated perspective. This refinement operation is termed "functional decomposition" in [Madisson, 1983].

Some methods for protocol development based on the definition of successively lower protocol layers (e.g. [Vissers et al., 1995]) also support the integrated and distributed system perspectives. At the highest abstraction level, the service level, the system under development is defined from the integrated perspective. At the next lower level, the system is decomposed into a number of protocol entities (the protocol layer) and a lower-level service. By considering the lower level service as the system to be further developed, the abstraction levels can be applied repeatedly.

6.3.2 Interaction system perspective of parts

A set of actions in a system can be distributed over the system parts in many different ways. Therefore, it is desirable to be able to represent the system intensionally, independent of any specific distribution of the internal actions over the system parts. Since these related internal actions represent the common behaviours of the parts,

i.e. their interaction systems, this is termed the *interaction system perspective of the parts*. Figure 6.6 illustrates this perspective.

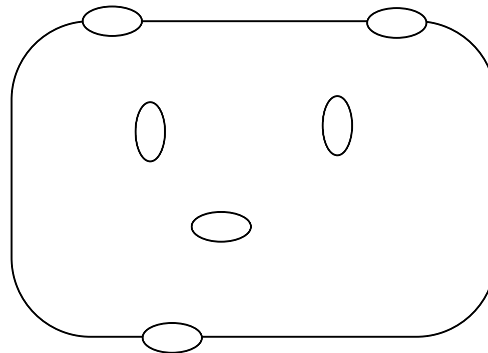


Figure 6.6 Example of system represented from interaction system perspective of parts.

The interaction system perspective of parts is an abstraction level in between the integrated system perspective and the distributed system perspective. Figure 6.7 gives an example of the use of these three perspectives. Because the parts in the distributed system perspective are themselves represented from the integrated perspective, the levels can again be applied repeatedly.

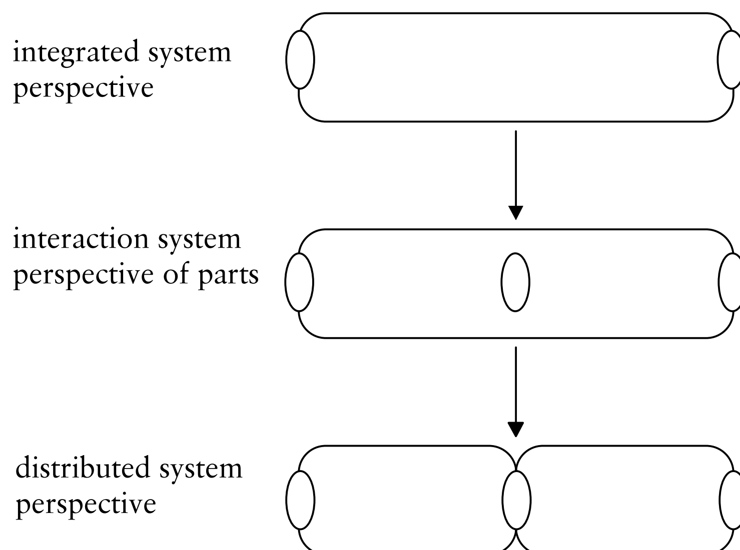


Figure 6.7 Three system perspectives as abstraction levels.

The interaction system perspective is supported by several development methods [De Weger & Vissers, 1994]. For example, the highest abstraction level in ISAC [Lundeberg, 1982] requires an information system to be represented intensionally without the representation of its parts. Only at a lower abstraction level the internal activities are distributed over information sub-systems. In the method for the

development of distributed information systems of Aue and Breu [1994], activities of an information system and a part of its environment are specified at the "business context level", without making a distinction between the information system and the entities in the environment. Only at the lower "service level" these activities are distributed over the information system and the entities in the environment. Similarly, in the framework of Ferreira Pires [1994] the interaction system of the system under development and its environment is to be specified before the actions are distributed over the system and its environment, at a lower abstraction level.

It appears that in most of these development methods the interaction system perspective is supported only at the highest abstraction level, where one considers the interaction system of the system under development and its environment. This can be explained from the purpose of the interaction system perspective: internal actions are introduced to allow for explicit consideration of the different ways in which they can be distributed over entities at a lower abstraction level. At a high abstraction level one has considerable freedom in the definition of entities that carry out the activities. Therefore, this type of separation of concerns is useful. However, at lower abstraction levels one becomes increasingly bound by the available implementation components. At those levels one generally introduces internal actions with the intent of distributing them over a particular intended structure of implementation components. Therefore, there is less need for explicit consideration of the different ways in which actions can be distributed over entities at lower abstraction levels.

6.3.3 Summary

Figure 6.8 summarises the aspects represented in the perspectives defined above. The figure shows that in none of the perspectives system parts are represented without their interactions. Even though such a perspective is conceivable, because the entity domain and the behaviour domain are orthogonal, it is of limited practical use, since its application results in incomplete models.

		representation of internal (inter)actions?	
		no (extensional)	*yes (intensional)*
representation of system parts?	*no*	integrated system	interaction system of parts
	yes		distributed system

Figure 6.8 Aspects of systems represented in three system perspectives.

6.4 Specific abstraction levels

The abstraction levels introduced in this section are based on the entities one should take into account during the development of a system, and on the different system perspectives defined in section 6.3. The entities to be taken into account are [De Weger et al., 1995a]:

- the system embedded in its environment;
- the system;
- the "logical" system parts;
- the "physical" system parts.

Some of the abstraction levels defined here are discussed in [Ferreira Pires, 1994] and [Van Sinderen et al., 1995].

6.4.1 System embedded in its environment

Objective. At this abstraction level the environment in which the system operates is defined. This is done for the following purposes:

- Various behaviours of the environment in which the system is embedded can be considered and evaluated during the development of a model at this abstraction level. This is especially important if a system is developed from scratch, since a new system may create opportunities for its environment to work in new ways.
- The resulting model forms a basis for determining which activities have to be supported by the system and to which extent they have to be supported.

Structure. A model at this abstraction level forms an intensional description of the entity consisting of the system under development and a bounded part of its environment. This entity is considered as a single entity without parts, i.e. the system and its environment are not explicitly defined.

In principle, a developer is free to delimit this entity. However, it is sensible to limit the activities represented in a model at this level to those activities possibly supported by the system under development. This prevents the danger of considering a far too large part of the environment, which may increase the development effort immensely, while not rendering any additional results that are useful in the development process [Yourdon, 1989].

Since a model at this abstraction level serves as a basis for distributing activities over the system under development and its environment, it ideally only represents the related activities shared by the system and the environment, i.e. their interaction system. However, in practice this is usually difficult, because the boundary between system and environment has not yet been decided upon.

Figure 6.9a illustrates the perspective of the system embedded in its environment.

Use in other development methods. See section 6.3.2.

6.4.2 Integrated system perspective

Objective. At this abstraction level the behaviour of the system as it is observed by its environment is defined. This is done for the following purposes:

- Various alternative distributions of behaviour responsibilities over the system and (entities in) its environment can be considered and evaluated during the development of a model at this abstraction level.
- The resulting model forms the basis for the development of models at lower abstraction levels, which represent the system as compositions of interacting parts; each composition of interacting system parts should conform to the model of the system from the integrated system perspective.

Structure. A model at this abstraction level represents the system as an integrated whole. Therefore, it does not represent any parts of the system. Preferably it does not describe any internal activities of the system either (i.e. it is extensional). If it does describe any internal activities, these are not prescriptive for implementations.

Relation with next higher level. A model at this abstraction level is a refinement of a model defined at the next higher level, in which the system and a part of its environment are viewed as a single entity. The following refinement operations are important for performing this refinement:

- decomposition of the entity consisting of system and environment into the system and (one or more entities of) its environment;
- distribution of the behaviour of the entity consisting of system and environment over the system and its environment.

This can be done in two steps:

1. delimitation of the interaction system of the system under development and its environment;
2. distribution of this interaction system over the system under development and its environment.

Figure 6.9 illustrates these steps.

Use in other development methods. See section 6.3.1. The behaviour of the system from the integrated perspective is called the *service* of the system. The system is called the *service provider* [Vissers & Logrippo, 1986].

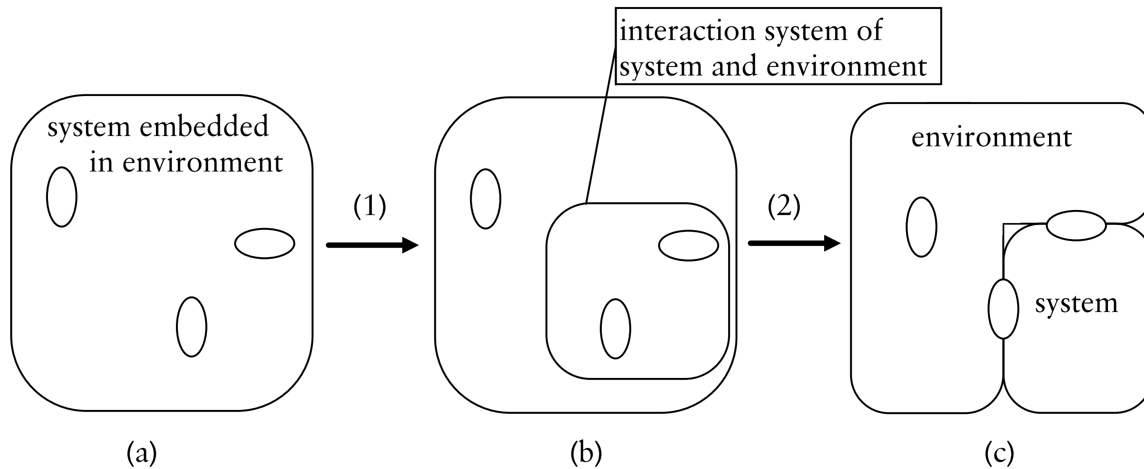


Figure 6.9 From system embedded in environment to integrated system perspective.

6.4.3 Logically distributed system perspective

Objective. At this abstraction level the system is defined as a composition of parts, while abstracting from the physical (i.e. geographical) distribution of these parts. At this abstraction level the parts can interact directly, i.e. without intermediate entities. This is done for the following purposes:

- Various decompositions of the system in logical parts that render the service of the integrated system perspective can be considered and evaluated during the development of a model at this abstraction level.
- The resulting model forms the basis for the development of a model at the next lower abstraction level, at which physical distribution is considered.

Structure. A model at this abstraction level represents the system from the distributed perspective. Therefore, it represents the system as a composition of interacting parts. Each of the parts is represented from the integrated perspective.

Relation with next higher level. A model at this abstraction level is a refinement of a model defined from the integrated system perspective. The following refinement operations are important in making this refinement:

- introduction of internal actions (by means of action refinement) and action points;
- decomposition of the system and distribution of the introduced actions and action points over the resulting entities.

Figure 6.10 illustrates this refinement operation.

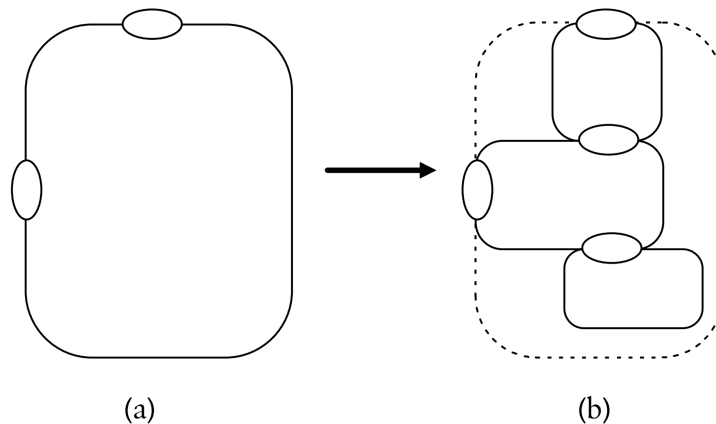


Figure 6.10 From (a) integrated system perspective to (b) logically distributed system perspective.

Use in other development methods. The distinction between the logically distributed perspective and the physically distributed perspective of a system can be found in some development methods. For example, in LOVEM [IBM, 1995] a distinction is made between the "logical line of visibility chart" and the "physical line of visibility chart". To proceed from the first perspective to the second, the "means of transportation" have to be introduced. In [Van Sinderen et al., 1995] a distinction is made between the "partitioned perspective" of a system, in which the geographical distribution is abstracted from, and the "distributed perspective" of a system, in which this geographical distribution is explicitly defined.

6.4.4 Physically distributed system perspective

Objective. At this abstraction level the system is defined as a composition of the lowest-level system parts. This means that each defined part is implemented in the real system by a single component. In comparison to the logically distributed system perspective, the physically distributed system perspective requires, for example, the additional specification of the system parts responsible for communication between geographically remote parts, the "communication infrastructures".

Structure. A model at this abstraction level represents the system from the distributed perspective. Therefore, it represents the system as a composition of interacting parts. Each of the parts is represented from the integrated perspective.

Relation with next higher level. A model at this abstraction level is a refinement of a model defined from the logically distributed system perspective. The following refinement operations are important for performing this refinement:

- introduction of internal actions and action points, followed by decomposition of system parts and distribution of the introduced actions and action points over the resulting entities;

- remote interface refinement.

The first group of refinement operations is useful for decomposing single logical system parts into multiple physically distributed system parts.

Remote interface refinement is the replacement of one or more abstract interactions of a group of entities by a set of related concrete interactions, such that each concrete interaction is shared by one or more of the original entities and one or more new entities that are introduced during this refinement. In the entity domain one or more interaction points are thus replaced by a structure of one or more entities with interaction points.

Remote interface refinement is a form of interface refinement. *Interface refinement* is the replacement of one or more abstract interactions by a set of related concrete interactions. Interface refinement can be described in terms of the basic refinement operations of section 6.2.2 as follows.

1. Abstract from the difference between the entities involved in the concerned interactions, i.e., perform the inverse of action (point) distribution and entity decomposition for each of the concerned interactions.
2. Decompose one or more of the actions resulting from step 1 and, possibly, their action points.
3. Decompose the entity resulting from step 1 and distribute each of the actions resulting from step 2 and their action points over them.

Figure 6.11 illustrates this refinement operation.

Figure 6.12 illustrates the transition from the logically distributed system perspective to the physically distributed system perspective.

Use in other development methods. Many development methods support an abstraction level at which a system is to be represented as a composition of its lowest-level parts. Examples are the “implementation model” levels of Jackson [1983], Essink [1986], Yourdon [1989], Ferreira Pires [1992], and Kremer [1995], the “technical design” level of MacDonald [1986], the “technical details model” level of Aue & Breuer[1984], and the “job line of visibility chart” level of [IBM, 1995].

6.4.5 Local interface refined system perspective

Objective. At this abstraction level the concrete interactions between system parts and between the system parts and the system environment are defined. Concrete interactions are interactions that can directly be mapped onto implementation constructs. The definition of these concrete interactions is required as the basis for the implementation of the system parts, so that these parts can interact.

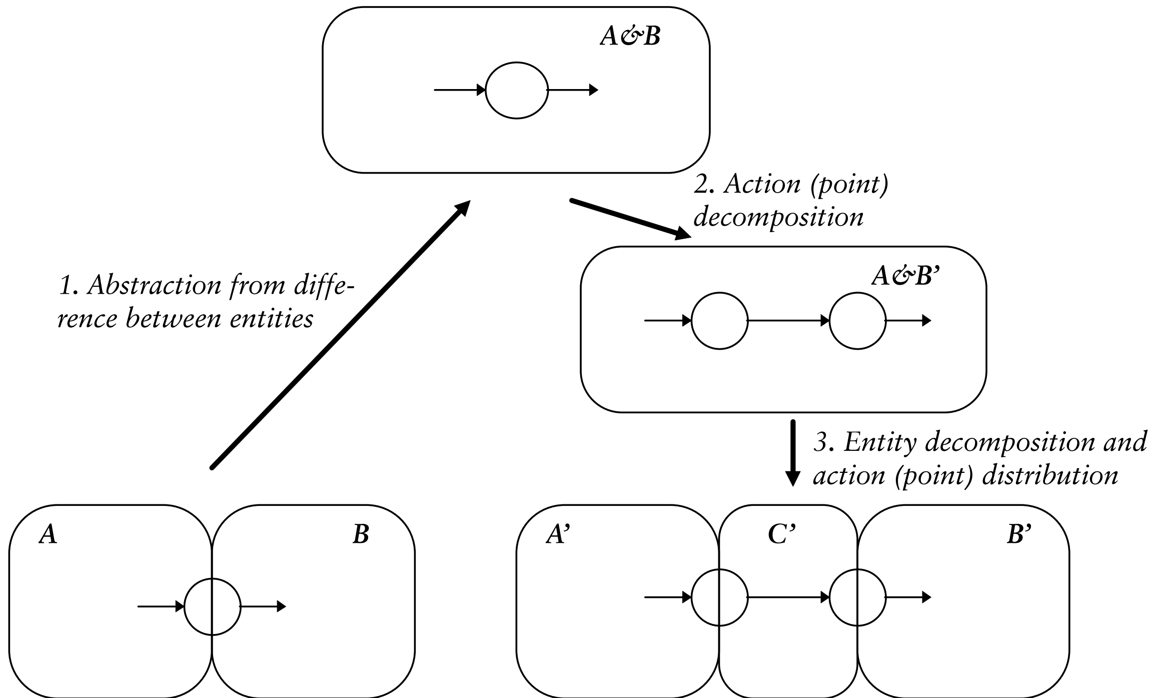


Figure 6.11 Example of remote interface refinement.

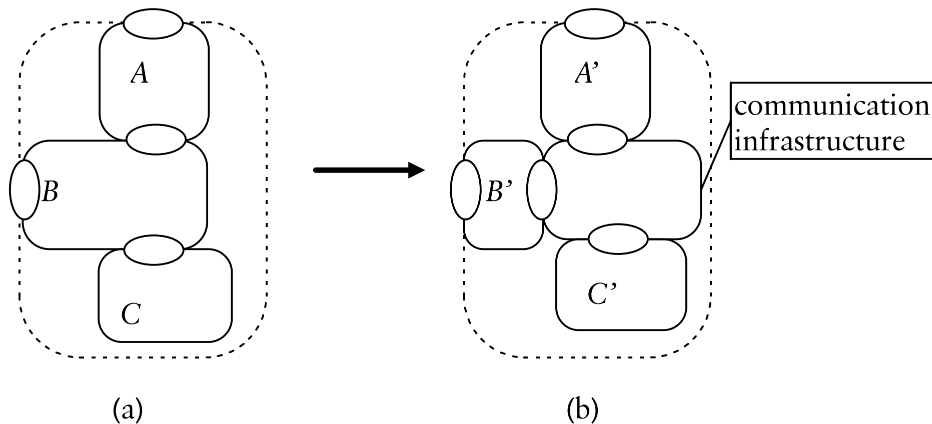


Figure 6.12 From (a) logically distributed system perspective to (b) physically distributed system perspective.

Structure. A model at this abstraction level represents the system as a composition of interacting parts, in which each of the parts is represented from the integrated perspective. The parts interact by means of concrete interactions.

Relation with next higher level. A model at this abstraction level is a refinement of a model defined from the physically distributed system perspective. Local interface refinement is required to make this refinement. *Local interface refinement* is the replacement of one or more abstract interactions of a number of entities by a set of related concrete interactions of the original entities or parts thereof.

Figure 6.13 shows two alternatives for local interface refinement. The first alternative (Figure 6.13a) is the replacement of an abstract interaction of a group of entities by multiple related concrete interactions of the original entities. An example is the replacement of the interaction “sales of goods” of an organisation and a client by the related interactions “sales of good 1” and “sales of good 2” of the same organisation and the same client.

The second alternative for local interface refinement (Figure 6.13b) is the replacement of an interaction of abstract entities by an interaction of concrete entities, where some of the concrete entities are parts of the abstract entities. An example is the replacement of the interaction “sales of goods” of an organisation and a client by the interaction “sales of goods” of the sales department of the organisation, the accounting department of the organisation, and the client.

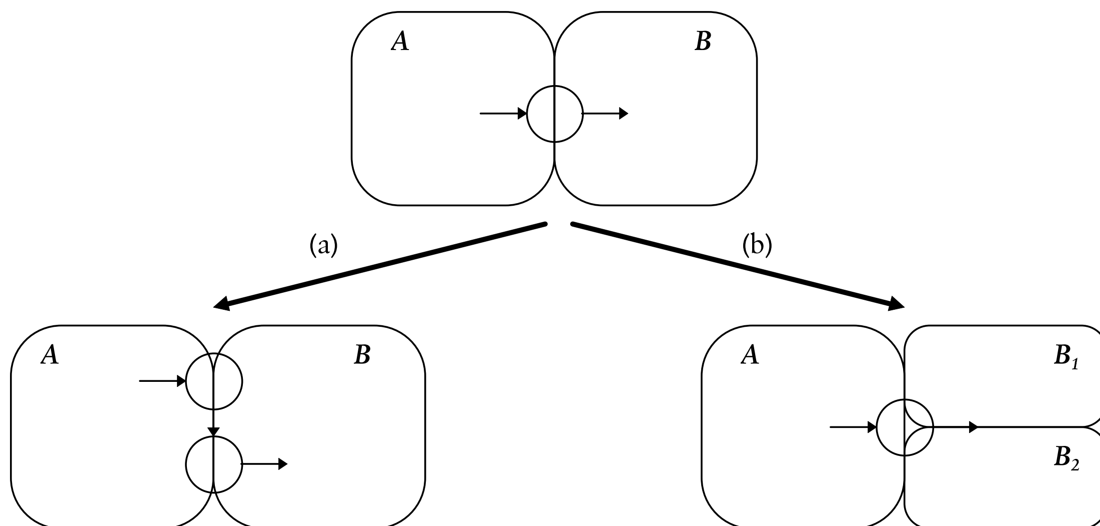


Figure 6.13 Two types of local interface refinement.

Another alternative for local interface refinement is the replacement of abstract values established in interactions by concrete values. An example is the replacement of the value “name and address” of type “personal info” by the values “first name” of type “string[20]”, “last name” of type “string[40]” and “address” of type “string[100]” in an interaction.

Use in other development methods. Local interface refinement may be required as a consequence of other refinement operations. Therefore, some local interface refinement is usually carried out throughout the development process. For example, section 6.2.2 showed that entity decomposition may require local interface refinement.

However, local interface refinement is not always required as a consequence of other refinement operations. Examples are the decomposition of an interaction into a sequence of interactions due to limitations of the interaction point (e.g. the

decomposition of "delivery of all goods" into "delivery of good 1", "delivery of good 2", etc. due to a limited capacity of the delivery point) and the replacement of abstract values established in interactions by concrete values (e.g. the development of a form on which customers specify their orders).

In order to prevent developers getting overwhelmed by unnecessarily detail early in a development process, such local interface refinement is best carried out as late as possible in a development process. Vissers & Scollo [1988] recommend that local interface refinement is performed in the implementation phase, but before the different system parts are implemented by different (groups of) designers, to preserve connectability of the parts. Also Ferreira Pires [1994] recommends that (local) interface refinement is performed at a low abstraction level, but before the system is finally implemented.

6.5 Application example

This section applies the abstraction levels of section 6.4 to the development of an example business process: the sales of an airline ticket by a travel agency. The purpose of this example is to illustrate the distinguished abstraction levels. Only the aspects of the business process that are relevant for this purpose are modelled.

6.5.1 System embedded in its environment

We consider the sales process of an airline ticket for a single customer. At the highest abstraction level three actions are distinguished in this process:

- *inform*, in which a number of ticket options are established;
- *book & pay*, in which an airline ticket is booked and paid for;
- *receive ticket*, in which the booked ticket is received.

Figure 6.14 depicts these actions and their relations.

6.5.2 Integrated system perspective

At this abstraction level the distinguished actions are distributed over the travel agency and the customer as follows:

- in the interaction *inform* the travel agency is responsible for supplying a number of ticket options that match the client's requirements;
- in the interaction *book & pay* the client is responsible for selecting a ticket to be booked from the options and for ensuring that he or she does not pay more for the ticket than the required price, whereas the travel agency should ensure that the payment amount is exactly the required price;

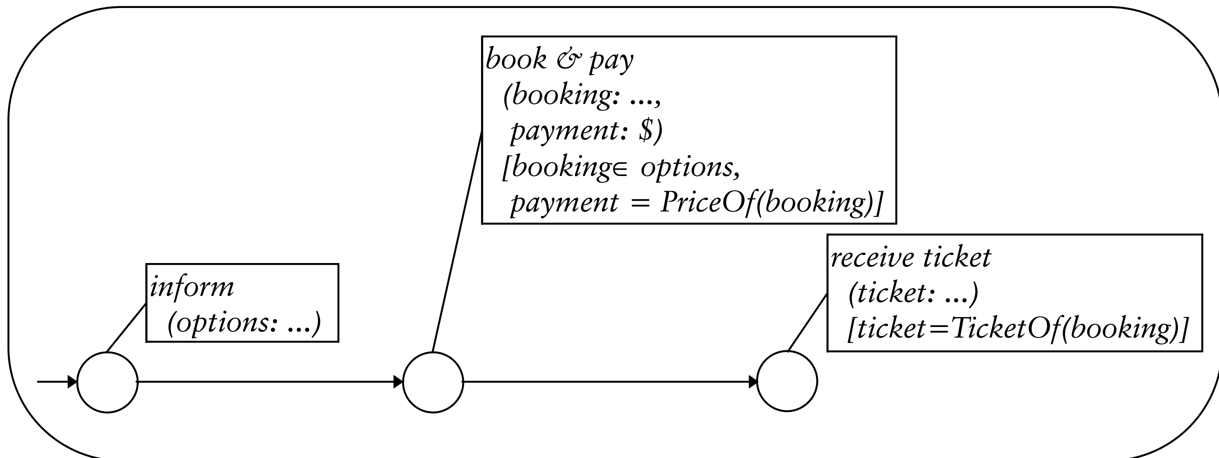


Figure 6.14 Travel agency embedded in its environment, viewed as a single entity.

- in the interaction *receive ticket* both parties are responsible for checking that the received ticket is the booked ticket.

Figure 6.15 represents this distribution of actions, as well as the distribution of relations over entities. The responsibility for the enabling relation between *inform* and *book & pay* has been wholly assigned to the client, to express that the client does not book without having been informed, whereas the travel agency is always prepared to book.

6.5.3 Logically distributed system perspective

At this abstraction level the travel agency is decomposed into logical parts. We distinguish:

- the travel shop at which the interactions with the client take place;
- an airline reservation system that is used to reserve tickets (*make reservation*) and to create tickets (*make ticket*);
- the head office at which, after a notice from the travel shop (*notify reservation*), the ticket is made and then transferred to the travel shop (*transfer ticket*).

Figure 6.16 represents the resulting system. Since the behaviour of the client remains the same as in Figure 6.15, we refrain from representing this behaviour in Figure 6.16 and Figure 6.17. For the sake of simplicity, these figures do not show the values established in interactions either.

The logical decomposition could go further, by distinguishing different entities in the travel shop and the head office, and the interactions of these entities.

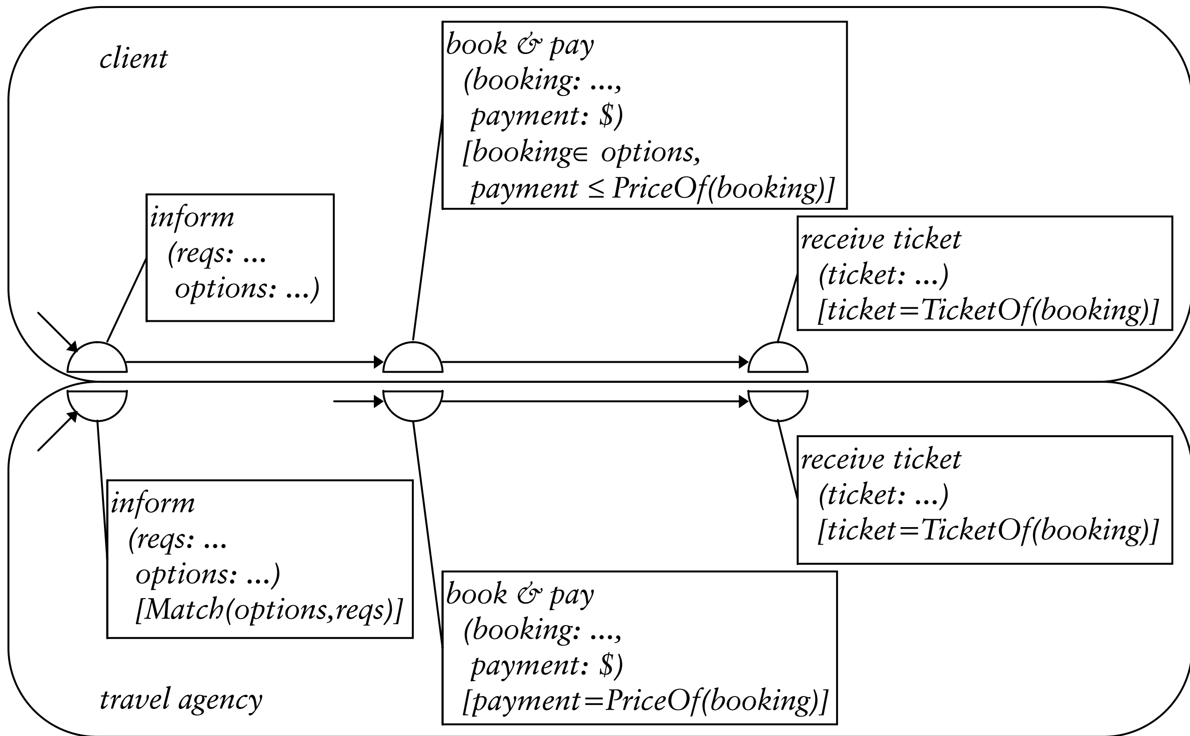


Figure 6.15 Travel agency and client from integrated system perspective.

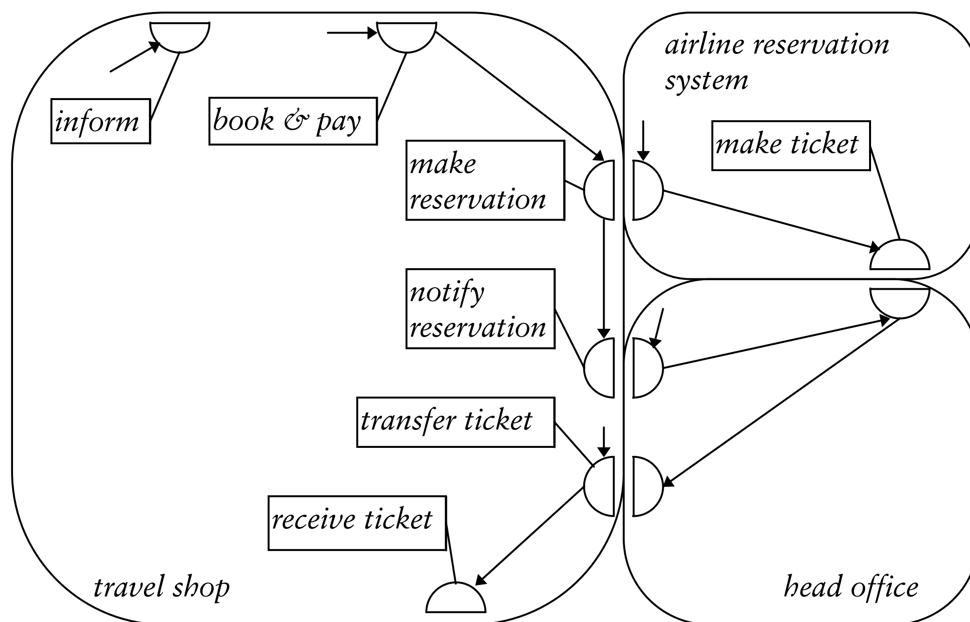


Figure 6.16 Travel agency from logically distributed system perspective.

6.5.4 Physically distributed system perspective

At this abstraction level the physical distribution of activities is taken into account. In our example most of the parts distinguished from the logically distributed perspective are in fact physically distributed. The following systems are identified to enable their interaction:

- an electronic communication system that allows the exchange of electronic messages between the travel shop and the airline reservation system, and between the travel shop and the head office.
- a courier service that supports the transfer of tickets from the head office to the travel shop;

Some of the interactions of the parts distinguished from the logically distributed perspective have to be decomposed at this abstraction level. For example, the interaction *make reservation* between the travel shop and the airline reservation system is decomposed into four related interactions, in which the electronic communication system participates: *reservation request*, *reservation indication, reservation response,* and *reservation confirm*.

Figure 6.17 represents the travel agency from the physically distributed perspective.

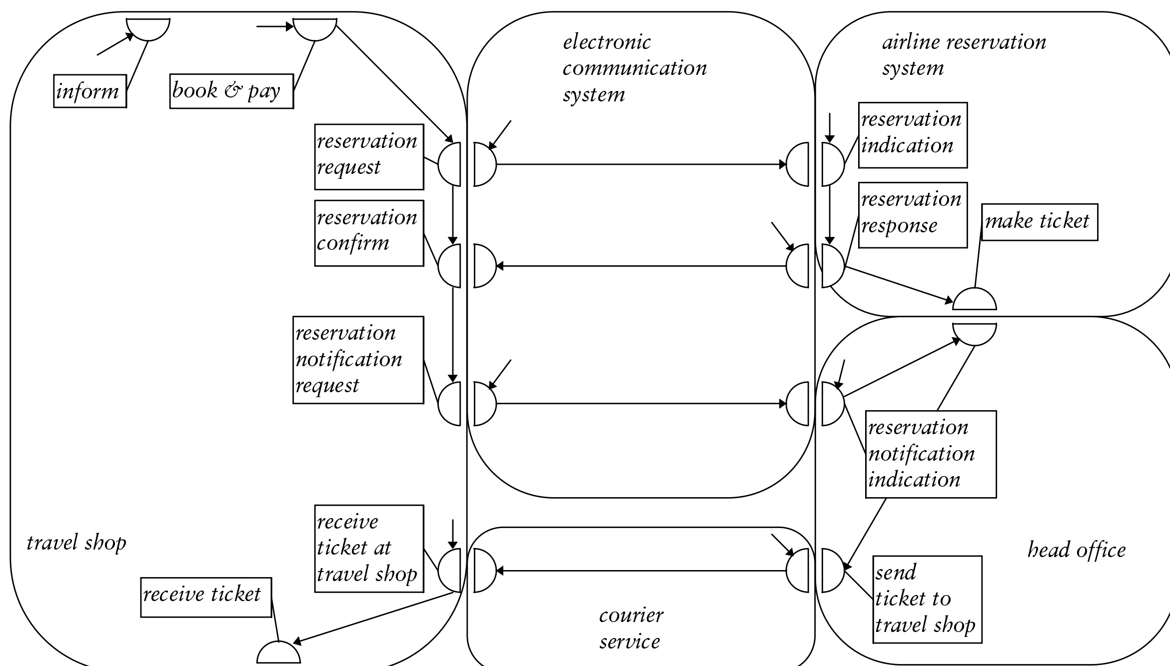


Figure 6.17 Travel agency from physically distributed perspective.

6.5.5 Local interface refined system perspective

At this level the abstract interactions of the physically distributed system perspective are replaced by concrete interactions. We give some examples of values established in interactions that have to be defined:

- the booking forms and reservation forms;
- the airline tickets and the rules according to which they should be filled out;
- the concrete values of the interactions of the travel shop, the head office, and the airline reservation system with the electronic communication system.

We do not define the concrete interactions here to keep the example brief.

6.6 Conclusions

Abstraction levels are system perspectives, such that each model of a system at a particular abstraction level contains either more or less detail than a model of the system at another abstraction level. In contrast to so-called viewpoints, the relations between abstraction levels are well-defined, and the construction of a complete model of a system is thereby eased.

The following generic abstraction levels were distinguished. From the integrated system perspective, a system is represented as an integrated whole: neither internal actions, nor internal parts of the system are represented. From the interaction system perspective of the parts, the related internal interactions of the system parts are represented; the parts are not represented. This allows one to consider the various ways in which these internal interactions can be distributed over the parts. From the distributed system perspective, a system is represented as a collection of interacting parts.

The following specific abstraction levels were distinguished. From the perspective of the system embedded in its environment, the system and its environment are represented without a distinction between the two. The integrated system perspective was defined above. From the logically distributed system perspective, a system is represented as a collection of interacting parts, where one abstracts from the physical distribution of these parts. From the physically distributed system perspective, a system is represented as a collection of physically distributed parts. From the local interface refined system perspective, the concrete interactions of parts are additionally represented.

7

Development strategies

7.1 Introduction

7.1.1 Motivation

This chapter discusses the selection and construction of development strategies. A *development strategy* prescribes the order in which development steps are carried out in a development process. A development step is an activity in which one or more development decisions are taken. Examples of development strategies are the waterfall model and rapid prototyping.

Synonyms of the term development strategy are “software process model” [Boehm, 1988] and “software development life cycle model” [Davis, Bersoff & Comer, 1988]. We prefer the term development strategy, because the first synonym is also used to refer to the dynamic aspect of a behaviour model, whereas the second synonym is also used to refer to the waterfall model.

Boehm [1988] states, based on his experience, that the choice for an appropriate development strategy is an important factor in the success of a development process. Even though developers are usually aware of the importance of a development strategy in a development process, the selection of a development strategy is not always carried out on the basis of an analysis of the development objectives the development strategy should support. Even when developers try to do so, two major problems with the use of current development strategies become apparent [Dawson & Cartwright, 1996]:

- It is often unclear for what development processes a particular development strategy is appropriate; authors often claim “universal” applicability of their development strategies, which, in practice, appears questionable.

- Often none of the current development strategies supports a particular development process completely. In such cases, a new development strategy, possibly a combination of current development strategies, should be used. It is, however, unclear how such a combined development strategy should be constructed.

The goal of this chapter is to show how these problems can be overcome. This chapter describes:

- criteria for the evaluation of development strategies;
- how a suitable basic development strategy for a development process can be selected on the basis of these criteria;
- how composite development strategies can be constructed from the basic development strategies.

7.1.2 Structure

Section 7.2 briefly describes some basic development strategies.

Section 7.3 describes the structure of these development strategies.

Section 7.4 introduces criteria for the evaluation of development strategies. These criteria are development objectives (such as the minimisation of re-work) that development strategies may support. The section then evaluates the discussed basic strategies by identifying how well these strategies support the distinguished development objectives.

Section 7.5 shows how the results of this evaluation can be used to select a development strategy that is appropriate for a particular development process. It shows that this selection should take place on the basis of the need for the support of particular development objectives in a development process.

Section 7.6 describes the construction of composite development strategies from basic development strategies.

Section 7.7 concludes with a discussion of suitability of the identified development strategies in some specific types of development processes.

Part of the work described in this chapter is based on [De Weger & Franken, 1997].

7.2 Some basic development strategies

This section describes some current development strategies, both to give the reader insight in these strategies, and to give our interpretation of the strategies (which is necessary because the terminology used in this area is sometimes inconsistent). The development strategies described here are basic in the sense that they serve as building blocks for the composite development strategies described in section 7.6.

The development strategies described in this section are applicable to distributed systems development. The names we use for the development strategies are the names customarily used in software development (e.g. the waterfall model and evolutionary development), since we assume the reader is familiar with most of these names. This does, however, not imply that the development strategies are only applicable to software development. They are applicable to business processes as well. Ganzevoort [1985], for example, who discusses strategies for organisation development, calls the waterfall model "design" and the evolutionary development strategy "development".

7.2.1 Waterfall model

The *waterfall model*, first described in [Royce, 1970], is based on the following guidelines:

1. design a system at successively lower abstraction levels;
2. complete the design at one abstraction level before proceeding to a lower level.

Because developers may make invalid designs that need correction, developers are allowed to correct designs at higher abstraction levels if they establish the invalidity of a design when working on a design at a lower abstraction level.

7.2.2 Evolutionary development strategy

The *evolutionary development strategy* [McCracken & Jackson, 1982; Hirsch, 1985] is based on the following guidelines:

1. elicit some requirements of the system and implement these;
2. elicit more requirements and add their implementation to the partially implemented system;
3. repeat step 2 until no more requirements have to be elicited and the system has been implemented.

The evolutionary development strategy should not be confused with the incremental development strategy, which is discussed in section 7.6.

7.2.3 Code-and-fix strategy

Boehm [1988] describes the *code-and-fix strategy* as the first development strategy applied to information systems development. In popular jargon this strategy is known as "hacking". It is based on the following guidelines:

1. make an implementation of the system (without creating designs at higher abstraction levels);
2. adapt the implementation to fix "bugs" and to make it better adhere to the requirements.

7.2.4 Rapid prototyping

Rapid prototyping [Luqi, 1989] is based on the following guidelines:

1. elicit the requirements regarding essential functions of the system;
2. rapidly create an implementation of the system (a "prototype") that satisfies these requirements of the system;
3. have users experiment with the system, elicit other requirements, and adapt the prototype until it meets all requirements.

The rapid implementation of a large system is usually hard to accomplish. Two approaches for overcoming this problem are:

- The use of tools that automate a part of the implementation process, such as fourth generation tools. This approach is sometimes used in software development. Its applicability is, however, restricted to the types of software for which such tools exist. An extreme form of rapid prototyping of software is the transform model [Balzer, 1985] or automated software synthesis [Partsch & Steinbruggen, 1983], in which the prototype is automatically generated on the basis of a formal specification of the requirements of the system.
- The implementation of the system on a small scale, for example for a small group of clients or users. This approach is often used in business process development [Davenport, 1993].

Rapid prototyping is usually followed by an implementation of the system according to the waterfall model, since it is hard to meet all requirements for a system with rapid prototyping only. Typical requirements that are hard to meet are scalability, reliability, maintainability, and performance requirements.

7.3 Structure of development strategies

7.3.1 Separation of concerns

A development process involves repeated separation of development concerns. At each point in a development process decisions are taken in development steps with regard to particular development concerns. If such a development decision does not result in a design of a component that can be implemented directly, further separation of concerns is required. The development decision thus results in new development steps in which new development decisions are taken. Therefore, the representation of a development process that shows the development decisions resulting from repeated separation of development concerns resembles a tree structure. Figure 7.1 shows a development process in terms of a tree of development decisions.

Since most development decisions originate in a development process as consequences of previously taken development decisions, different development processes

comprise different development decisions. It is therefore impossible for development strategies, which are applicable to many development processes, to fully support separation of concerns at the level of individual development decisions.

Instead, development strategies support the decomposition of development processes into *groups* of development decisions. Two forms of separation of concerns in development processes, which were identified in chapter 2, are supported by development strategies:

- horizontal separation of concerns, which leads to distinct abstraction levels;
- vertical separation of concerns, which leads to distinct aspects of a system; these aspects are also called increments.

Figure 7.1 shows how groups of development decisions relate to the development concerns of one abstraction level and one increment. An arrow represents that a development decision is to be taken as the consequence of another development decision.

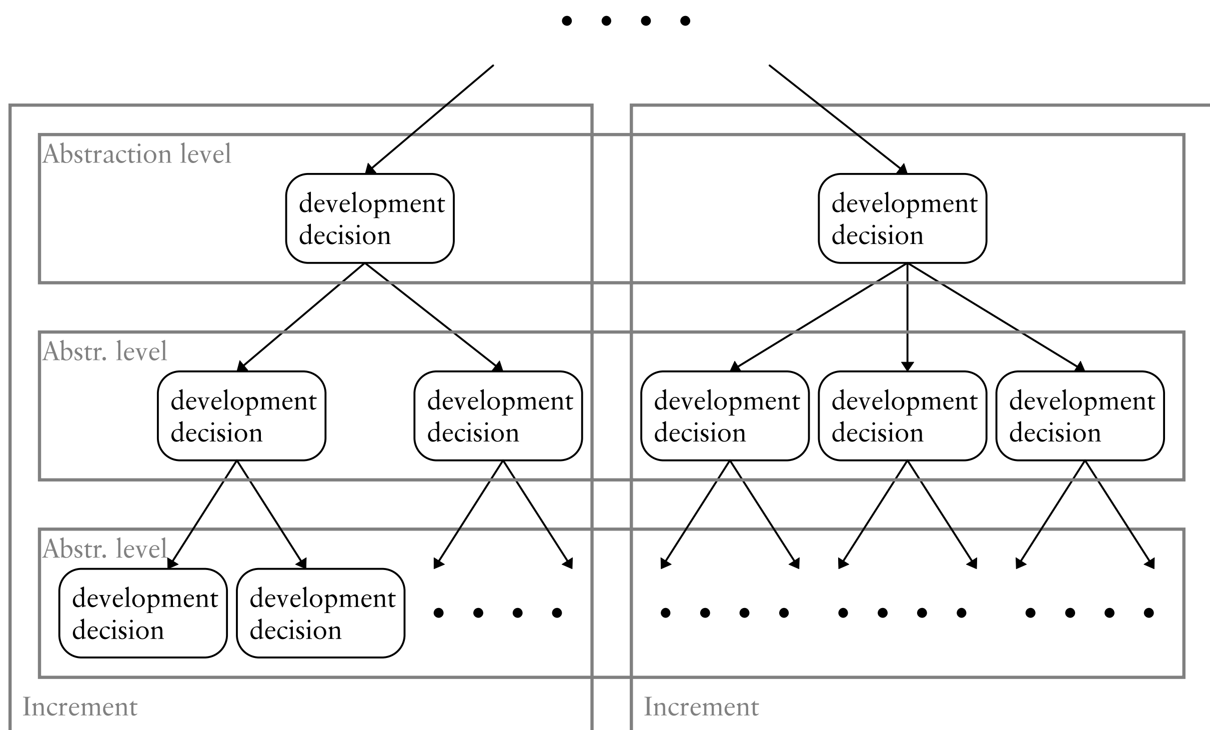


Figure 7.1 Development decisions resulting from repeated separation of concerns, and their relation to abstraction levels and increments.

Separation of concerns is supported by the identified basic development strategies as follows:

- the waterfall model only supports horizontal separation of concerns;

- the evolutionary development strategy only supports vertical separation of concerns;
- rapid prototyping poorly supports horizontal separation of concerns, since it only distinguishes between requirements and their implementation (one may consider that rapid prototyping also supports vertical separation of concerns, since it prescribes multiple cycles of requirements capturing and implementation; this is, however, not the case, since in the different cycles one may reconsider the entire system, and new cycles do not necessarily result in new increments);
- the code-and-fix strategy does not support any separation of concerns.

7.3.2 Order of development steps

Each basic development strategy only prescribes the order of development decisions *between* distinguished groups of development decisions, i.e. between the development steps in each of which a group of development decisions is made. A basic development strategy does not prescribe the order of development decisions *within* the distinguished groups of development decisions.

When we abstract from the reconsideration of development decisions allowed by the development strategies (which may in practice result in quite chaotic orders of development steps) the following orders of development steps are prescribed by the identified development strategies:

- the waterfall model prescribes a top-down order: development steps relating to one abstraction level should be performed before development steps relating to a lower abstraction level;
- the evolutionary model prescribes a breadth-first order: development steps relating to one increment should be performed before development steps relating to the next increment;
- rapid prototyping prescribes a top-down order: the essential user requirements are elicited before they are implemented;
- the code-and-fix strategy prescribes no order.

Figure 7.2 shows the structure of each basic development strategy. The grey boxes represent development decisions resulting from possible horizontal and vertical separation of concerns. The black boxes represent the development steps in which these development decisions are taken. Arrows between the development steps indicate the order of development steps. The figure abstracts from reconsideration of development decisions.

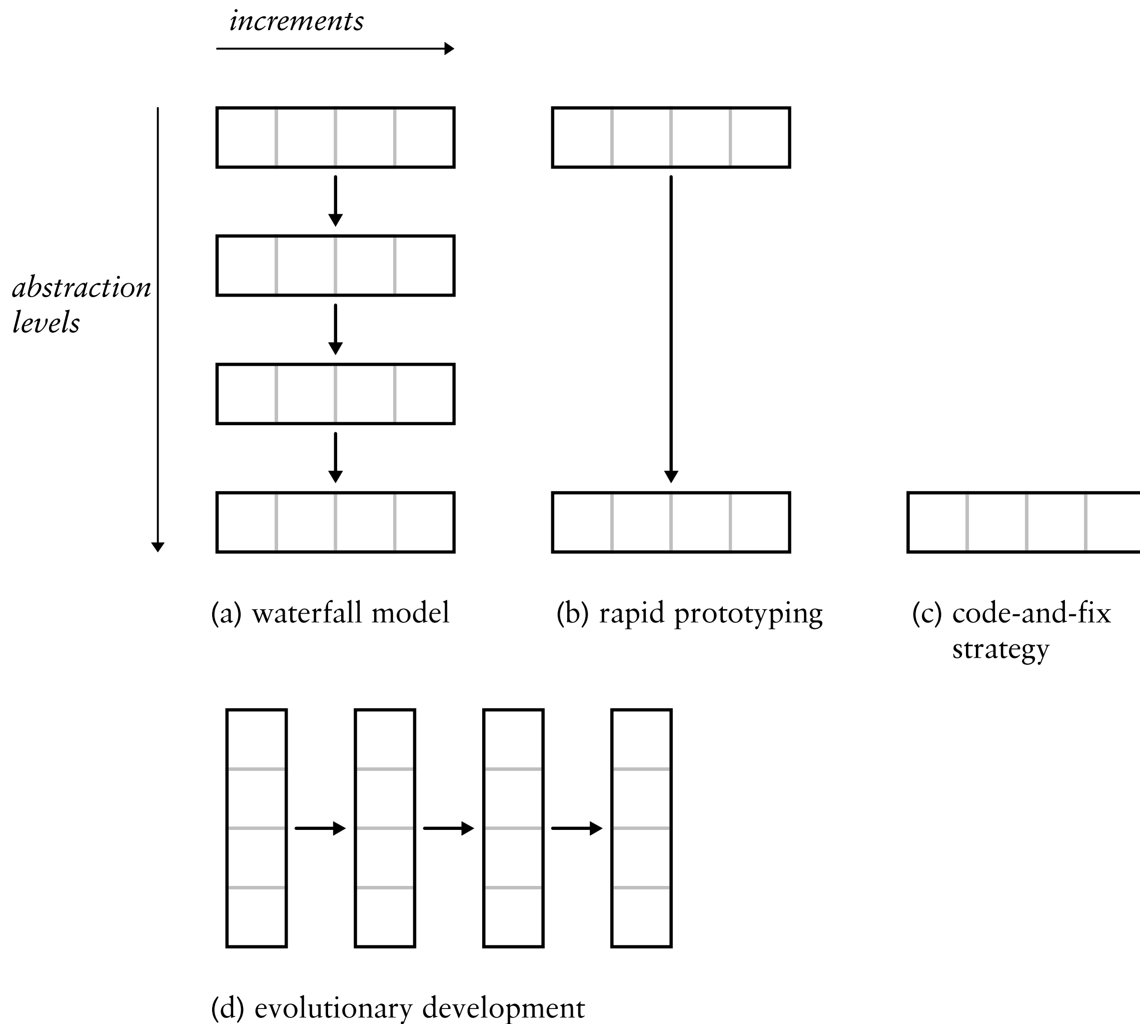


Figure 7.2 Structures of identified development strategies. Arrows between development steps represent order of development steps.

7.4 Evaluation of basic development strategies

7.4.1 Criteria

The objective of a development process is to obtain an implementation of a system such that the system and the development process satisfy certain conditions. These conditions usually concern [Wohlin, 1994; Van Sinderen, 1995]:

- The *requirements* the system should satisfy.
- The *costs* of developing, using, maintaining the system. These costs should, for example, be minimised or not exceed a certain limit.

- The *time to delivery* of the development process, i.e. the time between the start of a development process and the delivery of the product. This time should, for example, be minimised or not exceed a certain limit.

It therefore seems logical to evaluate the basic development strategies on the basis of their support for development objectives that concern these aspects. An impediment for doing so is, however, that development strategies concern only one aspect of development processes: the order of development steps. This order has little impact on requirements satisfaction. For example, the most important factor that influences the requirements satisfaction is the capability of the system developers [Boehm, 1981], on which a development strategy has no influence.

There are, however, some factors that influence costs and time to delivery that can be influenced by the use of development strategies.

- *Amount of re-work*. Re-work, extra activity due to reconsideration of development decisions, causes development costs and/or time to delivery to increase. By minimising re-work, development costs and/or time to delivery can be decreased.
- *Resource use*. If development steps use more resources than required for carrying them out properly, development costs and/or time to delivery will increase [Boehm, 1981]. Optimal resource use by each development step can decrease development costs and/or time to delivery.
- *Conformance to quality criteria*. When a system conforms to the quality criteria discussed in chapter 2, it is usually cheaper to develop, maintain, and use than when it does not. Moreover, conformance to quality criteria eases the re-use of parts of the system in other development processes. Re-use of parts usually decreases the costs and/or time to delivery of a development process.

Below we therefore evaluate each basic development strategy on the basis of 1) their support for minimisation of re-work, 2) their support for optimal resource use, and 3) their support for conformance to quality criteria. We also elaborate on the need for the support of these development objectives.

7.4.2 Support for minimisation of re-work

Re-work

Some of the development decisions made in a development process may be found incorrect or inappropriate and should be reconsidered. Unfortunately, this invalidity often becomes clear long after these decisions have been taken. As a consequence, later development decisions based on a previous invalid decision may also have to be reconsidered.

Figure 7.3 shows the effect of reconsideration of development decisions on the tree of development decisions. Assume that the development decisions shown in black have been carried out. Suppose that developers then discover that development deci-

sion 1a is invalid. This development decision, and possibly development decisions 1ai and 1aii, then have to be reconsidered. This requires re-work to make new development decisions, which are called 1a', 1ai', and 1aii' in the figure.

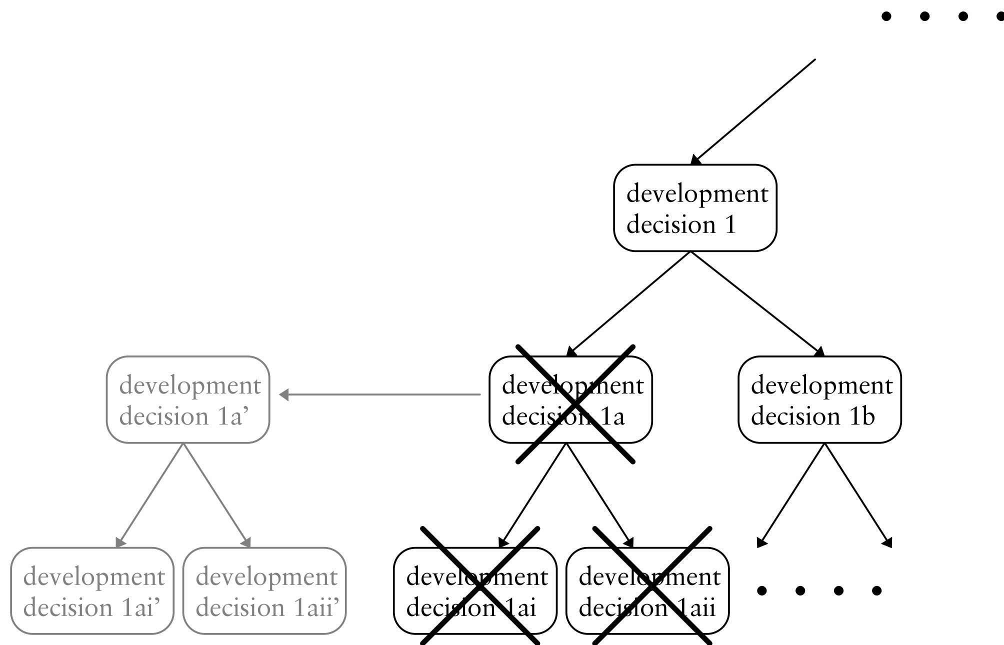


Figure 7.3 Re-work due to reconsideration of development decisions. Legend as for Figure 7.1. Crossed development decisions are reconsidered. Grey development decisions represent re-work.

Re-work may be necessary to give developers more insight in the system they develop, or to allow developers to improve the structure of the system. However, the extra work due to the reconsideration of development decisions increases the costs of the development process and/or the time to delivery. In practice, the extra work due to the reconsideration of development decisions causes a large part of the total costs of a development process. Boehm [1987] estimates that on average the costs due to this re-work make up 30 to 50 percent of the total costs of a development process.

Causes of reconsideration of development decisions

Two important causes of reconsideration of development decisions are the following:

- the specified requirements of the system do not reflect what users really need;
- a particular design cannot be implemented with the available implementation components.

We elaborate on these causes below.

- *The specified requirements of the system do not reflect what users really need.* In order to explain the reasons for this difference, three different types of require-

ments can be identified (based on [Roman, 1985] and [Dawson & Cartwright, 1996]):

- The *specified requirements* are the requirements obtained and specified by the developers after eliciting requirements from users at the beginning of the development process[1]. They are thus "how the developers interpret what the users say they need".
- The *initially conceived requirements* are the users' perception of the requirements at the beginning of the development process. They are thus "what the users initially think they need".
- The *actual requirements* are the requirements as they should have been established to satisfy the users. They are thus "what the users really need". The actual requirements often conform to the users' perception of the requirements when they have used the system after its development.

Differences between the actual and specified requirements may have the following causes, or a combination of these:

- The initially conceived requirements differ from the actual requirements. In this case the users do not know what they really need. Even if they think they know what they need, their conception of the requirements will change over time. Specified requirements that change over time for this reason are sometimes called "unstable" or "volatile" requirements [Boehm, 1987]. Unstable requirements may be caused by, e.g., a lack of experience of the users with the type of system that is being developed.
- The initially conceived requirements differ from the specified requirements. In this case the users may know what they want, but their intention is translated incorrectly into a high-level specification of the system. Specified requirements that change over time for this reason are sometimes called "uncertain" requirements. Uncertain requirements may be caused by, e.g., miscommunication between users and developers.

Figure 7.4 illustrates these reasons for differences between the actual requirements and the specified requirements.

- *A design cannot be implemented with the available implementation components.* Each development decision rules out the combinations of components that do not conform to the resulting design. It may therefore occur that after some development decisions no combination of available components exists that implements the design. This may e.g. occur when the system developers have little

1. Here we use the term "user" as the generic term for a person who determines the requirements. Users thus include, e.g., principals or clients.

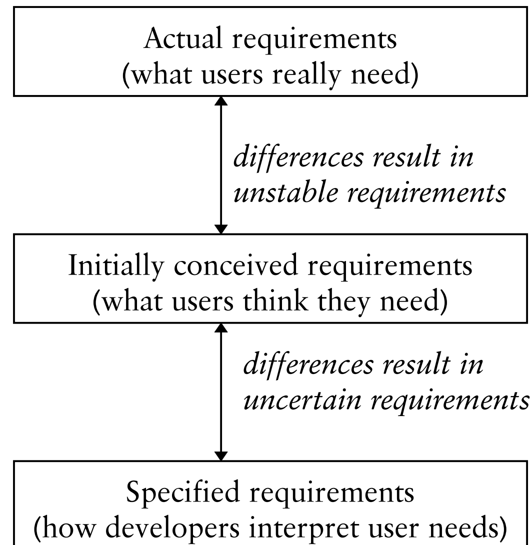


Figure 7.4 Reasons for differences between actual requirements and specified requirements

knowledge of implementation components (little "bottom-up knowledge"). An example of this cause of reconsideration of development decisions is the unavailability of sufficient properly educated employees to carry out a business process that has been designed.

It may therefore occur that after some development decisions no combination of available components exists that implements the design. This may e.g. occur when the system developers have little knowledge of implementation components (little "bottom-up knowledge"). An example of this cause of reconsideration of development decisions is the unavailability of sufficient properly educated employees to carry out a business process that has been designed.

Importance of early requirements validation and implementability validation

In order to minimise work due to the reconsideration of development decisions it is important to assess as early as possible the validity of development decisions that are likely to be invalid. By doing so, the amount of development decisions that are based on potentially invalid development decisions (and that may therefore have to be reconsidered as well) is minimised.

Empirical evidence confirms the importance of reconsidering invalid development decisions as early as possible. Several sources (e.g. [Boehm, 1981; Bunyard & Coward, 1982]) estimate that the costs of reconsidering invalid development decisions in the final stages of a development process are on average about 100 times larger than the costs of immediately reconsidering invalid development decisions.

Assessing the validity of development decisions with respect to the above two causes of re-work results in two types of validation:

- *Requirements validation* is the process of assessing whether the specified requirements for the system are equal to the actual requirements. Two techniques for requirements validation are [Davis, 1982]:
  - supplying the users with a prototype of the system, having them to experiment with the prototype, and asking them for feedback;
  - reviewing the specified requirements with the users (e.g. by analysing or simulating these specifications), pointing out inconsistencies in and consequences of their requirements, filling in gaps, and asking them for feedback.
- *Implementability validation* is the process of assessing whether the part of the design resulting from a particular development decision can be implemented with the available implementation components. The two generally applicable techniques for implementability validation of a part of the design are [Pressman, 1992]:
  - attempting to implement the part of the design;
  - assessing the part of the design attentively, for example by reviewing it with a team of developers who have knowledge of the implementation components.

Evaluation of development strategies

Premise 1a: early requirements validation

The early availability of a prototype of a system better supports early requirements validation than the early availability of the complete set of specified requirements or of a part of the system.

If no prototype is available early in a development process, the early availability of the complete set of specified requirements or of a part of the system better supports early requirements validation than early availability of neither of these.

Motivation. The motivation for the first part of premise *1a* is that early availability of a prototype of the system allows one to have users experiment with the system early in the development process. By experimenting with the system, users directly compare the specified requirements (which the prototype should satisfy) to the actual requirements.

Early availability of the complete set of specified requirements only allows developers to review the specified requirements with the users early in the development process. By reviewing the specified requirements with the users, these specified requirements are only compared to the initially conceived requirements; differences between the initially conceived requirements and the actual requirements are not considered.

Early availability of a part of the system allows users (at best) to experiment with a part of the system early in the development process. Thereby only a part of the specified requirements can be compared to the actual requirements.

Additional motivation for the first part of premise *1a* comes from Davis [1982]. By having the users experiment with the system, the "real thing", one can largely overcome a range of problems regarding human capabilities for information processing that may occur when reviewing the specified requirements with the users. These problems, which apply both to users and system developers, include the tendency to over-simplify, the tendency to focus on a limited number of aspects of the system, and an inability for abstract and structured thinking.

There are, however, cases in which the first part of premise *1a* does not hold. These are cases in which it is hard to observe whether a system satisfies certain requirements. An example of such requirements are stringent reliability requirements. To observe whether a system conforms to these requirements, immense testing efforts may be required. In such cases it is better to review the specified requirements with the users. (Appropriate development strategies for these cases are discussed in section 7.7.) Premise *1a* also only holds if one properly uses a prototype for evaluation, and does not, e.g., concentrate merely on the bells and whistles of a user interface. Finally, a prototype of a system cannot always be quickly constructed (see section 7.2.4).

The motivation for the second part of premise *1a* is that by the early availability of the complete set of specified requirements or of a part of the system, one can at least apply one of the above techniques for requirements validation. If neither the complete set of specified requirements, nor a part of the system, are available early in the development process, none of these requirements validation techniques can be applied.

Premise *1a* and the description of the structure of development strategies of section 7.3 lead to the following conclusion.

Conclusion 1a: support for early requirements validation

The rapid prototyping strategy best supports early requirements validation, because its application results in the early availability of a prototype of the system. The waterfall model and the evolutionary strategy better support early requirements validation than the code-and-fix strategy, because the application of the former strategies result in the early availability of the complete set of specified requirements, viz. a part of the system, whereas the application of the latter strategy results in neither of these.

Example/empirical support. Boehm et al. [1984] compared seven development processes in which the determination of user requirements was the most time-consuming factor. Each project was carried out by means of rapid prototyping and by means of a waterfall strategy. On average, the rapid prototyping projects required about 40 percent fewer man-hours than the projects carried out according to the waterfall strategy.

Conclusion *1a* does not tell whether a waterfall strategy or an evolutionary strategy is more successful in early requirements validation, because premise *1a* does not compare requirements validation on the basis of the complete set of specified requirements to requirements validation on the basis intermediate versions of the system that implement only part of the requirements.

Premise 1b: early requirements validation in case of unstable requirements
In case of unstable requirements, early requirements validation is better supported when users can experiment with intermediate versions of the system that implement important requirements than when users cannot experiment with such intermediate versions.

Motivation. If users can experiment with intermediate versions of the system that implement important requirements, the differences between the initially conceived requirements and the actual requirements can be considered and found relatively early in a development process. These are the differences that cause the unstability of the requirements. Moreover, the experimentation with a part of the system may lead to users make their conception of the requirements for the remainder of the system conform better to the actual requirements.

If users cannot experiment with intermediate versions of the system, requirements validation can only take place by reviewing the specified requirements with the users. In such reviews the differences that cause the unstability of the requirements (i.e. the differences between the initially conceived requirements and the actual requirements) are hardly considered.

Conclusion 1b: support for early requirements validation in case of unstable requirements
The evolutionary strategy supports early requirements validation better than the waterfall model in case of unstable requirements, because the evolutionary strategy results can be used to deliver a part of the system that implements important requirements early in a development process, whereas the application of the waterfall model only results in the early availability of the specified requirements.

Example/empirical support. On the basis of his experience with the application of the spiral model (see section 7.6) Boehm [1988, p. 63] concludes that evolutionary development is "well matched to situations in which users say, 'I can't tell you what I want, but I'll know it when I see it'". Davis et al. [1988] compare the use of the waterfall model to evolutionary development and conclude that evolutionary development is significantly better in matching changing user requirements than the use of the waterfall model.

Premise 2: early implementability validation
A development strategy supports early implementability validation of a part of design best when the strategy allows one to attempt to implement this part as quickly as possible.

Motivation. If a development strategy does not allow one to attempt to implement a design quickly, the only option for implementability validation is an assessment of the design. Such an assessment never allows one to obtain certainty about the implementability of the design, because of possible errors in judgement.

If a development strategy allows one to try to implement a design quickly, and the attempt succeeds, one has certainty the design can be implemented. (When the attempt does not succeed, one still does not have certainty that implementation is impossible, because there may still be possible implementations of the design one did not consider.)

In many cases, attempting to implement a design thus gives more certainty about the implementability of a design than an assessment of the implementability of the design, and therefore results in fewer reconsiderations of development decisions.

Conclusion 2: support for early implementability validation
The code-and-fix strategy gives the best support for early implementability validation, since it allows developers to start implementing immediately.
The evolutionary strategy gives some support for early implementability validation, since the evolutionary strategy allows developers to start implementing early, but only after some development activity.
The waterfall strategy and the rapid prototyping strategy give the least support for early implementability validation, since they allow developers to start implementing only after all other development activities have been performed.

Example/empirical support. Wohlin [1994] shows that the relative time to delivery of a development process carried out according to the waterfall model, compared to it being carried out according to the incremental strategy, increases in proportion to the amount of implementation errors.

7.4.3 Support for optimal resource use

Importance of resource planning and control

In order to keep a development process as economical as possible, it is important that each development step uses exactly the amount of resources required to carry out the step. (By resources we mean the people, support tools, etc. required to carry out development processes; the sources of development costs.)

- If a development step is not allowed to use the required resources, its execution is delayed, or it cannot be completed at all, implying corresponding consequences for the entire development process.
- If a development step uses more resources than required, this may lead to the development process becoming more expensive than required, or, in case the devel-

opment process has a fixed budget, this may cause future development steps to get insufficient resources.

Apart from re-work due to the reconsideration of development decisions, which was discussed in section 7.4.2, there are the following causes of non-optimal resource use in development steps:

- Development steps initially get allocated insufficient or superfluous resources, and this is not noticed during the development process.

  If a development step gets allocated insufficient resources and it is not allowed to use more, the required amount of resources cannot be used for the development step.
  If a development steps gets allocated superfluous resources, in nearly all cases these resources will be used for the development step. (This is Parkinson's law [Parkinson, 1957] which says that "work expands to fill the available volume".)

- Development steps initially get allocated the right amount of resources, but they use more resources than allocated, because of a lack of control.

The objective of resource planning and control is to eliminate these two causes. *Resource planning* is the process of substantiated allocation of resources to activities (development steps or groups of these in our case). *Resource control* is the comparison during the development process of the actually used or required resources to the allocated resources and the taking of measures in case of differences. Possible measures for compensating these differences are the allocation of less viz. more resources to a development steps (re-planning), and intervention to prevent further resource squandering [Pressman, 1992].

The application of current techniques for resource planning and control (e.g. the scheduling and control aspects of the Constructive Cost Model COCOMO [Boehm, 1981] and the Program Evaluation and Review Technique PERT [Cori, 1985]) to an entire development process require that [Cori, 1985]:

- the development process is split up into activities to which resources can be allocated;

- it is possible to give proper estimates of the complexity of each of these activities early in the development process.

Evaluation of development strategies

Premise 3a. A development strategy better supports resource planning and control when it properly supports separation of concerns, than when it offers little or no support for separation of concerns.

Motivation. The application of resource planning and control requires that a development process has been split up into (groups of) development steps. Therefore,

separation of concerns in a development process is required for resource planning and control.

Conclusion 3a. The waterfall model and the evolutionary strategy better support resource planning and control than the rapid prototyping strategy and the code-and-fix strategy, because the former strategies support separation of concerns well, whereas the latter strategies offer little or no support for separation of concerns.

Premise 3b. In case of proper support for separation of concerns, the early availability of the complete set of specified requirements supports resource planning and control better than when these requirements are not available early in a development process.

Motivation. The application of resource planning and control to an entire development process requires that it is possible to give proper estimates of the complexity of each of the distinguished development steps early in the development process. If the complete requirements are absent, it is very difficult to give such proper estimates for all of the development steps.

If only a part of these requirements are present, it may be possible to plan and control the resources for the development steps concerned with the part of the system that implements these partial requirements. However, it remains very difficult to distribute the resources optimally over the entire development process, because it is very difficult to give proper estimates of the complexity of the other development steps.

Conclusion 3b. The waterfall model better supports resource planning and control than the evolutionary strategy, because the application of the waterfall model results in the early availability of the complete set of specified requirements, whereas the application of the evolutionary strategy does not result in this.

Example/empirical support. All current techniques for resource planning and control can only be applied to an entire development process after the user requirements have been elicited; planning and control of requirements capturing is hardly supported by these techniques [Cori, 1985].

7.4.4 Support for conformance to quality criteria

Importance of pursuit of quality criteria

To all of the actors involved in the development, maintenance, and use of a system, it is important that the system and its designs conform to the quality criteria identified in chapter 2 for, among others, the following reasons (based on [Van Sinderen, 1985]):

- The system developers, who conceive, transform, and implement the designs during the development process, require that:

  - the functions of the designs are general enough to be robust to changes that concern their implementation;
  - the designs are proper to their objectives, so that future development steps are not unnecessarily constrained.
- The system developers and maintainers, who interpret the design and adapt and extend the system during and after the development process, require that
  - the designs are easy to comprehend;
  - the designs and the system are open-ended, so that they can be extended without major re-development.
- The system users, who learn and utilise the system, require that:
  - the system is proper to their requirements;
  - the extensional aspects of the system are easy to comprehend;

Pursuit of quality criteria requires effort by the developers and may therefore seem to increase development costs. However, it appears that the costs of developing, maintaining, and using a system are less when the quality criteria are pursued, for the following reasons [Van Sinderen, 1985]:

- General designs that are proper to their objectives diminish the likelihood that development decisions have to be reconsidered, and thus diminish development costs, and give maximal freedom to implementers to minimise the costs of these implementations. Moreover, conformance to generality eases the re-use of parts of the system in other development processes. Re-use of parts usually decreases the costs of a development process.
- Designs that are easy to comprehend require relatively little time to be understood by developers and maintainers. Open-ended designs and systems require relatively little effort to be extended, so that maintenance costs are reduced.
- A system that is proper to the users' requirements properly supports the users in their activities and thus increases their productivity. A system that is easy to comprehend is generally easy to learn and to use, and therefore reduces training costs and increases user productivity.

Evaluation of development strategies

Premise 4a. A development strategy better supports the pursuit of the identified quality criteria when it properly supports separation of concerns, than when it offers little or no support for separation of concerns.

Motivation. Separation of concerns is necessary in order to obtain orthogonal and general designs. Because proper separation of concerns increases the insightfulness of designs, and because the pursuit of consistency and propriety require insight, separation of concerns also supports the pursuit of consistency and propri-

ety. Therefore, support for separation of concerns implies support for the pursuit of all of the identified quality criteria.

Conclusion 4a. The waterfall model and the evolutionary strategy support the pursuit of quality criteria better than the rapid prototyping strategy and the code-and-fix strategy, because the former strategies properly support separation of concerns, whereas the latter offers little or no support for separation of concerns.

Example/empirical support. Rapid prototyping and the code-and-fix strategy are often used for experimentation only (i.e., for requirements validation and implementability validation, respectively). The initial system resulting from the application of these strategies is usually badly structured. Therefore, the final system is often built according to the waterfall model [Brooks, 1995].

Premise 4b. Consideration of entire designs at every abstraction level supports the pursuit of static generality and orthogonality better than consideration of different parts of designs in different increments.

Motivation. Static generality demands that an aspect that is common to multiple parts of a system is described in a design only once. Orthogonality demands that independent aspects of a design are separated.

By considering an entire design at a particular abstraction level, it becomes possible to recognise which aspects of the system are common to multiple parts and which aspects are independent.

If different parts of each design are considered in different increments, the recognition of aspects that are common to multiple parts or that are independent of each other, can, in the early increments, only be based on an assessment of the aspects that are potentially present in future increments. (This implies, among others, a strict obedience to open-endedness.) However, a prediction of such future aspects is prone to assessment errors. It is therefore more difficult to base the recognition of the commonality and independence of aspects on an assessment of (partially) unknown aspects than on fully known aspects.

Conclusion 4b. The waterfall model supports the pursuit of generality and orthogonality better than the evolutionary strategy, because the waterfall model allows the consideration of entire designs at every abstraction level, whereas the evolutionary strategy demands that different parts of a design are considered in different increments.

Example/empirical support. Boehm [1988, p. 63] states that in practice “it is generally difficult to distinguish [evolutionary development] from the old code-and-fix model, whose spaghetti code and lack of planning were the initial motivation for the waterfall model”. The evolutionary development strategy, which demands that different parts of designs are considered in different increments, thus often results in designs that are badly structured.

7.4.5 Summary of conclusions

Table 7.5 summarises the conclusions of this section. It shows the four criteria for evaluation of development strategies, the approach to support these criteria, and the ranking of the support offered by each basic development strategy.

criterion (development objective)	*approach*	*support by basic development strategies*
minimisation of re-work due to changing specified requirements	early requirements validation	1. rapid prototyping, 2/3. waterfall/evolutionary [a)] 4. code-and-fix
minimisation of re-work due to non-implementable designs	early implementability validation	1. code-and-fix 2. evolutionary 3/4. waterfall/ rapid prototyping
optimal resource allocation	resource planning and control	1. waterfall 2. evolutionary 3/4. rapid prototyping/code-and-fix
conformance to quality criteria	pursuit of quality criteria	1. waterfall 2. evolutionary 3/4 rapid prototyping/code-and-fix

Table 7.5 Summary of evaluation of basic development strategies. Third column lists support by development strategies in order from most to least. a) evolutionary strategy better in case of unstable requirements.

7.5 Selection of basic development strategies

7.5.1 Risks of development processes

None of the basic development strategies supports all of the identified development objectives in the best way possible. Therefore, the selection of a basic development strategy should not only be based on their support of these development objectives, but also on the *need* for the support of each development objective.

The need for the support of a development objective is high in a development process if the consequences of not supporting this development objective are large. Conversely, if the consequences of not supporting a development objective are marginal,

the need for this support is low. Since the development objectives were selected on the basis of their influence on costs and time to delivery of a development process, we also consider the consequences of not having support for these development objectives in terms of costs and time to delivery.

The actual consequences of not supporting a development objective can only be determined after the development process has been completed. A basic development strategy is, however, selected in the early phases of the development process. The selection of a development strategy should therefore be based on the *estimated* consequences of not supporting each of the development objectives *prior* to the selection.

After Boehm [1988] we call these prior estimated consequences *risks*. A high risk in a development process means a strong need for the support of a development objective by a development strategy. A low risk in a development process means little need for the support of a development objective by a development strategy. The risks of a development process are strongly influenced by human factors, such as the ability and experience of the developers, and the management of the development process.

The four identified development objectives lead to four types of risk:

- The *requirements risk* of a development process is the prior estimated consequences of re-work due to changing specified requirements if there is no support by the development strategy for requirements validation. A high requirements risk can be caused by e.g.:
  - users who do not know what they want;
  - developers who have poor communicative abilities.
- The *implementability risk* of a development process is the prior estimated consequences of re-work due to non-implementable designs if there is no support by the development strategy for implementability validation. A high implementability risk can be caused by e.g.:
  - little experience of the developers with the implementation components;
  - relatively high performance requirements.
- The *resource risk* of a development process is the prior estimated consequences of non-optimal resource use by development steps if there is no support by the development strategy for resource planning and control. A high resource risk can be caused by e.g.:
  - a tight and fixed budget for the development process;
  - developers who are only available for a fixed period.

- The *quality risk* of a development process is the prior estimated consequences of non-conformance to quality criteria if there is no support the pursuit of quality criteria. A high quality risk can be caused by e.g.:
  - a system that is likely to undergo much maintenance, for example, if it should function for a long time;
  - the demand that parts of the system under development should be re-useable in other systems.

7.5.2 Methods for risk assessment

In order to select a basic development strategy, the identified risks first have to be assessed. Two types of methods for doing so are mentioned in the literature:

- Methods based on extensive questionnaires (e.g. [SBA, 1986]). Lists covering many factors that influence risks are prepared on the basis of experience with development processes in the past. Then a group of developers and, e.g., system users and managers come together to rate all of these factors for the development process for which a strategy has to be selected.
- Methods for identifying relative risks (e.g. [Boehm, 1988]). Rather than being based on large lists of factors that influence risks, in these methods developers and users identify relative risks in, e.g., brainstorming and structured evaluation sessions.

The results of these methods are, to some extent, subjective and prone to errors. This is, however, unavoidable, since the selection of a development strategy is always based on an assessment of future events. Boehm [1988] therefore advises organisations to spend much effort to the development of systematic procedures for risk assessment on the basis of a generalisation of their experiences, and to involve experienced developers in risk assessment.

7.5.3 Selection of a basic development strategy

Once the risks of a development process have been assessed, a development strategy can be selected. Table 7.6 lists the situations in which the basic development strategies are most suitable in terms of the valued risks, which are ranked on an ordinal scale from high to low. The table was constructed by first considering the relations between the identified risks and the need for support of particular criteria (section 7.5.1) and then considering the support each basic development strategy offers for these criteria (section 7.4.5). Obviously, when a development strategy is suitable in case a particular risk is high or medium, it is also suitable when the risk is lower.

development strategy	*requirements risk*	*implementability risk*	*resource risk*	*quality risk*
waterfall model	high/medium [a)]	low	high	high
evolutionary strategy	high/medium [b)]	medium	medium	medium
code-and-fix strategy	low	high	low	low
rapid prototyping	high	low	low	low

Table 7.6 Suitability of basic development strategies. a) high in case of uncertain requirements, medium in case of unstable requirements. b) high in case of unstable requirements, medium in case of uncertain requirements.

7.6 Composite development strategies

Table 7.6 shows that each basic development strategy is appropriate in case of particular combinations of risk values. However, in many cases a development process possesses a different combination of risk values. In such cases, none of the identified basic development strategies optimally suits the development process. One should then be able to construct a tailored development strategy.

The construction of a new development strategy should be based on the way development strategies have been presented in section 7.3: first distinguish groups of development steps, and then determine the order in which they should be carried out. In this section we treat some composite development strategies, strategies which are constructed from the identified basic development strategies. The composite strategies discussed here can be classified as development strategies that use different basic development strategies:

- at different abstraction levels, or
- for different increments or versions of the system.

7.6.1 Use of different basic development strategies at different abstraction levels

An example of this type of composite development strategy is the incremental development strategy as described in [Alexander & Davis, 1991] and [Wohlin, 1994]. This development strategy is based on the following guidelines:

1. elicit all requirements;
2. select a part of the requirements and implement these;

3. expand the system by adding more increments according to step 2 until the complete system has been implemented.

This development strategy thus uses the waterfall model at the highest abstraction level and then uses the evolutionary development strategy. Figure 7.7 shows this graphically.

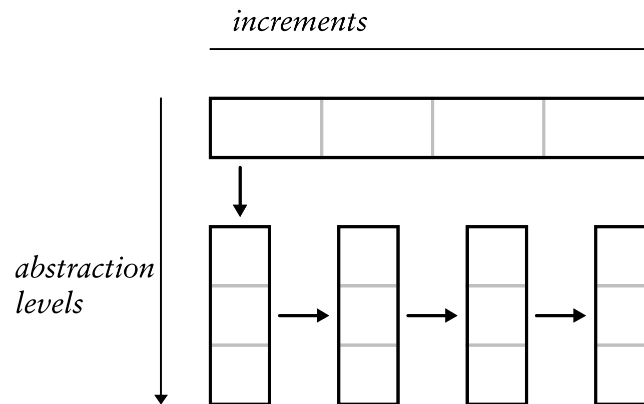


Figure 7.7 Structure of incremental development strategy. Legend as for Figure 7.2.

It is not surprising that the support for all of the criteria of section 7.4 by the incremental development strategy is somewhere between or equal to the support by the waterfall model and the support by the evolutionary strategy. The incremental strategy is particularly suited for development processes with a medium implementability risk (for which the waterfall model is unsuitable) and medium to high resource and quality risks (for which the evolutionary strategy is unsuitable).

The reason why incremental development is suitable for development processes with a medium to high resource risk is that its application results in the early availability of the complete set of specified user requirements. This allows proper estimates of the complexity of the following groups of development steps to be made.

The early availability of the complete set of specified user requirements is also the reason why incremental development is suitable for development processes with a medium to high quality risk. As Schot [1992] describes, this early availability of the user requirements allows for the selection of key functional elements, functions that have a major impact on the structure of the system. By implementing the functional elements with the largest structural impact first, and by strictly adhering to the criterion of open-endedness, a cleanly structured system can be developed.

Schot also describes an extension of incremental development as described by Alexander & Davis: the partly parallel implementation of key functional elements. This allows the time to market of a product to be reduced.

7.6.2 Use of different basic development strategies for different increments or versions of the system

An example of this type of composite development strategy is the "do it twice" strategy [Royce, 1970]. In this strategy a system is first built using a combination of rapid prototyping and the code-and-fix strategy. Then a new, cleaner, version of the system is built using the waterfall model.

A better worked out version of the "do it twice" strategy is the spiral model [Boehm, 1988]. It is based on the following guidelines:

1. select the aspect of the system with the highest risk, implement this aspect (either by means of rapid prototyping or by means of the code-and-fix strategy), and adapt it until this risk has been "resolved" (coped with);
2. expand the system by adding another increment according to step 1;
3. repeat step 2 until all high-risk aspects of the system have been resolved; then build the system according to the waterfall strategy.

Figure 7.8 shows a possible structure of a development process carried out according to the spiral model. In this development process the following basic development strategies were applied in the following order: rapid prototyping, code-and-fix strategy, rapid prototyping, and waterfall model.

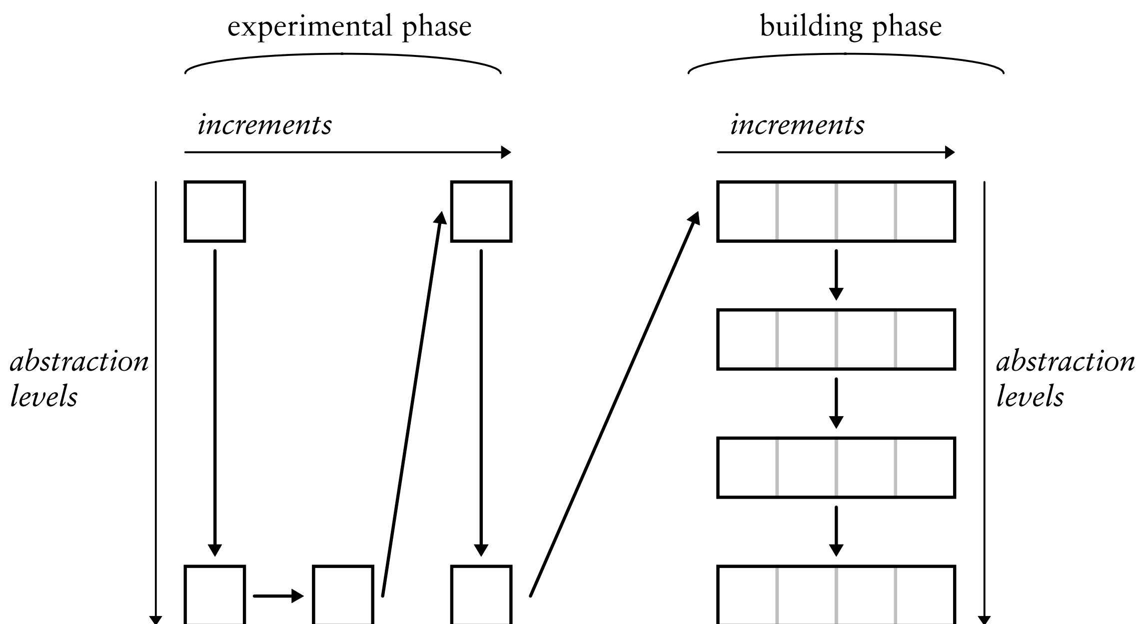


Figure 7.8 Possible structure of a development process carried out according to the spiral model. Legend as for Figure 7.2.

The spiral model offers good support for most of the criteria identified in section 7.4, because it allows for the use of any of the basic development strategies to

"resolve" the high risks of different aspects. The spiral model is, however, not always the most appropriate strategy (based on [Pressman, 1992]):

- Since the complete specified requirements only become available after the experimental phase, planning and control of this experimental phase is difficult. Due to the various basic development strategies that can be used in the experimental phase, the experimental phase is even harder to control.
- If both the requirements risk and the implementability risk are low, the waterfall model is the most appropriate strategy for a development process. The emphasis of the spiral model on experimentation may lead to far more experimentation than required before the system is developed with the waterfall model.

7.7 Development strategies for specific types of systems

The selection of a strategy for the development of certain specific types of systems is also influenced by other criteria than the ones discussed in this chapter. Here we give some examples of this.

An open system is a system, such that multiple parties agree on the external behaviour of its parts, each part can be implemented and produced by these parties, and the system can be composed by coupling the implemented parts [ISO, 1983]. This requires a specification of the external behaviour of each system part. If the implementation of a part is deferred until an agreement on this specification, this implies the use of the waterfall model in the initial stages of the development process.

In case a development process concerns the development of a system for an external party, a contract is often made that specifies the major requirements for the system. Since this contract is made before the system is implemented, this implies the use of the waterfall model in the initial stages of the development process.

In some systems it is hard or time-consuming to observe whether a system conforms to the stated requirements. This applies, for example, to systems with extreme reliability requirements [Courtois, 1985]. In such cases premise *1a* is not valid. (Premise *1a* says that a development strategy best supports early requirements validation when its application results in the early availability of a prototype of the complete system. Its motivation is that by experimenting with the system, users can directly compare the specified requirements—provided these have been implemented correctly—to their actual requirements.) In such cases early requirements validation is best performed by reviewing the complete set of specified requirements with the users early in the development process and by using analysis or simulation techniques to validate certain aspects of these requirements [Trivedi, 1984]. The waterfall model and the incremental development strategy are the most appropriate development strategies for this purpose.

7.8 Conclusions

The relation between the characteristics of development strategies and the need for the support of particular development objectives is important for the selection of an appropriate development strategy. However, little research has been carried out in this area. Our analysis of development strategies has led to:

- the distinction of important elements of development strategies: the support for separation of concerns, and the order in which distinguished development steps are to be carried out;
- the distinction of important development objectives that development strategies may support: minimisation of re-work due to changing specified requirements, minimisation of re-work due to non-implementable designs, optimal resource allocation, and conformance to quality criteria;
- the identification of the support of each (element of a) development strategy for each development objective.

The selection of a development strategy for a development process can then be based on the need for the support of each development objective. These needs can be formulated as risks.

In case no basic development strategy offers the required support for development objectives, a composite development strategy can be constructed by first distinguishing groups of development steps, and then determining the order in which they should be carried out.

8

Synthesis of development methods

8.1 Introduction

8.1.1 Motivation

This chapter discusses the synthesis of development methods from the previously discussed structuring techniques, i.e.

- architectural concepts (chapters 3 and 4);
- horizontal structuring techniques (chapter 5);
- vertical structuring techniques (chapter 6);
- development strategies (chapter 7).

By properly synthesising, and possibly refining, selected structuring techniques, one can construct a development method for business processes that gives developers more concrete guidance than a mere set of abstract structuring techniques from which developers select the ones they deem most appropriate.

8.1.2 Structure

Section 8.2 presents our approach to the construction of development methods and compares it with other approaches.

Section 8.3 illustrates our approach to the construction of development methods by presenting the outlines of four development methods.

8.2 Construction of development methods

8.2.1 Approach

Our approach to the construction of development methods can be summarised as follows:

1. Selection, and possibly refinement, of the horizontal and vertical structuring techniques to be supported by the method, resulting in groups of related development steps.
2. Selection, and possibly refinement, of a development strategy that determines the order in which these groups of development steps are taken, thus synthesising them in a development method.
3. Possibly further refinement of the resulting development method by the provision of heuristics, design criteria, and application examples.

The selection of structuring techniques in steps 1 and 2 should take place on the basis of the need for the support of development objectives that is offered by each structuring technique. For example, if one intends to develop a business process that is structured around work-flows, it is more appropriate to select a work-flow-oriented style for horizontal structuring than a business function-oriented style. Since the development objectives supported by each structuring technique were discussed in the chapter where the structuring technique was introduced, we do not further discuss these development objectives here.

The approach does not involve the selection or refinement of the architectural concepts: all identified architectural concepts are orthogonal to and apply to any development method constructed according to our approach.

Selection and refinement of horizontal and vertical structuring techniques

Selection. Chapter 2 showed that the combination of horizontal structuring (resulting in aspects) and vertical structuring (resulting in abstraction levels) leads to the structure of a development process as represented in figure 2.4. This figure is repeated as Figure 8.1 for the convenience of the reader. Each square in the figure represents a group of one or more development steps in which decisions are to be taken that regard one aspect of the system at one abstraction level. From here on we call such a group of development steps an (abstract) development step.

A development method need not support all aspects and abstraction levels identified in chapters 5 and 6. The constructor of a method may select which of the identified aspects and abstraction levels are to be supported by the development method.

One might, for example, select local and remote constraints with respect to work-flows for horizontal structuring and the integrated system perspective and the distributed system perspective for vertical structuring.

aspect / abstraction level	1	2	...	m
n				
..				
2				
1				

Figure 8.1 Combination of horizontal and vertical structuring resulting in groups of related development steps. Re-print of figure 2.4.

Refinement. The selected horizontal and vertical structuring techniques can be refined in order to provide further guidance to developers.

For example, the following further horizontal separation of concerns can be provided:

- modelling of entities;
- modelling of behaviour, consisting of:
  - modelling of (inter)actions, without their attributes;
  - modelling of causality relations between (inter)actions;
  - modelling of (inter)action attributes and reference relation.

Selection and refinement of a development strategy

Selection. Next, a development strategy should be selected. This development strategy determines the order in which the previously identified development steps are to be carried out. It thus synthesises these development steps in a development method.

Refinement. If the selected horizontal or vertical structuring techniques have been refined, resulting in smaller development steps, the selected development strategy needs to be refined as well, in order to prescribe the order in which these smaller development steps are to be carried out.

For example, given the further horizontal separation of concerns presented above, the following order of the resulting development steps can be prescribed:

- first, the modelling of entities;
- second, the modelling of behaviour, which is carried out as follows:
  - first, the modelling of (inter)actions, without their attributes;
  - second, the modelling of relations between interactions;
  - third, the modelling of (inter)action attributes.

Further refinement

Most of the elements of development methods discussed in chapters 3 to 7 are generic for distributed systems. In order to give more concrete guidance to developers of business processes, the method resulting from the previous steps can be refined further. In the example method of chapter 9 we do so by the provision of the following.

- *Heuristics*. Heuristics are "rules of the thumb" that usually originate from experience. An example is the "staple yourself to an order" heuristic, which says that when modelling a work-flow, it can be useful to follow through an organisation the route of the information or material objects passed in an initial interaction of the work-flow.
- *Examples of design criteria*. Design criteria determine what constitutes a "good" design. The provision of examples of design criteria enables developers to select appropriate ones or to develop their own design criteria. Examples of design criteria for business processes we provide are "a minimal amount of interactions with clients", and "a minimal number of checks and controls".
- *Application examples*. Examples can make a method more concrete for developers. Franken et al. [1997a] argue that many development methods are perceived as concrete mainly due to the large amount of examples of the application of the methods.

8.2.2 Development processes comprising both analysis and design

The approach to the construction of development methods of the previous section suits development processes in which one either analyses a current system or designs a new system. Since re-design processes comprise both types of activity, we consider such development processes here.

Figure 8.2 shows a simplified model of a re-design process. It is based on [DeMarco, 1979], but similar figures can be found in other literature on re-design. In Figure 8.2 the arrows represent relations between system models or between systems in the real world and their models.

Unfortunately, practice shows such figures are easily misunderstood. Examples of such misunderstanding are the following.

- *The current system should always be analysed in detail*. The figure does not intend to express this, since it does not define how detailed the model(s) of the current system should be. But the statement itself is incorrect as well. Ultimately, the goal of a re-design process is the design and implementation of a new system. Analysis of the current system is only useful insofar as it contributes to this goal, for example, by giving developers insight in the application domain or in the

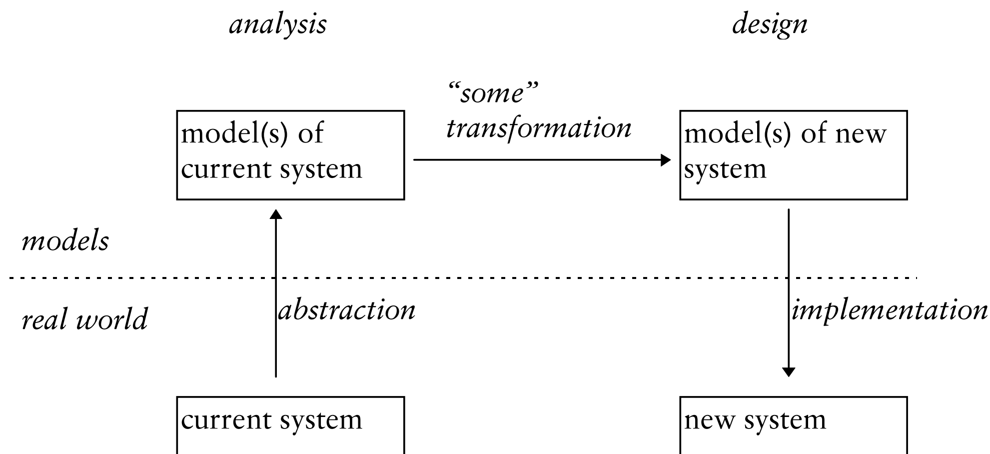


Figure 8.2 Model of a re-design process representing represent relations between models and between systems and their models.

implementation components, by helping them to trace problems of the current system, or by helping them to determine the constraints the new system has to satisfy.

- *Analysis of the current system should be finished before design of the new system starts*. The figure does not intend to express this, since the arrows do not represent order relations. In many cases one may start to design the new system after only a part of the current system has been analysed. This is, for example, prescribed by the evolutionary strategy.
- *Analysis should be carried out bottom-up and design should be carried out top-down*. Again, the figure does not intend to express this, since the arrows do not represent order relations. It may be intuitively appealing to model the current system according to a bottom-up approach, since all low-level components already exist, but our experience shows this is usually not an appropriate development strategy. A current system may be just as complex as a new system, and therefore one needs structuring techniques to make this complexity manageable (which the bottom-up strategy hardly supports). In fact, a current system may be more complex than a new system, since it often evolved in uncontrolled ways.

Since

- the objective of analysing a current system is different from the objective of designing a new system, and
- the structure and complexity of the current system may differ from those of the new system,

one may require different horizontal structuring techniques, different vertical structuring techniques, and a different development strategy for analysing a current system than for designing a new system. Therefore, the method for the analysis of a

current system may differ from the method for the design of a new system. Both methods can be constructed according to the approach of the previous section.

8.2.3 Comparison to other approaches

A class of development methods still used in practice has been called “methods of the cookbook” by Simon [1996]. Such methods prescribe detailed activities to be carried out when building a system. Although such methods are concrete, the metaphor immediately indicates a problem: whereas a recipe for a dish is meant to create the same dish over and over again, a development method should be applicable to a wide range of systems, most of which have never been built before. Methods of the cookbook, albeit structured, are therefore usually organised like large collections of recipes. They do not utilise structuring techniques to their full extent to make complexity manageable and are, due to their size and level of detail, usually based on a large number of implicit assumptions. This is why Simon [1996, p. 135] states that “nowhere do we need to return or retreat to the methods of the cookbook”.

Another approach to the construction of development methods is called “situational method engineering” [Harmsen et al., 1994]. A situational method is a development method whose underlying assumptions regarding the appropriateness of its elements have been made explicit; it is thus applicable to an explicated class of development processes, a so-called situation.

Harmsen et al. and others (e.g. [Kumar & Welke, 1992; Kronlöf, 1992]) propose to carry out situational method engineering by means of the selection of elements from one or more current development methods and their synthesis in a new development method. The use of current development methods has important pragmatic benefits: people used to one or more developments methods do not have to learn a new one. It has, however, drawbacks as well. First, if one uses a single development method as a basis, one is obviously limited to the concepts, the horizontal and vertical structuring techniques and the development strategy used in that method. Second, if one uses multiple development methods as a basis, it is very difficult to construct a complete and consistent method, especially if the modelling languages of the development methods are based on different underlying architectural concepts [Kronlöf, 1992].

Our approach tries to avoid these drawbacks. By the provision of an (expandable) set of horizontal and vertical structuring techniques and development strategies, one is not limited to the elements of a single current development method. By the use of a single set of architectural concepts, consistency problems with regard to the modelling language are avoided. Finally, because the relations between the identified horizontal structuring techniques, between the identified vertical structuring techniques, and between the identified development strategies have been defined, other consistency problems can be avoided as well.

8.3 Synthesis: some examples

This section gives some examples of the synthesis of elements of development methods. These examples consists of outlines of some development methods.

8.3.1 Initial development step

Before starting the development of any complex system, one needs to explicitly delimit the scope of the system under development and to determine the objectives of the development process [Davenport, 1993]. After doing so, the developers can focus on those elements of the system that fall within the selected scope and that influence the determined objectives. This prevents developers paying much attention to irrelevant issues and prevents unstructured discussion about the scope and objectives of a development process while it is being carried out.

The definition of the scope and objectives need initially not be as detailed as a complete service description of a system: when making such a detailed description, one is already deeply involved in the development process. Instead, one should initially define the objectives and scope at higher abstraction level, such that further, more detailed, specifications can be guided by these definitions.

Little guidance exists in the literature for the definition of scope and objectives at this high abstraction level, but Davenport [1993] and Brand & Van der Kolk [1995] provide some examples for business processes.

The scope can be defined in terms of, e.g.:

- the organisation units (or the "process owners", which are the responsible managers of organisation units) involved in the business process;
- the customers of the business process;
- other business processes that exchange materials or information with the process under development.

The objective can be defined in terms of, e.g.:

- short statements describing the product(s) and/or service(s) delivered to clients, or the changes to be made to current product(s) and/or service(s);
- various temporal aspects of the business process (e.g., processing time, waiting time, time to delivery);
- various cost aspects of the business process (e.g., overall costs, costs of various activities, costs of producing one product or servicing one customer)

8.3.2 Examples of development methods based on structuring around work-flows

In the examples below we select the following structuring techniques:

- horizontal structuring: local and remote constraints with respect to use-cases and work-flows;
- vertical structuring: the integrated system perspective and the distributed system perspective.

The following considerations are important when defining development steps on the basis of the above defined separation of development concerns.

- The number of use-cases and work-flows may differ per business process. The number of steps in which local constraints with respect to a use-case or a work-flow are defined should therefore be variable.
- The number of development steps in which remote constraints with respect to use-cases or work-flows are defined is potentially also large. Practice shows, however, that at a particular abstraction level remote constraints can often be defined in one development step. We therefore define only one development step in which these remote constraints are defined. If further separation of concerns is required, developers have to introduce this themselves.

Table 8.3 summarises the defined development steps, assuming that *m* use-cases and work-flows are considered. In steps *U1*, *U2*, ..., *Um* the local constraints are defined with respect to use-cases *1*, *2*, ..., *m* and assigned to entities. In step *UR* the remote constraints with respect to the use-cases are defined and assigned to entities. In steps *W1*, *W2*, ..., *Wm* the local constraints are defined with respect to the work-flows that are refinements of use-cases *1*, *2*, ..., *m*, respectively, and they are assigned to entities. In step *WR* the remote constraints with respect to the work-flows are defined and assigned to entities.

	local constr. use-case/ work-flow 1	local constr. use-case/ work-flow 2	• • •	local constr. use-case/ work-flow m	remote constr. use-cases/ work-flows
integrated perspective: use-cases	*U1*	*U2*	...	*Um*	*UR*
distributed perspective: work-flows	*W1*	*W2*	...	*Wm*	*WR*

Table 8.3 Development steps resulting from defined separation of concerns.

By carrying out step *UR* when steps *U1*, *U2*, ..., *Um* have been carried out, one completes the definition of the service of the business process. In many cases it is important to create a full service definition, since it is the most abstract definition of the business process that gives complete insight in the business process from the perspective of the environment of the system, notably of the customers.

There are, however, also cases where one does not require a full service definition, for example, if one merely wishes to get insight in the order of interactions in isolated use-cases (and not in e.g. the exact times of the interactions, which are usually determined by the remote constraints as well). Practice has shown that the definition of the remote constraints with respect to the use-cases is often difficult, particularly when one has no insight yet in the implementation of the business process. (This is because the remote constraints with respect to the use-cases depend on the resources shared by work-flows, which determine many of the remote constraints with respect to the work-flows.) In such cases one may therefore opt to skip step *UR*.

Development strategy: waterfall model

We call the initial development step in which an initial delimitation of scope and determination of objective should be performed step *I*.

The waterfall model does not prescribe any order in which the local constraints with respect to use-cases or work-flows are to be defined. However, a developer is free anyway in the selection of the use-cases and work-flows to be considered in the development steps *U1*, *U2*, ..., *Um* and *W1*, *W2*, ..., *Wm*. We therefore assume that steps *U1*, *U2*, ..., *Um* and steps *W1*, *W2*, ..., *Wm* are carried out in this order.

The waterfall model does not prescribe whether the remote constraints are to be defined before or after the local constraints either. It is, however, usually preferable to define the remote constraints after all local constraints have been defined, since a) remote constraints are often more complex than local constraints and b) by defining the local constraints first, one can define the remote constraints over already identified (inter)actions.

The waterfall model then prescribes the following order of development steps:

1. step *I*,
2. steps *U1*, *U2*, ..., *Um*, and *UR*, respectively,
3. steps *W1*, *W2*, ..., *Wm*, and *WR*, respectively.

If one considers implementation as well, this should be carried out in step 4.

Development strategy: evolutionary strategy

The evolutionary strategy only prescribes that a step *Ui* should be followed by a step *Wi*, $i=1..m$. It does not prescribe the order in which the use-cases or work-flows should be considered. It also does not prescribe whether the remote constraints are to be defined before or after the local constraints. For similar reasons as above we

assume that steps *U1*, *U2*, ..., *Um* and steps *W1*, *W2*, ..., *Wm* are carried out in this order, and we assume the remote constraints to be defined after all local constraints are defined.

The evolutionary strategy then prescribes the following order of development steps:

1. step *I*,
2. step *U1*, followed by *W1*,
3. step *U2*, followed by *W2*,

 ...

m+1. step *Um*, followed by *Wm*,

m+2. step *UR*, followed by *WR*.

If one considers implementation as well, each step 2, 3, ..., *m*+2 should be followed by an implementation step.

8.3.3 Examples of development methods based on structuring around business functions

In the examples below we select the following structuring techniques:

- horizontal structuring: business functions, extensionally or intensionally defined;
- vertical structuring: the integrated system perspective and the distributed system perspective.

Table 8.4 summarises the defined development steps, assuming that *n* business functions are considered. In steps *FE1*, *FE2*, ..., *FEn* the business functions *1*, *2*, ..., *n* are extensionally defined and assigned to entities. In steps *FI1*, *FI2*, ..., *FIn* the respective business functions are intensionally defined and assigned to entities.

	business function 1	business function 2	• • •	business function n
integrated perspective: extensional	*FE1*	*FE2*	...	*FEn*
distributed perspective: intensional	*FI1*	*FI2*	...	*FIn*

Table 8.4 Development steps resulting from defined separation of concerns.

Development strategy: waterfall model

For similar reasons as in section 8.3.2, we assume the development steps *FE1*, *FE2*, ..., *FEn* and *FI1*, *FI2*, ..., *FIn* are carried out in this order.

The waterfall model then prescribes the following order of development steps:

1. step *I*,
2. steps *FE1*, *FE2*, ..., *FEn*, respectively,
3. steps *FI1*, *FI2*, ..., *FIn*, respectively.

If one considers implementation as well, this should be carried out in step 4.

Development strategy: evolutionary strategy

The evolutionary strategy prescribes the following order of development steps:

1. step *I*,
2. step *FE1*, followed by *FI1*,
3. step *FE2*, followed by *FI2*,

...

n+1. step *FEn*, followed by *FIn*,

If one considers implementation as well, each step 2, 3, ..., $n+1$ should be followed by an implementation step.

8.4 Conclusions

We have shown that the construction of development methods need not be carried out by the mere induction of practical experience. Instead, development methods can be constructed by the synthesis and refinement of structuring techniques that are selected on the basis of the required support for particular development objectives. Since practical experience guided the development of our structuring techniques, development methods constructed in this way can both be concrete and aid developers in controlling the complexity of business processes and their development.

9

A work-flow-oriented evolutionary development method

9.1 Introduction

This chapter presents a work-flow-oriented, evolutionary development method for business processes. The method is constructed according to the approach of chapter 8.

The method, including an application example, is presented to illustrate the use of the structuring techniques discussed in this thesis. It also shows that these techniques, which may be perceived by some developers as abstract, can be synthesised and refined in a concretely applicable development method for business processes.

Section 9.2 gives an overview of the development method and introduces the application example used to illustrate the method.

Section 9.3 presents the development method in detail.

9.2 Overview of the method

Synthesis and refinement

The development method presented here is based on the synthesis of the following structuring techniques:

- horizontal structuring: structuring around use-cases and work-flows;
- vertical structuring: the abstraction levels: interaction system perspective of system and its environment, integrated system perspective, and distributed system perspective;
- development strategy: evolutionary strategy.

These choices result in a work-flow-oriented, evolutionary development method as outlined in section 8.3.2.

This method outline is further refined to give more concrete guidance to developers. The main refinements are the following.

- The initial step (initial delimitation of scope and determination of objective) is decomposed in two sub-steps. In the first sub-step a business process is selected and the objective of its development is determined. In the second sub-step the business process is further delimited.
- The steps where use-cases and work-flows are modelled are decomposed in two sub-steps: entity modelling and behaviour modelling.

  If required, behaviour modelling can be further decomposed in the modelling of separate phases of the behaviour, or in the modelling of normal situations and exceptional situations.

  Behaviour modelling is further decomposed in the modelling of (inter)actions, relations between (inter)actions, and (inter)action values and reference relations.

  Entity modelling is further decomposed in the modelling of entities and the modelling of their interaction points.
- Heuristics, design criteria and examples are provided. Many of the heuristics and design criteria come from literature on business process development. Our main objective of their presentation is to give more concrete guidance to developers. In addition, we aim to give additional relevance to the heuristics and design criteria (many of which originate from literature that does not have a strong methodological focus) by presenting them in the framework of a development method: we show when they are useful. The heuristics and design criteria serve as "rules of the thumb", rather than as formal guidelines. We present them concisely, since we assume their meaning to be clear to the reader.

The method does not consider implementation and the definition of remote constraints with respect to use-cases (see chapter 8). Since it is the objective of this chapter to present a self-contained method, some previously discussed elements are repeated in the description of the method.

The method prescribes that, in every step, entities should be modelled before behaviour is modelled. This somewhat arbitrary choice is based on the observation that many developers perceive entities as more concrete than behaviour. By first identifying and modelling entities, it becomes easier for them to identify and model behaviour. Sometimes, however, it is easier to first identify and model behaviour and then identify and model entities. This is the case, for example, if one models business processes in which the work-flows are distributed over many organisation units, whose interactions are complex. In such cases one may opt to model behaviour before entities.

Applicability

Since the method supports structuring around use-cases and work-flows for horizontal structuring, it is particularly suited for business processes consisting of relatively independent work-flows that are not subject to often changing non-customer requirements. It is less suited for business processes with strong interdependencies between work-flows due to scarce resources required by many work-flows, or business processes in which many work-flows are subject to often changing non-customer requirements. (See chapter 8.)

Since the method prescribed the modelling of a business process at the interaction system perspective of the system and its environment, integrated system perspective, and distributed system perspective abstraction levels, it is well suited for the initial phases of the development of a business process. These initial phases comprise requirements modelling and the definition of the main internal structure of the business process. In order to provide more support for implementation, the distributed perspective should be split into a logically distributed perspective and a physically distributed perspective, and a local interface refined perspective should be added. (See chapter 6.)

Since the method supports the evolutionary strategy, it is well suited for development processes with a medium or high requirements risk, a medium implementability risk, a medium resource risk, and a medium quality risk. (See chapter 7.)

Summary of development steps

For the convenience of the reader we summarise the development steps below. The name of the corresponding step of the outlined method of section 8.3.2 is given between brackets.

1. Determination of objective and delimitation of scope (step *I*)
   a. Selection of business process and determination of objectives
      i. selection of business process
      ii. determination of objectives
   b. Delimitation of scope
2. Use-case modelling (step *U1*)
   a. Selection of use-case
   b. Entity modelling
      i. modelling of entities
      ii. modelling of their interaction points
   c. Behaviour modelling

      (modelling of local constraints with respect to selected use-case)

i. modelling of interactions

ii. modelling of relations between interactions

iii. modelling of values established in interactions and reference relations

if required: separation of different phases of behaviour, or separation of normal situations from exceptional situations.

d. Assignment of behaviour to entities

3. Work-flow modelling (step *W1*)

a. Entity modelling

i. modelling of entities

ii. modelling of their interaction points

b. Behaviour modelling

(modelling of local constraints with respect to selected work-flow)

i. modelling of (inter)actions

ii. modelling of relations between (inter)actions

iii. modelling of values established in (inter)actions and reference relations

if required: separation of different phases of behaviour, or separation of normal situations from exceptional situations.

c. Assignment of behaviour to entities

4. Modelling of other use-cases and work-flows (steps *U2-Um* and *W2-Wm*)

(repetitions of steps 2 and 3 for other use-cases and work-flows)

5. Modelling of relations between work-flows (step *WR*)

a. Entity modelling

i. modelling of entities

ii. modelling of their interaction points

b. Behaviour modelling

i. modelling of remote constraints resulting from the availability of a limited number of entities that carry out a major part of work-flows (e.g., case workers)

ii. modelling of remote constraints resulting from entities involved in only small parts of work-flows (e.g., resources shared by case workers)

c. Assignment of behaviour to entities

Example

We use a realistic, but simplified example to illustrate the application of the method: the sales of insurances at the counter of an insurance company. The example is based on a real case and has been partly described in [Franken & De Weger, 1997].

Example

In the current process of insurance sales, a client fills in an application form at a counter. A counter employee then sends the application form to a back office, where the application is evaluated. A customer is notified by mail whether the application is admitted or not.

Analysis has shown that the time to process applications is long, partly due to the amount of controls performed, and thereby many clients cancel their applications before being sent a policy. Moreover, interviews have shown that clients perceive a rather low level of service.

Management of the insurance company has therefore decided that the current business process should be improved. The example covers the design of this new business process. An informal description of the new business process follows.

1. A client comes to the counter of a front office of an insurance company to effect an insurance. The client fills out an application form with the help of the counter employee.
2. On the basis of the information on the application form, possibly other documents provided by the client, and specified criteria, the counter employee makes one of three decisions: admittance, refusal, or reservation.
   - In case of acceptance, the client gets a preliminary insurance policy and the application form is sent to the back office for further processing.
   - In case of refusal, nothing happens further with the application.
   - In case of reservation, the application form is sent to the back office for further evaluation.
3. In the back office, an expert reviews reserved applications and makes one of two decisions: admittance or refusal. Clients are informed of refused applications.
4. Information regarding admitted applications is entered in an information system. Every day, at the end of the day, the applications entered in the information system are batch-processed, resulting in insurance policies. Each client is sent his or her policy, unless he or she previously cancelled the application.

9.3 The method

Step 1. Determination of objectives and delimitation of scope

Step 1a. Selection of business process and determination of objectives

Objective. Selection of the business process to be developed and determination of the objectives of the development process.

Motivation. The selection of the business process to be developed allows further development steps to be restricted to the ones that regard this business process. The determination of the objectives allows one to take development decisions that support the pursuit of these objectives.

Activities.

i. Select the business process to be analysed or designed. Give a brief, informal description of this business process.

ii. A business process is analysed or designed in order to reach particular objectives. Make these objectives explicit. In case a business process is re-designed, this is done because the current business process has deficiencies. Make these deficiencies explicit and formulate in what respects the new business process should overcome these deficiencies.

<u>Some general selection criteria</u>

- Select a business process that directly provides important products or services to customers.
  - Jacobsson et al. [1995]: Choose processes that are vital to the customers and central to the company's existence.
  - Maul et al. [1994]: Concentrate on operational processes, rather than on management or support processes.
  - Davenport [1993, p. 32]: "Select the processes most central to accomplishing the organisation's strategy."
- Select a business process that currently shows clear deficiencies.
  - Davenport [1993, p. 32]: Select "unhealthy" processes: "processes that are currently problematic and in obvious need of improvement".
- Select a business process in which the application of information technology can make an important difference.
  - Davenport & Short [1990, p.16]: "Identify IT levers... The role of IT in a process should be considered in the early stages of its redesign."

- Hammer & Champy [1993, p. 84-85]: "... applying information technology to business reengineering demands inductive thinking—the ability to first recognize a powerful solution and then seek the problems it might solve".

Some selection criteria/objectives

A list of aspects of business processes is given below. The selection criteria and objectives can be formulated in terms of these aspects.

- quality aspects, e.g.:
  - conformance of products or services to requirements;
  - clarity, reliability and maintainability of products or services;
  - subjective customer satisfaction.
- temporal aspects, e.g. (based on [Brand & Van der Kolk, 1995]):
  - delivery time of a work-flow: time between initial activity(ies) and final activity(ies) of a work flow (called response time in [Franken et al., 1997b]);
  - processing time of a work-flow: time spent on activities of this work-flow;
  - waiting time of a work-flow: delivery time minus processing time;
  - waiting time of a customer: time between the arrival of a customer and the start of the serving of the customer.
- cost aspects, e.g.:
  - fixed costs;
  - variable costs of a work-flow or of activities;
  - costs of a work-flow or of activities: variable costs plus some part of the fixed costs;
  - parts of the costs of a work-flow, e.g. transport costs or inventory costs.
- structural aspects, e.g. (some based on [Hammer & Champy, 1993]):
  - number of interactions with customers;
  - number of interactions or amount of information exchange between units;
  - size of inventory;
  - amount of checking and control, compared to the value added by these activities;
  - amount of re-work due to inadequate feedback.

Example

The selected business process is "insurance sales", which is concerned with insurance applications by clients at a counter, the evaluation of these applications, and the issue of insurance policies.

Deficiencies of the current business process:

- Delivery time is long, since counter employees cannot accept or refuse clients directly.
- Due to the long delivery time, clients often cancel their applications, or perceive the organisation as bureaucratic.
- Processing time is long due to the large number of controls.

The objectives of its improvement are the following:

- Delivery time and the number of clients cancelling their applications should be reduced. This is to be done by giving counter employees the authority to accept or refuse insurance applications according to set criteria. Accepted clients get a preliminary insurance policy. Only in case of doubt an expert should evaluate an application.
- Processing time should be reduced by the reduction of the number of controls.

Step 1b. Delimitation of scope

Objective. Informal further delimitation of the scope of the selected business process.

Motivation. The brief description of the business process of step 1a is insufficiently precise to restrict further development steps to the ones that regard this business process. Therefore, further delimitation of scope is required.

Activities.

Define informally:

- the product(s) or service(s) of the business process;
- the main activities of the business process;
- the initial and final activities of the work-flows of the business process;
- the organisation units involved in the business process; in case of re-design, the organisation units currently involved in the business process should also be defined.

Heuristics for delimitation of scope

- Proprietary: include only those activities relevant to the defined objectives.
- Completeness: “All major processes affecting [the defined objectives] should be included”. [Kaplan & Murdock, 1991, p. 36].
- Orthogonality or loose coupling: do not include activities that have little relations with the selected business process.
- Maintainability: try to localise the impact of changes in the environment (e.g., changing customer requirements or changing government regulations) in a single business process.

- Only re-design what one can change, or what one "owns" [Davenport, 1993].
- Try to keep a balance between too large a scope (resulting from the attempt to prevent sub-optimisation of small business processes) and too small a scope (resulting from the attempt to keep the development process manageable).
  - "The processes need to be defined at levels high enough that redesign can yield 'breakthrough' improvements, yet not so high as to be unmanageable" [Kaplan & Murdock, 1991, p. 37].
  - "Think of the entire core process, from the time the customer makes a connection with your company through the connections with the suppliers and back to the customer" [Johansson et al., 1993, p. 109].
    "Maintain focus: don't try to reengineer too many processes at once." [Car & Johansson, 1995, p. 124].

Example

Further delimitation of the scope of the process "insurance sales":

- the product of the business process is an insurance, represented by an insurance policy;
- the considered process is the sales of an insurance;
- a work-flow of the process starts when a client fills in an insurance application form; the work-flow ends when:
  - the client's application is refused, or
  - the client receives an insurance policy, or
  - the client cancels his or her application.
- the following organisation units are involved in the current and new business process: counter employees (front office), experts (back office), and an information system.

Step 2. Use-case modelling

Step 2a. Selection of use-case

Objective. Selection of the use-case to be considered in this step.

Motivation. According to the evolutionary approach of this method, different use-cases are considered separately. The remainder of the activities in this step are limited to the selected use-case.

Activities.

Select a use-case and describe informally:

- the entities involved in this use-case;
- the main interactions of the use-case.

Heuristics for selection of a use-case

Based on [Bal et al., 1997a]:

- Distinguish between the different types of customers (e.g. between private clients and business clients). Select one of the distinguished types of customers.
- Distinguish between the different products or services delivered to this type of customer. Select one of the distinguished types of products or services.
- The combination of the customer and the product or service delimits the use-case.

Example

In our example, we consider only one use case. The insurance company and a client are involved in the use-case. The use-case comprises all possible interactions of the insurance company and the client, i.e.:

- application for insurance;
- immediate admittance and refusal, reservation, and issuing of a preliminary policy
- deferred admittance and refusal;
- cancellation of an application;
- policy issue.

Step 2b. Entity modelling

Objective. Modelling of the entities involved in the selected use-case and their interaction points.

Motivation. This forms a basis for subsequent modelling of the local constraints with respect to the selected use-case and the assignment of this behaviour to entities.

Activities.

Since a use-case models the extensional aspects of a work-flow, the following entities should be modelled:

- the system under development, as an integrated whole;
- the systems in its environment it interacts with in the selected use-case;
- the interaction points of these systems at which interactions of the use-case occur.

In case the system under development is a part of an organisation, rather than an entire organisation, the system may interact with entities internal to the organisation. For example, a production department may have the sales department as its customer and the purchasing department as its supplier.

Heuristics for selection of entities in the system environment

- The following distinction between entities may be useful [Bal et al., 1997a]:
  - the object system, which is the system under development;
  - clients;
  - suppliers;
  - persons or organisations that are not clients or suppliers, but with which the object system interacts, for example, government agencies, or an accounting department;
  - intermediaries, which are the interaction systems of the object system and the other entities in its environment, for example, insurance agents, a postal company, or a telephone company.
- To keep models comprehensible, it is sometimes useful to abstract from an intermediary, for example if it provides a reliable service that does not add any value to the products or services of the object system, other than enabling interaction between the object system and clients or suppliers. For this reason one often abstracts from, e.g., a telephone company and models it as an interaction point, rather than an entity. When modelling an insurance company, however, one usually does not abstract from insurance agents, since these may provide additional value to customers.

Example

Figure 9.1 models the two entities (*client* and *insurance company*) involved in the selected use-case and their two interaction points (*counter* and *mail*). For reasons indicated in the last heuristic above, the postal company is modelled as an interaction point (*mail*), rather than an entity.

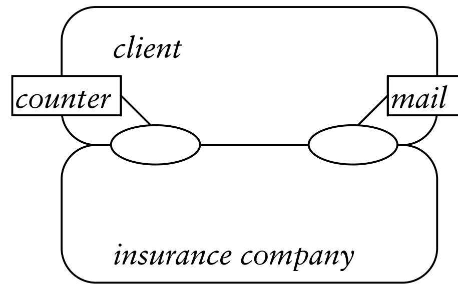


Figure 9.1 Entities involved in selected use-case and their interaction points.

Step 2c. Behaviour modelling

Objective. Modelling of the local constraints with respect to the selected use-case.

Motivation. These local constraints form a part of the interaction system of the system under development and the entities in its environment.

Activities.

The following aspects should be modelled:

i. the interactions of the entities involved in the selected use-case that occur on the interaction points of these entities;

ii. the relations between these interactions;

iii. the values established in these interactions and the reference relations.

In case the selected use-case is complex, the following separation of concerns may aid in keeping this complexity manageable.

- Consider different phases of the use-case separately. Structure the model as a causality-oriented composition of sub-behaviours, in which each sub-behaviour concerns a separate phase. An example of three phases that can frequently be distinguished [Brand & Van der Kolk, 1995]:
  - the related activities before a client order (“production preparation”);
  - the related activities from the client order to the delivery of a product (“production”);
  - the related activities after the delivery of the product (“post-production”).

  The points of time at which one of these phases ends and another one begins are sometimes called “customer order decoupling points” [Bal et al., 1997b].

  Phases that can frequently be distinguished in administrative business processes are: registration (e.g., application for a product, presentation of a claim), evaluation (e.g., evaluation of an application), informing of a decision, appeal against a decision [Bal et al., 1997a].

- First consider main stream behaviour and then consider the subsequent exceptions. Main stream behaviour is a set of related actions that are carried out in most copies of the selected use-case. Exceptions are only carried out for a small percentage of copies of this use-case.

  Examples of exceptions are the related actions due to:

  - cancellation of an order by a client;
  - no reaction by a client on a request for information by the organisation;
  - cancellation of an application by a client;
  - request for unusual services or products (e.g., an insurance with a coverage that is higher than some limit)

Heuristics for use-case modelling

Based on [Bal et al., 1997a]:

- "Staple yourself to an order". In order to identify the related interactions of a use-case, it can be useful to follow through an organisation the route of the information or material objects passed in an initial interaction of the work-flow.
- "Staple yourself to a result". In order to identify the related interactions of a use-case, it can be useful to trace back through an organisation the related activities required to bring about a particular product or service.

Design criteria for use-cases

- "[Cut] back the number of external contact points that a process has" [Hammer & Champy, 1993, p. 60]. This reduces delivery time, since waiting for external entities to provide information or materials often accounts for a significant percentage of the total waiting time.
- "Capture information once and at the source" [Hammer, 1990, p. 112]. By collecting all relevant information at once, one prevents inconsistencies resulting from different units collecting their own information.

Example

Figure 9.2 models the local constraints with respect to the selected use-case, which can be identified by "stapling yourself to an application". For brevity, only some interaction attributes are represented in one interaction. We assume, as in the remainder of the example, that it is clear from the names of the (inter)actions which activities they model.

To further enhance insight, this use-case could, for example, have been structured in the phases "initial contact" (comprising the interactions *application*, *refusal*, and *admittance*) and "later contact" (comprising the other actions).

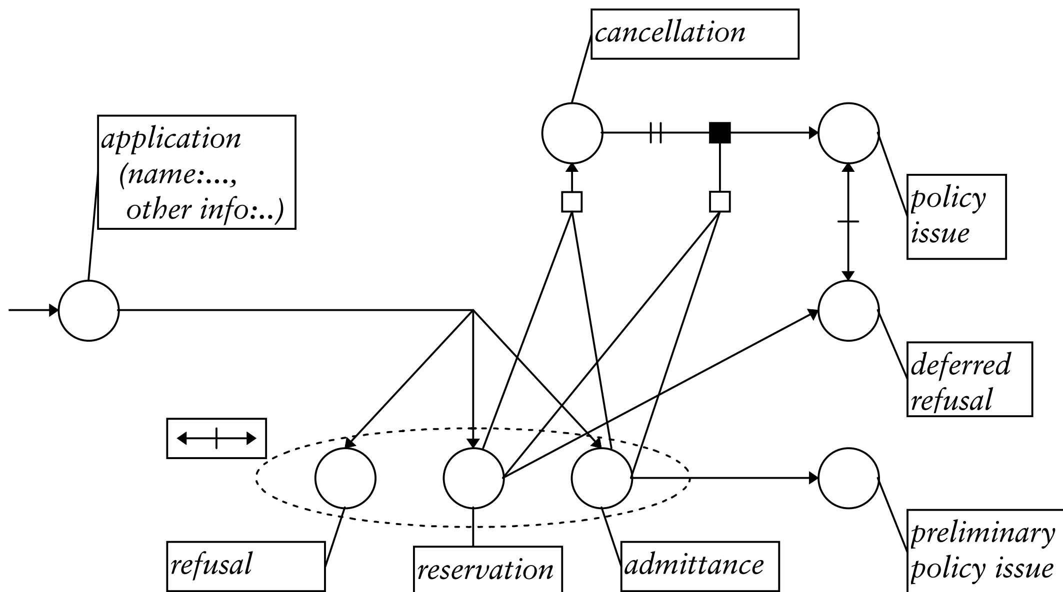


Figure 9.2 Model of selected use-case.

Step 2d. Assignment of behaviour to entities

Objective. Assignment of the modelled interactions and their relations to the modelled entities and their interaction points.

Motivation. This defines the contributions of the entities to the behaviour.

Activities.

i. Determine for each interaction the interaction point at which the interaction occurs.

ii. Determine for each relation the contribution of each entity to this relation.

iii. Determine for each value established in an interaction the contribution of each involved entity to the establishment of this value.

Example

Figure 9.3 models the assignment of the selected behaviour to the selected entities. If the contribution of the client to an interaction is not represented, this implies that the interaction is enabled at any time (start condition).

Figure 9.3 does not represent interaction points. The interactions *application*, *refusal*, *reservation*, *admittance*, and *preliminary policy issue* occur at the interaction point *counter*. The other interactions occur at the interaction point *mail*.

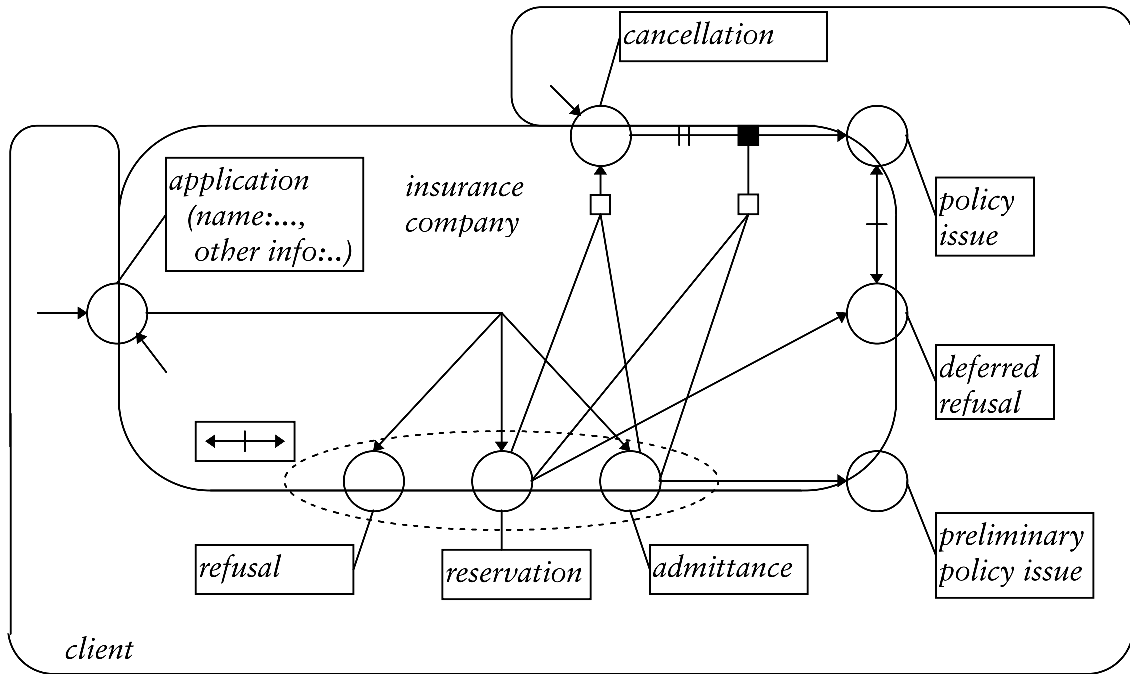


Figure 9.3 Assignment of selected behaviour to selected entities.

Step 3. Work-flow modelling

Step 3a. Entity modelling

Objective. Modelling of the entities of the system under development and their interaction points involved in the work-flow that comprises the internal mechanisms that support the use-case defined in step 2.

Motivation. This forms a basis for subsequent modelling of the local constraints with respect to the work-flow and the assignment of this behaviour to entities.

Activities.

Since a work-flow models the internal mechanisms that support a use-case, the following should be modelled:

i. the internal entities of the system under development involved in the work-flow;

ii. their interaction points at which interactions of the work-flow take place;

iii. (possibly) the action points at which internal actions take place.

Heuristics for identification of internal entities of the system under development

- The work-flow-oriented approach of this method is particularly suited to business processes whose entities carry out or control entire work-flows. Examples of such entities are:
  - Case workers (sometimes called case managers). Hammer & Champy [1993, p. 51] define a case worker as "an individual responsible for an end-to-end process". Davenport & Nohria [1994] define case managers as "individuals or small teams [that] perform a series of tasks, such as the fulfilment of a customer order from beginning to end, often with the help of information systems that reach through the organization".

    Thus, compared to more traditional structuring around business functions, "a case manager provides a single point of contact" and "work units change—from functional departments to process teams" [Hammer & Champy, 1993, p. 52, 65].
  - Work-flow management systems. These are systems that control an entire work-flow through an organisation, for example, by routing information, keeping track of tasks, and informing about the status of work-flows.
- One may opt to have all interactions with clients carried out by single unit, sometimes called a front-office, or a "single counter". Other activities that do not involve direct interaction with the customer can be carried out in a back-office.
- Telematics systems often enable geographically remote organisation units to co-operate on the same business function or work-flow. Therefore, such a group of physically remote units can often be viewed as one logical unit. This leads to Ham-

mer's [1990, p. 109] criterion: "Treat geographically dispersed resources as though they were centralized". This heuristic is only useful when modelling a business process at the logically distributed level, and not when modelling it at the physically distributed level.

Example

Figure 9.4 models the entities of the insurance company involved in the selected work-flow and their interaction points.

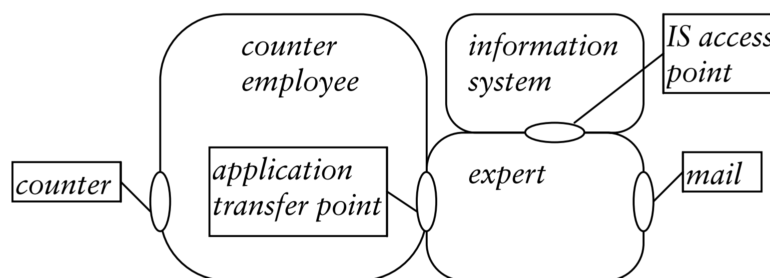


Figure 9.4 Entities inside insurance company involved in selected work-flow and their interaction points.

Step 3b. Behaviour modelling

Objective. Modelling of the local constraints with respect to the selected work-flow.

Motivation. The local constraints with respect to work-flows are part of the internal behaviour of the business process.

Activities.

The following aspects should be modelled:

i. the interactions of the entities of step 3a involved in the selected work-flow, and, possibly, internal actions of these entities;

ii. the relations between these (inter)actions;

iii. the values established in these (inter)actions and the reference relations.

In case the selected work-flow is complex, the structuring techniques of step 2c may aid in keeping this complexity manageable.

- Consider different phases of the work-flow separately. Structure the model as a causality-oriented composition of sub-behaviours, in which each sub-behaviour concerns a separate phase.
- First consider main stream behaviour and then consider the subsequent exceptions.

Heuristics for work-flow modelling

- "Staple yourself to an order" and "staple yourself to a result". See step 2c.

Design criteria for work-flows

[Hammer, 1990, pp. 109-112]:

- "Have those who use the output of the process perform the process". This is a consequence of structuring around work-flows. The criterion aids in preventing sub-optimisation of parts of a work-flow at the expense of the complete work-flow. Moreover, it prevents the transfer of information and materials (which are often time-consuming and error-prone), and the activities required to co-ordinate and control the transfer. Finally, people are best motivated to bring about results when they experience the usefulness of these results. This is the case if they use these results themselves.
- "Subsume information-processing work into the real work that produces the information". Again, this criterion is a consequence of structuring around work-flows, since it prevents specialised business functions for information processing. Application of the criterion should result in processed information being available quickly.
- "People who do the work should make the decisions." This prevents managers getting involved in every work-flow for making even simple decisions. Well-educated employees, supported by information systems, can nowadays usually make a significant amount of decisions. Managers then only need to deal with more difficult decisions. The criterion is sometimes referred to as "empowering" employees [Hammer & Champy, 1993].
- "Checks and controls are reduced". This criterion is partly a consequence of the previous one. The costs of strict control may be much higher than a failure once in a while. [Hammer & Champy, 1993, p. 58]. A cost-benefit analysis needs to be carried out for every control.

  Allen [1994] gives a variety of examples for banks, ranging from the elimination of dual controls on safe deposit box keys to the increase of cashiers' check limits

Example

The work-flow of the selected use-case is constructed in two steps. Figure 9.5 models what is considered the "main stream" behaviour of the work-flow. Attributes of (inter)actions are not represented. The following exceptions are not modelled in Figure 9.5.

- the interaction *reservation*, the actions it (directly or indirectly) enables, *transfer reservation*, *re-evaluate application* and *deferred refusal*, and the relations between these actions and between these actions and other actions;
- the interaction *cancellation* and its relations with other actions.

Figure 9.6 models all local constraints with respect to the selected work-flow, including these exceptions.

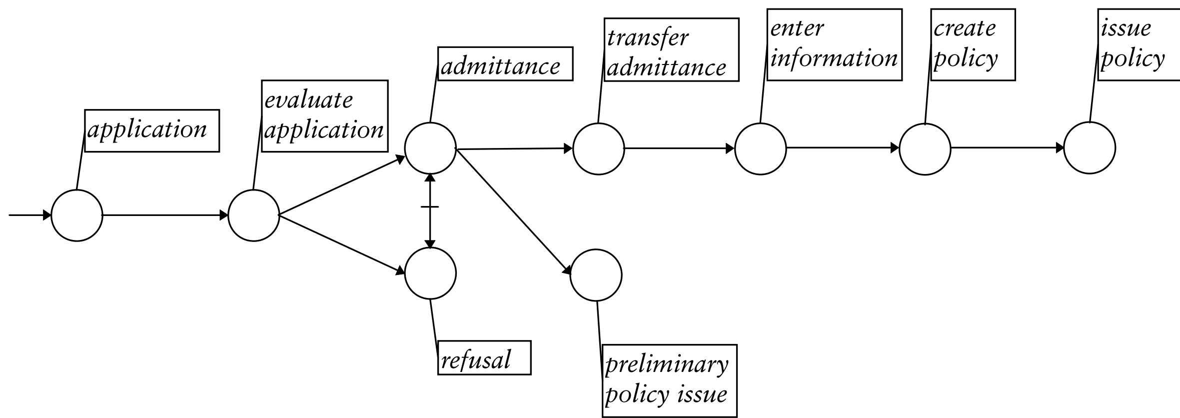


Figure 9.5 Model of part of selected work-flow: main stream behaviour.

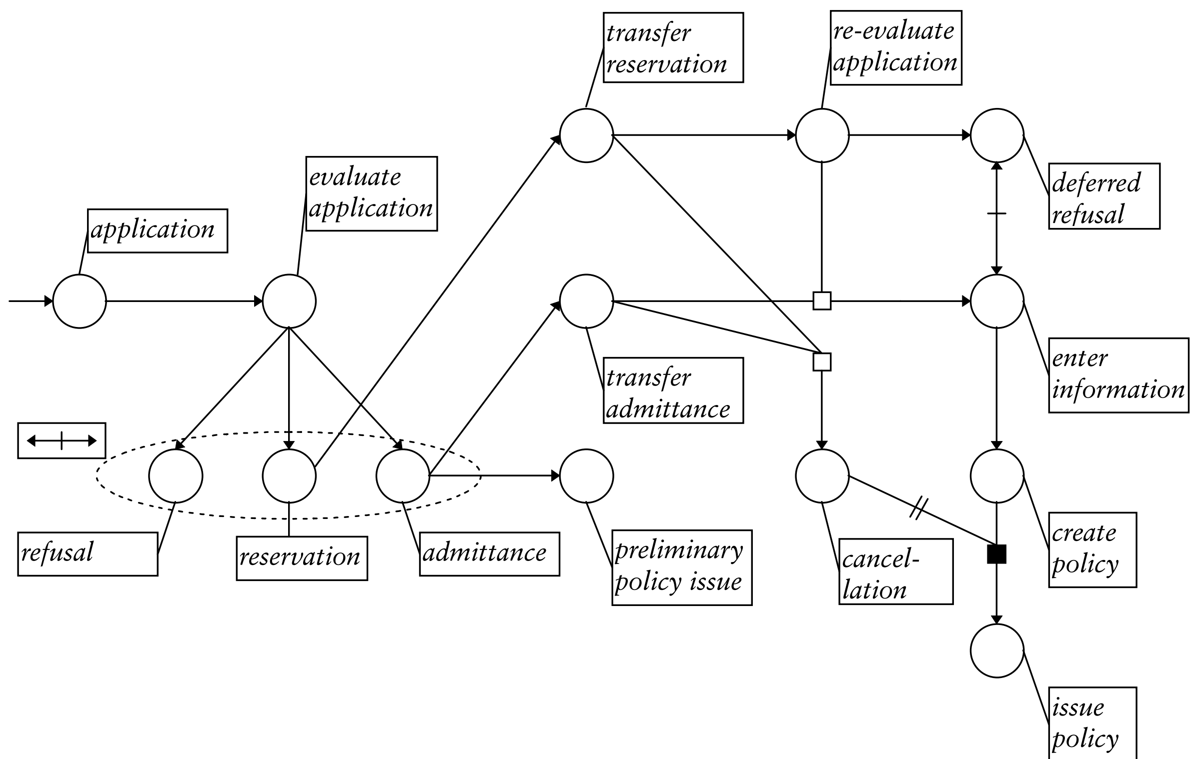


Figure 9.6 Model of local constraints with respect to selected work-flow.

Step 3c. Assignment of behaviour to entities

Objective. Assignment of the modelled (inter)actions and their relations to the modelled entities and their (inter)action points.

Motivation. This defines the contributions of the entities to the behaviour.

Activities.

i. Determine for each (inter)action the (inter)action point at which this (inter)action occurs.

ii. Determine for each relation the contribution of each entity to this relation.

iii. Determine for each value established in an interaction the contribution of each involved entity to the establishment of this value.

Example

Figure 9.7 models the assignment of the selected behaviour to the selected entities.

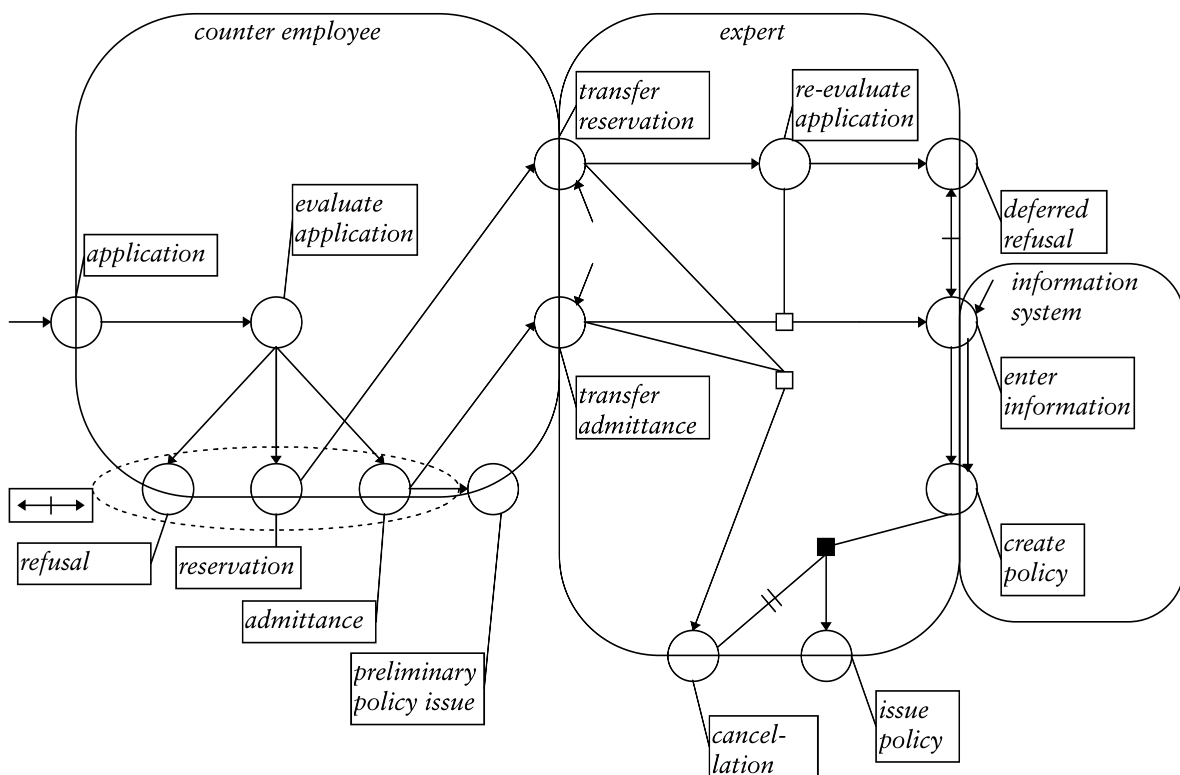


Figure 9.7 Assignment of selected behaviour to selected entities.

Step 4. Modelling of other use-cases and work-flows

Objective. Modelling of the entities involved in and the local constraints with respect to the other use-cases and work-flows of the business process under development.

Motivation. These entities and behaviours are all part of the integrated or distributed perspective of the selected business process.

Activities.

Repeatedly carry out steps 2 and 3 for other use-cases, as long as a use-case of the selected business process can be identified that has not been modelled before:

2. a. Select a use-case of the business process that has not been modelled before.
   b. Model the entities involved in this use-case
   c. Model the local constraints with respect to this use-case
   d. Assign the local constraints to the entities.
3. a. Model the entities of the system under development involved in the work-flow that forms the internal aspects of the selected use-case.
   b. Model the local constraints with respect to this work-flow.
   c. Assign the local constraints to the entities.

Example

In the example behaviour all use-cases and work-flows are copies of the use-case and work-flow modelled in steps 2 and 3. We therefore do not model any other use-cases and work-flows.

Step 5. Modelling of relations between work-flows

Step 5a. Entity modelling

Objective. Integration of the models made in steps 3 and 4 of the entities involved in the work-flows.

Motivation. This leads to a complete model of all entities, and their interaction points, involved in all work-flows of the selected business process. This model forms a basis for the identification of the remote constraints with respect to work-flows and the assignment of these remote constraints to entities.

Activities.

In steps 3 and 4 entities were modelled per work-flow. Entities involved in one work-flow may, however, be the same as entities involved in other work-flows. The same applies to interaction points. Therefore, the construction of a complete model of the entities involved in all work-flows and their interaction points involves:

i. the identification and modelling of distinct entities;

ii. the identification and modelling of the distinct interaction points of the entities.

This step does not comprise the identification of new entities or interaction points, since all entities and interaction points involved in work-flows have already been identified in steps 3 and 4.

> Example: Entity modelling
>
> We consider a single front office of the insurance company with three counter employees. The back office has two experts available for further processing of applications. Each counter employee can interact with each expert. There is one information system the experts can interact with.
>
> Figure 9.8 models these entities and their interaction points. A single interaction point is modelled for the interactions between the counter employees and the experts, to model that a single location (e.g., "internal mail") is used for the transfer of applications. This is similar for the interaction points *IS access point* and *mail*. Three interaction points *counter* are modelled to represent that clients interact with different counter employees at different locations.

Step 5b. Behaviour modelling

Objective. Integration of the local constraints with respect to work-flows with the remote constraints with respect to work-flows.

Motivation. This results in a complete behaviour model of the system under development.

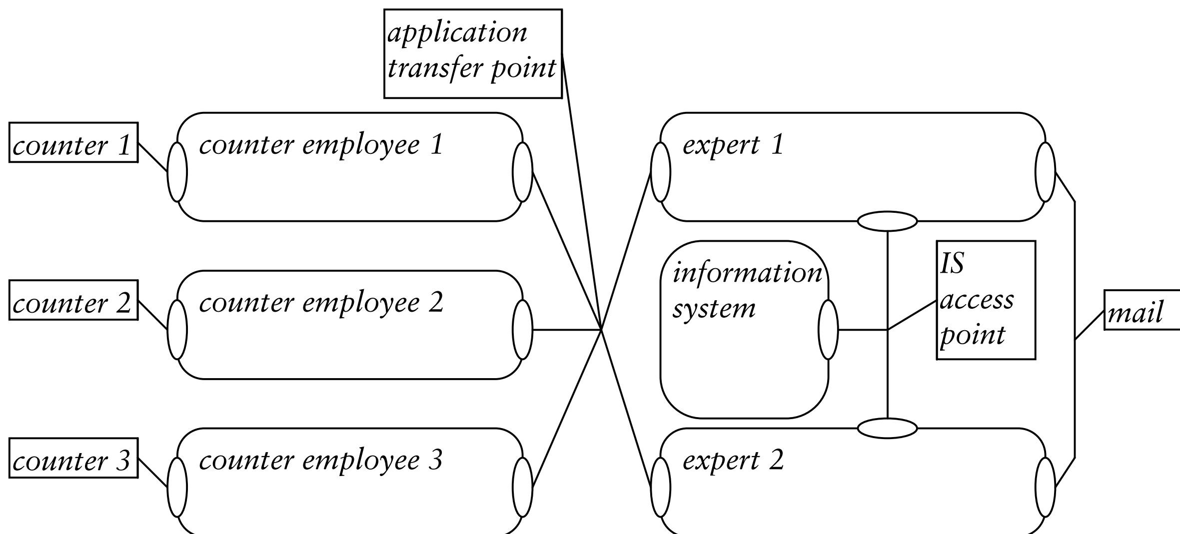


Figure 9.8 All entities of the insurance company involved in the insurance sales process and their interaction points.

Activities.

Remote constraints with respect to work-flows are the relations that determine the order of (inter)actions in different work-flows. These remote constraints result from (parts of) different work-flows being carried out by the same entities. The following separation of concerns is often useful in the modelling of remote constraints with respect to work-flows.

i. Model the remote constraints resulting from the availability of only a limited number of entities that carry out a major part of the work-flows they are involved in, such as case workers.

> Heuristic for modelling remote constraints due to limited number of case workers
>
> A case worker carries out a major part of every work-flow he or she is involved in. Assume that once a case-worker has finished working on one work-flow he or she starts working on a next one. Remote constraints with respect to work-flows result from organisations employing only a limited number of case workers. The behaviour of case workers can be modelled comprehensively according to the following structure, in which *n* is the number of case workers.
>
> *i=1..n: start → case_worker^i(entry)*
> ...
> *case_worker=*
> { *entry → ...,*
> ...
> *(* work on a work-flow finished *) → case_worker(entry)* }.

ii. Model the remote constraints resulting from entities involved in small parts of the work-flows, such as resources shared by case workers (e.g., a machine that can be used by one case worker at a time, or a store from which every case worker retrieves ordered items).

> <u>Heuristic for modelling remote constraints due resources shared by case workers</u>
>
> Just as the behaviour of the case-workers, the behaviour of resources shared by case workers can often be modelled comprehensibly as a recursive behaviour. In case a resource has a limited capacity (e.g. a limited stock in the store) that is reduced in interactions with case workers, this capacity can often be modelled comprehensibly by means of a state value of a state variable. The resource is then modelled according to a state-oriented style:
>
> *shared_resource=*
> { *entry(c:Capacity)* → ...,
> ...
> ... → *(* interaction with case worker that reduces capacity with r *)* < *c=c-r* >
> ...
> *(* shared resource available to other case workers *)* → *shared_resource(entry(c))* }.

This step does not comprise the identification of new interactions, since all interactions of all work-flows have already been identified in steps 3 and 4.

> <u>Example</u>
>
> In the insurance sales behaviour, the remote constraints can be divided into two categories:
>
> - remote constraints due to the availability at any time of three counter employees and two experts;
> - remote constraints due to the behaviour of the information system, which is shared by the experts.
>
> <u>Remote constraints due to limited number of counter employees and experts</u>
>
> Initially the insurance sales behaviour comprises three copies of the behaviour of a counter employee, two copies of the behaviour of an expert, and one copy of the information system behaviour. This is represented as follows.
>
> *i=1,2,3: start* → *counter*^{i}*(entry)*,
> *j=1,2: start* → *expert*^{j}*(entry)*,
> *start* → *information system(entry(∅))*.
>
> Each counter employee and each expert can begin activities of a new work-flow once their activities on an old work-flow have finished. Therefore, the counter employee and expert behaviours are defined recursively as follows.

counter=
{ *entry* → ...
(remainder of counter employee behaviour specified in step 3 *),*
refusal ∨ *transfer reservation* ∨ *(transfer admittance* ∧ *preliminary policy issue)*
→ *counter(entry)* }

expert=
{ *entry* → ...
(remainder of expert behaviour specified in step 3 *),*
deferred refusal ∨ *issue policy* ∨ *cancellation* → *expert(entry)* }.

To transfer a reserved or an admitted application, one counter employee interacts with one expert at the application transfer point. To enter information or to create a policy, one expert interacts with the information system at the IS access point. This is specified as follows.

interactions *counter, expert* on *application transfer point:*
transfer reservation, transfer admittance
expert, information system on *IS access point:*
enter information, create policy

Since the textual representation of this behaviour is more concise than the graphical representation, we do not give a graphical representation.

Remote constraints due to information system behaviour

The following remote constraints with respect to the work-flows are due to the behaviour of the information system:

- no two experts can enter information in the information system at the same time;
- all interactions *create policy* occur sequentially at the end of a day; once a sequence of *create policy* interactions has started, information can only be entered in the information system by the experts after all policies of the day have been created.

The specification of the information system behaviour follows below. In an action *get current time* a value is established that models the time at which the action occurs. This value is used to determine whether information can be entered or policies should be created. A function *is_end_of_day(t)* is assumed to be available that returns *true* if *t* is a point of time at the end of the day, and *false* otherwise. A function *info_of(p)* is assumed to be available that returns the information entered in the information system required to make insurance policy *p*.

information system=
{ *entry(S)* → *get current time(t:Time),*
get current time ∧ *not(is_end_of_day(t))* → *input system(entry(S)),*
get current time ∧ *is_end_of_day(t)* → *batch system(entry(S))* }

input system=
{ *entry(S)* → *enter information(i:Info)* <*S=S* ∪ {*i*}>,
enter_information → *information system(S)* }

```
batch system=
{ entry(S) ∧ (S≠∅) → create_policy(p:Policy) [info_of(p) ∈ S] <S=S\{info_of(p)}>,
  entry(S) ∧ (S=∅) → information system(entry(S)),
  create policy → batch system(entry(S))   }
```

Insurance sales behaviour

The following behaviour has been specified.

```
insurance sales=
{ interactions  counter, expert on application transfer point:
                            transfer reservation, transfer admittance,
                       expert, information system on IS access point:
                            enter information, create policy,
  i=1,2,3: start → counter^i(entry),
  i=1,2: start → expert^i(entry) [j=1,2],
  start → information system(entry(∅))   }

counter=
{ entry → ...,
  (* see Figure 9.7 for rest of counter behaviour *),,
  refusal ∨ transfer reservation ∨ (transfer admittance ∧ preliminary policy issue)
                            → counter(entry)   }

expert=
{ entry → ...,
  (* see Figure 9.7 for rest of expert behaviour *),
  deferred refusal ∨ issue policy ∨ cancellation → expert(entry)   }

information system=
{ entry(S) → get current time(t:Time),
  get current time ∧ not(is_end_of_day(t)) → input system,
  get current time ∧ is_end_of_day(t) → batch system   }

input system=
{ entry(S) → enter information(i:Info) <S=S ∪ {i}>,
  enter_information → information system(S)   }
```


```
batch system=
{ entry(S) ∧ (S≠∅) → create_policy(p:Policy) [info_of(p) ∈ S] <S=S\{info_of(p)}>,
  entry(S) ∧ (S=∅) → information system(entry(S)),
  create policy → batch system(entry(S))   }
```


Step 5c: Assignment of behaviour to entities

Objective. Assignment of the remote constraints with respect to work-flows to the modelled entities.

Motivation. This completely defines the contributions of the entities to the behaviour.

Activities.

i. Determine for each (inter)action the (inter)action point at which this (inter)action occurs.

ii. Determine for each relation the contribution of each entity to this relation.

iii. Determine for each value established in an interaction the contribution of each of the involved entities to the establishment of this value.

Example

The *counter* behaviours are assigned to the counter employees. The *expert* behaviours are assigned to the experts. The behaviour *information system*, including the sub-behaviours *input system* and *batch system*, is assigned to the information system entity.

All interactions between counter employees and experts (*transfer reservation*, *transfer admittance*) take place on the *application transfer point*. All interactions between experts and the information system (*enter information*, *create policy*) take place on the *IS access point*.

The interactions between counter employees and clients (*application*, *admittance*, *reservation*, *refusal*, *issue preliminary policy*) take place on the interaction point *counter 1*, *counter 2*, or *counter 3*, depending on the counter employee involved in the interactions. The interactions between experts and clients (*deferred refusal*, *issue policy*, *cancellation*) take place on the interaction point *mail*.

10

Conclusions and further research

10.1 Introduction

This chapter highlights the main research contributions of this thesis. Many of our structuring techniques have been applied in business process development projects. Therefore, this chapter also discusses the practical experience with the application of these techniques. Finally, it gives recommendations for further research, which should lead to improvements in the support of developers of business processes.

Section 10.2 describes our main research contributions.

Section 10.3 summarises the practical experience with the application of our structuring techniques.

Section 10.4 gives recommendations for further research.

10.2 Research contributions

Although the importance of structuring in system development is acknowledged in the literature, it appears that the role of structuring techniques in the control of complexity has been poorly addressed. Our analysis of complexity led to the identification of some important aspects of complexity, which allowed us to show how some current, generic structuring techniques aid in controlling complexity. By defining the structuring techniques for business processes in this thesis as refinements of these generic structuring techniques, we also clarified the role of the structuring techniques for business processes in the control of the complexity of business processes and of their development.

The provision of a well-defined, complete and parsimonious set of generally applicable basic architectural concepts is crucial for the proper support of developers [Vissers et al., 1995]. We provided, partly based on previous work (e.g. [Vissers,

1983] and [Ferreira Pires, 1994]), such a set of basic architectural concepts for distributed system modelling, and demonstrated their usefulness in business process modelling. We showed that these concepts are applicable at many abstraction levels and argued that these concepts cover the relevant aspects of distributed systems. A comparison by Quartel [1998] shows that our basic architectural concepts are generally more expressive than current models based on process algebras and event structures. Quartel's analysis applies to the extension of our basic concepts with integral and stochastic probability (see section 10.4.1). Van Sinderen et al. [1995] show that a subset of our architectural concepts can be mapped onto place/transition Petri nets [Reisig, 1985].

Data is often viewed as orthogonal to concepts for modelling system dynamics (see e.g. [De Weger et al., 1995]). We showed that data concepts can be defined in terms of the basic architectural concepts, which means that these concepts are not orthogonal. This insight allowed us to show in what circumstances data is useful for the concise modelling of behaviours, thus providing guidance to developers. This insight should also aid scientists and practitioners in the comparison of development methods and in the assessment of the applicability of development methods (as in e.g. [Verrijn-Stuart & Olle, 1994]). Finally, this insight should aid scientists and practitioners in the construction of development methods for systems in which both behaviour dynamics and data are important aspects. Given the rapid increase of large-scale telematics systems, in which behaviour dynamics and data play an important role, and the need for the integration of these systems in their user environments, this is an important research topic.

Modelling styles ease the structuring of system models and lead to more uniformly structured models. Our new insights regarding the basic architectural concepts and the role of data enabled us to extend the modelling styles for distributed systems of Vissers et al. [1988] and to define them in such a way that they are more orthogonal. Our distinction of frequently occurring structures of business processes enabled us to define concrete modelling styles for business processes, which are specialisations of the generic modelling styles. These styles aid developers in efficiently modelling business processes in an insightful way, both intensionally and extensionally.

The relation between the characteristics of development strategies and the need for the support of particular development objectives (often formulated as risks of development projects) is important for the selection of an appropriate development strategy [Boehm, 1988]. Little research, however, has been carried out in this area. Our analysis of development strategies led to an understanding of the extent to which (elements of) development strategies support particular development objectives. This relation can be used in practice to select or construct a development strategy that is appropriate for a development process. Our results can also be used as a foundation for further research in this area, which should lead to more concrete support for developers.

Many development methods used in practice are almost solely based on the induction of practical experience [Simon, 1996]. This results in development methods that provide little guidance to developers in controlling complexity. We have shown how development methods can alternatively be constructed by means of the synthesis and refinement of structuring techniques that are selected on the basis of the need for support of particular development objectives. Practical experience guided the development of our structuring techniques. Therefore, development methods constructed in this way can both be concrete and aid developers in controlling the complexity of business processes and their development.

10.3 Practical application

10.3.1 Background

In the past years, our project team has modelled, analysed, and/or re-designed about ten business processes of major organisations in the service industry. Most of these projects were carried out in the Testbed project [Franken, 1997]. All development projects concerned administrative business processes, such as the claim processing of an insurance company and the money transfer of a bank. One project involved the development of an architecture for the business processes of a large company. For reasons of confidentiality we cannot present details of these business processes in this thesis.

This section summarises the experiences with the practical use of our structuring techniques. We refer to the developers of business processes as the *users* of our structuring techniques.

10.3.2 Architectural concepts and their representation

Architectural concepts

The basic and derived architectural concepts presented in this thesis cover all relevant aspects of business processes. Although there appeared to be a constant need for users to construct their own derived concepts (see below), no additional basic concepts are required. Because the concepts are proper abstractions of system elements, and because the concepts are consistent and generally applicable, most of these concepts were readily used.

The action concept appeared to be intuitive and was readily used. Action decomposition was also applied without serious problems. Some users had to get used to the fact that an action models the result of an activity, and has therefore no duration.

The interaction concept was also readily used. Interaction decomposition, however, was initially applied far less than action decomposition. A possible explanation is that most users have previously been exposed to concepts that are similar to abstract

actions (for example, the processes of data flow diagrams, which can be decomposed into related sub-processes), but not to concepts that are similar to abstract interactions (for example, the events of extended data flow diagrams, which share some properties with interactions, cannot be decomposed into sub-events). Because of this, users often only specified concrete interactions, even when other actions were specified at a high abstraction level. For similar reasons, users often only specified value passing and did not use value checking and value generation. Since many users are not used to this abstract, but powerful, use of interactions, they need to be educated, such that the usefulness of the concept is made clear (see section 10.3.3).

The specification of causality relations gave little problems. The enabling relation and the disabling relation were intuitive for most users and were readily used. The synchronisation relation was hardly used in practice, because most users felt they did not need it. Complex causality relations, consisting of many disjunctions and conjunctions of elementary causality conditions, did not occur often in practice.

When we started to apply our concepts to business processes, the concepts of state variable and state value were not yet available. However, it quickly became apparent that there was a need for such concepts to model data bases, archives, etc. This led to the introduction of a concept called “item”, whose relation to the basic concepts was not well defined. Our current insights allow an item to be defined as an entity whose behaviour is defined in a fully state-oriented style.

Many users want to define their own derived concepts. Such derived concepts are combinations and/or specialisations of the basic architectural concepts. Derived concepts are sometimes called (design) patterns (e.g. [Coad, 1992; Ould, 1995]). One of the definitions of “pattern” given in [Webster’s] and used by Coad is: “a ... model proposed for imitation: something regarded as a normative exemplar to be copied”. A pattern is thus a generic, re-useable model of a system part or aspect. Currently, many patterns are being developed for object-oriented development (see e.g. [Gamma et al., 1995]). Some patterns based on our architectural concepts can be found in [Bal et al., 1997a]. Tool support for the definition of patterns, combined with a library of patterns, should improve the effectiveness of the development of business process models.

Parsimony appeared to be a very important quality criterion for the architectural concepts. Too many architectural concepts easily confuse users and one should therefore always strive for a minimal set of orthogonal architectural concepts.

Representation

The graphical representation of the architectural concepts is usually more comprehensible than the textual representation. However, some model constructions are hard to represent graphically. An example is the interaction of repetitive behaviours. This is why we represented the remote constraints in the behaviour of the applica-

tion example of chapter 9 textually, whereas all other aspects of the behaviour were represented graphically.

Some specifications of business processes are complex, due to the large amount of information they represent, even when the structuring techniques discussed in this thesis are applied. For example, behaviour specifications may become complex if not only (inter)actions and causality relations are represented, but also all action values, their data types, state values, and reference relations. This is why these data concepts were hardly specified in the application example of chapter 9. A specification of the assignment of behaviour to entities is potentially complex as well. We expect that a proper editing tool, in which the related aspects of business processes can be viewed and edited separately, aids in overcoming the difficulties caused by this complexity.

Further guidelines for the representation of business processes appeared to be useful. Examples of these guidelines concern the use of colour for different types of entities and actions, the representation of the name of an action inside the circle that represents the action (rather than in a text box attached to the action), and the drawing of arrows from left to right as much as possible. See [Bal et al., 1997a] for more examples.

10.3.3 Modelling styles

The monolithic style was used to model simple sub-behaviours of business processes. The modular style was used for more complex business processes.

The causality-oriented style appeared to be intuitive and was readily used. Initially, the constraint-oriented style was not used much, since it requires the use of the (abstract) interaction concept. After some time, however, when the users got accustomed to the interaction concept, the constraint-oriented style was used more frequently. The usefulness of the work-flow-oriented style, a specialisation of the constraint-oriented style, also promoted the use of the constraint-oriented style.

Both the work-flow-oriented style and the business function-oriented style appeared to be useful, since they are based on concretely perceived structures of business processes. Since many companies nowadays structure their business processes around work-flows, the work-flow-oriented style was used more than the business function-oriented style. Until recently, many users described business processes only in terms of the local constraints with respect to work-flows. Partly this is due to the difficulty they have in modelling remote constraints, but in many cases users do not even realise the existence of the remote constraints. We expect that with more concrete support for the identification and modelling of remote constraints with respect to work-flows (as in the development method of chapter 9), the construction of complete business process models can be better supported.

10.3.4 Abstraction levels

Two abstraction levels were most frequently used: the integrated system perspective and the distributed system perspective. The distinction between the two abstraction levels was clear. Particularly the integrated system perspective gave many users unexpected insight. Few employees, and even few higher managers, appeared to have an overview of their business processes and insight in the service these processes provide. Since our architectural concepts support (in contrast to many other modelling techniques) extensional modelling, the modelling of a business process from the integrated perspective was well supported. The (slightly incorrect) presentation of the integrated perspective as the "viewpoint of the customer" much promoted the use of this abstraction level.

Other abstraction levels have been little used in practice until now, mainly because they were not yet integrated in the development methods.

We expect that particularly the refinement of the distributed system perspective into the logically distributed system perspective and the physically distributed system perspective will be useful in the development of complex business processes. Given the emphasis many methods for business process re-design put on (initial) abstraction from physical distribution, the distinction between the logically distributed system perspective and the physically distributed system perspective may be perceived to be particularly useful in projects in which business processes are re-designed.

We also expect the perspective of the system embedded in its environment to be useful in practice, since users often find it difficult to properly delimit the system under development. Such difficulties occur particularly in cases where entities in the environment (e.g. insurance agents) contribute significantly to the service of an organisation.

There is a need among users for the support of consistency checking of models at different abstraction levels. In chapter 6 we presented a method for doing so. We hope that automated tool support for parts of this method will give users more support for consistency checking.

10.3.5 Development strategies

Initially some users attempted to model an existing business process according to a bottom-up strategy, but this attempt was quickly aborted for the reason stated in chapter 7: a bottom-up strategy provides too little support for separation of concerns, and is therefore of limited use in the development of complex business processes.

After that, all business processes were developed according to the waterfall model or the evolutionary strategy. The waterfall model was frequently used for the modelling of existing business processes, since it is best suited to development processes in which requirements do not change. The evolutionary strategy was frequently used

for the development of new business processes for which requirements might change.

The method for the selection of development strategies of chapter 7 was applied to evaluate the development strategy selected in an information system development project. By analysing the risks of the project, as they were estimated at the start of the project, it was identified that different parts of the development project needed different basic development strategies. Since the development project used a single basic development strategy (the waterfall model), the difference between the strategy that was actually used and the required strategies could be used to explain certain costly failures made in the development project. See [De Weger & Franken, 1996] for a more detailed account of this topic.

10.3.6 Development methods

Practice showed that there is a need for concrete development methods that guide users step by step through a development process. Initially, when such methods were not yet available, users could only develop business processes under the guidance of experienced developers. The mere provision of a set of structuring techniques inevitability resulted in the question: "what should we do next?".

Following a scientific approach, a development method should be presented in an abstract way, to prevent unnecessary and limiting design decisions regarding the method, and to show which aspects of a method are potentially common to other methods as well. For users, however, it is important that a development method is presented as concretely as possible. Users require a method that directly aids in solving the concrete problems they perceive. Structuring techniques that are perceived by the users as too general require interpretation by these users, which is often too hard for them.

During the various development projects it became apparent that users require concrete design criteria in order to determine what constitutes a "good" business process. Moreover, extra guidance was required for such issues as how to select and delimit business processes, when to stop refining models of business processes, and how to identify use-cases and work-flows. We provided this guidance by means of design criteria and heuristics.

10.4 Further research

10.4.1 Topics of this thesis

For certain purposes, for example, the analysis of particular quantitative properties of business processes (see section 10.4.2), one may need to model the probability of action occurrences more precisely than it is possible with the uncertainty attribute introduced in chapter 3. Quartel [1998] presents two possible refinements of uncer-

tainty: integral probability and stochastic probability. In the case of integral probability, the probability of an action occurrence is expressed as a real number in the range from 0 to 1. In the case of stochastic probability, the distribution of the probability of an action occurrence over time is considered as well.

Quartel also provides a formal semantics for the architectural concepts presented in this thesis. A textual syntax and a graphical syntax are being developed for the representation of our architectural concepts. Particular attention should be paid to the graphical syntax, because the graphical representation appears to be the most intuitive one, and because some modelling constructs cannot yet be represented graphically in an insightful way.

Further practical application will have to show the usefulness of the abstraction levels that have not yet been used in practice. Practical application should also lead to the further identification of specific structures of business processes, thus enabling the development of modelling styles and design patterns. Further research and practical application is required to improve the applicability of our method for the selection and construction of development strategies and to link it to current methods for risk analysis. Finally, practical application should lead to the provision of further design criteria and heuristics.

10.4.2 Quantitative analysis

This thesis has mainly focused on obtaining qualitative insight in business processes. Insight in quantitative properties, e.g. response time, utilisation, and production costs, is also important in business process development. Research is required to find out:

- exactly which quantitative properties are most relevant for analysis;
- how to compute the values of these quantitative properties efficiently using automated support tools.

In [Franken et al., 1997b] we report on initial research on the first issue. In the article we focus on the relation between particular structures of business processes and certain temporal properties of business processes, like response time, completion time, and throughput. Insight in this relation should ease the provision of quantitative insight in selected aspects of business processes, like a particular work-flow or a particular business function, and should clarify what information is required to compute the values of these properties.

10.4.3 Automated support tools

An automated support tool is a computer program that supports a developer in carrying out development steps by automating parts of them. The following automated support tools can be useful.

- An *editor* that eases the development of a specification of a business process. The editor should support both the graphical and the textual representation of a business process and should be able to convert a graphical representation into a textual representation, and vice versa. It should also support the separate editing of different aspects of a business process (e.g. entities and behaviour; behaviour dynamics and data). It should also support the refinement of system models, thus enabling the development of consistent models at different abstraction levels. Finally, it should support the use of modelling styles and design patterns.
- A *simulator* that allows a business process to be simulated, thus enabling developers and other people involved in a development process to analyse particular aspects of the business process.
- A *model checker* that allows one to verify the truth of certain propositions regarding a model. Examples of such propositions are “an order is always followed by a confirmation” and “the model contains no deadlock”.
- One or more *quantitative analysis tools* (or: performance analysis tools) that allow one to analyse certain quantitative properties of a model.

An editor has already been developed. Most of the other tools are planned to be developed in the Testbed project [Franken, 1997].

10.4.4 Implementation

This thesis focuses on the structuring of business processes and pays little attention to their implementation. Implementation is, of course, an essential part of development: the proof of the pudding.

An important difference between business processes and software is that business processes are partly carried out by humans. Attention should therefore be paid to the social and psychological aspects of business processes. This attention is even more important in case a business process is re-designed and migration to a new situation is necessary, since it is in the nature of many people to resist to changes. Human resource management and change management, including the education and motivation of people, is therefore crucial in the implementation of business processes [Betz et al., 1995].

People carrying out business processes are usually supported by automated systems. When a business process is re-designed these systems are frequently adapted or re-designed. Parts of the re-design of these systems, such as data conversion, can possibly be automated. Research is required to find out to what extent this is possible and how it should be done.

10.4.5 Telematics systems

Several of the structuring techniques described in this thesis are derived from and thus applicable to telematics systems as well. This is convenient, since telematics systems often enable business process re-design. The development of a telematics system is much eased if it can be carried out according to a method similar to the one used for the development of the business process to be supported by this system.

In order to provide more specific support for developers of telematics systems, our structuring techniques should be refined and support specific structures of telematics systems (see e.g. [Vissers et al., 1994]). Ferreira Pires [1994] applies architectural concepts that are similar to ours to the development of various telematics systems. Van Sinderen [1995] applies these concepts to the development of application protocols. Quartel [1998] specifies, among other things, the OSI Transport Service [ISO, 1986].

The specification of an implementation of a business process may serve as a specification of the services to be provided by one or more supporting telematics systems. Some telematics systems for which standard building blocks exist, such as work-flow management systems, may partly be generated automatically on the basis of a specification of their service [Leymann & Altenhuber, 1994]. Research is required to investigate how this should be done.

References

Alexander, Christopher. (1964). *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, MA.

Alexander, Linda C. & Davis, Alan M. (1991). Criteria for selecting software process models. In: Knafl, George J. (ed.), *The Fifteenth Annual International Computer Software & Applications Conference*, pp. 521-528. IEEE Computer Society Press, Los Alamos, CA.

Allen, Paul M. (1994). *Reengineering the Bank. A blueprint for survival and success.* Probus Publishing, Chicago, Il.

Aue, Alfred & Breu, Michael. (1994). Distributed information systems: an advanced methodology. *IEEE Transactions on Software Engineering*, 20(8), pp. 594–605.

Auramäki, Esa, Lehtinen, Erkki & Lyytinen, Kalle. (1988). A speech-act-based office modelling approach. *ACM Transactions on Office Information Systems*, 6(2), pp. 126-152.

Bal, René, Sinderen, Marten van & Teeuw, Wouter. (1997a). *Richtlijnen ten behoeve van het modelleren in Testbed.* Testbed Notitie WP3/N006/V002. Telematics Research Centre, Enschede, The Netherlands.

Bal, René, Jonkers, Henk & Oldenkamp, Johan. (1997b). *Methode voor use-case-geörienteerde case-uitwerking.* Testbed Notitie WP1/N001/V002. Telematics Research Centre, Enschede, The Netherlands.

Balzer, R.M. (1985). A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, November 1985, pp. 1268-1357.

Beinhocker, Eric D. (1997). Strategy at the edge of chaos. *The McKinsey Quarterly*, 1997(1), pp. 24-39.

Betz, Berend, Roelofs, José & Vrins, Jan. (1995). *Integraal ontwikkelen van organisatie en informatiesystemen.* Kluwer Bedrijfswetenschappen, Deventer, The Netherlands.

Blaauw, Gerrit A. & Brooks, Frederick P. (1980). *Computer Architecture.* Lecture Notes. Technische Hogeschool Twente, Enschede, The Netherlands. Revised

version published in: Blaauw, Gerrit A. & Brooks, Frederick P. (1997). Computer Architecture: Concepts and Evolution. Addison-Wesley, Reading, MA.

Boehm, Barry W. (1981). *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ.

Boehm, B.W., Gray, T.E. & Seewaldt, T. (1984). Prototyping vs. specifying: a multi-project experiment. *IEEE Transactions on Software Engineering*, May 1984, pp. 133-145.

Boehm, Barry W. (1987). Improving software productivity. *IEEE Computer*, September 1987, pp. 43-57.

Boehm, Barry W. & Papaccio, Philip N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, October 1988, pp. 1462-1477.

Boehm, Barry W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, May 1988, pp. 61-70.

Bolognesi, Tomasso & Brinksma, Ed. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, pp. 25-59.

Booch, Grady. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA.

Bowen, Dyfed. (1991). Open distributed processing. *Computer Networks and ISDN Systems*, 23, pp. 195-201.

Brand, N.A. & Kolk, J.R.P. van der. (1995). *Werkstroomanalyse en -ontwerp. Het logistiek vriendelijk ontwerpen van informatiesystemen*. Kluwer Bedrijfswetenschappen, Deventer, The Netherlands.

Brooks, Frederick P. (1995). *The Mythical Man-Month. Essays on Software Engineering.* Anniversary edition with four new chapters. Addison-Wesley, Reading, MA.

Buitelaar, Michiel & Groen, Udo. (1994). Business process redesign: een nieuwe kijk op automatisering?. *Informatie*, 36-6, pp. 388-397.

Bunyard, Jerry Max & Coward, James Mike. (1982). Today's risk in software development—can they be significantly reduced? Concepts: The Journal of Defence Systems Acquisition, 5(4), pp. 73-94. Re-printed in: Thayer, Richard H. (ed.), (1988), *Tutorial: Software Engineering Management*, pp. 75-90. IEEE Computer Society Press, Washington, DC.

Car, David K. & Johansson, Henry J. (1995). *Best practices in reengineering. What works and what doesn't in the reengineering process*. McGraw-Hill, New York, NY.

Carnap, Rudolf. (1956). *Meaning and Necessity. A Study in Semantics and Modal Logic*. The University of Chicago Press, Chicago, Il.

Chapple, E.D. & Sayles, L.R. (1961). *The Measure of Management*. MacMillan, London.

Coad, Peter. (1992). Object-oriented patterns. *Communications of the ACM*, 35(9), pp. 152-159.

Complex Systems. (1987). *Complex Systems: a journal devoted to the rapid publication of research on the science, mathematics and engineering of systems with simple components, but complex overall behaviour*, Vol. I. Complex Systems Publications, Champaign, Il.

Cori, Kent A. (1985). Fundamentals of master scheduling for the project manager. *Project Management Journal*, June 1985, pp. 78-89.

Courtois, P.-J. (1985). On time and space decomposition of complex structures. *Communications of the ACM*, June 1985, pp. 590-603.

Coyne, Kevin P., Hall, Stephen J.D. & Gorman Clifford, Patricia. (1997). Is your core competence a mirage?. *The McKinsey Quarterly*, 1997(1), pp. 24-39.

Davenport, Thomas H. & Short, James E. (1990). The new industrial engineering: information technology and business process re-design. *Sloan Management Review*, Summer 1990, pp. 11-27.

Davenport, Thomas H. (1993). *Process Innovation. Reengineering Work through Information Technology*. Harvard Business School Press, Boston, MA.

Davenport, Thomas H. & Nohria, Nitin. (1994). Case management and the integration of labor. *Sloan Management Review*, Winter 1994, pp. 11-23.

Davis, Alan M., Bersoff, Edward H. & Comer, Edward R. (1988). A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, October 1988, pp. 1453-1461.

Davis, G.B. (1982). Strategies for information requirements determination. *IBM Systems Journal*, 21(1), pp. 4-30.

Dawson, C.W. & Cartwright, T. (1996). Improving the software process with hybrid development models. In: Bray, M., Ross, M. & Staples, G. (eds.), *Software Quality Management: Improving Quality, Proceedings of the Fourth International Conference on Software Quality Management*, pp. 461-470. Mechanical Engineering Publications, London.

DeMarco, Tom. (1979). *Structured Analysis and Systems Specification*. Prentice Hall, Englewood Cliff, NJ.

Earl, Michael J. (1994). The new and the old of business process redesign. *Journal of Strategic Information Systems*, 3(1), pp. 5-22.

Ehrig, H. & Mahr, B. (1985). *Fundamentals of Algebraic Specification I*. Springer-Verlag, Berlin.

Essink, L.J.B. (1986). A modelling approach to information systems development. In: Olle, T.W., Sol, H.G. & Verrijn-Stuart, A.A. (eds.), *Information System Design Methodologies: Improving the Practice. Proceedings of the IFIP WG8.1 Working Conference*, pp. 55-86. North-Holland, Amsterdam.

Ferreira Pires, Luís (ed.). (1992). *The Lotosphere Design Methodology: Basic Concepts*. Lotosphere deliverable Lo/WP1/T1.1/N0045/V04. ESPRIT Project 2304, Brussels.

Ferreira Pires, Luís. (1994). *Architectural Notes: a Framework for Distributed Systems Development*. Ph.D. Thesis. University of Twente, Enschede, The Netherlands.

Franken, Henry M., Jonkers, Henk, Quartel, Dick A.C., Sinderen, Marten J. van, Teeuw, Wouter B., Vissers, Chris A. & Weger, Mark K. de. (1996a). *Basis voor BPR-modellering*. Testbed deliverable 3.1, Testbed/WP3/D3.1. Telematics Research Centre, Enschede, The Netherlands.

Franken, H.M., Weger, M.K. de, Quartel, D.A.C. & Ferreira Pires, L. (1996b). On engineering support for business process modelling and redesign. In Doumeingts, G. & Browne, J. (eds.), *Modelling techniques for business process re-engineering and benchmarking. IFIP TC5 WG5.7 International Workshop on Modelling Techniques for Business Process Re-engineering and Benchmarking*, pp. 103-120. Chapman and Hall, London. Also published as: CTIT Technical Report 96-34. University of Twente, Enschede, The Netherlands.

Franken, Henry, Jonkers, Henk, Mulaert, Ferial, Sinderen, Marten van, Teeuw, Wouter & Weger, Mark de. (1997a). *Ontwerprichtlijnen voor toepassing van AMBER voor bedrijfsprocesmodellering*. Testbed Deliverable WP3/D3.3/V05. Telematics Research Centre, Enschede, The Netherlands.

Franken, Henry M., Jonkers, Henk & Weger, Mark K. de. (1997b) Structural and quantitative perspectives on business process modelling and analysis. In: Kaylan, Ali Riza & Lehman, Axel (eds.), *Proceedings of the 11th European Simulation Multiconference ESM'97*, pp. 595-599. The Society for Computer Simulation, San Diego, CA.

Franken, H.M. (1997). A virtual test environment for business processes. *ACM Bulletin of Special Interest Group on Supporting Group Work*, April 1997, 18(1), pp. 63-67.

Franken, H.M. & De Weger, M.K. (1997). A modelling framework for capturing business process dynamics. To be published in: *Journal of Business Change and Re-engineering*.

Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John. (1995). *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA.

Ganzevoort, J.W. (1985). Ontwerpen en ontwikkelen, de veranderkundige dimensies van het organiseren. *Management & Organisatie*, 39(1).

Gotzhein, R. (1993). *Open Distributed Systems. On concepts, methods, and design from a logical point of view*. Vieweg Advanced Studies in Computer Science, Vieweg, Wiesbaden, Germany.

Hammer, Michael. (1990). Reengineering work: don't automate, obliterate. *Harvard Business Review*, July-August 1990, pp. 104-112.

Hammer, Michael & Champy, James. (1993). *Reengineering the corporation*. Paperback edition. HarperCollins, New York, NY.

Harmsen, Frank, Brinkkemper, Sjaak & Oei, Han. (1994). Situational method engineering for information system project approaches. In: Verrijn-Stuart, A.A. & Olle, T.W. (eds.), *Methods and Associated Tools for the Information Systems Life Cycle. Proceedings of the IFIP WG8.1 Working Conference on Methods and Associated Tools for the Information Systems Life Cycle*, pp. 169-194. Elsevier Science Publishers, Amsterdam.

Hennessy, Pippa, Harvey, Paul & Smith, Hugh. (1994). Support for enterprise modelling in CSCW. *Collaborative Computing*, 1, pp. 127-146.

Hirsch, E. (1985). Evolutionary acquisition of command and control systems. *Program Manager*, November-December 1985, pp. 18-22.

Horn, Stephen. (1996). *US Federal Government Year 2000 Survey*. Statement for the One Hundred Fourth Congress, Congress of the United States, July 30, 1996.

IBM. (1995). *LOVEM Consultant's Guide version 3.0*. IBM Canada Ltd., Toronto.

ISO. (1983). *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*. International Standard ISO 7498-4. International Organisation for Standardization, Geneva.

ISO. (1986). *Information Processing Systems - Open Systems Interconnection - Transport service definition*. International Standard ISO 8072. International Organisation for Standardization, Geneva.

ISO. (1987a). *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. DIS 8807. International Organisation for Standardization, Geneva.

ISO. (1987b). *Quality - Vocabulary*. International Standard ISO 8402. International Organisation for Standardization, Geneva.

Jackson, M. (1983). *System Development*. Prentice-Hall, Englewood Cliffs, NJ.

Jacobson, I., Ericsson, M. & Jacobson, A. (1994). *The object advantage: business process reengineering with object technology*. ACM Press Books, Amsterdam.

Johansson, Henry J., McHugh, Patrick, Pendlebury, A. John & Wheeler III, William A. (1993). *Business process reengineering. Breakpoint strategies for market dominance*. John Wiley & Sons, Chichester, UK.

Kaplan, Robert B. & Murdock, Laura. (1991). Core process redesign. *The McKinsey Quarterly*, 1991(2), pp. 27-43.

Kerklaan, Leo & Knaapen, Rob. *Kwaliteit in Kader*. Kluwer Technische Boeken, Deventer, The Netherlands.

Kernighan, Brian & Ritchie, Dennis. (1988). *The C Programming Language, second edition*. Prentice-Hall, Englewood Cliffs, NJ.

Kremer, Harro. (1995). *Protocol Implementation. Bridging the gap between Architecture and Realization*. Ph.D. Thesis. University of Twente, Enschede, The Netherlands.

Kronlöf, Klaus. (1992). *Method integration. Concepts and case studies*. John Wiley & Sons, Chichester, UK.

Kumar, K. & Welke, R.J. (1992). Methodology engineering: a proposal for situation-specific methodology construction. In: Cotterman, W.W. & Senn, J.A. (eds.), *Challenges and Strategies for Research in Systems Development*. John Wiley & Sons, Chichester, UK.

Kuutti, Kari. (1991). The concept of activity as a basic unit of analysis for CSCW research. In: Bannon, Liam, Robinson, Mike & Schmidt, Kjeld (eds.), *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Leymann, F. & Altenhuber, W. (1994). Managing business processes as an information resource. *IBM Systems Journal*, 33(2), pp. 326-348.

Linington, P.F. (1992). Introduction to the open distributed processing basic reference model. In: De Meer, J, Heymer, V. & Roth, R. (eds.), *Open Distributed Processing*, pp. 3-14. Elsevier Science Publishers, Amsterdam.

Lundeberg, M. (1982). The ISAC approach to specification of information systems and its application to the organization of an IFIP working conference. In: Olle, T.W., Sol, H.G. & Verrijn-Stuart, A.A. (eds.), *Information System Design Methodologies: a Comparative Review. Proceedings of the IFIP TC 8 Working Conference on Information Systems Design Methodologies*, pp. 173-234. North-Holland, Amsterdam.

Luqi, R. (1989). Software evolution through rapid prototyping. *IEEE Computer*, May 1989, pp. 13-25.

MacDonald, I.G. (1986). Information Engineering: an improved automatable methodology for the design of data sharing systems. In: Olle, T.W., Sol, H.G. & Verrijn-Stuart, A.A. (eds.), *Information System Design Methodologies: Improving the Practice. Proceedings of the IFIP WG8.1 Working Conference*, pp. 173–224. North-Holland, Amsterdam.

Maddison, R.N. (1983). *Information System Methodologies*. The British Computer Society/Wiley Heyden, Chichester, UK.

Maull, Roger, Childe, Stephen, Bennet, Jan, Weaver, Adam & Smart, Andi. (1994). *Report on Good Practice in Business Process Re-engineering in Manufacturing Industry.* Working paper EPSRC WP/GR/J95010-2. Engineering and Physical Sciences Research Council, Swindon, UK.

McCracken, D.D. & Jackson, M.A. (1982). Life cycle concept considered harmful. *ACM Software Engineering Notes*, April 1982, pp. 29-32.

Medina-Mora, Raúl, Winograd, Terry, Flores, Rodrigo & Flores, Fernando. (1992). The action workflow approach to workflow management technology. In: Turner, Jon & Kraut, Robert (eds.), *CSCW '92: Sharing Perspectives. Proceedings of the Conference on Computer-Supported Cooperative Work*, pp. 281-288. ACM Press, New York.

Minsky, Marvin L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, London.

Minsky, Marvin. (1985). *The Society of Mind*. Simon and Schuster, New York, NY.

Mintzberg, Henry. (1979). *The Structuring of Organizations*. Prentice-Hall, Englewood Cliffs, NJ.

Mitrani, I. (1982). *Simulation techniques for discrete event systems*. Cambridge University Press, Cambridge, UK.

Ould, Martyn A. (1995). *Business Processes. Modelling and analysis for re-engineering and improvement*. John Wiley & Sons, Chichester, UK.

Pall, G.A. (1987). *Quality Process Management*. Prentice-Hall, Englewood Cliffs, NJ.

Parkinson, G.N. (1957). *Parkinson's Law and Other Studies in Administration*. Houghton-Mifflin, Boston, MA.

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), pp. 1053-1058.

Partsch, H. & Steinbruggen, R. (1983). Program transformation systems. *ACM Computing Surveys*, September 1983, pp. 199-236.

Pirsig, Robert M. (1991). *Lila, an Inquiry into Morals*. Bantam Press, London.

Porter, M.E. (1980). *Competitive Strategy*. Free Press, New York, NY.

Pressman, Roger S. (1992). *Software Engineering: a practitioner's approach*. McGraw-Hill, New York, NY.

Quartel, Dick, Ferreira Pires, Luís & Van Sinderen, Marten. (1996). *On architectural support for behaviour refinement in distributed systems*. Unpublished manuscript. Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands.

Quartel, Dick A.C., Ferreira Pires, Luís, Sinderen, Marten van, Franken, Henry M. & Vissers, Chris A. (1997). On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, 29, pp. 413-436.

Quartel, Dick A.C. (1998). *Actions relations. Basic design concepts for behaviour modelling and refinement*. Ph.D. thesis, University of Twente, Enschede, The Netherlands. To be published.

Ramackers, Guus J. (1994). Model integration and model execution. In: Verrijn-Stuart, A.A. & Olle, T. William (eds.), *Methods and Associated Tools for the Information Systems Life Cycle. Proceedings of the IFIP WG8.1 Working Conference*, pp. 223-239. Elsevier Science Publishers, Amsterdam, The Netherlands.

Rechtin, Eberhardt. (1991). *Systems Architecting. Creating & Building Complex Systems*. Prentice Hall, Englewood Cliffs, NJ.

Rechtin, Eberhardt. (1992). The art of systems architecting. *IEEE Spectrum*, October 1992, pp. 66-69.

Reisig, W. (1995). *Petri Nets - An Introduction*. EATCS Monographs on Theoretical Computer Science, Vol. 4. Springer-Verlag, Berlin.

Roman, Gruia-Catalin. (1985). A taxonomy of current issues in requirements engineering. *IEEE Computer*, April 1985, pp. 14-22.

Rothenberg, Jeff. (1989). The nature of modelling. In: Widman, Lawrence E., Loparo, Kenneth A. & Nielsen, Norman R. (eds.), *Artificial Intelligence, Simulation, and Modelling*, pp. 75-92. John Wiley & Sons, New York, NY.

Royce, W.W. (1970). Managing the development of large software systems: concepts and techniques. Proceedings IEEE WESCON 1970, pp. 1-9. Re-printed in: Thayer, Richard H. (ed.). (1988). *Tutorial: Software Engineering Management*, pp. 118-127. IEEE Computer Society Press, Washington, DC.

Scharpf, F.W. (1977). Does organization matter? Task structure and interaction in the ministerial bureaucracy. In: Burack, E.H. & Negardhi, A.R. (eds.), *Organization Design: Theoretical Perspectives and Empirical Findings*. Kent State University Press, Kent, pp. 149-167.

Schot, Jeroen. (1992). *The role of Architectural Semantics in the formal approach of Distributed Systems Design*. Ph.D. thesis. University of Twente, Enschede, The Netherlands.

Simon, Herbert A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society*, 106, pp. 467-482. Re-printed with modifications in: [Simon, 1981].

Simon, Herbert A. (1981). *The Sciences of the Artificial, Second Edition*. The MIT Press, Cambridge, MA.

Simon, Herbert A. (1996). *The Sciences of the Artificial, Third Edition*. The MIT Press, Cambridge, MA.

Sinderen, Marten van, Ferreira Pires, Luís, Vissers, Chris A. & Katoen, Joost-Pieter. (1995). A design model for open distributed processing systems. *Computer Networks and ISDN Systems*, 27, pp. 1263-1285.

Sinderen, Marten van. (1995). *On the Design of Application Protocols*. Ph.D. thesis. University of Twente, Enschede, The Netherlands.

Sowa, J.F. & Zachman, J.A. (1992). Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, 31(3), pp. 590-616.

Stanat, Donald F. & McAllister, David F. (1977). *Discrete Mathematics in Computer Science*. Prentice-Hall, Englewood Cliffs, NJ.

Stein, D. (ed.). (1989). *Lectures in the Sciences of Complexity, SFI Studies in the Sciences of Complexity*, Vol. I. Addison-Wesley, Reading, MA.

Trivedi, K.S. Reliability evaluation for fault-tolerant systems. (1984). In: Iazolla, G., Courtois, P.J. & Hordijk, A. (eds.), *Mathematical Computer Performance and Reliability*, pp. 403-414. North-Holland, Amsterdam.

Venkatraman, N. (1994). IT-enabled business transformation: from automation to business scope redefinition. *Sloan Management Review*, Winter 1994, pp. 73-87.

Verrijn-Stuart, A.A. & Olle, T. William (eds.). (1994). *Methods and Associated Tools for the Information Systems Life Cycle. Proceedings of the IFIP WG8.1 Working Conference*. Elsevier Science Publishers, Amsterdam, The Netherlands.

Vissers, C.A. (1977). *Interface. Definition, Design, and Description of the Relation of Digital System Parts*. Ph.D. thesis. Technische Hogeschool Twente, Enschede, The Netherlands.

Vissers, C.A. (1983). Architectural requirements for the temporal ordering specification of distributed systems. In: Kalin, T. (ed.), *Proceedings of the European Teleinformatics Conference (EUTECO)*, Varese, 1983, pp. 79-97.

Vissers, Chris A. & Logrippo, Luigi. (1986). The importance of the service concept in the design of data communications protocols. In: Diaz, M. (ed.), *Protocol Specification, Testing, and Verification, V*, pp. 3-17. North-Holland, Amsterdam.

Vissers, C.A. & Scollo, G. (1988). *The Architecture of Interaction Systems*. Lecture notes. University of Twente, Enschede, The Netherlands.

Vissers, Chris A., Scollo, Giuseppe & van Sinderen, Marten. (1988). Architecture and specification style in formal descriptions of distributed systems. In: Aggarwal, S. & Sabnani, K. (eds.), *Protocol Specification, Testing, and Verification VIII*, pp. 189-204. IFIP, North Holland, Amsterdam.

Vissers, C.A., Scollo, G., van Sinderen, M. & Brinksma, E. (1991). Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89, pp. 179-206.

Vissers, Chris A., Sinderen, Marten van & Ferreira Pires, Luís. (1993). What makes industries believe in formal methods. In: Danthine, André, Leduc, Guy & Wolper, Pierre, *Protocol Specification, Testing, and Verification XIII*, pp. 3-26. Elsevier Science Publishers, Amsterdam.

Vissers, Chris A., Ferreira Pires, Luís & Quartel, Dick A.C. (1994). *Design of Telematics Systems*. Lecture notes. University of Twente, Enschede, The Netherlands.

Vissers, Chris A., Ferreira Pires, Luís & Lagemaat, Jeroen van de. (1995). Lotosphere, an attempt towards a design culture. In: Bolognesi, Tomasso, Lagemaat, Jeroen van de & Vissers, Chris, *LOTOSphere: Software Development with LOTOS*, pp. 3-28. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Wand, Y. & Weber, R. (1993). On the ontological expressiveness of information systems analysis and design grammars. *Journal of Information Systems*, 3, pp. 217-237.

Webster's. (1986). *Webster's Third New International Dictionary of the English Language*. Merriam-Webster, Springfield, MA.

Weger, Mark K. de, Knol, Henk-Jan & Essink, Leo J.B. (1994). Improving quality assurance in large information systems departments. In: Ross, M., Brebbia, C.A., Staples, G. & Stapleton, J. (eds.), *Software Quality Management II, Vol. I: Managing Quality Systems*, pp. 179-192. Computational Mechanic Publications, Southampton, UK.

Weger, Mark K. de & Vissers, Chris A. (1994). Issues in design methodologies for distributed information systems. In: Verrijn-Stuart, A.A. & Olle, T. William (eds.), *Methods and Associated Tools for the Information Systems Life Cycle.*

Proceedings of the IFIP WG8.1 Working Conference, pp. 195-208. Elsevier Science Publishers, Amsterdam, The Netherlands.

Weger, M.K. de, Franken, H.M. & Vissers, C.A. (1995a). *Architectural concepts for distributed information systems development*. Memoranda Informatica 95-20. University of Twente, Enschede, The Netherlands.

Weger, Mark K. de, Franken, Henry M. & Vissers, Chris A. (1995b). A development model for distributed information systems. In: *Proceedings First International Distributed Conference, IDC'95*, pp. 60-73. Portuguese Telecom, Portugal.

Weger, M.K. de & Franken, H.M. (1996). A situational approach to design strategies. In: Bray, M., Ross, M. & Staples, G. (eds.), *Software Quality Management IV: Improving Quality. Proceedings of the Fourth International Conference on Software Quality Management*, pp. 471-484. Mechanical Engineering Publications, London.

Weger, M.K. de. (1996). Structuring of business process models. In: Franken, H.M., Jonkers, H. & Weger, M.K. de, *Development aspects of business processes*, Platinum deliverable D3.5, PLATINUM/N021/V00, pp. 47-61. Telematics Research Centre, Enschede, The Netherlands.

Weger, Mark K. de & Franken, Henry M. (1997). A situational approach to design strategies. *Software Quality Journal*, 6(3), pp. 181-194.

Wirfs-Brock, Rebecca, Wilkerson, Brian & Wiener, Lauren. (1990). *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ.

Wohlin, C. (1994). Managing software quality through incremental development and certification. In: Ross, M., Brebbia, C.A., Staples, G. & Stapleton, J. (eds.), S*oftware Quality Management II, Vol. II: Building Quality into Software*, pp. 187-202. Computational Mechanic Publications, Southampton, UK.

Yourdon, Edward. (1989). *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

Index

A

abstraction, 17, 131
abstraction level, 131, 133
action, 32, 66
  basic, 67
action decomposition, 134
action distribution, 134
action point, 32
action point decomposition, 135
action point distribution, 135
action point refinement, 135
action refinement, 134
action value, 66
  assignment to action variable, 71
  establishment, 67
action variable, 70
analysis, 6
architectural concept, 7, 29
  basic, 7, 29
  data, 63
  derived, 7
architectural semantics, 7
architecture, 26

B

behaviour, 37
  monolithic, 45
  repetitive, 57
behaviour composition, 52
  causality-oriented, 53
  constraint-oriented, 58
behaviour execution, 49
business function, 115
business process, 3

C

categorisation, 17
causality condition, 38
  alternative, 46
causality relation, 38
causality-oriented style, 109
cleanliness, 22
code-and-fix, 157
completeness, 24
complexity, 13
composition, 21
conception, 5
conformance, 136
consistency, 23
constraint-oriented style, 110

D

data, 63
decomposition, 21
description, 6
design, 6
development decision, 155
development method, 7
development step, 155
  order of, 160
development strategy, 155
  basic, 156
  selection of, 176
disabling action, 40
disabling condition, 40
distributed system, 3

E

enabling action, 39
enabling condition, 39
entity, 30

entity decomposition, 135
entry point, 53
  parametrised, 99
evolutionary development, 157
exit point, 53
  parametrised, 99
extensional style, 105

F

future, 86

G

generalisation, 21
generality, 25
  dynamic, 25
  static, 25

H

history, 85
  equivalence classes of, 87

I

implementability validation, 166
implementation, 5
incremental development, 177
individualisation, 19
initial action, 38
intensional style, 107
interaction, 35, 59
  types of, 97
  value establishment in, 96
interaction point, 31
interface refinement, 146
  local, 147
  remote, 146

M

model, 5
modelling style, 103
modular style, 109
monolithic style, 108

O

open-endedness, 26
orthogonality, 23

P

prescription, 7
probability attribute, 38
process-oriented style, 112
propriety, 24

Q

quality, 22, 171

R

rapid prototyping, 158
reference, 72
reference relation, 71
reference rule, 72, 95
refinement, 19, 133
refinement operations, 134
requirements, 164
  actual, 164
  initially conceived, 164
  specified, 164
requirements validation, 166
resource control, 170
resource planning, 170
result action, 38
re-usability, 25
risk assessment, 176
risks of development processes, 175

S

separation of concerns, 159
  horizontal, 27, 159
  vertical, 27, 159
specialisation, 21
specification, 5
specification language, 7
spiral model, 179
start condition, 39
state, 87
state value, 79, 89
state-oriented style, 113
structure, 16
structure preservation, 26
structuring, 16, 26
  horizontal, 27, 159
  vertical, 27, 159
structuring technique, 9
synchronisation action, 40
synchronisation condition, 40
synthesis, 6
system, 2
  loosely coupled, 24
system perspective

distributed, 16, 138
integrated, 19, 138, 143
interaction system, 140
logically distributed, 144
physically distributed, 145
system development, 4

T

top-down, 160

U

uncertainty, 41

V

value checking, 97
value decomposition, 100
value generation, 98
value passing, 98

W

waterfall model, 157
work-flow, 115

Summary

This thesis treats the structuring of business processes. A *business process* is the set of related activities carried out by an organisation, or a part thereof, to deliver particular services to clients. Such services may include tangible products. The *structuring* of a system is the conception and subsequent representation of the system in such a way that the complexity of the system is kept manageable. Particular attention is paid to the role of structuring in system development.

This thesis treats the following structuring techniques:

- architectural concepts;
- horizontal structuring techniques (modelling styles);
- vertical structuring techniques (abstraction levels);
- development strategies.

Architectural concepts are abstractions of frequently occurring system elements and model required characteristics of these. They are the building blocks available to developers for modelling systems.

Basic architectural concepts are the elementary building blocks, from which other architectural concepts can be composed. Important basic architectural concepts include 1) entities, which are carriers of behaviour (a behaviour is a set of related actions), 2) actions, which model activities, 3) interactions, which model common activities of two or more objects, and 4) causality relations, which model relations between activities.

Data is a set of architectural concepts that comprises 1) action values, which model results of activities, 2) reference relations, which model relations between these results, and 3) state values, which are functions of action values and model one or more states of a behaviour that is being executed. It is shown in this thesis that all data concepts can be defined in terms of the basic concepts. Data concepts are thus not elementary building blocks of models, but can be used as "shorthand" concepts for the basic concepts, enabling the development of insightful structures and efficient implementations.

Horizontal structuring is the further structuring of a behaviour model at a particular abstraction level (see below). The horizontal structuring techniques considered here are *modelling styles:* sets of rules for the structuring of models to meet particular development objectives.

In the extensional style, internal (inter)actions of a behaviour are not represented, and it therefore supports the modelling of requirements for a system. In the intensional style, internal (inter)actions are represented, and it therefore supports implementation modelling.

In the monolithic style, a behaviour is not structured as a collection of related sub-behaviours, and it is therefore suitable for simple behaviours only. In the modular style, a behaviour is structured as a collection of related sub-behaviours. Two modular styles are distinguished: in the constraint-oriented style, behaviours are related by means of common actions (interactions), and in the causality-oriented style, the (non-)occurrences of one or more actions in one sub-behaviour are conditions for the occurrence of an action in another sub-behaviour.

In the state-oriented style, the occurrences of one or more actions depend on particular state values. It is suitable for the modelling of, for example, many aspects of database systems. In the process-oriented style, no action occurrences depend on state values. It is suitable for the modelling of, for example, many aspects of communication protocols.

This thesis shows how the defined styles aid in comprehensibly modelling frequently encountered structures of business processes: business processes structured around work-flows and business processes structured around business functions.

Vertical structuring is the structuring of a system in terms of related models at distinct abstraction levels. *Abstraction levels* are system perspectives, such that each model of a system at a particular abstraction level contains either more or less detail than a model of the system at another abstraction level. Two sets of abstraction levels are introduced: generic abstraction levels, which do not consider the types of entities one should take into account during a development process, and specific abstraction levels, which do consider this.

The following generic abstraction levels are distinguished. From the integrated system perspective, a system is represented as an integrated whole: neither internal actions, nor internal parts of the system are represented. From the interaction system perspective of the parts, the related internal interactions of the system parts are represented; the parts are not represented. This allows one to consider the various ways in which these internal interactions can be distributed over the parts. From the distributed system perspective, a system is represented as a collection of interacting parts.

The following specific abstraction levels are distinguished. From the perspective of the system embedded in its environment, the system and its environment are repre-

sented without a distinction between the two. The integrated system perspective was defined above. From the logically distributed system perspective, a system is represented as a collection of interacting parts, where one abstracts from the physical distribution of these parts. From the physically distributed system perspective, a system is represented as a collection of physically distributed parts. From the local interface refined system perspective, the concrete interactions of parts are additionally represented.

The combination of the horizontal and vertical structuring leads to the distinction of development steps that each regard one aspect of a system at one abstraction level.

A *development strategy* prescribes the order in which these development steps are to be carried out during a development process. For example, the waterfall model prescribes that a system is to be modelled at successively lower abstraction levels, and that the modelling at one abstraction level should be completed before proceeding to a lower level. The evolutionary strategy prescribes that aspects of a system should be modelled incrementally, and that the modelling of one increment should be completed before proceeding to another one.

The order in which development steps are carried out influences the ease of reaching various objectives of the development process. This thesis therefore analyses the support by a number of current development strategies for the following objectives: the minimisation of re-work due to changing requirements, the minimisation of re-work due to non-implementable designs, optimal resource allocation, and the quality of the system under development. Once one has determined the need for the support of each objective in a development process, the results of the analysis can be used to select or construct an appropriate development strategy.

The structuring techniques discussed above can be used as elements of methods for business process development. The following steps are required to construct such methods.

1. Selection, and possibly refinement, of the horizontal and vertical structuring techniques to be included in the method, resulting in groups of related development steps. Refinement requires the provision of further structuring techniques to provide more concrete guidance to developers of business processes.
2. Selection, and possibly refinement, of a development strategy that determines the order in which these groups of development steps are taken, thus synthesising them in a development method.
3. Further refinement of the resulting development method by the provision of heuristics, design criteria, and application examples.

Finally, this thesis presents a work-flow-oriented incremental development method for business processes as an example.

Samenvatting

Dit proefschrift behandelt de structurering van bedrijfsprocessen. Een *bedrijfsproces* is de verzameling gerelateerde activiteiten die worden uitgevoerd door een organisatie, of een deel daarvan, om bepaalde diensten te leveren aan klanten. Tastbare produkten kunnen onderdeel zijn van die diensten. De *structurering* van een systeem is de conceptie en de daaropvolgende representatie van het systeem op een dusdanige wijze dat de complexiteit van het systeem beheersbaar gehouden wordt. Bijzondere aandacht wordt besteed aan de rol van structurering in systeemontwikkeling.

Dit proefschrift behandelt de volgende structureringstechnieken:

- architecturale concepten;
- horizontale structureringstechnieken (modelleerstijlen);
- verticale structureringstechnieken (abstractieniveaus);
- ontwerpstrategieën.

Architecturale concepten zijn abstracties van veel voorkomende systeemelementen en modelleren vereiste eigenschappen daarvan. Ze vormen de bouwblokken die ter beschikking staan van ontwerpers voor het modelleren van systemen.

Architecturale basisconcepten zijn de elementaire bouwblokken, waaruit andere architecturale concepten opgebouwd kunnen worden. Belangrijke architecturale basisconcepten zijn: 1) entiteiten, de dragers van gedrag (een gedrag is een verzameling gerelateerde acties), 2) acties, die activiteiten modelleren, 3) interacties, die gezamenlijke activiteiten van entiteiten modelleren, en 4) causaliteitsrelaties, die de relaties tussen activiteiten modelleren.

Data is een verzameling architecturale concepten die onder andere bestaat uit 1) actiewaarden, die de resultaten van activiteiten modelleren, 2) referentierelaties, die de relaties tussen die resultaten modelleren, en 3) toestandswaarden, die functies van actiewaarden zijn en één of meer toestanden modelleren van een gedrag dat uitgevoerd wordt. Het wordt aangetoond in dit proefschrift dat alle data-concepten gedefinieerd kunnen worden in termen van de basisconcepten. Data-concepten zijn dus geen elementaire bouwblokken van modellen, maar kunnen gebruikt worden als "kortschrift"-concepten voor de basisconcepten die het ontwikkelen van inzichtelijke structuren en efficiënte implementaties mogelijk maken.

Horizontale structurering is de verdere structurering van een gedragsmodel op een bepaald abstractieniveau (zie hieronder). De horizontale structureringstechnieken die hier beschouwd worden zijn *modelleerstijlen*: verzamelingen regels voor de structurering van modellen om bepaalde ontwerpdoelen te bereiken.

In de extensionele stijl worden interne (inter)acties van een gedrag niet gerepresenteerd. De stijl is daarom geschikt voor het modelleren van de eisen aan een systeem. In de intensionele stijl worden interne (inter)acties wel gerepresenteerd. De stijl ondersteunt daarom het modelleren van implementaties.

In de monolithische stijl is een gedrag niet gestructureerd als een verzameling gerelateerde sub-gedragingen. De stijl is daarom enkel geschikt voor eenvoudige gedragingen. In de modulaire stijl wordt een gedrag wel gestructureerd als een verzameling gerelateerde sub-gedragingen. Twee modulaire stijlen worden onderscheiden: in de constraint-georiënteerde stijl zijn gedragingen gerelateerd door middel van gemeenschappelijke acties (interacties), en in de causaliteits-georiënteerde stijl is het (niet) gebeuren van één of meer acties in een sub-gedrag een conditie voor het gebeuren van een actie in een ander sub-gedrag.

In de toestands-georiënteerde stijl is het gebeuren van één of meer acties afhankelijk van bepaalde toestandswaarden. De stijl is geschikt voor het modelleren van, bijvoorbeeld, veel aspecten van database systemen. In de proces-georiënteerde stijl is het gebeuren van geen enkele actie afhankelijk van toestandswaarden. De stijl is geschikt voor het modelleren van, bijvoorbeeld, veel aspecten van communicatieprotocollen.

Dit proefschrift beschrijft hoe de gedefinieerde stijlen helpen bij het inzichtelijk modelleren van veel voorkomende structuren van bedrijfsprocessen: bedrijfsprocessen die zijn gestructureerd rond werkstromen en bedrijfsprocessen die zijn gestructureerd rond bedrijfsfuncties.

Verticale structurering is de structurering van een systeem in termen van gerelateerde modellen op verschillende abstractieniveaus. *Abstractieniveaus* zijn gezichtspunten of perspectieven, zodanig dat elk model van een systeem op een bepaald abstractieniveau óf meer óf minder detail bevat dan een model op een ander abstractieniveau. Twee verzamelingen abstractieniveaus worden geïntroduceerd: generieke abstractieniveaus, waarbij niet gekeken wordt naar de typen entiteiten die in beschouwing moeten worden genomen tijdens een ontwerpproces, en specifieke abstractieniveaus, waarbij daar wel naar wordt gekeken.

De volgende generieke abstractieniveaus worden onderkend. Vanuit het geïntegreerde systeem-perspectief wordt een systeem gerepresenteerd als een geïntegreerd geheel: noch interne acties, noch interne delen van het systeem worden gerepresenteerd. Vanuit het interactiesysteem-perspectief van de delen worden de gerelateerde interne interacties van systeemdelen gerepresenteerd; de delen zelf worden niet gerepresenteerd. Dit maakt het mogelijk de verschillende wijzen te beschouwen waarop

deze interne interacties over de delen kunnen worden verdeeld. Vanuit het gedistribueerde systeem-perspectief wordt een systeem gerepresenteerd als een verzameling interagerende delen.

De volgende specifieke abstractieniveaus worden onderkend. Vanuit het perspectief van het systeem geïntegreerd in zijn omgeving worden systeem en omgeving gerepresenteerd zonder onderscheid te maken tussen beide. Het geïntegreerde systeem-perspectief is hierboven gedefinieerd. Vanuit het logisch gedistribueerde systeem-perspectief wordt een systeem gerepresenteerd als een verzameling interagerende delen, waarbij geabstraheerd wordt van de fysieke distributie van deze delen. Vanuit het fysiek gedistribueerde systeem-perspectief wordt een systeem gerepresenteerd als een verzameling interagerende fysiek gedistribueerde delen. Vanuit het lokale interface verfijnde systeem-perspectief worden de concrete interacties van de delen ook nog gerepresenteerd.

De combinatie van horizontale en verticale structurering leidt tot het onderkennen van ontwerpstappen die elk betrekking hebben op één aspect van een systeem op één abstractieniveau.

Een *ontwerpstrategie* schrijft de volgorde voor waarin deze ontwerpstappen uitgevoerd dienen te worden in een ontwerpproces. Het waterval-model, bijvoorbeeld, schrijft voor dat een systeem op successievelijk lagere abstractieniveaus dient te worden gemodelleerd, en dat het ontwerp op één abstractieniveau afgerond dient te zijn, voordat verder gegaan wordt naar een lager niveau.

De volgorde waarin ontwerpstappen worden uitgevoerd beïnvloedt het gemak waarmee bepaalde ontwerpdoelen bereikt kunnen worden. Dit proefschrift analyseert daarom de ondersteuning door een aantal bestaande ontwerpstrategieën voor de volgende ontwerpdoelen: minimalisatie van het overdoen van werk als gevolg van wijzigende eisen aan het systeem, minimalisatie van het overdoen van werk als gevolg van niet implementeerbare ontwerpen, optimale allocatie van middelen, en de kwaliteit van het te ontwerpen systeem. Zodra men de gewenste behoefte heeft bepaald aan ondersteuning van deze ontwerpdoelen, kunnen de resultaten van de analyse gebruikt worden om een geschikte ontwerpstrategie te selecteren of te construeren.

De hierboven beschreven structureringstechnieken kunnen gebruikt worden als elementen van methoden voor de ontwikkeling van bedrijfsprocessen. De volgende stappen zijn vereist om dergelijke methoden te construeren.

1. Selectie, en mogelijk verfijning, van de horizontale en verticale structureringstechnieken die onderdeel gaan uitmaken van de methode, resulterend in groepen gerelateerde ontwerpstappen. Verfijning vereist het ter beschikking stellen van technieken voor verdere structurering om meer concrete ondersteuning te geven aan ontwerpers van bedrijfsprocessen.

2. Selectie, en mogelijk verfijning, van een ontwerpstrategie die de volgorde bepaalt waarin de onderkende groepen van ontwerpstappen worden uitgevoerd. Op deze wijze worden de elementen gesynthetiseerd in de methode.
3. Verdere verfijning van de resulterende methode door het ter beschikking stellen van heuristieken, ontwerpcriteria, en voorbeelden.

Dit proefschrift bevat tenslotte een werkstroom-georiënteerde, evolutionaire ontwikkelingsmethode voor bedrijfsprocessen als voorbeeld.