Object-Oriented Analysis
for Multi-Faceted Applications
with Distributed Control
and Localized Data

een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
maandag 14 december 1998

des namiddags om 1.30 uur precies

door

Johannes Wilhelmus Gerardus Mathijs Hubbers

geboren op 23 april 1970 te Nijmegen
Promotor: Prof. C.H.A. Koster
Co-promotor: Dr. A.H.M. ter Hofstede, Queensland University of Technology

Manuscriptcommissie:
Prof. Dr. G. Engels  Universität-GH Paderborn
Dr. J. Harrison  The University of Queensland
Dr. H.A. Proper  Origin B.V.

The author acknowledges the support of ID Research B.V.

CIP-GEGEvens Koninklijke Bibliotheek, Den Haag

Hubbers, Johannes Wilhelmus Gerardus Mathijs

Object-Oriented Analysis for multi-faceted applications with distributed control and localized data / Johannes Wilhelmus Gerardus Mathijs Hubbers. - [S.l.: s.n.]. - Ill.
ISBN 90-9012124-2
NUGI 851
Trefw.: object-oriëntatie / informatiesystemen.

Cover photograph: Chambers pillar, Northern Territory, Australia, by J.W.G.M. Hubbers

© 1998 by J.W.G.M. Hubbers

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

1 Problem Area .......................... 1
  1.1 Introduction ............................ 1
  1.2 Assessing analysis methods .............. 2
  1.3 Approaches to software development ... 3
  1.4 Overview of OO .......................... 4
  1.5 A closer look at OOA ...................... 6
  1.6 Characterization of application domains .. 7
  1.7 Problem statement and approach .......... 11
  1.8 Thesis outline ............................ 12

2 Principles ................................ 13
  2.1 Introduction ............................ 13
  2.2 Scenario Validation Principle .......... 13
  2.3 Service Decoupling Principle ............. 16
  2.4 Execution Resilience Principle .......... 17
  2.5 Model Integration Principle .............. 18
  2.6 Locality Principle ........................ 19
  2.7 Rightsizing Principle ..................... 21
  2.8 Conclusion ............................... 24

3 Structure Modeling ..................... 25
  3.1 Introduction ............................ 25
  3.2 Identification ............................ 26
Dankwoord

Hoewel het doen van promotieonderzoek voor het grootste deel van de tijd een solo-activiteit is, hebben in de loop der jaren veel verschillende mensen een rol gespeeld. Naast de hieronder genoemde personen en instellingen wil ik iedereen bedanken die, op wat voor manier dan ook, een bijdrage geleverd heeft aan het tot stand komen van dit proefschrift.

SERC en ID Research wil ik bedanken voor de mogelijkheid die beide bedrijven mij boden om tijdens mijn promotietraject theorie met de praktijk te combineren.

Voor mij waren twee gecombineerde conferentie/werkbezoeken aan Australië instrumenteel in het promotietraject. Zonder de steun van NWO, Shell, ID Research en de KUN waren deze bezoeken niet mogelijk geweest. I would also like to thank the people on the other side of the world who provided me with a relaxed and productive environment. I am particularly grateful to Maria Orlowska (The University of Queensland) and Athman Bouguettaya (Queensland University of Technology) for letting me work at their respective universities.

In hoeverre een werkbezoek een succes is, hangt niet alleen af van het werk dat je er doet. Op deze plaats wil ik dan ook Erik Proper, Arthur ter Hofstede en Mie Takagi bedanken voor toeristische ritten, gastvrijheid, boeiende gesprekken en nog veel meer. Dat je voor dergelijke dingen Nederland niet uit hoeft is meer dan eens aangetoond door Araminte Bleeker en Theo Wiggerts, mijn ex-kamergenoten op de KUN.

Ook wil ik de actieve lezers van het manuscript van dit proefschrift bedanken. Zij kregen in een korte periode een fikse hoeveelheid e-mail in hun mailbox gestort. Ondanks de tijdsdruk waaronder zij hebben gelezen, wisten ze met zeer constructieve opmerkingen te komen. Hun feedback heeft geresulteerd in een helderder proefschrift. In dit verband wil ik met name mijn waardering uitspreken voor de manier waarop Denis Verhoef ondanks alle drukte zinvolle vragen wist te stellen en relevante opmerkingen wist te maken.


Last but definitely not least, I would like to thank Alistair & Kylie, John, Andrew, and the rest of the gang for some very entertaining times in Brisbane. I don’t think I’ll ever forget the “five minute walk” (not recommended for fainthearted researchers). Moreover, Indian restaurants will never be the same...
Chapter 1

Problem Area

1.1 Introduction

The world of Information Technology (IT) is a complex and dynamic one. It is complex, because the reality that it has to capture is complex. Systems have to deal with many types of information, and have to perform many functions. Systems in domains such as banking and insurance consist of literally thousands of programs, have to implement hundreds of business rules, and probably just as many exceptions to those rules. Designing such a system is a difficult task, which becomes even more difficult when cooperation with other systems must be taken into account, for systems that are self-contained and can run in total isolation are becoming rarer and rarer. IT has penetrated many organizations at all levels. Where companies used to have separate systems to support production, keep track of finances, and deal with orders, they now integrate these separate systems to achieve a business advantage through synergy.

The dynamic nature of IT is the result of application domains that keep changing. Users demand ever more complex applications to solve new problems and fulfill an increasing appetite for information. In many cases, the financial well-being of a corporation hinges on a number of mission-critical applications. In a practical sense, it is impossible to replace such systems in their entirety (a so-called cold turkey migration). As a result, the applications themselves are in a continuous flux, changing to keep on meeting increased demands for functionality or performance. Sommerville remarks in [Som89] that “As long-lifetime software is subject to regular change, it is important that the software is written and documented in such a way that changes can be made without undue costs.”

In order to deal with these difficulties, methods and techniques have been developed to develop and maintain systems. Their objective is to improve the productivity of developers and the quality of the resulting system [Sol85]. Software development methods usually employ a “divide and conquer” strategy. The process of software development is split into a number of phases, e.g. analysis, design, implementation, and testing. The results of the analysis phase can be viewed as the blueprint for the design and implementation of the
ultimate system. This phase is often considered the bottleneck of system development, since the acquisition of requirements is notoriously difficult, and it is a well-known fact that the later in the development process an error is detected, the more expensive it is to correct it [Dav90].

1.2 Assessing analysis methods

Unfortunately over the past few decades, a plethora of methods and techniques have been introduced. This has led to a situation that is often referred to as the methodology jungle [AF95]. Given this jungle, it is imperative that criteria are provided that help assessing strengths and weaknesses of methods. In the context of analysis methods such criteria were developed in [Hof93]. Each of them will be briefly discussed subsequently.

To be successful as an analysis method, a method must be able to specify the problem domain on a conceptual level. The conceptual level focuses on the essence of a specification, and hides implementation and representation details [Gri82]. This allows analysts to avoid premature implementation-oriented decisions that may lead to sub-optimal solutions.

Strongly related to conceptualization is the issue of expressiveness. A technique should allow the modeling of all relevant static and dynamic aspects of problem domain. A technique without sufficient expressive power cannot be used for conceptual modeling. As Embley et al. state in [EKW92], “an analyst has enough to do just studying, understanding, and documenting a system, without also having to transform ideas into an overly constrained system model”.

For a precise and complete understanding, a technique should have a formal foundation. In [Jac93], Jacobson remarks: “To be effective, the modeling language for analysis and design must be formalized”. This is underlined more recently by Henderson-Sellers, who notes that “...a formal underpinning is needed” in [Hen98]. And still many methods lack a formal foundation, especially analysis methods. The majority of these methods are only explained intuitively through examples and informal explanations. A formal foundation defines what well-formed models are (syntax), as well as their meaning (semantics). This way, open ends and ambiguities can be detected and dealt with. At the same time, a deeper understanding of the concepts and insight in the workings of the method that is being formalized can be achieved.

A formalized method is also useful when CASE tools are developed that support the modeling process through sophisticated checks, and help verification and validation of models. In this context, verification means checking if a model has certain properties, using the rules of a technique, whereas validation is concerned with the question whether or not a well-formed model meets its intended requirements.

Furthermore, for a technique to be practically useful, its models should be easy to understand, especially for communication purposes. This property is called comprehensibility. Especially in the analysis phase, models need to be communicated with domain experts, who are typically not educated in computer science.
A fundamental observation that has surfaced in the eighties is that no one method is equally suitable to all kinds of problem domains (the well-known *silver bullet* problem, see e.g. [BS87, Sol83, Bro87, AW91, LL87, ML83]). Each method has its particular strengths and weaknesses, and is therefore more suited to certain problem areas than it is to others. This property is referred to as *suitability*.

### 1.3 Approaches to software development

Main directions taken in the development of methods are *structured* approaches (e.g. JSD [Jac83], IE [Mar86], and Yourdon [You89]), *formal* approaches (e.g. Z [Spi88], VDM [Jon86]), *natural language* approaches (e.g. KISS [Kri94]), and *object-oriented* approaches (e.g. Objectory [JCJO92], OMT [RBP+91], OOA [CY90], Object-Oriented Design with Applications [Boo91], and Wirfs-Brock [WWW90]).

The weakness of *formal methods* is comprehensibility. In general, they suffer from a large cognitive gap between the models and reality, which discourages user participation [AF88]. In [EKW92] Embley et al. note that “analysts are not mathematicians. We cannot expect an analyst to document his understanding of a system in first-order predicate calculus or any other highly formal notation.”

*Natural language approaches*, such as KISS [Kri94], take natural language as input, and transform it to a model of the intended system. To validate the system model, sentences can be generated that express the contents of the models in a language that is familiar to domain experts. This distributes the characteristics of an object in the problem domain over many pages of text, due to the linear structure of natural language [EKW92].

*Structured approaches* to software development generally use either a process-oriented or a data-oriented approach. A disadvantage of structured approaches is the strict separation of data and operations. As a result, every object in the problem domain is distributed over (at least) two models, which makes it difficult to comprehend the meaning of the combined models. Processes and data are only weakly integrated. Objects that occur in the real world are associated with multiple processes in a way that is comparable to the natural language approach. Furthermore, the relations that exist between objects are hidden within processes.

The *data-oriented* approach offers a number of inherent strengths. The data that is being used within a problem domain is usually less prone to be changed than the processes that operate on those data. As a result, a design that was derived using a data-oriented approach is likely to be better suited to deal with the inevitable changes it will have to undergo during its lifetime. The data-oriented approach is of course particularly well suited to be used in information-intensive domains. Information-intensive domains form a large percentage of the entire IT-domain.

A risk that is inherent to a *process-oriented* approach is that the strong emphasis on the dynamic nature of a system invites the analyst to concentrate on *how* to do things, as opposed to *what* should be done. *How* things should be done is a question that belongs in
the area of design. If the question of how is considered during analysis, the quality of the analysis is likely to suffer. Champeaux et al. identify another drawback of using a process-oriented approach in [CLF93]: “The starting point for process modeling resides in the required behavior of the desired system. This makes S[tructured] A[nalysist] a predominantly top-down method. High level process descriptions are consequently target system specific, and thus unlikely to be reusable for even similar systems.”

Object-orientation has become a “buzz word”. Even though it is not always clear what the distinguishing marks of OO are, it is associated with all the “good” things that analysts, system developers, and users need: reuse, graphical user interfaces, evolutionary development, rapid prototyping, easily adaptable systems, simple maintenance, and so on. Although OO is certainly capable of delivering a positive contribution in these areas, a more precise understanding of OO is needed to judge the advantages of the OO approach. In short, OO claims to be particularly well suited for complex domains, because of the integration of data and operations, which decreases the gap between an application domain and its model. There is a straightforward correspondence between system objects and analysis document components [EKW92]. The lack of emphasis on processes facilitates the analyst in modeling on a conceptual level [CLF93]. Before we continue the discussion of strengths and weaknesses of OO, we will give a brief overview of OO.

1.4 Overview of OO

1.4.1 History

Object-orientation has its roots in the late 1960’s, when it started out as a programming technique. The programming language Simula-67 [DNM69] was the first language that offered support for the definition of classes and the creation of objects. The seventies saw a limited development of OO, e.g. through the introduction of de language Smalltalk. It was not until the 1980’s that OO really took hold. The introduction of languages such as Smalltalk-80 [GR83], C++ [Str91], and Eiffel [Mey88] caused people to become interested in the concept of objects.

Aside from the introduction of programming languages and programming environments, in the late eighties a start was made with the research into system development methods that were based on the concept of OO. Furthermore, OO ideas were incorporated in tools ranging from database systems to graphical user interface packages.

1.4.2 Objects

The central idea of OO is that a system consists of cooperating objects. Objects encapsulate state information and functionality. State information is kept by means of attributes and associations with other objects. The functionality of an object is modeled through a set of methods (also known as operations or services). The methods of an object can be compared to procedures working on the internal data stored within that object. Together, the methods form the interface of an object.
1.4.3 Encapsulation

Due to the encapsulation of data within each object, it is impossible for an object to manipulate the data contained within another object directly. The benefit of this approach is that side-effects are tamed. The state of an object can only be changed by the object itself, through the invocation of one of its methods. Therefore, the internal structure and implementation of methods can be changed, as long as the interface remains unchanged. In effect, objects can be viewed as black boxes. Analysts only have to be concerned with the services that are offered by an object, without having to pay attention to the way those services are implemented. This characteristic of OO makes it much easier to understand what a system of objects does or means.

1.4.4 Components

Combining data and methods into a single object is a first step towards components. Instead of completely separated, global descriptions of the functionality and the data of a system as it is used in traditional modeling approaches, OO describes functionality and data on a per class basis. There is often no need to examine the entire design/model if the system or its design needs to be changed. Usually, it is sufficient to single out one or a few objects for closer examination. This makes OO systems and designs more tolerant with regard to maintenance and functionality changes.

1.4.5 Classes and instances

Objects are defined by means of classes. A class is a blueprint that describes both the methods and the structure (the data) of a particular type of object. In other words, the class defines which associations and attributes an object has, as well as what its interface is, and how the methods behave.

Every object is an instance of a class. An object maps the attributes and associations that have been defined in the class to values. The methods that are defined in the class can be invoked on an object that belongs to the class. The identity of the object whose operation is invoked is an implicit parameter in the method invocation, allowing the method to act upon the attributes and associations of the object.

An alternative way of looking at the relationship of objects and classes is the following. A class can be seen as a collection of instances. All objects of a class together comprise a collection of instances of that class. As a result, the term “class” can be interpreted in two different ways: (1) a class defines the structure and methods of objects, and (2) a class is a collection of instances. The first definition corresponds to the notion of implicit types, whereas the second definition is in line with explicit types.

An important characteristic of objects in this context is that each object has its own inherent, unique identity. Consequently, two objects that are instances of the same class can have identical values for their attributes and associations, but they can still be distinguished as separate objects.
1.4.6 Models

Object-oriented analysis methods usually deliver models of three types, each of which captures a dimension of the system:

1. structure model (also known as object model)
2. object behavior model (OBM)
3. interaction model

The structure model is used to describe how classes are associated with each other. The behavior models describe how the objects of each class handle service requests. Interaction models describe how objects cooperate to realize the system’s functionality.

The system is fully described by the structure model in conjunction with the behavior models. This thesis refers to that description as the system model.

1.5 A closer look at OOA

Although OO might not be the definitive answer to the problems associated with software development or information systems development, it does offer a number of unique and desirable characteristics that can help raise the quality of the system development process, and thus produce a better system. These benefits range from a close match with the way of thinking of “ordinary” people (hence increasing comprehensibility), to opportunities to reuse (parts of) models. As Goldberg puts it in [GR95]: “Four characteristics of objects facilitate software systems development: objects separate interface from implementation; objects closely map the real world; objects come in different sizes; and objects live all the way down—from analysis to design to implementation.” In addition, OO tightly couples the data and process perspectives, contrary to e.g. traditional structured approaches. This achieves a substantial increase in expressive power.

As a result of the promises of object-orientation, many OO analysis methods have been developed in a relatively short time. The results of this jungle are ambiguities and fuzzy definitions of concepts. It is often hard to determine the exact meaning of a concept, and similar concepts (often with the same name) of different methods have subtly or deceptively differing meanings [EJW95]. A lack of formalization stands in the way of a precise and complete understanding of a method.

Yet many methods are accompanied by little or no formal definitions. The methods that do have a formal foundation (e.g. CGOOD [TG96], TM [ABBV90], O₂ [BDK92]) appear to aim more at the design level than at the analysis level.

As remarked before, the suitability of a technique depends on the type of application domain. The next section provides a characterization scheme of domains for which we consider OO to be particularly well suited.
1.6 Characterization of application domains

According to Jackson [Jac95], “it is always right to pay serious attention to the application domain. Yet there is no standard comprehensive taxonomy of domain characteristics”. Because the suitability of a technique is strongly linked to the domain for which it is intended, this section provides a characterization of application domains based on three properties:

1. Control structure
2. Data structure
3. Faceting

This list is not necessarily definitive or exhaustive, nor is it by any means the only way of characterizing application domains. However, this way of characterizing is rooted in the Model-View-Controller (MVC) paradigm, which is described in e.g. [GR83] and [KP88]. The goal of MVC is to separate the representation (model), presentation (view), and way it is controlled by the outside world (controller). The following sections discuss how the application domain properties are linked to the MVC perspectives, and how they describe the application domains that are discussed in this thesis. A similar three-pronged approach to describe applications is described in [GC96]. Their perspectives are abstraction, control, and presentation.

The examples use the notation proposed by Jacobson et al. in [JCJ092], because it offers distinct notations for control objects, data objects (or entity objects), and interface objects as shown in figure 1.1.

![Figure 1.1: Notation used in the examples](image)

1.6.1 Control structure

In this section focus is on the control structure of domains, or more precisely, the prevalent control structure of typical applications in those domains. It is possible to talk about the control structure of a domain, because a domain often suggests a particular control structure.
Control structures range from *centralized control* to fully *distributed control* control. This sliding scale can be expressed in a *distribution of control* property. Figure 1.2 contains schematic drawings of various control structures.

![Diagram of control structures](image)

Figure 1.2: Centralized and distributed control

In an application with a centralized control structure, there is one subsystem (or object) that controls the behavior of the system as a whole. This subsystem is called the *controller*, or *master*. By nature, a controller is tightly coupled to the subsystems it controls (the *slaves*).

Examples of centralized control can be found in the process industry, such as manufacturing of animal food, or chemical plants that transform crude oil to more sophisticated products such as diesel fuel and gasoline. The application that is responsible for the manufacturing process typically controls all variables. It regulates how much of the raw material is processed at any one time, adds other chemicals, makes sure that the reaction process takes place at the correct temperature and pressure, and opens and closes valves to extract the end products.

At the other end of the scale are the fully distributed control structures. An application with such a control structure consists almost exclusively of controllers. Typical domains that utilize this kind of control structure are simulations and monitoring of industrial processes. A monitoring application typically consists of a large number of independently operating objects, each of which monitoring a limited number of variables, such as the temperature at the top of a reactor, or the pressure in a piece of pipeline. It is very conceivable that all these monitors report to a single reporting facility, but the point is that, even though it is a central facility, it does not actively control the monitors. The *Observer pattern* [GHJV95] is a good example of distributed control in the area of design patterns.

Between the extreme cases of fully centralized and fully distributed control flow are an infinite number of intermediary control structures, which mix some of the characteristics of both. Such a mixed control structure is usually built using a limited number of controllers, each responsible for a part of the domain. Workflows form an example of mixed
control structures. The separate units that make up a workflow constitute a distributed control structure, while the applications that implement the workflow processes each can be controlled in a centralized manner.

1.6.2 Data structure

The data structure property is related to the model part in MVC. It gives an indication how the data that is used in the application domain is partitioned. The extremes are:

- a global data structure, which involves no partitioning at all;
- many local data structures, which makes extensive use of partitioning.

In an application that employs a global data structure, all data can and will be accessed by all subsystems. Take for example a CASE tool, whose main task is to assist in the development of a consistent model of a system. Such a tool might consist of different editors and employ scores of rule verification algorithms. Yet all subsystems (in this case, editors and verification routines) perform accesses that stretch across the entire system model, which is shown in figure 1.3.

![Figure 1.3: Global data structure in a CASE-tool](image)

Highly localized data structures can be found in CSCW (Computer Supported Collaborative Work) systems, where each collaborating unit presides over its own data. Furthermore, a unit has no direct interest in the data that is maintained by other units. Simulations provide another example of a domain that employs localized data structures, e.g. for each of the simulated actors and resources.

Partially localized data structures are typically found in back offices of financial institutions and ERP (Enterprise Resource Planning). A full-featured ERP package offers company-wide functionalities, which are divided over a number of components. This makes the all encompassing scope more manageable and creates the possibility to assemble specially tailored systems out of the components that are really needed. Examples of components
in this context are subsystems that handle stock control, sales, personnel, production, or billing. Each component is responsible for its own data, yet all operate on a global database (figure 1.4).

Figure 1.4: Partially localized data structure in ERP-context

1.6.3 Faceting

The “view” part in MVC usually refers to the presentation of data (the model) through a user interface. In [GC96], it is referred to as representation multiplicity. For the purpose of characterizing domains, a broader perspective is taken, referred to as faceting. In this perspective, presentation is not solely concerned with presenting results to a human user, but also includes presenting the same output in different ways to different subsystems.

Sophisticated graphical user interfaces and business layers of front-office applications form a typical domain with multiple facets, because of the demands placed upon easy customization and flexibility. In contrast, embedded software tends to be tailored for a specific purpose, which reduces the need for presenting multiple facets.

A concrete example of multi-faceting (not directly related to user interfaces) would be a database of publications (see figure 1.5). Required presentations might be HTML for presentation of publications on the Word Wide Web, PostScript for easy one-stop downloading and printing of entire publications, citation data in the shape of \LaTeX’s BIB\LaTeX-format, and citation data and entire publications in RTF (Rich Text Format) for non-\LaTeX-users. The example makes clear that multi-faceting not only involves user interface aspects.

The flexibility that results from multi-faceting is not always desirable, e.g. in the area of time-critical and embedded systems. In these domains, the inherent cost of achieving flexibility might be too high a price to pay.
1.7 Problem statement and approach

The advantages of an object-oriented approach are most clear when it is used in an environment which places high demands not only on information retrieval and storage, but also on performing operations on the data. As it is put in [GC96]: “An important feature of the software running in any modern computer system is the ability for applications to share both control and data.” Examples of application domains that software development is branching out to, are geographical information systems, multi-media, meteorological systems, air traffic control, and medical information systems. Informally, these domains can be characterized by

1. Complex operations on data. Storage and retrieval is no longer the main task of a system. Its main purpose is to perform operations and calculations based on stored data.

2. Need for integration and interoperability. Systems are used in an organization-wide fashion, or, alternatively, used in organisations that work in a less hierarchical manner.

3. Complex data types. Modern systems tend to use more complex data types, such as sounds and images.

Using the characterization scheme that was presented in section 1.6, these domains share the following common properties: they are multi-faceted, employ distributed control structures, and operate on local data.

The aim of this thesis is to bring some order to the OO methodology jungle and hence to deduce what the core concepts of OO analysis are, in the context of multi-faceted applications with distributed control and localized data. The approach is a synthesizing one, whereby a number of OOA methods that have proven themselves in practice are formally
analyzed and extended. The resulting formal model is then empirically validated through its deployment in a number of practical case studies. Its expressiveness is shown through the formal description of the Parallel Random-Access Machine [Sav97]. Using this research approach, insight into object-orientation can be gained without developing yet another modeling technique, which would only increase the number of trees in the jungle.

1.8 Thesis outline

As it is important to formally ground the notion of suitability, this thesis starts with a derivation of suitability principles in the context of our domains of interest in chapter 2. The principles have been synthesized based on the cases that are described in chapter 7. In addition, parts from [Hub97] and [Hub98] are used.

Chapter 3, which is based on [HH98], investigates the modeling of data within an object-oriented context, resulting both in a synthesis of data modeling and OO, and in a formal foundation. This chapter also uses parts of [Hub97].

Chapter 4 addresses the behavioral perspective in object-oriented modeling. Again, the results are described in a formal way. This chapter is based on [HH97] and [HH96].

OO’s promise of reusability is discussed in chapter 5, using a real-life case study described in [Hub94, OH93, OH95]. The chapter is partially based on [HV97].

The expressiveness of the technique described in this thesis is studied in chapter 6.

The theoretical work is offset against practical experiences in chapter 7, which validates our approach, and yields feedback on comprehensibility and suitability. Detailed documentation of the cases can be found in [HO95a, HO95b, HO95c, HVW97].
Chapter 2

Principles

2.1 Introduction

Techniques should fulfill the general requirements discussed in chapter 1. In this chapter, the general requirement of suitability is refined. The suitability of a technique depends heavily on the domain. This chapter provides a number of more detailed principles that deal with suitability of OO-techniques for domains that match the characterization given in chapter 1. Modeling techniques that are suitable for these domains have to abide by (at least) the principles presented in the remainder of this chapter.

The approach and the Scenario Validation and Execution Resilience principles are inspired by [BHP97]. Of course, these principles have been migrated from a business suitability perspective to the context of suitability of OOA techniques in multi-faceted domains with distributed control and localized data. The principles are synthesized and abstracted based on experiences that were gained in modeling practical applications. Therefore, the set of principles presented in the remainder of this chapter cannot be claimed to be complete.

2.2 Scenario Validation Principle

The Scenario Validation Principle is derived from the distributed control that is characteristic for the domains that are under investigation. Distributing control over multiple objects makes it more difficult to understand what the system does, how it works, and whether it is correct. This is because the distribution of responsibilities over smaller tasks makes it necessary to integrate their actions into a scenario that can be validated by a human being.

Consider the example presented in figure 1.4. Sales and Stock Control can both operate autonomously. Yet both subsystems require services from Stocks. Distribution of control implies the sharing of the responsibility to maintain accurate stock data. Designing a distributed control system that guarantees the correctness of the stock data is more difficult than designing a similar system that uses a global control structure. The global control structure minimizes dependencies, and offers a de facto scenario that can be validated. This
shows a clear contrast to the validation of the working of the system in figure 1.4, where a scenario must be pieced together on the basis of the workings of the various units.

**Principle – Scenario Validation**

“A technique should provide an explicit notion of scenario to support model validation”

Validation involves the ability to trace execution sequences and the interaction between objects. The challenge that is presented is that the system model describes multiple different execution sequences. The *actual* execution sequence that results from the execution of the system depends on an unpredictable context, e.g. user inputs or messages from other systems.

In general, scenarios can be validated on three levels of granularity:

1. use case
2. interaction diagram
3. traces

### 2.2.1 Use cases

A *use case* describes a particular way of using the system at a high level of abstraction. Current modeling techniques offer few facilities to include decisions in use cases. Typically, one type of input to the system is mapped to a single use case. Though the lack of decisions lessens the expressive power, it is convenient from a validation point of view. Inherent to use cases is the one-to-one mapping of user input to a model. As a consequence, validation on the use case level is simple, but will only seldom uncover hidden problems. Use cases (introduced in Jacobson’s Objectory (OOSE) method [JCJO92]) have proven their use, and have been adopted by by many of the more recent OOA methods. As a result, support for validation in earlier stages is growing, in particular through the absorption of Jacobson’s Objectory method by the Unified Modeling Language (UML, [BJR97a, BJR97b, BJR97c]).

### 2.2.2 Interaction diagrams

Interaction diagrams are provided by nearly all OOA methods. Notable exceptions are CY and OMT. Interaction diagrams operate on a finer level of granularity than the use cases. They are potentially useful in discovering conflicts such as starvation and deadlock.

Figure 2.1 contains an example interaction diagram, illustrating the handover of an aircraft from one air traffic controller (ATC) to another. First, the *aircraft* receives the message that it is handed over. The *aircraft* then requests to be removed of the list of responsibilities of ATC A, and requests to be added to the list of responsibilities of ATC B. A’s display is updated to reflect that the aircraft is in the process of being handed over. At the same
time, B’s display receives the request to display the new responsibility. If no errors are encountered, both displays are updated to show that the handover has been completed.

Responsibility-Driven Design (RDD, [WWW90]) uses collaboration graphs for similar purposes. The collaboration graphs show how the objects operate together to realize their responsibilities (services).

2.2.3 Traces

The most extensive and exhaustive form of scenario validation is offered by the trace mechanism. An execution trace is a sequence of actions that are performed by objects in the system. It shows which object requests services from other objects, gives insight into the order in which operations are performed, and can show whether all objects are in fact being used.

Execution traces can be deduced from object behavior models. An object behavior model, which specifies behavior at the class level, defines a set of all possible traces that any instance of that class can generate if executed.

As mentioned in [HH97], one of the advantages of using Process Algebra as a means of formalizing is that it allows for easy simulation of the model. Chapter 4 applies an instantiation mechanism to object behavior models to provide trace information on the level of object instances. This makes it possible to execute the entire system model, and generate a trace of all the actions it performs. Such a trace can of course be very verbose. To reduce its complexity, a trace can be filtered for a more comprehensible overview. Frederiks and van der Weide use traces extensively for the validation of object life models, which are closely related to object behavior models, in [FW96].
2.3 Service Decoupling Principle

Most OO techniques do not distinguish external interaction (service requests originating from outside the object) from internal interaction (the sequence of actions that results from a service request) at the modeling level. Rather, the behavior of an object is usually specified as a state machine with multiple entry points, all of which can be used to request a particular service. This specification mechanism makes it hard to comprehend the services that are offered by an object, particularly in a multi-faceted domain, where many services provide access to the same information.

**Principle – Service Decoupling**

“A technique should support the decoupling of service requests from their actual definition”

At the programming level, this principle is widely recognized. Most third generation programming languages such as Modula [Wir85], C [KR88], and Pascal [JW85], already incorporate constructs to separate the interface definition (“what”) from the implementation (“how”). OO languages such as Eiffel [Mey88], C++ [Str91], and Java [AG98], also cater for the need to hide the implementation from the outer world.

Yet most object-oriented analysis methods provide no adequate means to specify object behavior and the object interface separately. The typical approach to describe object behavior is to use state transition diagrams (or derivatives thereof), such as the ones used by OOSE [JCJO92], OMT [RBP+91], UML, or OOSD [CLF93].

State transition diagrams (STD) force a tight coupling of the method signature (as used in e.g. the “receive message” construct) and its implementation, by mixing the interface with the implementation. An example of this situation is given in figure 2.2. It shows a small STD, in which state $S$ is followed by either the reception of a message identified by the construct $R_1$ (say $x$), or reception of a message identified by the construct $R_2$ (say $y$). The implementation of $x$ consists of performing task $T_1$. Performing task $T_2$ comprises the implementation of $y$. It is not excluded that both $R_1$ and $R_2$ indicate reception of the same message ($x = y$). If this is the case, two different implementations can be specified for the same service in one object. At the very least, any change to the implementation of the request must be reflected by all other implementations of that request in the same OBM. Therefore, if $x = y$ in the example in figure 2.2, any change to the sequence that starts with $R_1$ and ends with $F_1$ must be applied to the sequence $R_2-F_2$.

OSA [EKW92] uses high-level states and high-level transitions. This makes it possible to partially abide by the Service Decoupling Principle, because multiple high-level transitions can refer to the same lower-level ("implementation") transition. RDD provides service decoupling through the use of their Class Responsibilities and Collaborations (CRC) cards. A CRC card is used to determine what the responsibilities (services) of a class are. Responsibilities are refined using *signatures*, which map the services of an object to an implementation. Figure 2.3 shows an example CRC card, describing a pressure sensor.
Another feature of RDD is its notion of *subsystem*. Subsystems are used to divide a design in smaller elements. Relevant in the context of service decoupling is the possibility to designate contracts either as *public* or *internal*. Public contracts carry over to other subsystems, whereas internal contracts are hidden from the outside world.

### 2.4 Execution Resilience Principle

Another important aspect of distributed control relates to handling conflicts, exceptions, and unexpected behavior. These phenomena should be dealt with in a graceful manner. The natural place for handling exceptions is within the boundaries of each object. Unfortunately, when responsibilities are distributed over multiple objects, this solution no longer suffices. In a distributed control situation it is more likely that multiple objects are involved in an exception.

Another factor that promotes the Execution Resilience Principle can be found in the flex-
ibility required by the application domain. Multi-faceting is a convenient hook for future functional extensions of a system. To maintain the stability of the system when it is changed (either for maintenance purposes or to add new functionality), fail-safes that deal with the problems that further facets may introduce should be modeled explicitly.

The best way to deal with exceptions is to provide support to clearly model them. Built-in support for dealing with the unexpected can be regarded as flexibility at the modeling level.

**Principle – Execution Resilience**

“A technique should support the handling of operational errors, so that a resilient execution of the conceptual model results”

Many techniques choose to postpone paying attention to exception handling to the design or implementation stages. Noteworthy exceptions are OOSE and UML, OSA, and HOOD [DHM93]. Both OOSE and UML offer an elegant way of specifying exceptions on a conceptual level through the “extends” concept. The “extends” concept operates on the use case level. An extends association from use case A to use case B indicates that use case B may include use case A if certain conditions (e.g. an exception is encountered) are met. OSA deals with exceptions through exception transitions. They can be used in the specification of an object’s behavior, and are triggered when an exception occurs. HOOD includes a concept called exception flow. Using exception flows, abnormal returns of control flow during execution of an operation can be channeled.

### 2.5 Model Integration Principle

In object-oriented techniques, different model types show different perspectives of objects. From a comprehensibility point of view, separating the different perspectives into multiple model types is a good thing. It is much easier to comprehend the working of a system if it is possible to temporarily disregard other perspectives in favour of concentrating focus on a single perspective.

However, one of the strengths of OO is the integration of perspectives in the concept of objects. Therefore, the models that represent the various perspectives of objects should exhibit characteristics that allow for their easy integration. That way, OO’s claim of providing a natural integration of behavior and structure can be substantiated. It is essential from a comprehensibility point of view that the different perspectives can be unified to form a clear overall picture. The overall picture in its turn can then be mapped down to a design.

**Principle – Model Integration**

“A technique should support the integration of individual models by providing an explicit mapping from individual models to an overall design”
The UML, through its integration of a variety of techniques, covers a wide range of model types. Some of these techniques appear very similar, which makes it hard to decide when to use which model type [Hub97]. This is a result of insufficient model integration, because if a method abides by the Model Integration Principle, it is clear what each model contributes to the system model.

A weakness that the UML has inherited from Jacobson’s OOSE method is the integration of use cases with the rest of the models. The concept of use cases has proven its worth, but the lack of an explicit connection between use cases and other models makes the transition from initial modeling to more detailed activities (“modeling the use cases” – [JCJ99]) difficult. On the positive side, model integration of object interactions and object behavior is given sufficient attention. The same can be said to hold true for the integration of the structure model (somewhat misleadingly called “analysis” model in OOSE terminology) and the object behavior models. The object behavior models refer directly to properties and associations that are defined in the structure model. This is described in more detail in section 4.3.

A good example of model integration is provided by the contracts of the Responsibility-Driven Design method. Informally, objects in RDD can play two roles. An object can request a service from another object, in which case it acts as a client. Alternatively, an object may provide a service to a client, in which case it acts as a server. The consistency of models can be verified by checking the contracts. A contract states which services are offered under which conditions. As there are always at least one server and one client involved in any one contract, contracts provide an explicit notion of model integration.

2.6 Locality Principle

The modeling of applications that employ local data gives rise to the identification of a locality principle. Object-orientation offers encapsulation to provide locality on the level of objects. The Locality Principle takes this concept one step further, and applies the underlying idea to all aspects of a technique. For example, a characteristic of data modeling techniques is that they allow constraints to connect entities that are otherwise not directly connected, such as the example constraint c in figure 2.4. The non-local nature of the constraint makes it difficult to assign the responsibility of ensuring that this constraint is not violated to a particular object in a natural way.

**Principle – Locality**

“A technique should actively support the modeling of properties at a local level”

A good example of a technique that goes a long way in adhering to the Locality Principle in the area of constraint modeling is OMT. OMT uses three types of constraints:

1. constraints on objects
2. constraints on links (also known as associations)

3. general constraints

A constraint on an object is, by definition, a constraint with a local scope (one object). It is natural to assign the responsibility of maintaining the constraint to the object it refers to. Since OMT allows the modeling of a link as a class, it is possible to give a similar argument for constraints on links.

Figure 2.5 presents examples of two OMT constraints. It shows a constraint on an object, which expresses that the priority of a Task may never increase, and a constraint on a link that expresses that each Consultant works on at least one Project.

Unfortunately, OMT uses general constraints as a catch-all construct, in that they may be used to express any constraint that cannot be expressed in terms of constraints on objects or links. The global character of this type of constraint is indicated by the authors of [RBP+91] as they refer to “unattached constraints” and a constraint that is “attached to all the classes”. Obviously, such constructions violate the Locality Principle, which opens the gates to problems with respect to comprehensibility, maintainability, and adaptability.

Inheritance is a common construct that supports the Locality Principle: it can be used to group common functionality at a local level. However, the use of inheritance is limited by
the fact that inheritance hierarchies become very dense and convoluted if they are used to distribute little groups of behavior to a multitude of classes.

![Class Hierarchy Diagram]

Figure 2.6: “Simple” class hierarchy

Consider the class hierarchy in figure 2.6, which deals with the relatively simple domain of sensors. In this example, a sensor is characterized by three aspects: it can measure, it can report, and it might log. This is reflected in the inheritance hierarchy by the direct or indirect inheritance of MeasuringDevice, Reporter, and Logger. Unfortunately, this introduces “unwanted” classes whose sole reason for existing is that they represent three ways of looking at a sensor.

In [Bap94] Bapat introduces the concept of capsules. Capsules group common functionality into reusable, encapsulated units. Figure 2.7 presents a better solution to the problem posed in figure 2.6. Here, the capsules allow the designer to focus on how sensors fit into the system. The capsules permit the adding of standardized functionality where it is required.

### 2.7 Rightsizing Principle

The rightsizing principle deals with two related but subtly different issues:

1. scaleability
2. granularity
Scaleability is the ability to model systems with sizes that range from small to very large. In practice, most techniques have no problems modeling small systems. The challenge of many techniques is coping with the complexity that results from modeling large systems. Techniques that deal well with granularity provide support for modeling at different levels of abstraction.

### 2.7.1 Scaleability

The domain characterization that is used in chapter 1 does not make any assumptions about the size of the system. Systems that are relevant in the perspective of this thesis can be small or very large. The unit of measurement for the size of a system is the number of classes, because we are interested in the size of the system model. As a result of this definition, we do not consider a system that consists of a single class that has many instances to be a large system.

A technique that is suitable for systems that consist of many classes should provide a means to reduce the sheer size of the models to a more manageable level, for example through well-defined subsystem constructs.

### 2.7.2 Granularity

In practice, real-life objects are rarely all of the same granularity. Specifically in the early stages of analysis, an analyst usually encounters objects that span a wide range of granularities. For example, objects that appear in an initial object model of a library system might be: title, publication, author, borrower, library branch, and library.

Another reason why support for modeling different levels of granularity is important can be found in the distribution of control that is present in the application domains under
Rightsizing Principle

investigation. Distribution of control almost always results in objects of at least two granularities: the controlling objects and the controlled objects. Also, distribution of control is a property that is common when resorting to the reuse of previously designed software components. A more extreme case of the use of components is the reuse of, or interfacing with, entire existing applications.

2.7.3 Scaleability + Granularity = Rightsizing

Because of the strong ties between modeling systems of vastly different size and dealing with different levels of granularity, both aspects have been combined into a single Rightsizing Principle.

Principle – Rightsizing

“A technique should provide modeling constructs that facilitate dealing with modeling on different scales and on different levels of granularity”

A useful starting point to cater for the needs of the Rightsizing Principle is provided by a well-defined, powerful composition mechanism. Most object-oriented techniques provide the concept of aggregation. Most aggregation concepts exhibit two weaknesses. First of all, they lack a precise definition. In particular the handling of service requests is often unclear. Can a request be addressed to the aggregate as a whole? And if so, is the request forwarded to all constituting parts? Secondly, the current crop of aggregation constructs hardly differs from standard associations. As remarked in [RBP+91], the choice between an association and an aggregation is often entirely a subjective matter. For a composition mechanism to be useful from a rightsizing perspective, it must be able to hide details from lower levels of abstraction, while at the same time presenting a recognizable face to the analyst. The composition mechanism of OSA is a promising step in that direction, in that it allows for the definition of higher-level behavior based on lower-level specifications.

Subsystems, such as employed by e.g. OOSE, OMT, and RDD can be another useful construct, operating on a larger scale than the composition concept. Composition is typically used to group a small number of classes, where subsystems can consist of a large number of classes. The concept of “sheets”, as used in OMT, is not sufficient. The sole purpose of sheets is to limit the physical size of a model, without adding semantics that provide encapsulation or defining a subsystem interface.

Rightsizing poses a serious challenge for the natural language oriented techniques, such as KISS [Kri94]. The natural language approach can easily cause the growth of models to equal the growth of the domain. This becomes clear when looking at the object interaction model of the KISS method. It describes the possible interactions between objects on the basis of the actions that can be performed on them. If we assume that the size of an application consists of the sum of the number of classes and the number of actions, model growth equals domain growth because KISS lacks abstraction mechanisms such as subsystems or composition mechanisms.
2.8 Conclusion

In this chapter, a number of suitability principles for object-oriented approaches in the context of multi-faceted applications with distributed control and localized data were defined.

Table 2.1 summarizes the principles by showing for each principle to which model type it is primarily applicable, and to which domain characteristics it is particularly supportive.

<table>
<thead>
<tr>
<th>Principle</th>
<th>Primarily applicable to model type:</th>
<th>Supportive to domain characteristic:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Structure</td>
<td>Behavior</td>
</tr>
<tr>
<td>Scenario Validation</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Service Decoupling</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Execution Resilience</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Model Integration</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Locality</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Rightsizing</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of principles

Although this set of principles cannot be claimed to be complete, we believe that they are instrumental in guiding our synthesis and application of object-oriented methods. The principles give us a qualitative means to assess the benefits of alternative modeling approaches.
Chapter 3

Structure Modeling

3.1 Introduction

An often used term for the model that describes the relations between object classes is the object model, which is defined in [RBP+91] as follows: “The object model describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations. The object model provides the essential framework into which the dynamic and functional models can be placed.” In other words, the static dimension of a system describes the structural aspects of a system, i.e. the way individual classes are related to one another [Hut94]. Many names exist for this kind of model, e.g. object model [RBP+91, KR94], information model [SM88], object-oriented data model [ADM+89], analysis model [JCJO92], object-relationship model [EKW92], or static characterization [CLF93], or most recently class diagram in [BJR97a].

Unfortunately, there is a lack of consensus about what an OO data model is. There is confusion when it comes to terminology, and to proper definitions of concepts. In some cases concepts with the same name represent fundamentally different ideas, while in other cases concepts with different names seem very similar.

The aim of this chapter is to untangle this web. The approach is a synthetic one, in the sense that existing techniques are analyzed with respect to the concepts they offer. This leads to a small number of orthogonal fundamental notions. They may be combined to characterize the various concepts used by the various techniques. This chapter does not define a concrete graphical notation. The notation used in the diagrams merely reflects the abstract syntax suggested by the formalization.

The concepts are provided with a formal semantics that is inspired by a highly flexible approach based on category theory [HLW97]. Category theory is a relatively young branch of mathematics designed to describe various structural concepts from different mathematical fields in a uniform way. Although it provides the necessary level of abstraction and representation independence needed for the issue at hand, the formalization presented in
this chapter only partly uses the flexibility provided in [HLW97]. Instead, a slightly less flexible but more accessible set-theoretic formalization is given.

In this thesis, focus is only on conceptual data modeling. The importance of the early phases of system development is well-known [Dav90]. In our view it is essential that the techniques that are used during the early stages have an unambiguous, and therefore formal, basis. This facilitates lucid communication between different developers, and removes unintended inaccuracies from analysis models. As the focus of this chapter is on data modeling, the modeling of behavior is postponed to chapter 4.

The chapter is organized as follows. Section 3.2 compares identification in object-oriented to non-object-oriented systems. Section 3.3 lays the necessary formal groundwork that is needed for the definition of the semantics of associations by introducing and defining features and constraints. Dynamic semantics are described in section 3.4. Section 3.5 deals with different types of associations between classes, and how those association types can be characterized using features and constraints. Section 3.6 presents examples of the application of features and constraints. One of the examples characterizes the class diagram from the Unified Modeling Language. Section 3.7 concludes the chapter, and identifies topics for further research.

### 3.2 Identification

According to [CAB+94], an object is a “thing” that can be distinctly identified. At the appropriate level of abstraction almost anything can be considered to be an object. In traditional information modeling, developing a scheme that can be used to identify entities can be challenging. In classical (non-OO) systems, identification is necessary to be able to retrieve everything that was at some time input into the system. Units of data are identified through their properties in the domain (weak identification). Therefore, denotation is important. In object-oriented systems, denotation is far less important, as OO has a different way of dealing with identification. An object-oriented system provides each object with a unique object identifier (OID), which relieves the analyst of the burden of providing an identification scheme. The OID comes from an abstract domain, and is foreign to the problem domain. Though it may not be practical to generate a unique system identifier for every value [EN94], this is not a concern at the analysis level.

Being able to resort to OID’s for identification purposes can be an advantage, particularly when weak identification is not desirable, for instance when modeling a simulation of a rabbit-farm. All rabbits have an identical appearance, but each rabbit has its own identity. Another characteristic application of OID’s can be found in domains that deal with incomplete knowledge. In those domains, classical systems often resort to introducing an explicit OID (called surrogate [WJ95]) to distinguish between two objects that possess identical properties. An example of this is the introduction of the social security number in the USA, the tax file number in Australia, and the SoFi (SocialFiscal) number in the Netherlands.
However, sometimes the explicit distinction of two objects that “look” the same, i.e. two objects that have the same external appearance, is undesirable. Instances of basic types such as integers or booleans are clear examples of this kind of object. To accommodate the need to model these objects, so-called value objects are introduced. A value object represents a value from a concrete domain, such as integer or boolean. Its OID is equal to the value it represents. This ensures that any concrete value exists at most once as an object. Value objects can be compared to nameless objects as described in [WJ95].

### 3.3 Features and constraints

This section lays the groundwork for the definition of association types by introducing features and constraints, which are embedded in the formal context of type graphs.

The type graph, comparable to [TG96] or [Sie90], captures the structure of the system by describing the relations between object classes. The characteristics of associations are captured by means of features and constraints. Features provide a low-level characterization of the connection between two objects. Constraints deal with more complicated situations, and can be used to characterize the connection a single object has with a number of other objects.

### 3.3.1 Type graph

The various object types in the data model correspond to the nodes in the type graph, while the constructors are mapped to arrows. Formally, a type graph \( G \) is a directed multigraph in which the nodes in \( G_0 \) are object classes and association classes. The edges in \( G_1 \) between the nodes of the type graph are induced by the relations between the object classes and the association classes. In addition to the edges that result directly from the relations between classes, a number of extra edges is incorporated into the type graph, to aid in the definition of the semantics of features. Usually they are not shown explicitly. In this chapter they are represented by dashed arrows.

The behavior of sets is dealt with in the following way. Objects modeling elements and objects modeling sets are related by a bijective function \( \text{clt} : G_1 \rightarrow G_1 \). If \( \text{clt}(e) = f \), where \( e : A \rightarrow B \) and \( f : A \rightarrow C \), then \( B \) is a set with element type \( C \). The definition of a type graph is very liberal: the only restriction placed on the type graph is that cyclic inheritance structures are excluded. There are no cycles consisting solely of inheritance edges.

The set \( V \) is a subset of the set of nodes, and contains the value type nodes. \( K \) is a set of concrete domains, which provide the values for the value types. \( \text{Dom} : V \rightarrow K \) is the function that links a value type with its domain.

### 3.3.2 Populations

The semantics of populations, features, and constraints is based on conventional set theory but builds upon earlier work in the area of category theory described in [HLW97].
The semantics of a data model is the set of all its possible instantiations, which are also called *populations*. A population is defined as a *model* $\text{Pop}$, which is a graph homomorphism from a type graph to a state graph. To avoid notational clutter the model is sometimes omitted if it is clear from the context.

The model $\text{Pop}$ maps the nodes in a type graph onto sets and the edges onto relations of the state graph. The objects in the state graph are finite sets of object identifiers from the infinite set $\mathcal{O}$. The arrows represent relations between objects: an arrow $R : A \to B$ from $A$ to $B$ exists if and only if $R \subseteq A \times B$. Using *relations* as the basis for the population of arrows matches the approach taken in [Kas96].

For a model $\text{Pop}$ to be a valid model, the population of a class should consist exclusively of instances of that class. For this purpose the function $\text{Class} : \mathcal{O} \to \mathcal{G}_0$, which yields the class of an object, is introduced. Hence, for each $X \in \mathcal{G}_0$, $X$ should be mapped to a set of instances which are of type $X$:

\[
\text{Pop}(X) \subseteq \{ \alpha \in \mathcal{O} | \text{Class}(\alpha) = X \}
\]

**Example 3.3.1**

Figure 3.1 contains an example of a populated type graph. $G$ is an association that relates the instance $a_1$ of $A$ with $b_3$ of $B$ through the association instance $g_1$. $E$ represents the element-of relation between set type $S$ and element type $C$. The set type $S$ contains two sets: $s_1$ and $s_2$. $s_1$ consists of the elements $c_1$ and $c_2$, and $s_2$ consists of the single element $c_1$. $P$ associates both the sets $s_1$ and $s_2$ with $a_1$. The dashed arrow $i$ from $B$ to $C$ indicates that $B$ is a subtype of $C$. In that subtyping, $b_3$ corresponds to $c_1$. $B$ inherits $C$’s properties, which explains dotted arrow $c$. $c$ is a derived arrow, and is usually omitted from the diagrams.

**3.3.3 Features**

The characteristics that capture the essence of the associations described in sections 3.5.1 to 3.5.6 are *dependency*, *no-knowledge*, *inheritance*, *not-null*, and *function*. Naturally, *direction* plays a role, e.g. to indicate which class is dependent on another class. Giving an
association a direction enables the use of the terms *source class* and *destination class*. In this framework, the notion of *class* refers to a type of object, or to a type of association because associations are considered to be first class citizens.

Features find their way into the type graph by attaching them to edges. Of course, multiple features may be attached to a single edge.

**Dependency** The meaning of the *dependency* feature is as follows. If an object \(a\) of class \(A\) is associated with an object \(b\) of class \(B\) where \(A\) depends on \(B\), and \(b\) is deleted, \(a\) is deleted too. For the definition of inheritance structures, a *double*, or bi-directional dependency is used. In this case, the source depends on the target, and vice versa.

Syntactically, all edges that possess the dependency feature comprise the set \(\mathcal{D}\). The edges that possess the *double dependency* feature comprise the set \(\mathcal{H}\).

The semantics of the dependency feature is defined in section 3.4, which deals with dynamic semantics.

**Not-null** The *not-null* feature indicates that its associated arrow is a total relation. The arrows that possess the not-null feature comprise the set \(\mathcal{M}\).

**Function** In general, the population of an arrow can be a relation. The *function* feature is used to restrict the possible populations of an arrow to functions. The set \(\mathcal{F}\) contains all arrows that have the function feature.

**Inheritance** Arrows that possess the *inheritance* feature are contained in the set \(\mathcal{I}\). The population of an arrow that has the inheritance feature is a total and injective function.

Access to inherited properties is provided by the addition of inferred arrows: \(\text{InhProp} : \mathcal{I} \times \mathcal{G}_1 \to \mathcal{G}_1\). If \(\text{InhProp}(i, g) = h\), \(i : A \to B\) and \(g : B \to C\), then \(h : A \to C\). The population of the inferred arrow consists of the composition of the inheritance arrow \((i)\) and the inverse property arrow \((g)\): \(\text{Pop}(g) \circ \text{Pop}(i) = \text{Pop}(h)\).

**No-knowledge** The final feature deals with knowledge. If an object \(a\) has no knowledge of an object \(b\), the methods of \(a\) cannot refer to the association of \(a\) with \(b\), which should be reflected in the behavior that is exhibited by \(a\). The edges with the *no-knowledge* feature comprise the set \(\mathcal{K}\).

To define the meaning of the no-knowledge feature, extra arrows are used. All arrows that do not have the no-knowledge feature can be traversed in both directions. This is shown in the type graph by inserting the reverse arrows. However, if an arrow has the no-knowledge feature, its reverse arrow is not incorporated in the type graph. This is reflected by the inverse function \(\text{Inv} : \mathcal{G}_1 \setminus \mathcal{K} \to \mathcal{G}_1\), which does not include arrows with the no-knowledge feature in its domain.

Semantically, \(\text{Pop}(\text{Inv}(e)) = \text{Conv}(\text{Pop}(e))\), where \(\text{Conv}(R)\) yields the converse of relation \(R\).
3.3.4 Constraints

Certain properties of associations cannot be expressed solely by combining features, because they pertain to a group of arrows. These properties are captured by constraints. Depending on the type of constraint, this group of arrows must either have the same source or the same destination, which is called the base of the constraint.

The constraints presented in this section are very common, and have proven their importance with regard to e.g. identification purposes and defining mappings to implementation platforms [Hal95].

Characteristic for the constraints used in information modeling is that, in general, they may be used to relate a number of otherwise unrelated entity types. In an object-oriented context, having a constraint span arbitrary object types might be considered to be a violation of the Locality Principle, as the responsibility to maintain the invariant specified by the constraint cannot be assigned to a particular object type. Therefore, the constraints presented in this chapter only range over a set of arrows that are connected to a single object type.

Syntactically, a constraint consists of a set of edges. All constraints are contained in the set \( \mathcal{C} \). The function \( \text{Base} : \mathcal{L} \rightarrow \mathcal{G}_0 \) yields the base of a constraint.

### 3.3.4.1 Cardinality

The cardinality constraint imposes limits on the number of times a combination of values may occur in the population of an object type.

Consider the sample cardinality constraint in figure 3.2, denoted as \( \text{card} \{ p, q \}, 2, 3 \), where class \( A \) is associated with class \( B \) with cardinality \( \langle 2, 3 \rangle \). There are two ways to satisfy this constraint. Either \( f \) is empty, or each combination of instances of \( A \) and \( B \) is associated with a minimum of 2 and a maximum of 3 instances of \( C \). This occurrence frequency can be found by examining the population of \( f \). The upper bound may be infinite, in general.

![Figure 3.2: Example cardinality constraint](image)

Syntactically, a cardinality constraint consists of a set of edges, a minimum, and a maximum cardinality. The edges that are involved in the constraint are yielded by the function \( \text{lEdges} : \mathcal{L} \rightarrow \varphi(\mathcal{G}_1) \), where \( \mathcal{C} \subseteq \mathcal{L} \) is the set of cardinality constraints. The upper and lower
bounds of a cardinality constraint are yielded by the functions $\text{Upb} : \mathcal{C} \rightarrow \mathbb{N} \cup \{\infty\}$ and $\text{Lwb} : \mathcal{C} \rightarrow \mathbb{N}$ respectively.

The edges that are involved in a syntactically correct cardinality constraint $c \in \mathcal{C}$ must all have the same source:

$$\forall e \in \text{Edges}(c) \ [\text{src}(e) = \text{Base}(c)]$$

$\text{card}(C, n, m)$ means there is a $c \in \mathcal{C}$ such that $|\text{Edges}(c)| = C$, $\text{Lwb}(c) = n$, and $\text{Upb}(c) = m$. The notation $\text{Pop} \models c$ is used to denote the fact that a population $\text{Pop}$ complies with the constraint $c$.

For the definition of the semantics of the cardinality constraint a relational view is taken. All edges that are involved in the constraint are joined to form a new relation. The type of join that is used is close to the natural join. The result of joining relations $R_1 : A \rightarrow S_1$, $R_2 : A \rightarrow S_2$, \ldots $R_n : A \rightarrow S_n$ is $\forall_{1 \leq i \leq n} R_i$, which is a subset of $A \times (R_1 \times \ldots \times R_n)$:

$$\forall_{1 \leq i \leq n} R_i = \{(a, (s_1, \ldots, s_n)) | \bigwedge_{1 \leq i \leq n} (a, s_i) \in R_i\}$$

This leads to the following definition of the semantics of a cardinality constraint $c \in \mathcal{C}$:

$$\text{Pop} \models c \iff \forall_{p \in |\text{Edges}(c)|} \text{an } [\text{Lwb}(c), \text{Upb}(c)]\text{-relation over } \text{Base}(c)$$

$R \equiv A \times (R_1 \times \ldots \times R_n)$ is an $[m-n]$-relation over $A$ iff

$$\forall r_1 \in R_1, \ldots, r_n \in R_n [m \leq |\{a \in A | (a, \langle r_1, \ldots, r_n \rangle) \in R\}| \leq n]$$

**Example 3.3.2**

In this example the semantics of the constraint $\text{card}([p, q], 2, 3)$ of figure 3.2 is illustrated.

Consider a population of $f$ that consists of three instances, $f_1$, $f_2$, and $f_3$, associated with instances from $A$ and $B$ as follows:

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$f_3$</td>
<td>$a_2$</td>
</tr>
</tbody>
</table>

Because $r$ does not take part in the definition of the semantics of the constraint, its population is omitted.

Using this population as a starting point, the joining of $p$ and $q$ yields the results shown in the table below:

<table>
<thead>
<tr>
<th>p $\bowtie$ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$ $\langle a_1, b_1 \rangle$</td>
</tr>
<tr>
<td>$f_2$ $\langle a_1, b_1 \rangle$</td>
</tr>
<tr>
<td>$f_3$ $\langle a_2, b_2 \rangle$</td>
</tr>
</tbody>
</table>
Syntactically speaking, this constraint is correct because both edges that are involved in the constraint \((p \text{ and } q)\) have the same source \((f)\).

As far as the semantics of the constraint is concerned, this population violates the constraint because \(p \not\propto q\) is not a \([2-3]\)-relation over \(f\): \(\langle a_2, b_2 \rangle\) is associated with an instance from \(f\) only once in the population of \(p \not\propto q\).

Omitting \(f_3\) from the population of \(f\) would result in a valid population, as \(p \not\propto q\) would contain two instances of \(\langle a_1, b_1 \rangle\), fulfilling the requirements of a \([2-3]\)-relation. \(\square\)

### 3.3.4.2 Uniqueness

The uniqueness constraint is a specialized case of the cardinality constraint. In effect, a uniqueness constraint \(u\) over a set \(U \subseteq \mathcal{L}\) of edges (denoted as \(\text{unique}(U)\)) is equivalent to a cardinality constraint that eliminates multiple occurrences: \(\text{unique}(U) \equiv \text{card}(U, 0, 1) \equiv \text{card}(U, 1, 1)\). This matches the informal meaning that a combination of values of the population of the edges in the set \(U\) may only occur a single time, or not at all. Therefore, if \(c = \text{card}(U, 0, 1)\):

\[
\text{Pop} \models u \iff \text{Pop} \models c
\]

The set \(U\) contains the uniqueness constraints. To provide access to the edges that are involved in a uniqueness constraint, the \(\text{lEdges}\) function is extended to range over uniqueness constraints as well: \(\text{lEdges} : \mathcal{C} \cup U \rightarrow \varphi(\mathcal{G}_1)\).

Consider for example the symmetric association in example 3.5.1. To ensure that the population of the \text{privilege} association does not contain two identical facts, a uniqueness constraint has to be placed on both \(p\) and \(q\): \(\text{unique}\{p, q\}\).

### 3.3.4.3 Total role

The definition of the cardinality constraint implies that it cannot be used to specify that an object of a class \textit{must} participate in a particular association. Cardinality constraints become relevant after an association exists. A different constraint, the total role constraint, is used to enforce mandatory participation of objects in an association.

Syntactically, a total role constraint \(c\) is a non-empty set of edges \(T\), and is denoted as \(\text{total}(T)\). The total role constraints comprise the set \(\mathcal{T}\). As is the case for uniqueness and cardinality constraints, the \(\text{lEdges}\) function yields the edges that are involved in a total role constraint. It is therefore extended to encompass total role constraints as well: \(\text{lEdges} : \mathcal{C} \cup U \cup \mathcal{T} \rightarrow \varphi(\mathcal{G}_1)\). In order to be meaningful, the edges that are involved in a total role constraint \(c\) must all have the same target, \(\text{Base}(c)\).

Formally, the meaning of a total role constraint \(c\) is the following:

\[
\text{Pop} \models c \iff \bigcup_{x \in \text{lEdges}(c)} \text{ran} \circ \text{Pop}(x) = \text{Pop}(\text{Base}(c))
\]

This reflects the intuitive meaning that all instances of \(\text{Base}(c)\) participate in the population of at least one of the edges involved in the constraint.
Example 3.3.3
Consider the sample total role constraint $c = \text{total} \{ \{p,q\}\}$ of figure 3.3, with the following population:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$p$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a_1, a_2, a_3}$</td>
<td>$f_1$</td>
<td>$a_1$</td>
</tr>
</tbody>
</table>

It is clear that $\text{Pop} \not\models c$, because $\text{ran} (\text{Pop}(p)) \cup \text{ran} (\text{Pop}(q)) = \{a_1, a_2\} \neq \{a_1, a_2, a_3\} = \text{Pop}(A)$. Were $a_3$ to be removed from the population of $A$, constraint $c$ would hold.

3.3.4.4 Extensional uniqueness

The formalization of the extensional uniqueness constraint follows from the observation that such a constraint is not violated if and only if two sets that consist of exactly the same elements are the same set.

Syntactically, an extensional uniqueness constraint $c \in \mathcal{E}$ (the set of EU-constraints) consists of a single edge in the domain of the function $\text{clt}$. This singe edge is yielded by the function $\text{IEdge} : \mathcal{E} \rightarrow \text{dom} (\text{clt})$. For an EU-constraint to make sense, the base of the extensionally unique edge must be the source of a single edge that has the set type as its destination. This unique edge is yielded by the function $\text{CoEdge} : \mathcal{E} \rightarrow \text{ran} (\text{clt})$. $\text{CoEdge}$ can be expressed in terms of the $\text{clt}$-function: $\text{CoEdge}(c) = \text{clt}(\text{IEdge}(c))$.

To express the semantics of an EU-constraint formally, the notion of relational image is used. If $R : A \rightarrow B$ a relation and $a \in A$, then the relational image $\{a\}$ of $a$ is defined as follows:

$$R(\{a\}) \equiv \{ b \in B | R(a,b) \}$$

Using this definition, the formal semantics of an EU-constraint $c$ is:

$$\text{Inv}(q) \circ p(\{a_1\}) = \text{Inv}(q) \circ p(\{a_2\}) \Rightarrow a_1 = a_2$$

where $q = \text{IEdge}(c)$, and $p = \text{CoEdge}(c)$.  

Example 3.3.4

The portfolios used in the world of fashion models form a good example of sets. A modeling agency has a large amount of pictures of each model. From this range of pictures different portfolios can be constructed, for instance based on the type of prospective customer. Figure 3.4 depicts this situation, explicitly modeling the association between Photos and Portfolios.

![Figure 3.4: Set example](image)

Example 3.3.5

Figure 3.4 contains an example of a set called Portfolio that consists of Photos. To enforce real set-like behavior of the inclusion association, an EU-constraint is placed on q. As an example, consider the following population of inclusion:

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>a1</td>
<td>e1</td>
</tr>
<tr>
<td>e2</td>
<td>a2</td>
<td>e2</td>
</tr>
<tr>
<td>e3</td>
<td>a1</td>
<td>e3</td>
</tr>
<tr>
<td>e4</td>
<td>a2</td>
<td>e4</td>
</tr>
</tbody>
</table>

This population would violate the constraint $\text{extunique}(q)$, because both the set represented by $b_1$ and the set represented by $b_2$ consist of the same elements ($\{a_1, a_2\}$). One way to obtain a population that does not violate the example EU-constraint is to remove $a_2$ from the set $b_1$ by deleting the instances $(e_2, a_2)$ and $(e_2, b_1)$ from $p$ and $q$ respectively.

3.4 Dynamic semantics

Not all features can be captured purely in terms of instantiations. The dependency feature can only be formalized through focussing on population changes (dynamic semantics).

The dynamic semantics described in this section is based on a transaction model. Multiple objects and associations are updated in a single transaction. Adding new objects and associations to the system is relatively simple. The population is updated and constraints are verified. If all constraints still hold, the transaction is valid. If not, the whole transaction is rolled back, and the population is not changed.
Deleting objects or associations is more complex, because of the dependencies between objects. The new population that results from a delete operation is calculated in two steps. First all instances of the populations of nodes are collected that depend directly or indirectly on the instance that is deleted. This is done using a fixed point calculation. The dependency of \( x \) on \( y \) in a population \( \text{Pop} \) is notated as \( x \triangleright_{\text{Pop}} y \). Of course, the dependency relation is reflexive and transitive.

[D1] \( x \triangleright_{\text{Pop}} x \)

[D2] \( x \triangleright_{\text{Pop}} y \land y \triangleright_{\text{Pop}} z \implies x \triangleright_{\text{Pop}} z \)

The collection of instances-to-be-removed is expanded to encompass any instances that directly depend on the current collection of instances-to-be-removed. This is done by following the edges with the dependency feature, and checking the population of their destination object types.

[D3] \( \exists e \in \mathcal{E} [\text{Pop}(e)(x) = y] \implies y \triangleright_{\text{Pop}} x \)

All parents of instances in an inheritance hierarchy must be removed.

[D4] \( \exists e \in \mathcal{H} [\text{Pop}(e)(x) = y] \implies x \triangleright_{\text{Pop}} y \)

All descendants of an instance that is to be removed must also be deleted.

[D5] \( \exists e \in \mathcal{H} [\text{Pop}(e)(y) = x] \implies x \triangleright_{\text{Pop}} y \)

A singleton set depends on its sole element. As a result, the removal of that single element causes the removal of the then empty set. The function \( \text{dom} \) yields the domain of its argument.

[D6] \( e \in \text{dom}(\text{clt}) \land y \in \text{Pop}(\text{src}(e)) \land \neg \exists y' \in \text{Pop}(\text{src}(e)), y' \neq y [\text{Pop}(e)(y) = \text{Pop}(e)(y')] \implies \text{Pop}(e)(y) \triangleright_{\text{Pop}} y \)

Once all dependent instances are known, the population is updated. The population of the object types is updated using the function \( \text{Delete} (\text{Pop}, V) \), where \( V \) is the set of instances to be deleted (determined through applying a fixed point calculation to axioms D1 to D6). The \( \text{Delete} \) function consists of two parts. The first, \( \text{UpdObj}(\text{Pop}, V) \), updates the population of the object types, while the second part, \( \text{UpdEdg}(\text{Pop}, V) \), updates the population of the edges.

The new population of the object types consists of those instances that are not dependent on any instance that is to be deleted.

\[
\text{UpdObj}(\text{Pop}, V) = \lambda x \in \mathcal{G}_{\alpha}. \{ z \in \text{Pop}(x) | \forall v \in V [z \not\in_{\text{Pop} v}] \}
\]
The definition of the updated population of the edges of the state graph uses a relation restriction which is defined as follows. If \( f : A \rightarrow B \) a relation, then \( f[A][B] \) is a relation defined by

\[
f \cap A \times B
\]

Due to the removal of instances from the source and target object types, the domain and range of the relations represented by the edges may be changed. Intersecting the old population of the edges with the product of the new domain and range ensures that the new edge population indeed still represents the correct relation. Intuitively, references to deleted objects are removed at this stage.

\[
\text{UpdEdg}(\text{Pop}, V) = \lambda x \in \mathcal{G}_1.x[\text{UpdObj}(\text{Pop}, V)(\text{src}(x))][\text{UpdObj}(\text{Pop}, V)(\text{tgt}(x))]
\]

Using these two definitions, the \textit{Delete} function that yields a new population based on the old population and a set of instances that is to be deleted can be defined as follows:

\[
\text{Delete}(\text{Pop}, V) = \text{UpdObj}(\text{Pop}, V) \cup \text{UpdEdg}(\text{Pop}, V)
\]

Figure 3.5: Cascading object removals due to the impact of dependencies when deleting object \( b_2 \)

\textbf{Example 3.4.1}

Consider the model of figure 3.5 with a population \( \text{Pop} \), where \( S \) represents a set. \( E \) are the elements of \( S \), which is indicated by \( \text{clt}(q) = p \). \( B \) and \( C \) are symmetric associations \( (x, y, r, s \in D) \).

To compute the population of this model if \( b_2 \) is deleted, the axioms D1–D6 are used to determine the set \( V \) of objects that depend on \( b_2 \). Because of D1, initially \( V = \{b_2\} \). According to D3, \( f_2 \) and \( r_1 \) depend on \( b_2 \) because \( p \in D \) and \( x \in D \). Again using
sec. 3.5

3.5 Associations between classes

Object-oriented systems exist by the grace of the cooperation between objects. In order to enable cooperation between two classes, these two classes must be related in some way. At least one class must know of the existence of the other before anything useful can take place. Associations reflect the fact that two classes are connected. Such a connection can express many things, depending on the interpretation given to it by the analyst. In [KE94], Kilov and Ross state that “Associations can and should be an integral part of a standardized object model.” This thesis takes this thought one step further by providing associations with their own implicit identity, analogous to objects. In effect this makes associations first class citizens.

Section 3.2 provided an introduction to classes and objects, the building blocks for an OO data model. The ability to connect classes by means of “associations” provides the means to capture the overall structure of a system. Obviously, to express differences in the way objects are connected, different kinds of associations are needed. Unfortunately, in the area of associations (or relations, attributes, properties, features), the OO jungle is particularly dense. Different methods use many different association types. Often, the precise meaning of a particular association type is left to the intuition (or worse, to the imagination) of the reader. Distinct association types can carry the same name, and a particular association type can be referred to by various names. For example, in [CLF93] it is remarked that there is a difference between their relationships and OMT’s associations [RBP+91], but that difference (if it exists at all) is very hard to pin down without precise definitions of those concepts.

The modeling of information relies heavily on the availability of relation types. Object types in a data model are often meaningless without a clear description of how they are related to each other. Wirfs-Brock et al. [WWW90] describe several special relationships including PartOf, DependsUpon, HasKnowledgeOf, IsAnalogousTo, and IsKindOf. [KR94] describe a modeling library consisting, among other things, of a number of relationship types such as symmetric relationships, components, and dependencies. Jacobson’s analysis model features static acquaintance associations, attributes, dynamic associations, inheritance, uses, extends, and aggregation relationship types. In order to shed some light on what different association types mean, we try to express them in terms of the general features and constraints defined in section 3.3.
The remainder of this section presents a number of basic association types. The association types are taken from the OO domain, and, where possible, strengthened with aspects from information modeling. For each type we provide both an informal and a formal meaning.

3.5.1 Symmetric associations

The relation type of the OO data model framework is based on the fact types of object-role modeling techniques such as NIAM [Hal95] or PSM [HW93], and the relationships of ER [Che76] and its many variants. The symmetric association is a common relation in object-oriented modeling techniques, albeit that different names are used. We prefer the name symmetric association to distinguish it from other association types, and because it points to the characteristic feature, which is its symmetry. [RBP+91] calls it association, [EKW92] relationship, Coad and Yourdon call it instance connection in [CY90], whereas [KR94] actually use the term symmetric relationship. All these relations have a number of characteristics in common:

1. Each fact (instance of a relation) has exactly one identity. Facts are uniquely determined by their components.
2. A symmetric association consists of a fixed but arbitrary number of components. Associations consisting of a single component, called unary associations, are allowed.
3. The components, often referred to as roles, of a fact have knowledge of each other’s existence. The static acquaintance associations of [JCJO92] do not conform to this property; they can only be accessed from one end.
4. All components of a fact instance must have a value.
5. To maintain referential integrity and avoid the well-known problem of dangling pointers, instances depend on the existence of their components. If one of the component instances is destroyed, the associated association instance vanishes too. Naturally, there are no consequences for the previously-associated instances if the association is deleted.

Referential integrity in data modeling is linked with the concept of foreign keys. If a column \(q\) in a relational table \(A\) is the source of a foreign key, the values in that column must form a subset of the values of the target column \(r\) of the table \(B\) the foreign key refers to (see figure 3.6). This concept can be mapped directly onto the object-oriented case. In OO, an association of an object type \(C\) with another object type \(D\) does not violate referential integrity if the population of each role \((u\text{ and } w)\) of the association is a subset of the extent \((t\text{ resp. } x)\) of the class it is linked with \((C\text{ and } D,\text{ respectively})\).

This definition coincides with the conformity rule of the Predicator Model [BHW91], a formal object-role modeling technique based on NIAM. The conformity rule states that the instances of roles \((u, w)\) of a fact type \((f)\) must be members of their associated object type \((C, D)\).
Example 3.5.1
The mobile phone is a popular phenomenon in this era of global communications. Figure 3.7 models a very small part of this area. The object types are Phone and Network, which are associated by a privilege association type. The privilege association type expresses which (cellular) Phone has the right to use which Network.

It is obvious that the associations between a phone and its networks are no longer relevant when that phone is removed from the system. Therefore those associations should also be deleted. The same situation occurs if a network is removed (e.g. due to a reorganisation or new legislation). In that case the previously existing associations with the phones of that network must be deleted, implying that a privilege instance depends on both its object types.

As far as notation is concerned, arrows instead of lines are used to emphasize the dependency of the association type on its object types. The association type is drawn using the same symbol as an object type because associations are first class citizens, and therefore objects in their own right.

To elaborate on this example of a binary association type, assume that because of free market forces individual phone users can negotiate their own system of rates, allowing them to gear the cost of using different networks to their specific needs. This leads to the introduction of a new object type Rate, and a ternary association between Phones, Networks, and Rates as shown in figure 3.8.

A symmetric association type is defined using dependency, not-null, and function features, in addition to the uniqueness constraint. First of all, an instance of an association depends
on all its constituting parts. Therefore, all outgoing edges of a symmetric association should possess the dependency feature. Applying the not-null feature to all the arrows ensures that all the components of each fact instance have a value. The function feature ensures that the components behave like functions. Furthermore, all instances of the association should be unique, which is achieved by adding a uniqueness constraint that encompasses all outgoing edges of the association type.

In figure 3.7, this leads to the following: $p$ and $q$ should have the dependency feature, so $p, q \in D$. The not-null and function features are assigned to all arrows of the fact type, so $p, q \in M$ and $p, q \in F$. Each combination of $(p, q)$ values must be unique, which is expressed by the uniqueness constraint $\text{unique} \{(p, q)\}$.

### 3.5.2 Uni-directional associations

Uni-directional associations are used to model situations where two objects are associated, but only one has knowledge of the existence of the other. For example, if class $A$ is uni-directionally related with class $B$, instances of $A$ have knowledge of the existence of objects of class $B$, but instances of $B$ are not aware of the objects of $A$. This can be expressed by giving the appropriate edge in the type graph the no-knowledge feature.

The choice whether or not to support uni-directional associations is an interesting issue. Basically there are two approaches to it. The first argues that uni-directional associations are able to express certain situations with greater accuracy, and force analysts to consider the symmetry of the relations they use [JCJO92]. Therefore, it would be desirable to incorporate one-way associations into our framework. The other school of thought takes a more information-modeling oriented approach [CLF93, EKW92, RBP+91], and argues that the issue of access paths should not be addressed in the analysis phase, but in the design phase [EKW92] uses bi-directional associations, although their notation uses arrows which suggest a certain directional element. As Rumbaugh puts it in [RBP+91]: “Although associations are modeled as bidirectional they do not have to be implemented in both directions.” The design stage must accommodate specific access paths (and directions), e.g. by means of one or more pointers.
Whereas information hiding and encapsulation are very important to high quality designs, it is equally important that information is accessible from both sides of an association. Therefore, if the association happens to be uni-directional, its inverse must be available in the construction of queries. In fact, two levels of knowledge can be distinguished. First there is the knowledge that an individual object has. Secondly, there is the knowledge that can be used at the system level. This is the knowledge that is available when constructing queries, independent of the uni- or bidirectionality of associations. Our framework accommodates both uni- and bidirectional associations, because a justification for the existence of uni-directional associations in analysis models does exist (see example 3.5.2), even though they are quite seldomly encountered in practice. Uni-directional associations do have their use when constructing layers of abstraction or subsystems. Associations between different layers/subsystems can be made using uni-directional associations, thus hiding the rest of the layer/subsystem they are part of from the rest of the world.

Example 3.5.2

Figure 3.9 contains a model of a spy ring. In this model, uni-directional associations are used to express the fact that instances of higher levels have knowledge of the existence of instances at the lower levels. In this example, a US-spy knows the US-citizens that gather information for him, and they know him. On the other hand, he does not know his US-controller. But the US-controllers know their US-spies, of course.

Uni-directional associations suggest the incorporation of data into the source class. From an integrity point of view this may lead to a problem with dangling references, in this case a relation with a no-longer existing object. One sure way to avoid this is to add a dependency feature to the relation. This would probably involve a bookkeeping system to keep track of which object is related to which other objects. Instead of cluttering the definition of a class, this responsibility can be moved beyond the encapsulation barrier of the source object and placed in a special association construct.
A completely different approach to the integrity problem when using uni-directional associations is to limit their target types to concrete types. Concrete types cannot have associations with other objects: they are always found on the fringes of a model. Therefore, deletion of an object whose sole relations are to concrete types is perfectly safe.

Objectory’s static acquaintance associations [JCJO92] can be modeled as uni-directional associations without the dependency feature: maintaining referential integrity becomes the explicit responsibility of the analyst specifying the behavior associated with that relation.

Uni-directional associations enter the type graph by introducing a new association which connects the source of the uni-directional association explicitly to its destination. This new association is treated in a manner similar to a symmetric association, with the following exception: the edge from the new association to the target of the uni-directional association is given the no-knowledge feature. This ensures that the association can only be traversed in the intended direction.

Using figure 3.10 as an example, this means that \( f \) is introduced to model the uni-directional association, which results in \( p, q \in D, p, q \in M, p, q \in F, \) and unique(\{p, q\}). The unidirectionality is arrived at by giving \( q \) the no-knowledge feature: \( q \in K \).

![Figure 3.10: Translation of uni-directional association to type graph](image)

### 3.5.3 Attributes

Just as is the case with uni-directional associations, the inclusion or exclusion of attributes in the models of a method is surrounded by controversy. [CLF93] motivate their choice to incorporate attributes by stating that attribute relationships represent intrinsic, definitional properties of an object, whereas other relationships describe contingent, incidental connections between objects. Other methods (e.g. [RBP+91, JCJO92, WWW90]) also use attributes. Often (e.g. in [SL96]), attributes are associated with concrete types (see section 3.2), something that is not relevant in an object-oriented analysis context, although it guarantees the absence of dangling references when performing object deletions (see section 3.5.2). This also appeared in [HH97], where attributes and associations as described in [JCJO92] could be consistently treated uniformly, indicating that there is no fundamental difference between the two if viewed from a sufficiently high level of abstraction.

At the design level, Meyer’s view of an attribute comprises both relations with other objects and with predefined types [Mey88], and Embley et al. describe a method that bans attributes from the object model [EKW92]. In describing the object-oriented approach to information modeling in [KR94], it is said that “Time and again, the abstraction level used by the analyst is based directly and improperly on outdated implementation technology...
The overemphasis on attributes in information modeling is another example. These are ideas drawn from database implementation.

This thesis models attributes as symmetric associations, which allows the state of an object to be described through the values of its associations. Note that this is made possible because objects are the sole instances in populations of our models. Values from concrete domains are encapsulated in value objects. Therefore, there are no values from concrete domains that can be used as attributes, but cannot be used as instances of an association.

3.5.4 Multi-valued properties

Some methods (see e.g. [CD94]) use a special construct, the multi-valued property, to associate an object with a number of objects. For example, an employee of a large company may have more than one phone number under which he or she can be reached. In this case it is convenient to group those numbers in a single multi-valued “phones” property. In the examples of this chapter, multi-valued properties are drawn with a double arrow, as can be seen in figure 3.11. The behavior of a multi-valued property resembles a set, in the way that a single object is associated with a set of object instances.

![Figure 3.11: A multi-valued property](image)

Other methods [JCJO92, RBP+91, EKW92, Bap94, CLF93] do not include an explicit construct for multi-valued properties, but allow for all properties to point to multiple objects. Constraints (cardinality, multiplicity, participation) are used to indicate how many objects can be associated using a single property. Although multi-valued properties can be simulated by means of constraints, true set-like behavior is not offered by any of the methods mentioned above. Two objects can be associated with sets consisting of the same elements, but with a different identity, opening the possibility of update anomalies (see section 3.5.6). Furthermore, there is nothing to prevent an object to occur more than once in a “set” that is the destination of a multi-valued property, so in terms of set-theory, conventional multi-valued properties resemble multi-sets (or bags) more closely than they resemble real sets. True set-like behavior can be enforced with the use of a more powerful constraint, the extensional uniqueness constraint [HW93, HW94]. From a conceptual point of view a direct representation of sets is desirable. This is argued in more detail in section 3.5.6.

Basically, multi-valued properties can be regarded as attributes that can have multiple values. Because the symmetric associations presented earlier in this section are capable of associating a single instance from one class with multiple instances of another class, multi-valued properties are modeled using symmetric associations.
3.5.5 Inheritance

Instead of trying to give yet another definition of what inheritance means, we adopt the definition given in [CLF93], where they state that if $P$ is a superclass of $Q$, then “every property that applies to all instances of $P$ applies as well to all instances of $Q.” Placed in the context of the framework presented in this chapter, this means that every property that is defined for a superclass applies also to the subclass.

There is not much doubt that inheritance is one of the features that distinguishes object-orientation from other methodologies. However, the availability of inheritance at the analysis level may provoke the argument that object-oriented methods are not really on a conceptual level, as inheritance is seen as an implementation technique. For instance, in [WWW90] is mentioned that “Inheritance also allows us to reuse code; the wheel need not be reinvented every time.” This is true, but the use of inheritance is definitely not limited to the application as an implementation or design technique to achieve more efficient ways of coding a system. The relevance of inheritance for the analysis of systems is made clear in [CLF93], where the following is noted about the concept of inheritance: “Inheritance is a core concept of the object-oriented paradigm, emerging in two basic contexts, abstraction and reuse.” It is abstraction that justifies the use of inheritance in analysis models. When properly used, the inheritance mechanism can be employed to describe objects and the way in which they are related in terms of their associations and behavior in an elegant fashion. Inheritance provides a means to reuse parts of specifications.

Abstraction is also the aim of subtyping in information modeling. Sometimes the literature about object-oriented methods uses the term subtyping when discussing inheritance. [KR94] for example uses the term static subtyping for inheritance. In their terminology, static subtyping implies that the set of instances of a subtype is a subset of the set of instances of its supertype. There is a marked difference between subtyping as it is used in information modeling on the one hand, and inheritance in object oriented methods on the other hand. For instance, in NIAM the way instances are distributed over a subtype hierarchy depends on subtype defining rules [Hal95]. Object-oriented inheritance hierarchies on the other hand allow an arbitrary distribution of objects over the hierarchy. There are no subtype defining rules, except that most methods do not allow an object to change its class once it has been created (metamorphism).

Actually, the characteristics of inheritance are more like those of generalization. The definition of generalization in an OO context of [EKW92] is equivalent to the definition of generalization given in [HW93]. All instances of subtypes (specializations in the terminology of [EKW92]) are also instances of the supertype, and can participate in the relationships of the supertype. The most general approach to subtyping, though, is to link instances of subtypes to instances of supertypes, e.g. by means of an injective function. The approach taken in e.g. [EKW92] can be considered to be a special case of the more general approach. Intuitively, the properties of an inheriting object are distributed over the class hierarchy.

Just like in information modeling subtype hierarchies, it is possible for a class to inherit from more than one superclass. Again paraphrasing [EKW92], “Another way of viewing
multiple inheritance is to see the set of objects in the specialization object class as a subset of the intersection of two or more generalization classes. The specialization having multiple inheritance need not be a proper subset of the intersection, but each member of the specialization must be a member of the intersection."

This leads to the following characteristics for the inheritance association:

1. An instance of a subtype has at least the same associations as those specified for its superclass(es).
2. Distribution of properties over the inheritance hierarchy: instances of subtypes are linked to instances of their supertype(s). Each subtype instance is linked with a single instance of each of its supertypes.
3. Linked instances depend on each other. If a single instance in an inheritance chain is deleted, the whole chain of instances that are linked with that object is to be deleted too.

Because this chapter focuses on the structural aspects of object oriented analysis, inheritance of behavior is not discussed here.

Example 3.5.3

Currently, there are a number of types of telephones available. The first is the conventional phone, connected to the telephone network by means of a wire. The modern, wireless variant connects to a base station, and allows wireless telephoning around the house. And then there is the mobile GSM telephone. Based on these types, a new multi-function phone could be defined, suitable to be used as a wireless phone.
around the house, and as a true global phone when out of reach of the base station (figure 3.12).

All four types have their own associations. For example, the GSM-phone might have associations with networks (see example 3.5.1), and the wireless phone has some encoding schemes associated with it that ensure secure communications around the house, while all share the properties of an ordinary phone, e.g. they all have a phone number. The dashed arrows a, b, and c show the inferred arrows that show the inheritance of the Number property.

An intuitive view of a subtype object is that it consists of a number of objects, each an instance of a different class. Linking these instances yields all associations of the inheriting object. The association is an injective function from the set of instances of the subtype to the set of instances of the supertype. When considering the inheritance association from an update perspective, dependencies between parents and children work both ways. Deletion of the parent implies deletion of the child, and vice versa. The type graph reflects this by awarding the corresponding edge in the type graph with the double dependency feature.

The inheritance feature is used to ensure that an inheritance arrow is a total and injective function. It is total because a child object must be associated with a parent object. It is injective, because even though a subclass might have multiple superclasses, a child can only be associated with a single parent in each superclass. The inheritance of properties of the superclass is handled by the inferred arrows that connect the subclass to the property of the superclass. In figure 3.12 the arrows \( \text{InhProp}(p, \text{Inv}(x)), \text{InhProp}(q, \text{Inv}(x)), \) and \( \text{InhProp}(q \circ s, \text{Inv}(x)) \) (represented in a dashed style) are such arrows. They provide access to the Number property in a manner that is similar to arrow \( \text{Inv}(x) \). Using this structure, function composition gives inheritance.

Supporting multiple inheritance means that it is possible that without further restraints conflicting populations may be introduced. This situation arises when a subtype inherits from a number of supertypes, some of which have a common ancestor. This is the case in figure 3.12, where Phone is an indirect ancestor of Multi-phone via both GSM-phone and Wireless phone. Conflicts are avoided if the composition of all inheritance arrows leading from an instance of a subtype to a common ancestor type result in the same ancestor instance. For figure 3.12 this means that \( p \circ r = q \circ s \).

### 3.5.6 Sets

Gathering a number of instances of a particular type and associating such a group with another object has already been discussed in section 3.5.4, where it was remarked that sets are a construct that is desirable from a conceptual point of view (see also [HM81]). This section introduces true sets into the framework.

An approach to avoid update anomalies is to provide each set with its own identity, which makes it possible to associate objects not with sets directly, but with objects that act as if they are sets. This solution follows the philosophy of object-orientation by hiding the
“implementation” of the set, and delegating the responsibility to uphold the laws of set-like behavior to a single class. This responsibility is expressed in a formal way by the *extensional uniqueness* constraint [HLF96]. The EU-constraint specifies that no two sets consist of exactly the same elements.

Summarizing, the characteristic properties of a set are: each set has its own identity, and sets consisting of the same elements have the same identity. Any object may participate at most once in one particular set. A set depends on the existence of its members. If the last element of a set is deleted, the set is deleted too.

Incorporating sets into the type graph requires the introduction of an additional association that links the set type with its element type. A set is modeled using two edges, a “set” edge from the element association to the set type (say $q$), and an “element” edge from the element association to the element type (say $p$) (see figure 3.4). These two edges are mated in the $\text{clt}$ function: $\text{clt}(q) = p$.

As was the case with symmetric associations, both $p$ and $q$ have the dependency, not-null, and function features ($p, q \in D, p, q \in M, p, q \in F$). Also, combinations of $p$ and $q$ must be unique ($\text{unique}([p, q])$). Finally, to ensure that the element association inclusion behaves like a true set, an extensional uniqueness constraint is placed on the set edge: $\text{extunique}(q)$.

### 3.6 Examples

The use of features and constraints to compose different association types is illustrated by means of examples. Section 3.6.1 presents a language independent example that is based on a common design pattern. Sections 3.6.2 and 3.6.3 apply the features and constraints to the structure models of the Unified Modeling Language and of Jacobson’s Objectory method.

#### 3.6.1 Composite pattern

The example is based on the *Composite* pattern, presented in [GHJV95].

The aim of the *composite* pattern is to provide a uniform appearance for both individual objects and composites. This eliminates the need for client objects to distinguish between composite objects and “elementary” objects. A common example of this situation occurs in drawing editors. A drawing editor lets the user build complex diagrams out of simple components. These complex diagrams can in turn be grouped to create still larger components (see figure 3.13).

The main participants of the *composite* pattern are the *Component* (e.g. Graphic), *Leaf* (Rectangle, Line, …), *Composite* (Picture), and *Client* (Drawing Editor). The *Component* class declares the interface for objects in the composition. The *Leaf* class represents leaf objects in the composition, and defines behavior for primitive objects. The *Composite* defines behavior for components that do have children, and maintains the relations with its
child components. Finally, the Client manipulates objects in the composition through the Component interface.

The symmetric association uses results in $p, q \in D$, $p, q \in M$, $p, q \in F$ and $\text{unique}(\{p, q\})$. The inheritance arrows $r$ and $s$ yield $r, s \in H$ and $r, s \in I$. Not shown in figure 3.13 are the inferred arrows from Leaf to uses and from Component to uses that give access to the inherited properties.

3.6.2 UML

In this section, a selection of the modeling constructs from the Class Diagram of the Universal Modeling Language [BJR97a, BJR97b, BJR97c] are described using the framework of features and constraints. The following relationships are distinguished:

- association (binary or N-ary)
- aggregation
- composition
- attributes
- generalization

UML has separate notations for binary and N-ary ($N > 2$) associations. As they do not differ from a semantic point of view, both can be modeled in the same way. The UML allows tuning of the association type to specific situations. The basic, untuned, version corresponds to our symmetric association. Therefore, the translation of a UML-association follows the pattern described earlier, which results in the following assignment of features and constraints to the association in figure 3.14: $p, q, r \in D, M, F$ and $\text{unique}(\{p, q, r\})$.

The property isNavigable is a good example of the tuning capabilities of UML. It is placed on the target end of an association to specify whether or not traversal from a source instance to its associated target is possible. This UML-property can be captured by the no-knowledge feature, by attaching it to non-navigable association ends.
UML offers two constructs to group objects: *aggregation* and *composition*. Both act in large parts in the same way as symmetric associations. Composition however, as shown in figure 3.15, implies that the lifetime of a part coincides with the lifetime of the whole. That means that in the example of figure 3.15, \( p, q, \) and \( r \) all possess the double dependency feature \( (p,q,r \in H) \), because the double dependency ensures that deleting the Window also results in deletion of the associated Header, Scrollbar, and Panel. This dependency is comparable to the component dependence of [KP97]. The similarity of composition with symmetric associations is illustrated by the mapping of the uniqueness constraints and the assignment of the features: \( \text{unique}([x,p]), \text{unique}([y,q]), \text{unique}([z,r]) \), and \( x, y, z, p, q, r \in M, \mathcal{F} \). Attributes have the exact same semantics as compositions.

Ordinary aggregation in UML differs from composition in that it does **not** require that the lifetime of a part coincides with the lifetime of the whole, because a part of one aggregate can also be a part in another aggregate. This is distinction is reflected in the use of single dependencies instead of double dependencies. Therefore, if the composition associations in figure 3.15 were to be interpreted as aggregation associations, \( p, q, \) and \( r \) would possess the single dependency feature \( (p,q,r \in D) \).

Figure 3.16 contains an example in which a *SavingsAccount* and a *CreditAccount* inherit from a general *Account* class. Generalization in UML corresponds to the general notion of inheritance. The UML generalization association can therefore be characterized using the
inheritance and double dependency features. For figure 3.16 this means that \( p \) and \( q \) have both the inheritance and the double dependency feature: \( p, q \in \mathcal{I} \) and \( p, q \in \mathcal{H} \).

### 3.6.3 Objectory

The Objectory method uses an uncomplicated structure model. It uses acquaintance associations, attributes, and inheritance associations. Aggregates are modeled using \textit{consists-of} associations, but as they have the same semantics as acquaintance associations, they will not be discussed in this section. The semantics of inheritance associations is equal to inheritance as described by the UML.

The distinguishing feature of acquaintance associations is the use of \textit{directionality}. An acquaintance association between two objects means that the source object knows of the existence of the destination object. Figure 3.17 contains an example acquaintance association, where \textbf{MotorManagement} objects know of the existence of \textbf{FuelInjector} objects, but not vice versa. This is captured by assigning the no-knowledge feature to \( q \). In all other aspects, acquaintance associations act in the same way as the symmetric association presented in section 3.5.1. Therefore, the combination of \( p \) and \( q \) must be unique (\textit{unique(\{p, q\})}), instances of the relation \( R \) depend on their constituting components (\( p, q \in \mathcal{D} \)), are total (\( p, q \in \mathcal{M} \)), and are functions (\( p, q \in \textit{funcs} \)).

Attribute associations are identical to acquaintance associations, with the exception of the type of the target: acquaintance associations connect two classes, whereas attribute associations connect a class to a primitive type. As this has no impact on the characterization...
of the semantics of the association, attribute associations are characterized indentically to acquaintance associations.

### 3.7 Conclusion

This chapter presents an investigation into the concepts that underly object-oriented conceptual data modeling. In the course of the investigation many different ways of approaching object-oriented analysis were encountered. Although these interpretations of object-oriented data modeling differ in many particular issues (e.g. whether or not to include attributes, or whether or not to view associations as first class citizens), they all gravitate to a clearly identifiable area, the core of OO data modeling.

The OO data core consists of a small number of features and constraints. Combining features and constraints allows the definition of a wide range of concepts that are relevant to the modeling of the structure of a system, both in the field of OO, and in the field of information modeling.

Therefore it was natural that an effort was made to integrate classical information modeling in the object-oriented way of modeling without violating the ideas behind OO. These extensions provide OO with a much more effective means to handle analysis in contexts where information modeling plays a larger role than in the traditional OO domains.

The strong focus on the structural side of object-oriented modeling has resulted in the nearly total exclusion of object behavior and communication from this chapter. As a result, all features (with the exception of the dependency feature) and constraints deal with the static aspects of OO-modeling. One area where this is felt concerns inheritance. The relation between the behavior of supertypes and subtypes might influence the structure of an object model, but that is a research area in its own right.

A side effect of the incorporation of information modeling concepts is that the gap between analysis and design becomes smaller. It is much claimed that the transition from analysis to design in OO is relatively smooth, but this hinges on the ability to completely describe at the analysis level what a system is supposed to do. With the added conceptual power of information modeling concepts, achieving a complete analysis model is easier, leaving less gaps to be filled during the design stages.

In this chapter, a first effort is made to capture the precise meaning of (sometimes rather informal) concepts used in the field of object-oriented analysis, using a number of OOA methods from varying backgrounds. The concepts of the OO data core presented in this chapter have been formalized to provide an unambiguous semantics. The formalization is based on set theory.

It is unavoidable that some open ends, e.g. a powerful composition mechanism, still remain. We think that this chapter provides a step towards more clarity of exactly what these open ends are, and how they might be filled in. It has therefore been an objective to present a framework that is easily extensible, both in the sense that new features and constraint
types can be added to the existing work in a straightforward way, and in the sense that this framework allows the definition of other association types.
Chapter 4

Behavior Modeling

4.1 Introduction

This chapter addresses the need for formal descriptions of techniques by providing a formalization of object behavior models. The starting point for the formalization is Jacobson’s Objejectory [JCJO92]. This choice is based on the underlying philosophy, and its proven value in practice. Equally important, Objejectory offers an extensive treatment of behavior modeling and represents the object-oriented approach to behavior modeling well. Furthermore, in contrast to many other methods, it is not just an evolution of a structured approach, and has not inherited the characteristics of such an approach.

Objejectory emphasizes that software development is a process of change. The motivation for this emphasis lies in the costs of performing changes to a system. These costs constitute the main part of the total life cycle costs of most systems. As the functionalities of real-life systems have to be changed on a fairly regular basis, it is important for a software engineering method to deliver a system that is easily extensible and robust.

Extensibility is important because it has to be relatively easy to adapt a system to changes of the environment or to add new functionalities required by the customer. A concept that is strongly related to extensibility is robustness. In [JCJO92] Jacobson defines a system to be robust if its structure remains stable, despite the multitude of changes during its lifetime. In Jacobson’s view, robustness counters the deterioration of maintainability of a system that can be caused by frequent changes. Hence, Objejectory focuses on system changes.

Another important aspect of Objejectory is reuse. The appropriate use of reuse mechanisms can decrease the costs of system development and maintenance, while increasing the quality of the system. Objejectory pays attention not only to the reuse of code at the implementation level, but also to reuse at the level of modules or documents.

\[\text{This definition of robustness differs from the one Meyer gives in [Mey88] in that Meyer emphasizes the ability of a system to function under abnormal conditions. This thesis calls this property execution resilience.}\]
Unfortunately, Objectory lacks a number of desirable features. In [EJW95], Embley et al. present a list of important features that should be present in object-oriented analysis methods. This list is used to extend the basis offered by Objectory to enable the definition of a general framework that encompasses core behavior modeling concepts.

The formalization is defined in terms of expressions in Process Algebra. Process Algebra [BW90] offers a formal framework of sufficient expressive power, is flexible, and supports communication. In previous research, Process Algebra has also been used to describe complex interaction and behavior, e.g. in [Vaa90] where the object-oriented programming language POOL is formalized, and in [Wie90], where it is used to formally define the conceptual modelling language CSML. In [DS95], the dynamic model of the object-oriented specification method MERODE is formally defined using an algebra quite similar to Process Algebra. In contrast to Objectory, MERODE does not incorporate explicit communication mechanisms, in particular asynchronous communication cannot be expressed.

The chapter is organized as follows. Section 4.2 starts with a short overview of Objectory, followed by an examination of the models involved in Objectory’s analysis phase. The formalization of the data model is built on the framework presented in chapter 3. Section 4.3 deals with the integration of the data modeling framework with OOSE’s behavior models. The emphasis of the formalization lies on section 4.4, which formalizes behavior modeling. Section 4.5 presents a formalization of a number of important extensions to the basic communication model, and section 4.6 presents the conclusions.

4.2 Overview of Objectory

This section focuses on the overall structure of Objectory. Objectory consists of three phases: analysis, construction, and testing. The analysis phase uses an informal requirements specification to develop a requirements model, which in its turn is used to develop the analysis model. Both the requirements model and the analysis model are used by the construction phase, which produces a design model and an implementation model. Objectory’s construction phase consists of two sub-phases, design and implementation. These sub-phases correspond roughly to the usual design and realization phases. The testing process uses these models along with the requirements model to develop a test model. Figure 4.1 shows the connections between the various models that are used in Objectory. The analysis models are presented in an informal way in the following sections. The overview concludes with a discussion of strengths and weaknesses of the core models.

4.2.1 Requirements model

The development of the requirements model is based on the requirements specification. The requirements model consists of a use case model, interface descriptions, and a problem domain model. As this thesis focuses on the analysis phase of the software development cycle, the following sections are restricted to the use case and problem domain models.
4.2.1.1 Use case model

The use case model is the most important part of the requirements model. The concepts used in the use case model are actors and use cases.

The actors represent the things and people that interact with the system. An actor can be anything that needs to exchange information with the system. As actors are not part of the actual system, they will not be described in detail in the use case model. A distinction is made between an actor and a user of the system: the user is the actual person who uses the system, whereas an actor represents a certain role that a user can play.

An instance of an actor requests that the system performs certain operations. When the user uses the system, he performs a sequence of transactions in a dialogue with the system. Together, the transactions accomplish a goal that is desired by the user. Such a sequence is called a use case. Each use case is a specific way of using the system. Every execution of a use case can be viewed as an instance of the use case. A use case starts when a user inputs a stimulus. Once started, the use case performs the associated operations.

The use case concept fits smoothly into the object-oriented approach. The description of a use case can be seen as a class description, whereas individual use cases that are created as a response to a request of an actor correspond with instances of these classes. Actors can be viewed from the same perspective.

The set of all use case descriptions defines the complete functionality the system has to offer. The detailed descriptions of the use cases are crucial to the identification of the actual objects in the system. Figure 4.2 contains a description of an example use case.
Use case: Retrieve playlist

The manager identifies himself by typing his name and password in the login window.

The system verifies the password, closes the login window and opens the management window.

The management window displays the list of bands that are managed by this manager.

The manager selects a band.

The management window displays the list of performances of that band.

The manager selects a performance and a window is opened that contains the details of the performance, such as the date, the location and its capacity, the playlist and the total duration of the songs in the playlist.

Figure 4.2: Description of the Retrieve playlist use case

As the focus of this chapter is more on the modelling concepts used in Objectory than on the way these concepts are used, the way (candidate) objects can be found is not discussed. The words typeset in italics in the example use case description indicate the occurrence of candidate objects or object types.

In addition to giving the first pointers that can be used to identify object types, the use case descriptions provide information about the interaction within the system. The key to this interaction can be found in the operations (also called services) that make up a use case. An operation is requested by an object, and carried out by an object. The details involving the interaction of objects are modelled at a later stage of the development process in the interaction diagrams and object behavior models.

4.2.1.2 Problem domain model

To communicate with the potential users of the system and to get a stable basis for the detailed descriptions of the use cases, it is often appropriate to sketch a logical and surveryable problem domain model. This model supports the development of the requirements model by defining the concepts that the system is going to use. The requirements model is developed using the informal requirements specification.

The problem domain model offers a wide range of constructs. Which concepts are actually used in a particular situation depends strongly on the level of detail that is suitable. In its most bare form, only objects and their descriptions are used, although it is possible to identify attributes, static instance associations, inheritance, dynamic instance associations, and operations. A common level of detail includes attributes, inheritance, and static instance associations. The various relationships and associations are described in more detail in section 4.2.2, as part of the description of the analysis model. Figure 4.3 contains an example problem domain model. Instance associations are represented by solid arrows with a name, communication associations are represented by arrows without names, and inheritance hierarchies are drawn with dotted arrows.
4.2.2 Analysis model

After the requirements model has been completed (and approved by the system users or orderers), the development of the actual system starts with the development of the analysis model. The aim of this model is to structure the system, independent of the implementation environment that is to be used. This means that the analysis model focuses on the logical structure of the system. The analysis model forms the foundation for a stable, robust, maintainable, and extensible system.

The analysis model uses the problem domain model as a starting point. The analysis model contains more detail, and distinguishes three types of objects: entity objects, control objects, and interface objects. The availability of three object types is not relevant in the scope of the formal framework presented in this chapter.

Because of the distribution of the functionality of a use case over different objects, the objects will have to cooperate to offer the same functionality as the use cases. The way a number of objects can cooperate is modelled by relationships. There are five relationship types available:

1. inheritance
2. acquaintance association (static instance association)
3. consists-of relation
4. attribute relation
5. communication association

Refer to section 3.6.3 for a description of the formal semantics of the relationship types of Objectory.
4.2.2.1 Inheritance

The *inheritance relationship* (figure 4.4) can be used to describe new objects in terms of existing objects. Only the properties of the new object that differ from the ancestor have to be described. The descendant has all the properties of the ancestor, so properties are inherited “downward”.

![Figure 4.4: A guitar player inherits the properties of a general member of the band](image)

When properly used, the inheritance mechanism can be used to describe objects and the relatedness of their properties in an elegant fashion. This way it is possible to reuse the descriptions of the common properties in multiple classes. In the example in figure 4.4 the classes *Guitar player* and *Singer* are defined in terms of the more general class *Band member*.

In this chapter, a simple form of inheritance is used. If a class $x$ inherits from a class $y$, all attributes and instance associations of $y$ are also attributes and instance associations of $x$. This kind of inheritance is called *monotonic* inheritance [Bap94], while in TM [ABBV90] it is called *pure* inheritance. With monotonic inheritance, it is not possible to selectively inherit properties. Of course, $x$ may add other attributes and instance associations to those that are inherited. The semantics of inheritance associations is described in section 3.6.3.

4.2.2.2 Instance associations

*Instance associations* (acquaintance relationships) are used if an object needs to know of the existence of another object. Figure 4.5 contains an example of an acquaintance association. The number in square parentheses is the cardinality assigned to the association. The cardinality in this example states that a band has exactly one agent. The semantics of cardinality constraints are defined in section 3.3.4.1.

Acquaintance relationships in Objectory are uni-directional: if an object $A$ is acquainted with an object $B$, $B$ is not automatically acquainted with $A$. This allows for more precise specifications. Refer to section 3.6.3 for a description of the semantics of uni-directional associations.
4.2.2.3 Aggregates

A special kind of acquaintance relationship is the consists-of relationship. This kind of relationship is used to express that an object is composed of other objects. Common names for such a structure, where a uniting object has associations with its participating parts, are aggregate or part-of. Like acquaintance associations, part-of relations can have an associated cardinality. Objectory assigns aggregates the same semantics as symmetric associations. Parts know to which whole they belong, and the whole knows its parts. Therefore, the part-of relation is not distinguished as a special construction in the formal framework: part-of’s are treated as symmetric associations.

4.2.2.4 Attributes

The attribute relationship can be seen as a kind of acquaintance association. The difference between attribute relations and acquaintance associations is that the target of the former is a concrete domain, whereas the target of the latter is an object. Attributes can be used in all object types to describe the information the object needs to store.

![Diagram showing instance association and attribute relation](image)

Figure 4.5: Examples of instance association and attribute relation

Each attribute has a type, which can be a primitive data type such as integer (figure 4.5) or string, or a more complex composite data type. This chapter does not explicitly distinguish concrete domains. Instead, populations consist solely of objects. Jacobson’s concrete types are encapsulated in value objects, as described in section 3.2. Summarizing: from a technical point of view attributes receive the same treatment as instance associations, but in this chapter the term “attribute” will be used to adhere more closely to Jacobson’s terminology.

4.2.2.5 Communication relations

The last relationship type is the communication association. A communication association between an object A and an object B means that object A can send stimuli to object B. No restrictions are imposed on which kind of object can communicate with which other kind. To avoid crowding the diagrams, no labels are assigned to the arrows representing communication associations. Communication associations have no impact on the data model, so they are not incorporated into the formal framework.

4.2.3 Interaction diagrams

Interaction diagrams (also known as communication diagrams) are used to model the way in which objects cooperate. They show which objects request what services from which
other objects, and in what order. In effect, they describe how the functionality of use cases is realized through cooperating objects. Each object is represented by a bar. The rectangles on the bars indicate services. Service requests are drawn using an arrow from the requesting object to the object that provides the service.

Interaction diagrams are often used to aid in the transition from the informal use cases to the object behavior models. They offer a higher-level view on the workings of objects than object behavior models, which cover the internal workings of a single object. Due to their higher modeling precision, object behavior models could be used to derive interaction diagrams. An object behavior model describes all the possible traces that an object might generate, whereas an object interaction diagram describes only a single, partial trace. Consider the example object interaction diagram in figure 2.1 on page 15. It shows that a scenario exists in which an Aircraft object first receives a request to perform a handOver service, after which it requests a remove and an add service. When both these services have been completed, it sends status requests to two Display objects.

![Figure 4.6: Integration of object behavior models and interaction models](image)

Figure 4.6 show the relation between the (simplified) interaction diagram and the skeleton of the Aircraft object behavior model. Note that the sequence of actions on the bar for the aircraft in the interaction diagram reappear in the object behavior model. This is an example of good support for model integration. A legend to the notation that is used in the object behavior model can be found in figure 4.7.
4.2.4 Object behavior

The distribution of use cases over objects assigns responsibilities to the objects with regard to functionality. As stated before, the objects must collaborate to offer the functionality that is described in the use cases. To model this collaboration, an interaction diagram is developed for each concrete use case. This activity is called “designing the use cases”. The interaction diagrams are used as a basis to model the behavior of individual objects. Jacobson places both the interaction diagrams and the object behavior models in the design phase. To be able to fully describe a system, this chapter includes object behavior models in the framework.

To obtain a complete description of the system, it is necessary to increase the level of detail of the object descriptions beyond the list of operations that can be extracted from the interaction diagrams. Especially in the case of complex objects, a diagram describing the behavior of individual objects helps to increase the understanding of the system. For this purpose object behavior models can be used. An object behavior model (OBM) is a state transition graph that describes which service requests can be received by an object, and what happens when a particular request is received in a given state [DS95].

States and state transitions can be described in many ways. Jacobson uses a notation that supports the use of the interaction diagrams as a basis for the object behavior diagrams. In this thesis an alternative notation that communicates the meaning of the constructs more clearly is used (figure 4.7). Informally, the relationship between use cases, interaction diagrams, and object behavior diagrams is as follows: a use case describes a single execution path through the system. This path is divided into subpaths, and each subpath is modelled in a single interaction diagram. All interaction diagrams together define the interface of an object. The workings behind this interface (the behavior of a single object) are modelled in an object behavior model.

A number of constructs is available to describe the behavior of an object. The constructs are connected to each other with arrows. If a construct has more than one successor, a choice is made between the possible continuations of the execution path. The life of an object starts with its initial state. Communication is handled through the send, receive, and return from message constructs. Objectory does not define the semantical differences between messages and signals. This chapter uses messages for synchronous communication, and reserves signals for asynchronous communication. The execution of primitive tasks is specified by means of the perform task construct, which can modify the state of the object. The execution flow may branch out by means of the decision construct. The effect of a previous state construct is that execution continues at the state construct that was encountered most recently. New objects are created with a create object construct, and removed from the system upon encountering a destroy object construct. The synchronizer and collector are described in section 4.5.2.

Example 4.2.1

Figure 4.8 contains an example of an object behavior model. This example is based on the pattern Observer from [GHJV95].
A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects, without creating a tight coupling. For example, it is good practice to separate presentational aspects from the underlying application data. If both a spreadsheet and a bar chart are used to display this year’s profits, it is desirable that they behave as though they know of each other’s existence, even if they are not aware of each other. A way to achieve this is to use **Subjects** and **Observers**. The Observers (spreadsheet and bar chart) can request modifications of the Subject (profits), and are notified with an **Update** message when the internal state of the Subject changes. Then, the Observer requests the new Subject state through a **GetState** request, and updates its own state accordingly. For simplicity’s sake, the behavior of Subjects and Observers has been restricted. For example, figure 4.8 does not allow an Observer to detach itself from its Subject.

### 4.2.5 Analysis of core models

Like many other object-oriented methods, Objectory does not always provide precise definitions of the concepts it uses. For example, it is not clear if a previous state construct can always be replaced by an arrow (see figure 4.9), implying that it in fact adds no new functionality. In this thesis, the previous state construct is interpreted *dynamically* to offer
more than a notational abbreviation (see figure 4.14 and example 4.4.3). Therefore, it is necessary to develop a formal framework that describes the exact syntax and semantics of concepts.

In [EJW95] a number of features is described that should be supported by an object-oriented analysis method. Objectory includes most of these features, but a few important ones are lacking. On the dynamic side, the basic framework provided by Objectory was extended with constructs that offer support for the specification of

1. nondeterministic behavior

2. intraobject concurrency
Nondeterministic behavior is added by means of the decision construct. The original decision construct can be compared to an if-then-else construct. The extension provides a choice between an arbitrary number of alternatives, and the conditions associated with the alternatives need not be disjoint.

Concurrency within an object is not supported by Objectory. Especially in the case of complex objects, this can be a severe handicap.

Interobject concurrency is only partially supported by Objectory: it is possible to specify asynchronous communication, but there is no construct that enables the retrieval of an asynchronously computed value.

A similar asymmetric peculiarity can be found in the handling of object creation: although an explicit object destruction construct exists, it has no counterpart in an object creation construct. As a result, the consequences of creating a new object and its initialization are far from clear. Therefore, this chapter introduces an object creation construct.

The extensions mentioned in this section are formalized in sections 4.4 and 4.5.

4.3 Integration with data model

The data model defines the state of the system. It holds the populations of the classes, and stores the values of attributes and instance relations. The behavior side of the framework operates as a client of the data model. The constructs of the object behavior models retrieve values that are stored in the system state, and update them.

The set $\mathcal{E}$ contains all attributes and instance associations. Jacobson’s properties are mapped to symmetric associations. Furthermore, $\mathcal{O}$ is used as an abbreviation for $\mathcal{G}_0$, and $S$ is used as an abbreviation for Pop. The function $\text{Dom} : \mathcal{E} \rightarrow \mathcal{O}$ yields the source class of instance associations and attributes, whereas the function $\text{Ran} : \mathcal{E} \rightarrow \mathcal{O}$ is used to indicate the target class of a property. The name of a property is indicated by the injective function $\text{name} : \mathcal{E} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is a finite set of names. $\mathcal{N}$ can be determined statically by gathering the names of attributes and instance associations from the analysis model.
The retrieval of the value of a property $p$ from an object $\alpha$ is denoted as $S(\alpha, p)$ with $\alpha \in S(Dom(p))$ and $p \in \mathcal{L}$, where $S(\alpha, p) = S(Ran(p)) \circ \text{inv}(S(Dom(p)))[\alpha]$. Because the combination of an object identifier and a property name suffices to identify a particular property, this notation extends naturally to $S(\alpha, n)$ with $n \in \mathcal{N}$.

The notation that is used for updates uses a similar approach. To express that a property $p$ of an object $\alpha$ is changed to a value $v$ (either a single element or a set) in a state $S$, the notation $S \oplus \{ (\alpha, p) : v \}$ is used (see appendix A). Of course, the values must be elements of the correct domain $Ran(p)$. Often a number of attribute values must be copied from one object to attributes of another object. The mapping of attributes may be implied by their order in a tuple. For example, suppose that the values of the attributes, present in a tuple $P$ of object $\alpha$ have to be copied to the attributes in a tuple $Q$ of object $\beta$. Formally, this corresponds to: $s \oplus \{ (\beta, Q_i) : s(\alpha, P_i) | 1 \leq i \leq n \}$ which is abbreviated to: $s \oplus \{ (\alpha, P) \mapsto (\beta, Q) \}$. The notation $P_i$ is used to select the $i$-th element of the tuple $P$. Of course, the lengths of both tuples should be the same: $|P| = |Q|$. It is also possible to indicate an explicit mapping. The corresponding notation is: $s \oplus \{ \alpha \mapsto Q \beta \}$, where $Q$ is an attribute mapping.

Naturally, the creation and destruction of objects has consequences for the system state. If a new object is created, it is added to the active objects of its class. Likewise, if an object is destroyed, it is removed from the system state. These effects are handled by the semantic functions $\mathbb{I}$ (the initialization function) and $\mathbb{D}$ (the destruction function). They are described in more detail in section 4.4.3.4.

### 4.4 Behavior model

The object behavior models provide the basis for the communication model. The formal semantics of the communication model is defined by means of a translation of the object behavior models to Process Algebra [BW90]. Process Algebra offers a formal framework of sufficient expressive power, is flexible, and supports communication. This section presents a description of the syntax of object behavior models, a short overview of the relevant parts of Process Algebra to make this chapter self-contained, and the translation of object behavior models to Process Algebra expressions.

#### 4.4.1 Syntax of object behavior models

The ways an object of a certain class can behave is specified by the object behavior model of that class. A number of object behavior primitives is used to construct an object behavior model. These constructs offer the means to create or destroy objects, communicate with other objects, make choices, and perform elementary tasks.

An object behavior model is named after the class it models, and consists of the finite set $\mathcal{X}$ of behavior objects which is a union of the set of states $\mathcal{L}$, the set of tasks $\mathcal{T}$, the set of communication primitives $\mathcal{C}$, the set of decisions $\mathcal{D}$, the set of previous state constructs $\mathcal{P}$, the set of object creators $\mathcal{G}$ and the set of object terminations $\mathcal{F}$. 

65
To support synchronous as well as asynchronous communication, a number of different communication primitives is needed. Objects can receive messages or signals by means of the “receive message” and “receive signal” primitives respectively, which are contained in the sets $M \times$ and $S \times$. To send messages or signals, the primitives “send message” and “send signal” are used. These are gathered in the sets $M \circ$ and $S \circ$. To complete the list of communication primitives, the set $R$ of “return from message” primitives is introduced. $C$ is the union of the sets $M \times$, $S \times$, $M \circ$, $S \circ$, and $R$. Messages are used for synchronous communication, while signals are used to model asynchronous communication.

A service (either a message or a signal) can be requested from any object (the service provider) that matches the communication criteria specified in the function $\text{CommExpr} : C \setminus R \rightarrow CE$. $\text{CommExpr}$ associates an expression in the domain $CE$ with every service request. It associates another expression with every service. Upon the execution of a service request, both the expression associated with the requesting object and the expression associated with the service itself are evaluated. For communication to take place, the result of both evaluations must be equal. If more objects can handle a service request, one is chosen at random.

The communication expressions can be used to request a service from a particular object in the following way. The communication expression of the sender consists of the name of an attribute that contains the OID of the object that is to service the request. The communication expression of the receiver consists of the “self” attribute, which yields the receiver’s OID.

The name of the service request is yielded by the function $\text{Request} : C \setminus R \rightarrow \mathcal{N}$. The outgoing parameters that accompany the request are yielded by the function $\text{OutPar}$. The $\text{OutPar}$ function also yields the attributes of the servicing object that contain the return values: $\text{OutPar} : (M \circ \cup S \circ \rightarrow \mathcal{E}^{+}) \cup (R \rightarrow \mathcal{E}^{+})$. The attribute(s) that receive any values that are returned by the servicing object are indicated by the function $\text{InPar}$. The same function is used to indicate which attributes of the servicing object receive the outgoing parameters of a request: $\text{InPar} : (M \times \cup S \times \rightarrow \mathcal{E}^{+}) \cup (M \circ \rightarrow \mathcal{E}^{+})$. Figure 4.10 shows how outgoing parameters are mapped to incoming parameters. As can be seen in this figure, it is not possible in Objectory to return values to the requesting object after the completion of an asynchronous service request. Section 4.5.2 presents the constructs to support this mechanism.

The connections between the various primitives in an object behavior model are captured in the relation $\text{Follows} \subseteq \mathcal{X} \times \mathcal{X}$. The starting point is given by the initial state $I \in \mathcal{L}$. When convenient, infix notation will be used for the $\text{Follows}$ relation. The connections between a decision construct and its followers have a decision condition (an element of $\mathcal{H}$) associated with them: $\text{DecCond} : (\mathcal{X} \times D \cap \text{Follows}) \rightarrow \mathcal{H}$. In Objectory, there is one condition associated with a decision construct, and therefore two continuations are possible: one is taken if the condition is true, and one if it is false. The approach that is presented here offers more flexibility. There is no limit to the number of continuations after a decision, and the decision conditions need not be disjoint: if the conditions associated with multiple
continuations are true, a nondeterministic choice is made. This solution can be compared with Dijkstra’s guarded commands [Dij75].

A new object is created by the object creation construct. The class of the new object is given by the function $\text{OfClass} : G \rightarrow O$. For initialization purposes, the creating object can pass a number of initialization parameters to the newly created object. These parameters are yielded by the function $\text{OutPar}$, which is extended to range over $\text{OutPar} : (G \rightarrow E^+) \cup (M \cup S \rightarrow E^+) \cup (R \rightarrow E^+)$. Upon initialization of the newly created object, the initialization parameters are copied to the attributes indicated by the function $\text{InPar}$, applied to the initial state $I$. The OID of the new object is returned to the creator, and placed in the attribute given by the function $\text{NewID} : G \rightarrow E$.

Jacobson uses two special constructs to simplify the notation of the diagrams: labels and a symbol that signals a jump to the previous state. The “jumps” that result from the use of labels are expanded before they are entered in the Follows relation. In the example in figure 4.11, $A$ Follows $B$ in both diagrams. The jumps that are the result of the previous state construct are handled in a dynamic fashion. For each object information is kept about the state it was most recently in. If a previous state construct is encountered, execution resumes in the state that was most recently encountered.
4.4.2 Process Algebra

Although the name Process Algebra suggests a single algebra to describe processes, it actually refers to a whole family of algebras based on the same principles. Traditionally, only the family member used is presented.

The units of Process Algebra are atomic actions. The set of all atomic actions is called \( \mathcal{A} \).

Although they are units of calculation, atomic actions need not be indivisible (see [GW96]).

Starting with atomic actions, new processes can be constructed by applying sequential and alternative composition (\( \cdot \) resp. \( + \)). Table 4.1 summarises the axioms defining these operators. The algebra that results is called basic process algebra (BPA). As a convention, the names of atomic actions are written in lowercase (e.g. \( a, b, \text{rudolph} \)), while process variables are written in uppercase (e.g. \( A, B, \text{RED_NOSE_REINDEER} \)). Normally, the \( \cdot \) will be omitted unless this results in ambiguity.

\[
\begin{align*}
X + Y &= Y + X \quad (A1) \\
(X + Y) + Z &= X + (Y + Z) \quad (A2) \\
X + X &= X \quad (A3) \\
(X + Y) \cdot Z &= X \cdot Z + Y \cdot Z \quad (A4) \\
(X \cdot Y) \cdot Z &= X \cdot (Y \cdot Z) \quad (A5)
\end{align*}
\]

Table 4.1: BPA

A special constant \( \delta \), deadlock, denotes the inaction, or impossibility to proceed. As a rule, such a situation is to be avoided. The axioms in table 4.2 are therefore obvious.

\[
\begin{align*}
X + \delta &= X \quad (A6) \\
\delta \cdot X &= \delta \quad (A7)
\end{align*}
\]

Table 4.2: \( \delta \) axioms

Within BPA process expressions describing sequential processes with choices can be formulated. To add parallelism, an additional operator has to be introduced. This operator, called (free) merge and denoted as \( \parallel \), is defined with the aid of an auxiliary operator \( \ll \), the left-merge (see table 4.3).

Another special constant \( \varepsilon \), the empty action, is used to denote the process that does nothing but terminate successfully. Its axioms are stated in table 4.4.

After adding \( \varepsilon \), processes may terminate directly. The termination operator \( \sqrt{\cdot} \) determines whether or not this termination option is present for a given process. Its axioms are listed in table 4.5.
\[ X \parallel Y = X \parallel Y + Y \parallel X \]  \hspace{1cm} (M1)
\[ a \parallel X = aX \] \hspace{1cm} (M2)
\[ aX \parallel Y = a(X \parallel Y) \] \hspace{1cm} (M3)
\[ (X + Y) \parallel Z = X \parallel Z + Y \parallel Z \] \hspace{1cm} (M4)

Table 4.3: Merge

\[ X \cdot \varepsilon = X \] \hspace{1cm} (A8)
\[ \varepsilon \cdot X = X \] \hspace{1cm} (A9)

Table 4.4: \( \varepsilon \) axioms

\[ \sqrt{\varepsilon} = \varepsilon \] \hspace{1cm} (TE1)
\[ \sqrt{a} = \delta \] \hspace{1cm} (TE2)
\[ \sqrt{(X + Y)} = \sqrt{(X)} + \sqrt{(Y)} \] \hspace{1cm} (TE3)
\[ \sqrt{(X \cdot Y)} = \sqrt{(X)} \cdot \sqrt{(Y)} \] \hspace{1cm} (TE4)

Table 4.5: Termination operator

\[ X \parallel Y = X \parallel Y + Y \parallel X + \sqrt{(X)} \cdot \sqrt{(Y)} \] \hspace{1cm} (TM1)
\[ \varepsilon \parallel X = \delta \] \hspace{1cm} (TM2)
\[ aX \parallel Y = a(X \parallel Y) \] \hspace{1cm} (TM3)
\[ (X + Y) \parallel Z = X \parallel Z + Y \parallel Z \] \hspace{1cm} (TM4)

Table 4.6: Merge with \( \varepsilon \)

To include the empty process in parallel composition, the definition of the merge needs to be modified (see table 4.6). Specifically, parallel processes are now able to choose termination at any moment. Note that from axioms M2, A9 and TM2 it follows that \( \varepsilon \notin \mathcal{AA} \).

Axioms A1-9, TE1-4 and TM1-4 define Process Algebra with the empty action (PAe).

To allow parallel processes to exchange information (i.e. communicate) the definition of parallel composition, TM1-4, has to be modified. The extended version (see table 4.7) introduces the communication merge \( | \).

The modified definition has to be completed with the definition of the communication function \( \gamma \), defined over pairs of atomic actions. The axioms covering this definition are listed in table 4.8. Specific process specifications will have to define the (partial) communication function \( \gamma \). This function is both commutative and associative.

Finally, one needs a way to prevent the isolated occurrence of atomic actions meant to
\[
X \parallel Y = X \parallel Y + X + X \mid Y + \sqrt{(X)} \cdot \sqrt{(Y)} \tag{CM1}
\]
\[
\varepsilon \parallel X = \delta \tag{CM2}
\]
\[
aX \parallel Y = a(X \parallel Y) \tag{CM3}
\]
\[
(X + Y) \parallel Z = X \parallel Z + Y \parallel Z \tag{CM4}
\]
\[
\varepsilon \mid X = \delta \tag{CM5}
\]
\[
X \mid \varepsilon = \delta \tag{CM6}
\]
\[
aX \mid bY = (a \mid b)(X \parallel Y) \tag{CM7}
\]
\[
(X + Y) \mid Z = X \mid Z + Y \mid Z \tag{CM8}
\]
\[
X \mid (Y + Z) = X \mid Y + X \mid Z \tag{CM9}
\]

Table 4.7: Merge for communicating processes

\[
a \mid b = \gamma(a, b) \text{ if } \gamma(a, b) \downarrow \tag{CF1}
\]
\[
a \mid b = \delta \text{ otherwise} \tag{CF2}
\]

Table 4.8: The communication function

communicate with other actions. This is achieved through the encapsulation operator \( \varnothing_H \). In fact it is a whole family of operators, one for each \( H \subseteq AA \). The axioms of \( \varnothing_H \) are listed in table 4.9. Note that by this definition the termination operator \( \sqrt{\cdot} \) is equal to \( \varnothing_{AA} \). The system BPA + A6-9 + TE1-4 + CM1-9 + CF1-2 + D1-4 is called ACP\(_{e} \) (Algebra of Communicating Processes). It is within ACP\(_{e} \) that a translation of object behavior models is defined.

The state operator \( \lambda \) in Process Algebra is used to describe processes with an independent global state. Informally, the expression \( \lambda^m_s(X) \) represents the execution of process \( X \) on machine \( m \) in state \( s \). The action function action calculates which action has to be performed as a result of executing \( X \) in state \( s \) on machine \( m \), while the effect function effect calculates the new state. Table 4.10 shows the relevant axioms of the state operator.
4.4.3 Translation of object behavior models to Process Algebra

An object behavior model provides a means to describe the life cycle of objects of a certain class. The behavior of each object that belongs to the class is consistent with the description in the object behavior model. To describe the semantics of this dynamic side of the system, all object behavior models are translated to Process Algebra expressions. Each object is represented by the Process Algebra translation of the object behavior model of its class. The complete system is represented by the expression $E_S$, which states that all these $ACP_{e}$ expressions are executed in parallel:

$$E_S = \lambda_{s}^{\mathcal{C}} \circ \partial_{H} \left( \bigg\| \mu_{a}^{I} \circ \rho^{a} \circ T_{\text{Class}(a)} \right)$$

To eliminate most of the parentheses, $\circ$ is used to denote function composition. The state operator $\lambda_{s}$ uses an environment $e$ and a state $s$. The environment $e$ is a partial function from $\mathcal{OID}$ to $\mathcal{OID}$. If $e(\beta) = a$, this implies that object $\beta$ is servicing a request from object $a$. The state $s$ contains the values of attributes of individual objects, and parameters that are used in service requests. Note that this system state should not be confused with the state of an individual object. Also note that there is difference between $\lambda$ and a “classical” state operator as described in table 4.10, where a process is run on a particular machine. The $\lambda$ used in this chapter uses a split state space ($e$ and $s$), while the machine is irrelevant.

The set $H$ used by the encapsulation operator $\partial$ contains, among others, the communication operations ($\text{Send, Receive, Wait, Create, WaitCreate}$) with their parameters, ensuring that these atomic actions can only occur in conjunction with other communication actions.

The rewrite operator $\rho^{a}$ is inspired by [Vaa90], where it was used for the formal definition of the Parallel Object Oriented Language (POOL). This rewrite operator instantiates PA expressions with an $\alpha \in \mathcal{OID}$ that contains the object identifier of the executing object, and provides the formal equivalent of a self attribute: all objects have knowledge of their identity. This way, the expressions that are generated based on a model at the class (type) level are customized so that they can be used on the object (instance) level. It is assumed that $\rho^{a}$ operates solely on PA actions if explicitly defined. Otherwise, it passes over them without changing anything.

Like the operator $\rho^{a}$, the rewrite operator $\mu_{a}^{ps}$ is a local operator. It operates on a single object $\alpha$ and is used to translate the previous state construct. For this purpose, the most recently encountered state $ps$ of each object $\alpha$ is remembered. Initially, the previous state
is the initial state $I$ of the corresponding object class. The $\mu$ operator passes over all PA actions, except the $st$ and $ps$ actions.

The function $T$ yields the translation of the object behavior model of the class of the object with object ID $a$. For every behavior object $x \in X$ a process variable $E_x$ is defined that represents the entry point of $x$. All objects are created beforehand. To avoid the uncontrolled execution of all the expressions representing these objects, the $Tc$ function ensures that execution is suspended until an explicit creation command is received.

The translation $Tc$ of a certain class $c$ consists of waiting for a create action that delivers the initialization parameters yielded by $InPar$, followed by the translation of the initial state of the OBM of the class, indicated by the function $Init$:

$$Tc = wc[InPar(Init(c))] \cdot E_{Init(c)}$$

To avoid a bootstrapping problem, a single object called $System$ is created that is allowed to start its execution without having to wait for an explicit $Create$ action. This object is responsible for the creation of additional objects. The $Tc$ function translates it the same way as a normal object, but without the $wc$.

The following sections present the translations of the various concepts that may occur in object behavior models.

### 4.4.3.1 States

The main purpose of the translation of a state symbol is an administrative one. The execution of the process algebra expression ensures that the system state reflects that the object represented by the expression is in a certain state.

The translation of either an initial state or an ordinary state $t$ offers the information needed to implement the previous state construct. The $st[t]$ action indicates that the object has entered state $t$.

$$t \in \mathcal{L} \Rightarrow E_t = st[t] \cdot Tail(t)$$

where $Tail(t)$ is the expression that formalizes the possible continuations of the execution path of the object:

$$t \in \mathcal{X} \Rightarrow Tail(t) = \sum_{x \text{Follows } t} E_x$$

In this expression, it is implicitly assumed that $x \in \mathcal{X}$, a convention that is adopted throughout the rest of this chapter.

This definition of the $Tail$ reflects the intuition that a single path is chosen when multiple continuations are available. It is possible that a behavior object is not followed by any other behavior object. In that case, the sum is specified over the empty set. This exception is dealt with by defining the sum over the empty set to be the empty action $\varepsilon$, as this is the
neutral element for sequential composition. Therefore, if no behavior object \( x \) exists such that \( x \text{ Follows} t \), the expression for \( \text{Tail}(t) \) reduces to \( \varepsilon \).

As the objects (or more precisely, their translations) are all started at the beginning, it must be ensured that the execution of an object is suspended until it is explicitly created by the system itself, using the \textit{create object} construct. For this purpose, the suspended object must know to which class it belongs, and its own identity:

\[
\rho^o(wc[P] \cdot X) = \text{WaitCreate}(\text{Class}(\alpha), \alpha, P) \cdot \rho^o(X)
\]

It is the task of the rewrite operator \( \mu \) to update the previous state \( ps \) of object \( \alpha \) to point to the newly encountered state \( t \):

\[
\mu^t_{ps}(st[t] \cdot X) = \mu^t_{\alpha}(X)
\]

If an object encounters a \textit{previous state} symbol in an object behavior model, it continues its execution with the translation of the most recently encountered state. In the formal framework, the \( \mu^t_{ps} \) handles the bulk of this task. Therefore, the translation of a previous state construct \( t \) is uncomplicated:

\[
t \in \mathcal{P} \Rightarrow E_t = ps
\]

Any construct following a previous state construct is ignored, as it will not be executed in any case, due to the effect of the operator \( \mu^t_{ps} \). Note that because there is no \textit{Tail} for the operator \( \rho \) to process, it vanishes upon encountering a \( ps \):

\[
\rho^o(ps) = ps
\]

It is the responsibility of the \( \mu^t_{ps} \) to insert not only the translation of the previous state, but also the \( \rho^o \) to customize the inserted expression to the object executing it.

\[
\mu^t_{\alpha}(ps) = \mu^t_{\alpha} \circ \rho^o(E_{ps})
\]

This dynamic handling of the previous state is a widening of the scope of the original definition, where the previous state can be determined statically. An example of the dynamic use of the previous state construct can be found in example 4.4.3.

### 4.4.3.2 Communication

The extensions of Objectory support two different kinds of communication: either synchronous (messages) or asynchronous (signals). In the case of synchronous communication, the requesting object suspends execution until the servicing object has finished servicing the request. If asynchronous communication is used, the requesting object continues its execution in parallel with the servicing of the request. In case the servicing object is busy and cannot service a request, the execution of the requesting object is suspended until the servicing object is ready to communicate.
Usually, if an object requests a service, the requesting object asks a specific object to handle the request. The formal framework has been extended to handle communication schemes that allow for more latitude in addressing objects through the use of communication expressions. These allow the requesting object to only requests that a service be performed, without specifying which particular object should handle the request.

Many refinements to asynchronous communication exist, for example one may consider unbuffered asynchronous communication: any event that is not accepted in a timely manner is lost [CLF93]. Such further refinements are not considered as they make assumptions about the underlying communication media. The consequences of these assumptions are not relevant to systems analysis. The remainder of this subsection presents the translations of the various concepts used to model communication.

### 4.4.3.2.1 Synchronous communication

A request for synchronous communication is modeled by the *send message* construct.

The servicing of a request is mapped to two pairs of communicating process algebra atoms. The initiation of a service request consists of a *sn* and a *rec* pair. The *sn* appears in the translation of the *send* construct of the requesting object, while the *rec* surfaces in the translation of a corresponding *receive* construct. The conclusion of a service request consists of a *rfm* atom in the translation of a *return from message* construct of the servicing object, and a *wt* atom that completes the translation of the *send*. Informally, the requesting object initiates a service request with a *sn*. The servicing object starts handling the request (*rec*), while the requesting object waits (*wt*) until the request is completed (*rfm*).

A *send message* primitive $t$ is translated to a *sn* operation, which requests the execution of service $\text{Request}(t)$, followed by a *wt* to suspend the execution until the requested operation has been completed. After reception of confirmation of the completion of the service, execution continues with the constructs following the send message:

$$
t \in \mathcal{MO} \quad \Rightarrow \quad E_t = \text{sn}[\text{CommExpr}(t); \text{Request}(t), \text{OutPar}(t)] \cdot \text{wt}[\text{InPar}(t)] \cdot \text{Tail}(t)
$$

where $\text{OutPar}(t)$ are the actual parameters that accompany the service, $\text{InPar}(t)$ are the attributes that receive any values that are returned by the servicing object, and $\text{CommExpr}(t)$ indicates the communication expression. The passing of the values of the parameters is handled by the state operator $\lambda_x$ at a later stage.

The $\rho^a$ operator adds the identity of the requesting object to the communication parameters:

$$
\rho^a(\text{sn}[c; r, O] \cdot X) = \text{Send}(c; r, a, O) \cdot \rho^a(X)
$$

The *Wait* operation provides the means to handle values that are returned by the servicing object to the calling object:

$$
\rho^a(\text{wt}[R] \cdot X) = \text{Wait}(a; R) \cdot \rho^a(X)
$$
The counterpart of the send message construct is the receive message primitive. The translation of such a construct $t \in \mathcal{MI}$ is straightforward:

$$t \in \mathcal{MI} \Rightarrow E_t = \text{rec}[\text{CommExpr}(t); \text{Request}(t), \text{InPar}(t)] \cdot \text{Tail}(t)$$

As usual, the object identity $\beta$ is added:

$$\rho^\beta(\text{rec}[d; r, Q] \cdot X) = \text{Receive}(d; r, \beta, Q) \cdot \rho^\beta(X)$$

The actual parameters of the requesting object are placed in the formal parameter attributes indicated by $\text{InPar}(t)$. The rec operator does not know which object requested execution of the operation. This knowledge is gathered at the point where a Send and a Receive communicate:

$$\text{Send}(c; r, \alpha, P) \mid \text{Receive}(d; r, \beta, Q) = \text{ServiceReq}(c; d; \beta, \alpha, P, Q)$$

where $\alpha$ is the OID of the requesting object, $c$ characterizes the intended provider of the service by means of a communication expression, $d$ is the communication expression that characterizes the receiving object, and $\beta$ is the OID of the object that services the request ($\beta$ services the request of $\alpha$). $P$ and $Q$ are the actual and formal parameters involved in servicing request $r$.

Note that as soon as the actual servicing of the request begins ($\text{ServiceReq}(c; d; \beta, \alpha, P, Q)$), the name of the request is no longer needed. In an object behavior model this name is only used to identify the entry point of the request.

Because the state operator $\lambda^x$ cannot be used until after communication between Send and Receive has taken place, the communication function $\gamma$ allows communication of all Send-Receive pairs that have the same name of the request ($r$). Selection based upon the values of the two communication expressions $c$ and $d$ is handled at the time the state operator evaluates the result of the communication of a Send-Receive pair. Erroneous communication is then filtered out.

When servicing a request, the state operator $\lambda^x$ performs a number of tasks. First of all, it checks if this Send-Receive pair is allowed to communicate by comparing the value that results from the evaluation of the communication expression $c$ with the value that results from the evaluation of the communication expression $d$. The semantic function $\text{IM}(\mathcal{CE} \times \mathcal{OTD} \times S) \rightarrow \mathcal{VACL}$ is used to evaluate communication expressions by yielding a result in $\mathcal{VACL}$, a set of concrete values. If both communication expressions yield the same result, the request is serviced:

$$\lambda^x(\text{ServiceReq}(c; d; \beta, \alpha, P, Q) \cdot X) = \begin{cases} \lambda^x_{\beta, \alpha}(X) & \text{if } \text{IM}(c, \alpha, S) = \text{IM}(d, \beta, S) \\ \delta & \text{otherwise} \end{cases}$$

Furthermore, the state operator updates the environment $(e \oplus \{\beta : \alpha\})$ in order to remember that $\beta$ is servicing a request from $\alpha$, and it adds the values of the parameters of the operation to the state. The passing of the parameters is done by copying the values of the actual
parameters in $P$ to the corresponding formal parameters in $Q$. At this point, ServiceReq has served its purpose of providing $\lambda^e_s$ with the necessary information and disappears.

If the evaluation of the sender's communication expression $c$ does not match the result of the evaluation of the receiver's communication $d$, this branch of the process algebra evaluation is halted by means of a $\delta$.

The completion of an operation is indicated by a return from message construct. Such a construct $t$ is translated as follows:

$$t \in R \Rightarrow E_t = rfm[OutPar(t)] \cdot \text{Tail}(t)$$

Intuitively, this indicates that object $\alpha$ has completed a request, and is ready to confirm that to the requesting object. On top of this, the servicing object provides the requested return values $OutPar(t)$.

$$\rho^\alpha(rfm[V] \cdot X) = \text{Return}(\alpha, V) \cdot \rho^\alpha(X)$$

The return values $V$ are accepted by the requesting object when a Wait and a Return action communicate. As in the case of communication between Send and Receive actions, a loose communication strategy must be used because the state operator $\lambda$ is only available after communication has taken place. Communication between Wait and Return actions is only restricted by the length of the tuples of parameters. Both tuples must have the same number of elements: $|R| = |V|$.

$$\text{Wait}(\alpha; R) | \text{Return}(\beta, V) = \text{AcceptValues}(\alpha, \beta, R, V)$$

It is the responsibility of the state operator to decide if the matching of two particular Wait and Return actions is a valid one. If not, a deadlock results. If on the other hand the result of the communication (AcceptValues) is valid, i.e. object $\beta$ was servicing a request from object $\alpha$ ($e(\beta) = \alpha$), the environment is updated to reflect the current state of the system by “forgetting” that the object with OID $\beta$ is servicing a synchronous request ($e \circ \{\beta\}$). Furthermore, the actual return values in $V$ are retrieved from the state $s$ and passed on to attributes $R$ of $\alpha$:

$$\lambda^e_s(\text{AcceptValues}(\alpha, \beta, R, V) \cdot X) = \lambda^e_{s \circ \{(\beta, V) \rightarrow (\alpha, R)\}}(X)$$

**Example 4.4.1**

Figure 4.12 contains an example of synchronous communication. The sender object, a read-out for a digital thermometer, asks the receiver, an indoor/outdoor temperature sensor, what the current temperature is by requesting service temp. Along with the request, the parameter indoorOutdoor is passed. The sensor interprets the parameter to mean that the temperature from the outdoor sensor is to be returned. Upon completion of the service, the temperature is read from attribute degreesC, and copied to attribute $t$ from the sender.
The following set of Process Algebra expression is the result of applying the previous translation rules to the object behavior models in figure 4.12:

\[ E_{t_1} = \text{sn}[\text{sensor}; \text{temp}, \langle \text{indoorOutdoor} \rangle] \cdot \text{wt}[\langle t \rangle] \cdot T \]

\[ E_{u_1} = \text{rec}[\text{self}; \text{temp}, \langle \text{insideOutside} \rangle] \cdot E_{u_2} \]

\[ E_{u_2} = \text{rfm}[\langle \text{degreesC} \rangle] \cdot U \]

T and U represent the respective tails of \( t_1 \) and \( u_2 \). Two objects exist, \( \alpha \) and \( \beta \). \( \text{Class}(\alpha) = \text{Sender}, \text{Class}(\beta) = \text{Receiver} \). This system \( S \) is represented by the following expression \( E_S \):

\[ E_S = \lambda^\omega \circ \partial_H \left( \mu^I_\alpha \circ \rho^\alpha(E_{t_1}) \parallel \mu^I_\beta \circ \rho^\beta(E_{u_1}) \right) \]

The following rewritings show the workings of the state operator \( \lambda \) and the rewrite operator \( \rho \). The \( \mu \) operator is omitted from the expressions, as it has no impact on the current example, which contains no previous state constructs. State \( s \) is the system state at the beginning of the example.

\[ E_S = \lambda^\omega \circ \partial_H \left( \text{Send}(\text{sensor}; \text{temp}, \alpha, \langle \text{indoorOutdoor} \rangle) \cdot \text{Wait}(\alpha; \langle t \rangle) \cdot \rho^\alpha(T) \right. \]

\[ \quad \parallel \text{Receive}(\text{self}; \text{temp}, \beta, \langle \text{insideOutside} \rangle) \]

\[ \quad \cdot \text{Return}(\beta, \langle \text{degreesC} \rangle) \cdot \rho^\beta(U) \big) \]

\[ = \lambda^\omega \circ \partial_H \left( \text{ServiceReq}(\text{sensor}; \text{self}; \beta, \alpha, \langle \text{indoorOutdoor} \rangle, \langle \text{insideOutside} \rangle) \right. \]

\[ \quad \cdot \text{Wait}(\alpha; \langle t \rangle) \cdot \rho^\alpha(T) \parallel \text{Return}(\beta, \langle \text{degreesC} \rangle) \cdot \rho^\beta(U) \big) \]

\[ = \lambda^\omega \circ \partial_H \left( \text{AcceptValues}(\alpha, \beta, \langle t \rangle, \langle \text{degreesC} \rangle) \right. \]

\[ \quad \cdot \rho^\alpha(T) \parallel \rho^\beta(U) \big) \]

\[ = \lambda^\omega \circ \partial_H \left( \rho^\alpha(T) \parallel \rho^\beta(U) \right) \]
where
\[
\begin{align*}
    s' &= s \oplus \{(\alpha, \{\text{indoorOutdoor}\}) \mapsto (\beta, \{\text{insideOutside}\})\} \\
    s'' &= s' \oplus \{(\beta, \{\text{degreesC}\}) \mapsto (\alpha, \{t\})\}
\end{align*}
\]

Hence, attribute \( t \) of object \( \alpha \) now contains the temperature as measured by \( \beta \). □

### 4.4.3.2 Asynchronous communication

The construct for asynchronous communication is translated the same way as its synchronous counterpart. Informally, the requesting object can continue its execution immediately after initiating the service request by means of (synchronous) communication between the send and receive. Therefore, the only difference with the translation of synchronous communication is that the \( \mathfrak{w}t \) atom is omitted from the translation of a send signal construct, as there is no need to wait for the completion of the requested operation:

\[
t \in \mathcal{SO} \Rightarrow E_t = \text{sn}[\text{CommExpr}(t); \text{Request}(t), \text{OutPar}(t)] \cdot \text{Tail}(t)
\]

The translation of a receive signal construct \( t \) is the same as the translation of a receive message construct:

\[
t \in \mathcal{SI} \Rightarrow E_t = \text{rec}[\text{CommExpr}(t); \text{Request}(t), \text{InPar}(t)] \cdot \text{Tail}(t)
\]

Because there is no “return from signal” construct, there is no way to pass results of the signal call back to the calling object. To fill this gap in Objectory, section 4.5 adds the necessary constructs to the original object behavior models, and expands the formal framework.

![Figure 4.13: Example of asynchronous communication](image)

**Example 4.4.2**

This example deals with asynchronous communication, in this case between an editor and a compiler. Note that the communication expressions have been omitted from this example.

78
The source code editor has facilities to start the compiler, and can handle multiple files. Therefore, the compiler is started in parallel to the editing. This allows the user of the editor to keep working while the source code of one file is being compiled. The two object behavior models of figure 4.13 are a simplification of this situation, and have the following translation (the translation of the perform task construct is presented in section 4.4.3.3):

\[
\begin{align*}
E_{e_1} &= pf[e_1] \cdot E_{e_2} \\
E_{e_2} &= sn[compile,(f)] \cdot E_{e_3} \\
E_{e_3} &= rec[compile,(s)] \cdot E_{e_2}
\end{align*}
\]

where \( f \) is the attribute referencing the file that is to be compiled. If \( \alpha \) is an instance of the class Editor, and \( \beta \) is a Compiler, the whole system \( S \) is represented by:

\[
E_S = \lambda_s^\alpha \circ \partial_H (\mu_\alpha^l \circ \rho^\alpha(E_{e_1})) || \mu_\beta^l \circ \rho^\beta(E_{c_1}))
\]

The following calculations form an example of the possible interaction between the various process algebra actions, assuming that the system state \( s \) is empty at the beginning. Again, the operator \( \mu \) is omitted from the calculations:

\[
E_S = \lambda_s^\alpha \circ \partial_H ( perform(\alpha, e_1) \cdot send(compile, \alpha, (f))
\]

\[
\cdot \rho^\alpha(E_{e_1})
\]

\[
|| receive(compile, \beta, (s)) \cdot perform(\beta, c_2)
\]

\[
\cdot \rho^\beta(C)
\]

\[
= \lambda_{E(e_1, \alpha, \omega)}^\alpha \circ \partial_H ( service_req(\beta, \alpha, (f), (s))
\]

\[
\cdot (\rho^\alpha(E_{e_1}) || perform(\beta, c_2) \cdot \rho^\beta(C))
\]

\[
= \lambda_{E(e_1, \alpha, \omega) \in \{(\alpha, (f)) \rightarrow (\beta, (s))\}}^\beta \circ \partial_H ( perform(\alpha, e_1) \cdot \rho^\alpha(E_{e_1})
\]

\[
|| perform(\beta, c_2) \cdot \rho^\beta(C)
\]

at which point object \( \alpha \) continues with the editing \( (perform(\alpha, e_1)) \), while object \( \beta \) handles the compilation of the source code \( (perform(\beta, c_2)) \). \qed

4.4.3.3 Tasks and decisions

The core of the formal framework consists of the formalization of the interaction between objects. But in order to actually do something, an object must be able to perform tasks and make decisions.

A perform task symbol represents the execution of a basic operation. For its translation a semantic function \( \lambda : T \times OTD \times S \rightarrow S \) is introduced. \( T \) is the set of basic operations used in the state transition diagrams, \( OTD \) the set of all object identifiers, and \( S \) the set of all possible states that can occur in the system. An object identifier is needed, because the task is performed in the context of a certain state by a certain object.

The translation of a perform task symbol \( t \) leaves the essence to the rewrite operator \( \rho^\alpha \) and the state operator \( \lambda_s^\alpha \):

\[
t \in T \Rightarrow E_t = pf[t] \cdot \text{Tail}(t)
\]

\[
\rho^\alpha(pf[t] \cdot X) = perform(\alpha, t) \cdot \rho^\alpha(X)
\]
The semantic function $\lambda_e$ calculates the new system state, based upon the primitive task that is associated with $t$, the object $\alpha$ that performs the task, and the previous state $s$.

$$\lambda^e_e(\text{Perform}(\alpha, t) \cdot X) = \lambda^e_e(t, \alpha, s)(X)$$

A decision symbol occurs when there are two or more alternative paths in an object behavior model (see figure 4.14). The translation of a decision $d$ is defined as follows:

$$d \in D \Rightarrow E_d = \sum_{x \text{ follows } d} \text{cond}[\text{DecCond}(x, d)] \cdot E_x$$

To provide the state operator with the identity $\alpha$ of the object that evaluates condition $c$, $\alpha$ is added by the rewrite operator $\rho^\alpha$:

$$\rho^\alpha(\text{cond}[c] \cdot X) = \text{Condition}(c, \alpha) \cdot \rho^\alpha(X)$$

The semantic function $\Phi : \mathcal{H} \times \mathcal{OID} \times \mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ evaluates the condition $c$ to true or false in the context of the current state $s$ and object $\alpha$.

$$\lambda^e_e(\text{Condition}(c, \alpha) \cdot X) = \langle \Phi(c, \alpha, s) \rangle \cdot \lambda^e_e(X)$$

The execution path that is preceded with a condition that results in false is blocked by a deadlock, and the complementary path continues.

The evaluated condition is handled by conditional processes:

$$\langle \text{true} \rangle = \varepsilon$$
$$\langle \text{false} \rangle = \delta$$

---

Figure 4.14: Example of a decision construct
Example 4.4.3

Assume that the groundsman of a soccer stadium can decide if he wants to mow the grass in lines perpendicular to the sidelines, or mow it in somewhat more challenging diagonal lines. Either way, once he has mown the first line he is committed: all further mowing has to be done in the same direction. Figure 4.14 contains a fragment of a simplified object behavior model of this situation. In the beginning a decision is made: either task $t_1$ (mowing a perpendicular line) or task $t_2$ (mowing a diagonal line) will be repeatedly performed through the dynamic use of the previous state construct $s_3$. If the groundsman has decided to mow the field perpendicularly, the most recently encountered state is $s_1$. If, on the other hand, he has started mowing diagonally, the most recently encountered state is $s_2$. The effect of encountering the previous state construct $s_3$ is that execution continues in the most recently encountered state, thereby ensuring a consistently mowed field.

The translation of this model to process algebra:

\[
E_d = \text{cond}[\text{perpendicular}] \cdot E_{s_1} + \text{cond}[\text{diagonal}] \cdot E_{s_2} \quad E_{s_1} = \text{st}[s_1] \cdot E_{t_1} \\
E_{t_1} = \text{pf}[t_1] \cdot E_{s_3} \\
E_{t_2} = \text{pf}[t_2] \cdot E_{s_3} \\
E_{s_2} = \text{st}[s_2] \cdot E_{t_2} \\
E_{s_3} = \text{ps}
\]

Under the assumption that the groundsman decides to mow the grass perpendicularly, the full system $s$ (consisting of a single groundsman object $\alpha$) can be rewritten as shown below. At the beginning of the fragment, the environment is presumed to be empty, and the system is in state $s$.

\[
E_s = \lambda_s^\circ \partial_H\left( \mu_{\alpha} \circ \rho^\alpha(E_d) \right) \\
= \lambda_s^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\text{Cond}(\text{perpendicular}, \alpha) \cdot \text{st}[s_1] \cdot \text{Perform}(\alpha, t_1) \cdot \text{ps} \\
+ \text{Cond}(\text{diagonal}, \alpha) \cdot \text{st}[s_2] \cdot \text{Perform}(\alpha, t_2) \cdot \text{ps}) \right) \\
= \lambda_s^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\text{Cond}(\text{perpendicular}, \alpha) \cdot \text{st}[s_1] \cdot \text{Perform}(\alpha, t_1) \cdot \text{ps} \\
+ \mu_{\alpha}^{\mu}(\text{Cond}(\text{diagonal}, \alpha) \cdot \text{st}[s_2] \cdot \text{Perform}(\alpha, t_2) \cdot \text{ps}) \\
+ \varepsilon \cdot \lambda_s^\circ \partial_H(\mu_{\alpha}^{\mu}(\text{st}[s_1] \cdot \text{Perform}(\alpha, t_1) \cdot \text{ps})) \\
+ \delta \cdot \lambda_s^\circ \partial_H(\mu_{\alpha}^{\mu}(\text{st}[s_2] \cdot \text{Perform}(\alpha, t_2) \cdot \text{ps})) \\
= \lambda_s^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\text{Perform}(\alpha, t_1) \cdot \text{ps}) \right) \\
+ \delta \\
= \lambda_{E(t_1, \alpha, s)}^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\text{ps}) \right) \\
= \lambda_{E(t_1, \alpha, s)}^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\rho^\alpha(E_{s_1})) \right) \\
= \lambda_{E(t_1, \alpha, s)}^\circ \partial_H\left( \mu_{\alpha}^{\mu}(\text{st}[s_1] \cdot \text{Perform}(\alpha, t_1) \cdot \text{ps}) \right) \\
= \ldots
\]

□

4.4.3.4 Object creation and destruction

An important issue in object-orientation is the creation and destruction of objects. Objectory provides an explicit object destruction symbol, but lacks a create object symbol. This
framework provides support for both symbols.

A create object construct $t$ with initialization parameters $\text{OutPar}(t)$ is translated as follows:

$$t \in \mathcal{G} \Rightarrow E_t = \text{cr}[\text{OfClass}(t), \text{NewID}(t), \text{OutPar}(t)] \cdot \text{Tail}(t)$$

The $\text{OfClass}(t)$ function yields the class to which the new object belongs, $\text{NewID}(t)$ is the attribute that receives the OID of the newly created object, so that the new object can be referenced by the creating object. The OID $\beta$ of the creating object is inserted by the $\rho^\beta$:

$$\rho^\beta(\text{cr}[^c, ^id, ^P] \cdot X) = \text{Create}(\beta, ^c, ^id, ^P) \cdot \rho^\beta(X)$$

As all process algebra translations of the objects are already created, the creation of a new object boils down to granting the new object $\alpha$ permission to continue its execution beyond the $\text{WaitCreate}$ operation:

$$\text{Create}(\beta, ^c, ^id, ^P) \mid \text{WaitCreate}(c, ^a, ^Q) = \text{Created}(\alpha, \beta; ^id, ^P, ^Q)$$

Updating the system state is handled by the state operator $\lambda^s_\sharp$ in conjunction with a semantic function $\sharp : \mathcal{OTD} \times \mathcal{S} \rightarrow \mathcal{S}$, the initialization function. The task of the initialization function is to ensure that the newly created object conforms to invariants and constraints that are placed upon it. In the case of Objectory, these constraints are limited to cardinality constraints.

$$\lambda^s_\sharp(\text{Created}(\alpha, \beta; ^id, ^P, ^Q) \cdot X) = \lambda^{\sharp}_s(\alpha, ^a; (\beta, ^id); \alpha; (\beta, ^P)\rightarrow (\alpha, ^Q)))(X)$$

The new system state contains the initialization parameters and the new OID $\alpha$, and has been modified by the initialization function $\sharp$. Also, at this point the initialization function adds object $\alpha$ to the active objects of its class.

The end of the life of an object is indicated by a destroy construct $t$, which has the following simple translation:

$$t \in \mathcal{F} \Rightarrow E_t = \text{des}$$

$$\rho^\alpha(\text{des} \cdot X) = \text{Destroy}(\alpha) \cdot \rho^\alpha(X)$$

Because of the disappearance of object $\alpha$, the system state must be updated to reflect that $\alpha$ is no longer an active object. This is handled by the semantic function $\sharp : \mathcal{OTD} \times \mathcal{S} \rightarrow \mathcal{S}$, the destruction function.

$$\lambda^s_\sharp(\text{Destroy}(\alpha) \cdot X) = \lambda^{\sharp}_s(\alpha, ^a)\sharp(X)$$

Dangling links as a result of the destruction of an object can be avoided if the semantic function $\sharp$ conforms to the dynamic semantics axioms of section 3.4.
4.4.4 Validation support

Process Algebra is a formalism that supports the execution of its specifications. Executing a specification stepwise is a natural mechanism to yield a trace of the actions that are performed by the system. In process algebra, such a trace consists of atomic actions, and can be used for validation purposes as described in the Scenario Validation Principle.

The state operator $\lambda s$ incorporates the effect of actions into the state $s$. Therefore, the atoms are no longer visible in the rewriting after the state operator has passed over them. However, a trace can be visualized in a trivial way by regarding the sequence of atoms that is processed by the state operator as a trace. Example 4.4.4 constructs a simple trace from a rewriting.

Example 4.4.4

This example constructs the trace that results from the rewritings in example 4.4.2. The rewriting is repeated in a shortened, symbolic form on the left, while the resulting trace is constructed on the right.

<table>
<thead>
<tr>
<th>Abbreviated rewriting</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perform(1) • Send...</td>
<td>ReceivePerform(2)...</td>
</tr>
</tbody>
</table>

As yet, nothing has happened.

<table>
<thead>
<tr>
<th>ServiceReq(Perform(1)...</th>
<th>Perform(2)...)</th>
<th>Perform(1)</th>
</tr>
</thead>
</table>

Perform has been interpreted by the state operator.

<table>
<thead>
<tr>
<th>Perform(1)...</th>
<th>Perform(2)...)</th>
<th>Perform(1) • ServiceReq</th>
</tr>
</thead>
</table>

The service request was accepted, and processing of the request has started.

\[ \ldots \]

\[ \ldots \]

4.5 Intraobject concurrency

Objectory supports synchronous and asynchronous communication between objects, but does not support parallelism within an object. In [EJW95], this kind of parallelism is referred to as intraobject concurrency. Intraobject concurrency can be especially helpful when modelling complex objects, and is important for (recursive) system decomposition. In order to support intraobject concurrency in full, a number of concepts is added to object behavior models. This section describes these concepts and their formalization.

4.5.1 Internal parallelism

First of all, it must be possible to express that two object behavior constructs are executed in parallel. This is accomplished by identifying the points in the object behavior model where alternatives branch out. These points ($x$, $y$, $z$ in figure 4.15) are provided with their own identity, and incorporated in the formal framework. The identities of these splitpoints
make up the set $B$ ($B \subseteq X$). At such a splitpoint only one of the possible continuations of the execution path is chosen. In all other situations, if two arrows depart from the same construct, both continuations are executed in parallel.

The translation of a splitpoint $t$ consists of its Tail:

$$ t \in B \Rightarrow E_t = \text{Tail}(t) $$

Thus, internal parallelism is incorporated into the formal framework by means of a redefinition of the Tail. If $t$ is a splitpoint ($t \in B$), a choice is made between all possible continuations of the execution path:

$$ t \in B \Rightarrow \text{Tail}(t) = \sum_{x \text{ follows } t} E_x $$

Otherwise, the available continuations are started in parallel:

$$ t \in X \setminus B \Rightarrow \text{Tail}(t) = \big|_{x \text{ follows } t} E_x $$

**Example 4.5.1**

Using the new definition of Tail, the translations of the object behavior models in figure 4.15 are:

$$
\begin{align*}
E_a &= \text{st}[a] \cdot E_x \\
E_x &= E_b + E_c \\
E_d &= \text{st}[d] \cdot (E_e \parallel E_f) \\
E_g &= \text{st}[g] \cdot (E_y \parallel E_z) \\
E_y &= E_h + E_i \\
E_z &= E_j + E_k
\end{align*}
$$

4.5.2 Synchronization

Introduction of internal parallelism allows divergence of the execution flow. To force the convergence of several parallel flows, the synchronizer is added to the framework. Informally, a synchronizer suspends execution of an object until all activities “upstream” are
completed. At that point, execution of the waiting object may resume. The process algebra translation of synchronizers is based on the scheme presented in [HN93].

Synchronizers are translated with the help of the communication function. Each behavior model construct $x \in \mathcal{X}$ that is followed by a synchronizer $y \in \mathcal{Y}$ produces a unique atomic action $\sigma_{x,y}$ upon completion of its own work. A synchronizer is started as soon as all its input atomic actions are available. This is done by means of the communication function, which is defined for each synchronizer. Because a synchronizer can well have more than two inputs, communication is not necessarily binary. The production of the synchronization atoms is reflected in the modified Tail function. As an abbreviation, the set of non-synchronizers $\mathcal{U}$ is defined as $\mathcal{X} \setminus \mathcal{Y}$, and following the adopted convention this implies $u \in \mathcal{U}$.

For $t \in \mathcal{B}$, a single continuation of the execution path is chosen:

$$t \in \mathcal{B} \implies \text{Tail}(t) = \sum_{u \text{ Follows } t} E_u + \sum_{y \text{ Follows } t} \sigma_{t,y}$$

Otherwise, all subsequent execution paths are started in parallel:

$$t \in \mathcal{X} \setminus \mathcal{B} \implies \text{Tail}(t) = \| E_u \|_{u \text{ Follows } t} \| \sigma_{t,y} \|_{y \text{ Follows } t}$$

Because the translation of a decision construct does not use the Tail function, its translation is adapted to produce a synchronization atom $\sigma$ for the continuations of decision $d$ that start with a synchronizer $y$:

$$d \in \mathcal{D} \implies E_d = \sum_{u \text{ Follows } d} \text{condDecCond}(u, d) \cdot E_u + \sum_{y \text{ Follows } d} \text{condDecCond}(y, d) \cdot \sigma_{d,y}$$

The definition of a synchronizer consists of three parts: an ACP expression, a definition of the communication function, and a definition of its encapsulation set. The expression for a synchronizer $y \in \mathcal{Y}$ consists of its Tail, as its only task is to start its followers in parallel:

$$y \in \mathcal{Y} \implies E_y = \text{Tail}(y)$$

The communication function is defined as:

$$\sigma_{x,y} = E_y$$

This definition implies that all inputs have to be finished before synchronizer $y$ is started. Of course, the above expression is only present for synchronizers that are followers of other object behavior model constructs, i.e. $y \in \mathcal{Y}$ such that an $x \in \mathcal{X}$ exists such that $y \text{ Follows } x$.

In order to assure that the $\sigma$'s are used exclusively to trigger the synchronizer, the encapsulation operator $\partial_H$ has to be applied. Every $\sigma_{t,s}$ is a member of the encapsulation set $H$. 

85
Example 4.5.2

Figure 4.16 contains an example of the use of a synchronizer: synchronizer $s_1$ ensures that task $t_3$ will not be executed until both task $t_1$ and the synchronous message request $t_2$ have been completed. The translation of the constructs in this example is:

\[
\begin{align*}
E_{t_1} &= pf[t_1] \cdot \sigma_{t_1,s_1} \\
E_{t_2} &= sn[...] \cdot \sigma_{t_2,s_1} \\
E_{t_3} &= pf[t_3]
\end{align*}
\]

where

\[\sigma_{t_1,s_1} \cdot \sigma_{t_2,s_1} = \sigma_{t_3,s_1} = E_{t_1}\]

In addition to internal synchronization, external synchronization is needed. An external synchronizer, called collector, waits until a number of return values is computed by certain objects. This enables an object to asynchronously start a number of services from other objects while continuing its own execution, and collect the values computed by the “helper” objects at a convenient time. The collector waits until all return values have been received and entered into the environment of the requesting object, after which it allows execution to continue.

For the definition of the translation of a collector $z \in Z$ a function \texttt{Collect} is introduced that indicates which values are to be collected from which providers. \texttt{Collect}(z) yields tuples $\langle p, R \rangle$, with $p$ the attribute that contains the OID of the provider, and $R$ a tuple of attributes that receive the values computed by the object indicated by $p$. Therefore, no two tuples may share the same provider attribute. Using this function, the translation of a collector $z$ is:

\[z \in Z \implies E_z = cl[\texttt{Collect}(z)]\]

Besides adding the identity of the executing object $\alpha$, the $\rho^\alpha$ adds the auxiliary functions $Q$ and $F$, both initially $\emptyset$. $Q$ is used to gather the values that have already been collected. Given a provider attribute $p$, $F(p)$ yields the OID of the actual provider of the collected values.

\[\rho^\alpha(cl[C] \cdot X) = \texttt{Coll}(\alpha, C, \emptyset, \emptyset) \cdot \rho^\alpha(X)\]
To make the computed values available to the requesting object, a *return from signal* construct is introduced. Its translation and graphical representation are identical to the translation and representation of a return from message construct.

The way the communication function is used to collect the values resembles the way it is used in synchronous communication. The major difference is that it is possible that the collector has to wait on more than one servicing object. As soon as a servicing object has computed its return value(s), it can communicate with the collector. The collector incorporates the computed values in $Q$, updates $F$ to reflect that $\beta$ is the actual provider of the values, removes $p$ from the list of pending collects $C$, and resumes waiting for the remaining collects. Of course, values can be provided only once for each provider $p$. Therefore, communication is possible only if $F(p)^\dagger$: no actual provider has delivered values for $p$.

$$\text{Coll}(\alpha, C, Q, F) \mid \text{Return}(\beta, V) = \text{Coll}(\alpha, C \uplus \{p\}, Q \uplus \{p : C(p) \mapsto V\}, F \uplus \{p : \beta\})$$

As is the case with the other communicating actions, the Coll and its parameters is part of the encapsulation set $H$ of the encapsulation operator $\partial_H$. To enable the state operator $\lambda$ to process a fully served request, that request must be allowed to pass through the encapsulation operator. Hence, the Coll actions that have an empty $C$ parameter are not part of the encapsulation set $H$.

When all values have been computed ($C = \emptyset$), the state operator $\lambda$ transfers the computed values to the state of the requesting object. However, the values are only transferred if they originate from the correct provider.

Let $\langle \alpha, \emptyset, Q, F \rangle$ be a fully served request from an object $\alpha$, and $s$ a system state, such that for all $p \in \text{dom}(Q)$, the actual provider $F(p)$ corresponds with the requested provider $s(\alpha, p)$, then:

$$\lambda^e_s(\text{Coll}(\alpha, \emptyset, Q, F) \cdot X) = \lambda^e_{s \uplus \{F(p)^\dagger \mid p \in \text{dom}(Q)\}}(X)$$

Otherwise this rewrites to $\delta$.

![Figure 4.17: Accepting values from asynchronous service requests](image)

Example 4.5.3

Consider the example in figure 4.17. Suppose that collector $c$ waits for values $r_1$.
and \( r_2 \) from the respective providers \( p \) and \( q \), which happen to be objects \( \alpha \) and \( \beta \). \( t_1 \) and \( t_2 \) are signals to \( \alpha \) and \( \beta \). In that case, synchronizer \( s_1 \) merges the diverged execution flows into one flow, and the collector is started. As soon as both \( \alpha \) and \( \beta \) have indicated that they have computed values for \( r_1 \) and \( r_2 \), those values are retrieved by the collector, and execution continues.

Example 4.5.4

Figure 4.18 contains object behavior models of two classes designed to compute a Fibonacci number. The first variant uses communication expressions that always yield the same result (in this case, the constant “\( \text{any} \)”), irrespective of the OID of the servicing object. This allows it to use two “anonymous” objects to recursively compute the values of \( \text{fibb}(x-1) \) and \( \text{fibb}(x-2) \) in parallel. After both values have been computed, they are added and returned to the requesting object.

In the second variant, two new fibonacci calculators are created to calculate the values of \( \text{fibb}(x-1) \) and \( \text{fibb}(x-2) \). In principle, after delegating the computation to the

\[ \text{collect} \begin{pmatrix} \langle k, \text{any} \rangle, \langle l, \text{any} \rangle \end{pmatrix} \]

\[ r = r + s \]

\[ \text{outpar} \]
two new objects, the creating object is free to perform other tasks. However, in this example this opportunity is not used: the first object collects the values of $\text{fibb}(x - 1)$ and $\text{fibb}(x - 2)$, adds them and returns them to the object that requested the service $\text{fibb}(x)$.

Example 4.5.5
This example is based on the Chain of Responsibility presented in [GHJV95]. Consider a context-sensitive help facility. The help that is provided depends on the part of the interface that is selected and its context. The help message that accompanies a standard Continue/Abort/Help dialog varies with the associated action.

A way to handle this problem is to organize help information according to its generality, and associate a “helpdesk” with each object that offers context sensitive help. Requests for help can then be handled by the helpdesk of the dialog, or it can be passed along to a more general helpdesk (bigDesk in figure 4.19). This way, there is no need for the requesting object to know the identity of the object that serves the request for help. It suffices that the requesting object knows its own, little helpdesk. Help requests are communicated to that helpdesk, and are passed along a chain of desks until some desk provides the information. Note that the object behavior model of the Application uses the extended decision construct: it has three continuations, each with its own condition.
4.6 Conclusion

This chapter presented a general, formal framework for communication in object-oriented analysis methods. The core of the framework is based on Objectory, which is representative for current object-oriented behavior modeling techniques. The framework is extended to offer all common ways of communication and support the specification of (complex) object behavior and interaction, such as synchronous and asynchronous communication, parallelism and synchronization, and nondeterministic behavior. These extensions facilitate dealing with distributed control and multi-faceted domains.

The formal foundation in Process Algebra allows for verification and validation of models by allowing dynamic simulation, which provides rapid feedback to the model designer. Although the formal model may seem complex, especially in computations, it should be kept in mind that process specifications, in particular those using asynchronous communication and concurrency, are by nature very complex.

Furthermore, additional models are needed to further facilitate communication with users in the very early phases of system development. These should be formally related to the core models defined in this chapter. Finally, we aim at a more sophisticated declarative specification formalism, as the current communication model is sometimes too operationally oriented. For instance, there is a direct link between service signatures and service implementations, which is a violation of the Service Decoupling Principle.
Chapter 5

Component Modeling

5.1 Introduction

The software engineering discipline has adopted many engineering and design principles to guarantee the fulfillment of quality requirements and to increase the engineers’ productivity. A central quality requirement is the flexibility of the resulting software product. Flexibility is measured by estimating the effort necessary to adapt a software system if the environment of that system undergoes changes, albeit unforeseen. In today’s practice, where companies rapidly introduce new product lines and services, where the society happens to introduce new currencies (the Euro) and new tax rates, and where the Earth circling around the Sun introduces new millennia, flexible software systems are a key success factor to survive.

In particular, the design principle of modularity has been applied to the software engineering process, giving rise to the development of reusable software components. Using software components is not a recent invention. One of the benefits of UNIX [Sal94] is its possibility to chain programs by the so-called pipe mechanism. Programs read from their standard input and write their results into their standard output.

The notion of a software component is a fuzzy one and therefore, it has been gratefully used as part of development tool suppliers’ selling machinery. It is a hard job to define this concept precisely. Many factors may play a role in this demarcation:

- Size. Are scrollbars considered to be a software component? And “Open File”? And customer databases?
- Type. Should data components, process components and user interface components be distinguished from one another?
- Technology. Is the technology independent description of the functionality of “a” spelling checker’s functionality perceived to be a software component? And what about Oracle tables, MS Access forms and Smalltalk libraries?
• Genericity. Are report writers or DBMSs software components? And code parts which realise the addition of 5 to an input value? Is it possible to tune software components, e.g. a production planning module in an ERP (Enterprise Resource Planning) package?

In [Szy98], Szyperski distinguishes three characteristic properties of components:

1. A component is a unit of independent deployment.
2. A component is a unit of third-party composition.
3. A component has no persistent state.

To ensure that a component is independently deployable, the component needs to be well separated from its environment and from other components. This requirement is captured by the Locality Principle of chapter 2. In this context, a third party is defined as a party that does not have access to the internal workings of a component. For a component to be used in conjunction with other components by such a third party, it must be completely self-contained. Furthermore, it has to provide a clear specification of the services it provides. In other words, a component needs to interact with the outer world through a well-defined interface, which is stimulated by adherence to the Service Decoupling Principle. The lack of a persistent state ensures that components can easily be reused. For example, a database server can act as a component. The database itself is clearly not a component. To use components in a specific setting, they can be parameterized. Note that adjusting the behavior of component this way is far removed from having a persistent state, and, from an outside perspective, is similar to changing the compiled code.

In [VW96], a technique has been presented to describe software components. This technique is an attempt to define software components precisely. In the design of this technique (called CMT, in full, Component Modeling Technique), the position is taken that components can be used to create larger components and that it is possible to make components more specific or more generic as needed. This provides support for modeling software at different levels of granularity, which is required by the Rightsizing Principle.

CMT is a technique that provides constructs to model functionality, while making no assumptions whatsoever with regard to the way that functionality may be implemented. This implies conformance with the Service Decoupling Principle. Because of the high level of abstraction of CMT, technical standards such as OLE, OpenDoc, CORBA or DCOM are not considered in this chapter.

Compared to Object-Oriented approaches CMT offers extensions for the composition of components. As stated in [Bas97], reuse is not merely a matter of stringing modules together in different combinations. Reuse must be more general, because it has to be possible to change the component’s properties that remain fixed during execution. In general, even though Meyer associates OO with a “component culture” [MM92], object-oriented techniques lack a mechanism to build more complex objects based on existing objects, which is
in violation with the Rightsizing Principle. Furthermore, object reuse is largely restricted to inheriting properties from closely related but more general ancestors. CMT provides a structural mechanism to define new components not only by similarities with other components, but also by means of dissimilarities. This results in models that emphasize the distinctions between different components more clearly than purely object-oriented techniques can.

Compared to traditional process modeling approaches (e.g. [LGN81, MM87], CMT offers an elegant instantiation mechanism to refine specifications. In addition to that, partial specifications in the form of components can be reused in a natural way. An important difference with dataflows is that CMT exclusively uses typed connections between components.

This chapter aims at demonstrating CMT’s use and benefits, and embeds it in an object-oriented context to enhance its functionality. CMT is applied to model a complex real-life software application. It is shown that the concepts in the technique provide for a good insight into the software components underlying the design of the application. Furthermore, insight is given into the potential flexibility of the application, whereas it is possible to abstract from its implementation. Finally, CMT is positioned in a larger modeling context to provide an integrated technique for component modeling which can model control flows. Focus is on the application of these ideas, not on their formal embedding.

This chapter is structured as follows. Section 5.2 presents the main modeling concepts in CMT, as far as they are necessary to comprehend the subsequent sections. Section 5.3 presents the real-life application which is chosen to be a realistic and complex casus. Section 5.4 applies the CMT concepts to this application and illustrates the types of insights to be gained. Section 5.5 uses object behavior models and object-oriented data models to extend the reach of CMT. The extended technique, called Integrated CMT, is applied to the real-life application of section 5.3. Section 5.7 summarises the main observations.

5.2 CMT

The notion of a software component is tentatively described as “a potentially reusable part of a software system”. To be able to define the notion of a software component more precisely, the following preliminaries are used:

- A software component realises a particular functionality, which is determined by the set of requests from the environment that the software component can handle.

- It is not relevant how the software component realises its functionality, so its internal structure is not relevant.

- However, the interface that the software component offers its environment should be taken into account. It should be known which external knowledge the software component needs to handle a particular request. It should also be determined which results are delivered by the software component.
• It should be possible to use software components to realise new software components
  which are larger or more complex. Composition principles should be taken into ac-
  count.

This set of preliminaries reflects a system theoretical view in which software components
are perceived as black boxes that are part of a larger system (the environment) and react
to stimuli from this environment. The notion of a software component should be defined
using the notions of *communication* and *composition*. These notions are explored in the
next two subsections, the core composition notions of CMT are found in sections 5.2.2.1,
5.2.2.2, and 5.2.2.3. In the remainder of this chapter, we will refer to a software component
as a “component”.

### 5.2.1 Communication

A component is a software unit which can handle different request types. For each request
type, a component is able to receive information from its environment, necessary to handle
the request: its input parameters. Handling a request provides for a result: its output
parameters.

In CMT, a component is represented as a “plug” (see e.g. figure 5.1), each line represents
a parameter. The set of request types is seen as an input parameter, represented in the
header of the component description (again, see figure 5.1).

Input and output parameters have a name and a type. The type indicates which values the
parameter may have and which operations are allowed on these values. Therefore, a (data)
type is an elementary component. Parameters can be instantiated with a value taken from
the corresponding domain.

![Figure 5.1: The component Integer](image)

Figure 5.1 shows an example of a component: the component *Integer*. This component
can perform calculations based upon input information which consists of a request (*zero*,
*add*, *subtract*, *multiply*, *divide*, *modulo*) and the necessary additional information: one or
two integer arguments. It is assumed that the component “knows” for each request type
which input parameters should be instantiated to be able to perform the corresponding
calculation. For example, the request *zero* does not need any input, as it generates the
initial integer value, whereas the request *modulo* needs a value for both input parameters.
Instantiating this component with the request modulo, with \( x \) “9” and with \( y \) “4” gives as a result that result takes the value “1”. The way in which integers are represented and in which the requests are implemented remains hidden.

This sample component shows that requests can involve the class of integers: the request zero does not need any additional input information and its execution results in a new instance of an integer. At the same time, requests can involve particular instances of integers: for handling one of the other requests, the component needs to know the values of the input parameters to be used in the calculation. Note that the type of the input parameters is the component itself.

Figure 5.2 shows a more complex example of a component: the Memory manager. This component can write values of any type (integers, reals, strings, lists, and so on) into a global memory and read these values when requested. The request type has the type \{read, write\}. The read request needs the name and the type of the variable to be read as input information, the value of that variable is then returned. The write request needs the variable name, the type of the variable, and the value to be written.

![Figure 5.2: The Memory manager component](image)

5.2.2 Composition

In CMT, three basic composition principles are distinguished to create new components using existing ones: instantiation, aggregation, and concatenation.

5.2.2.1 Instantiation

Some input parameters are given one fixed value by which a new, more specific, component is created. This mechanism is called instantiation in CMT. The integer component (figure 5.1) can be tuned towards an addition component by giving the value “add” to the request type. The result is a specific component without a set of request types. By instantiating the memory manager (figure 5.2), a simple customer database can be described by instantiating the input parameter type indicator with the predefined type “customer”. Instantiation can be applied on any input parameter, although the resulting component will not always make sense. For example, the integer component can be adapted by instantiating the
input parameter $x$ with the value “5”. The resulting component can perform “5-related calculations”.

CMT’s notion of instantiation differs fundamentally from instantiation as used in an object-oriented context. All instances of a class of objects conform exactly to the same blueprint (the class definition). At the abstraction level CMT operates on, no attention is paid to the creation or destruction of instances of particular component types. CMT focuses on the function that must be performed, and which type of component is suitable to perform that function. Instantiation in CMT takes on a different meaning, which is actually closer to OO’s specialization or inheritance concept. CMT views a component as a general, high level building block. To apply such a building block in a particular situation, it must be tailored to the situation at hand. The result of this tailoring is a more specialized component, which is called an “instance” of the component it is derived from. Bhapat’s so-called capsules [Bap94] provide a reuse mechanism that walks a middle ground between OO’s inheritance and CMT’s instantiation.

5.2.2.2 Aggregation

A component is an aggregation of a number of other components if it is specified for each of these components under which conditions that component will be executed. Aggregation may be compared to the “case of” construction in programming languages. This is represented by a circle in which the left hand of the condition is shown, and a number of lines from this circle to the constituent components which are labeled with the individual right hands of the conditions.

Aggregation as it is used in CMT exhibits a slight resemblance to the aggregation concept of object-orientation, as both aggregations group a number of components to perform a particular task. The main difference is that in contrast to OO, CMT provides an unambiguous definition of the responsibility of each of the components that take part in the aggregation. On top of that, CMT offers the grouping of components into a larger component. Because the world on the outside of the component is not aware of its composed nature, this is a very powerful mechanism. OO’s aggregations usually do not encapsulate their internal structure from the outside world. A notable exception is the OSA method described in [EKW92], which provides so-called high-level objects.

5.2.2.3 Concatenation

Concatenation enables the construction of networks of components. Concatenation may be compared to the way in which process structures are composed in data flow diagrams or in which integrated circuits are composed using elementary chips or in which processes are “piped” in UNIX environments. If two components are concatenated, some of the output parameters of a component are related to some of the input parameters of the other component. For each output parameter, its type should conform to the type of the related input parameter. This is graphically represented by connecting the lines that represent the parameters involved.

It turns out that this form of abstractly coupling components works very well in practice,
as it is very flexible. Adding or deleting connections between components can be done independent of the interior of the components. In a way, CMT’s concatenation can be compared to object-oriented interaction models, but without taking timing considerations into account. The interaction model describes in what order objects pass information, whereas CMT only models what type of information is passed between components.

5.2.2.4 An example of composition

The stopwatch component from figure 5.3 illustrates the three mechanisms to build large components. It has been constructed using instantiations of the components “Integer” and “Integer memory”. In this example, the Integer component is seen as a component which can perform the operations increment which increments the input parameter, and zero which returns the integer 0. The Integer memory component is seen as an instantiation of the Memory manager from figure 5.3 where type indication is fixed on the value “Integer”.

This stopwatch component has only one input parameter which is its request, called stpw-hndl. The stopwatch is able to handle three request types: display, reset, and count (which increments the stopwatch value). It has also one output parameter (for the display request). Figure 5.3 shows that the stopwatch is constructed as an aggregation of three smaller components, where the request acts as the condition for the choice of one of these three. The components which are related to the reset and the count requests are concatenations of elementary components.

5.3 Case description

The case that is described in this section is inspired by a real-life front-office application that has been developed in the realm of banking and insurance [OH93, Hub94, OH95]. In
the remainder of this chapter, we will refer to this application as the Advice Application (AA).

The project, which was performed over the course of eight months, started with determining the *look-and-feel* of the application. This was done using panels of stakeholders (including future users) and domain experts. The panels were supported by a number of user interface specialists and object-oriented software development specialists.

The results of the panel sessions were captured in a design of a prototype of the advice application, to gain insights into the feasibility of realizing the ideas of the panel sessions. This prototype was then transformed into an industrial strength application.

The main task of the application is to provide support for financial advisers during their conversations with customers. During such a conversation, the adviser tries to understand exactly what the customer’s needs are, and how to satisfy them with the available products.

### 5.3.1 Flexibility

Most conversations between an adviser and his customer follow a reasonably standardized path, although each conversation follows a specific path, depending on the need of the customer. There are standard paths for borrowing money, investing money, obtaining insurance, or buying a house. We call these standard paths *business scenarios*, or just *scenarios*.

From the point of view of the adviser the ideal customer does not deviate from the standard path. In reality, many conversations are less structured. Customers re-visit previous topics at a later time, and customers bring topics up ahead of time. Supporting this process requires the application to be highly flexible. If the application provides enough flexibility, it is the *customer* who dictates the flow of the conversation, and not the *application*. The sole purpose of the application is supporting the adviser to follow (and guide) the customer, without dictating the course of the conversation.

Another aspect of flexibility is found in the need to react to the information that is provided by the customer. Conversation topics that were interesting at the start of the conversation may become irrelevant. Of course, the opposite can happen too: a topic that was not relevant at the outset can become highly relevant. The application must support this dynamic behavior by notifying the adviser of new, relevant topics, and by removing topics that have become irrelevant. This type of functionality suggests a natural distribution of control to individual topics. Topics then monitor their own relevancy, and send a signal if they become relevant enough to warrant a change in their presentation.

Furthermore, a sufficiently flexible application allows the adviser to adapt the standard scenarios to his own preferred conversation style, or tailor the conversation to the expectations of the customer. Some advisers and customers prefer to explore various global options first, and do an in-depth analysis later. Others might find a more methodical approach that follows the standard path more closely more appropriate.

A final aspect of flexibility follows from the need to quickly react to changes in the market, and the ability to quickly create new products. This can be made possible by encapsulating
all data relevant to a product in a local data structure. To be effective, new products must be supported by scenarios. Creation of new scenarios must be possible with minimum effort. In this respect, AA is a prime example of a multi-faceted application that uses localized data and distributed control.

5.3.2 Application core

The *scenario* is the central concept of the application. A scenario provides a structured starting point to compose an advice that fits the situation of the customer. In the case of this organisation, many scenarios are related to a number of products, such as specific mortgage types, loans, or life insurance.

A scenario is composed of a number of relevant *topics*. Each topic addresses a particular issue, such as the income of the customer, his financial situation, or the fiscal consequences of buying a house. A topic can be compared to a screen or a calculation. See figure 5.4 for a sample scenario, and figure 5.5 for a sample topic.

| • Check address data  
• Determine credit need  
• Calculate income  
• Calculate maximal mortgage  
• Calculate credit limit  
• Mortgage  
• Fiscal consequences |

Figure 5.4: Credit need inventarisation scenario

| • Net monthly income?  
• Number of extra months?  
• Amount of corporate savings per month?  
• Yields a gross yearly salary of . . . |

Figure 5.5: Salary conversion topic

The course of the conversation is captured by the *Conversation course* component. At the start of the conversation, the *Conversation course* is initialized to a specific scenario. During the conversation, information provided by the customer while discussing a topic can give rise to an extension of the *Conversation course*. In that case, a new topic that has become
relevant in the current situation is appended to the Conversation course. For example, if the customer decides to take out a mortgage on his house, a new topic “Life insurance” becomes relevant, and is added to the Conversation course. This growth of the Conversation course points at the most important distinction between a scenario and the Conversation course. Although the Conversation course starts with the same set of topics as a scenario, topics may be added to or removed from the Conversation course in the course of a conversation. A scenario on the other hand is completely static: it consists of a fixed set of topics.

The application is designed using four layers, see figure 5.6. The domain layer contains components that are specific for the banking and insurance world, such as Income, Taxes, Security, Customer, and Contract. These components form the basis upon which topics are built, and act as uniform intermediaries between the persistent data in the database and the application core. Their relations are shown in figure 5.7, which also introduces a partial concrete syntax for the object-oriented data models of chapter 3. Inheritance associations are depicted using dashed arrows that connect a subtype with its supertypes. Binary symmetric associations are drawn as a solid line that connects the two participating classes.

The focus of this chapter is on the topic and domain layers.
5.4 Application of CMT to the Advice Application

This section presents examples of some of the more complex components in the topic layer. The components presented here are topics of the scenario in figure 5.4.

5.4.1 Income calculation

"Income" plays a central role in a multitude of scenarios. However, it is no exception that a customer asks for a preliminary advice while not knowing his gross yearly income. If the customer does know his monthly income after taxes, a topic that gives a good estimate of the gross yearly income based on the monthly income can be used.

Figure 5.8: Salary conversion

Figure 5.8 depicts the **Salary conversion** component. Its incoming parameters are the Customer-id, a number of Extra months, the Net monthly income, and a parameter that indicates if the customer participates in a special corporate savings plan.

The component can be divided into three distinct parts. The first two parts calculate the yearly income and factor in the effects of tax deductions. The third component receives the result of the calculation and stores it in the income data of the correct customer, to make it available for future use.

The first and second component are responsible for the actual calculation. The first component converts the monthly income to a yearly income by multiplying the net monthly income by 12 (months in a year) times 1.08 (yearly allowance for holiday expenses), and adding a thirteenth and fourteenth month if applicable. The second component uses its knowledge of tax laws (e.g. tariff groups) to convert the net yearly income to a gross yearly income. Naturally, the **Salary conversion** component can be used in higher level models as a basic component.
5.4.2 Obtaining approval

After the necessary data (such as income) are gathered, an agreement can be made that formalizes the sale of a banking product (e.g. a loan) to the customer. As such an agreement is legally binding, the adviser must first obtain final approval before the agreement can be made up. Whether the adviser himself is authorized to approve the agreement depends on the particular product that forms the subject of the agreement and the amount of money involved. Generally speaking, advisers of higher ranks are allowed to approve agreements that involve larger amounts. Figure 5.9 contains the Obtain approval component.

![Diagram of Obtain approval component]

As can be seen in figure 5.9, the first component checks if the adviser is authorized to approve the agreement. If he is, the actual approval is handled based on an assessment of the customer (which includes a credit check with a central registry). Finally, a record is kept of the approval in the Approval registry component.

5.4.3 Conversation course component

The component examples presented in sections 5.4.1 and 5.4.2 deal with topics that are linked directly to the domain layer. The Conversation course component in figure 5.10 is responsible for much of the flexibility of the application.

The Conversation course handles three requests: add, remove, or activate a topic. The add request adds a topic that has become relevant to the list of topics that is discussed during the conversation. Conversely, the remove request deletes a topic that is no longer relevant in the current situation. The activate request switches the focus of the conversation to a particular topic. The add topic and remove topic components are instantiations of a generic memory component. They are used to keep track of which topics are currently of interest.
5.5 Integrated CMT

The previous section showed models of individual components. Though CMT is good at modeling the dependencies between components in terms of input-output couplings and interfaces, it does not provide a model of a system. For that, it lacks the ability to describe the functionality of a component. The services that a component offers are only described on the syntax level. We propose to use the object behavior models of chapter 4 to describe the semantics of a component’s services. Furthermore, a CMT-model does not provide clues about instantiation of components, which is necessary for a complete system model. To request a service from a component, the requesting component must know from which component it can request the service. To make the system structure explicit, the components are embedded in the data model of chapter 3, in which components will be treated as objects.

5.5.1 Overview

Figure 5.11 shows the embedding of a component model in object behavior models and a structure model. It uses the concrete syntax for the object-oriented data models of chapter 3.

Each component is fully independent of other components, because all interactions with the outside world happen through the invocation of services. The behavior of each component is specified in a separate object behavior model, while the system structure is captured in a single data model. Within that data model, each component has its own cluster that shows how a composite consists of components. Because a component type can occur in multiple composite types, the data clusters of the individual components may overlap.
5.5.2 Structural aspects

The global data model is constructed by combining the local structure clusters of all component types into a single global structure model.

A component’s structure cluster models its part-whole structure. It is centered around an object that represents the whole, which gives the component a way of referring to itself. This object is the equivalent of a self attribute in object-oriented programming languages, and consistent with the view that an object knows its own identity. The component’s parts are connected to the object that represents the whole using symmetric associations. If desirable, inheritance associations may be used to indicate similarities between component types.

The global data model is built by fitting the local clusters together. The set of classes of the global data model is the union of the classes that appear in all the local clusters. This ensures that a class appears only once in the global model. The associations are inferred from the associations in the local clusters: two classes are associated in the global model if and only if there is a cluster that associated those two classes.

The input-output connections between components as found in CMT are typed. There is no restriction on the types that can be used. Hence, it is quite possible that the type of a
CMT connection does not occur in a component’s cluster. The use of a global data model ensures that a component can use any type of parameter without running the risk of using an undefined type.

### 5.5.3 Behavior

The functionality of components is modeled using object behavior models. Obviously, the analyst has a large degree of freedom in specifying how the composite uses its components to realize its functionality. However, the CMT-models already suggest a number of natural translations to OBM’s. For example, concatenation of components is translated in a natural way into a communication association, as is shown in example 5.5.1. This is a good example of adherence to the Model Integration Principle.

#### Example 5.5.1

Figure 5.12 shows how a connection between two components finds its way into the OBM’s of both their representing classes. Composite C is linked to its single component P through a connection that “transports” an e. The OBM of C shows its initialization (it receives the e), followed by the request of a service comp with parameter e. P starts by receiving that request, performs a calculation, and goes back to waiting for another comp request.

The single init can be replaced by a choice between all the available services if a component provides more services. Section 5.6.1 shows an example of this situation.

A CMT aggregate corresponds to a decision in an OBM. An aggregation specifies for each of the aggregated components under which condition it will be executed. In its extended form, the decision construct of chapter 4 can deal with aggregates of more than two components. This is shown in figure 5.13, which translates part of the stopwatch example of figure 5.3 to an OBM.

The instantiation mechanism of CMT operates in two ways. It fixes (“instantiates”) the value of an input parameter, or it fixes the request type. Both cases can be handled using
Component Modeling

Ch. 5

Figure 5.13: Incorporating aggregation in an OBM

Figure 5.14: Using inheritance to model CMT’s instantiation mechanism

Inheritance, as is shown in figure 5.14. Instantiating an input parameter can be dealt with by creating a subtype of the general component, and overriding the services that use the instantiated parameter with services that accept the other (non-fixed) parameters. The replacement service calls the original service, passing the correct value for the fixed parameter along with the parameters that were received in the normal way. In figure 5.14, the init(x,y) service of AddingInteger is overruled in Inc5Integer by an Init(x) service that fixes one parameter to the value 5, creating a subtype of the AddingInteger component that adds 5 to its input parameter.

Instantiation of the request type can be captured by creating a subtype that offers an init service, which invokes the fixed service of the supertype. In figure 5.14, this construction is used to create an AddingInteger component that is based on a general Integer component.
5.6 The Advice Application revisited

This section applies the integrated component modeling strategy to the AA application, illustrating the workings of CMT’s composition mechanisms. The global data model will not be constructed, because this section only discusses selected components of AA.

5.6.1 Conversation course component

The Conversation course component consists of five sub-components, as can be seen in figure 5.10. Figure 5.15 shows the resulting data structure, which connects the Conversation course with Add topic, Remove topic, Salary conversion, Obtain approval, and Credit check. The associations in this example have not been named. The figure also used the inheritance association to indicate the similarities between a general Topic and its specializations. The inheritance associations with Memory Manager as target follow from the fact that the Add topic and Remove topic components are CMT-instantiations of a generic Memory Manager: the Add topic can only add topics to the memory, whereas the Remove topic component is only able to remove topics from the memory.

The OBM of the Conversation course component in figure 5.16 shows that a Conversation course handles three types of requests. It can add, remove, or activate a topic. If a Topic is to be added or removed from the list, the Add topic or Remove topic component is activated by sending it an init signal, after which the Conversation course returns to its initial state. Activation of a topic results in deciding which topic must be activated, followed by sending the appropriate topic the act (short for activation) signal.
5.6.2 Salary conversion

The data cluster of figure 5.17 shows that the Salary conversion component uses three topics to realize its functionality. Figure 5.18 contains the corresponding OBM. It shows that the raw data is passed to the MonthToYear and the NetToGross components. The values that
are calculated by these components are stored in the Income component.

![Diagram](image)

Figure 5.18: Behavior of the Salary conversion component

### 5.7 Conclusion

This chapter has applied modeling concepts for the recognition of software components in existing applications or the recognition of coherent functionalities in organisations. These concepts have been applied to a complex real-life case: a front-office support system.

The model of the components in the AA application shows two types of components:

- **Process intensive components**, which are characterised by the fact that their input parameters and their output parameters have the same type, grosso modo. An example is the component which performs several calculations on salaries.

- **Data intensive components**, which realise the transformation of data types of their constituent components. Usually, these components employ some kind of memory management. Examples are the customer and the income component. Most of these components allow a user to navigate the objects in the domain layer, as the types of their input parameters relate to other objects than the types of their output parameters (see figure 5.10).

The use of the CMT concepts demonstrates that both large-scale and small-scale components may be modelled as black boxes. Salary, being an abstract data type, is an example of a generic component with a coarse granularity, whereas the Obtain approval component has a fine granularity. Naturally, most of the black boxes may be further decomposed, dependent of the designer’s needs to have more insight or the designer’s wish to abstract from implementation details. CMT provides easy scaling to enable the analyst to model at the level of detail that is desired at a particular moment.
CMT seems to be well suited to a responsibility-driven approach [WWW90]. The CRC-cards that are used in the responsibility-driven approach capture information that is similar to the information that is captured in CMT’s component models. Components can be refined to reflect refinements of the system’s responsibilities. CMT is a suitable technique to record responsibilities in a more formal way.

The composition mechanisms were necessary to show the application structure. Aggregation and concatenation are used in the specification of almost every component. Although it is not directly visible, the instantiation mechanism has been widely used also. Each of the small-sized calculation components within the Salary conversion component is an instantiation of the Salary, as these components are the result of the instantiation of the request type to the addition or the multiplication operation. The Remove topic from list and the Add topic to list components within the Conversation course are good examples of instantiations of a topic memory, which in turn may be perceived to be an instantiation of the generic memory manager introduced in section 5.2.

It should be clear that CMT is a technique which describes the functionality of software components in a static way. It abstracts from the dynamic aspects in a fundamental way, which allows the analyst to concentrate fully on what the system should do without being hampered by how it must do so. As is inherent to abstraction, certain gaps are left. In the case of CMT, the most notable gaps are the lack of indicating which component has the initiative to trigger other components (explicit modeling of control flow).

It has been shown that the static and dynamic modeling techniques of chapters 3 (OO data modeling) and 4 (behavior modeling) can be used to fill in gaps left by CMT, by providing constructs that enable the modeling of an outline of a system architecture. Obviously, this approach will not yield a ready-to-implement model. However, it does provide a bridge to the later stages of system design, while retaining the advantages of a component oriented approach. The integrated technique abides by the Model Integration Principle, the Locality Principle, and the Rightsizing Principle. It supports scenario validation, and allows decoupling of services.

Further research is needed to establish a complete formal foundation for Integrated CMT.
Chapter 6

A Study in Expressiveness: PRAM

6.1 Introduction

This chapter presents an exercise in expressive power. It shows how the techniques that are described in this thesis can be used to model a Parallel Random-Access Machine (PRAM).

The PRAM is primarily an abstract programming model, not a machine to be built. However, it does exhibit the properties that characterize our application domains. Specifically, a PRAM relies extensively on distributed control and localized data storage. This is what distinguishes it from a Turing machine. The Turing machine does not have parallelism or distribution.

6.2 Description of PRAM

Following [Sav97], the parallel random-access machine (PRAM) consists of a bounded set of processors and a common memory containing a potentially unlimited number of words (see figure 6.1). Each processor is a random-access machine (RAM), whose CPU can access locations in both its local random-access memory and the common memory. It implements a fetch-and-execute cycle in which it alternately reads an instruction from a program stored in the local random-access memory and executes it. Instructions are read from and executed from consecutive memory locations, unless a jump instruction is encountered.

During each PRAM step, the RAMs execute the following steps in synchrony: they (a) read from the common memory, (b) perform a local computation, and (c) write to the common memory. Each RAM has its own program and program counter, as well as a unique identifying number that it can access to make processor-dependent decisions.

6.3 Structure model of a PRAM

This chapter models the PRAM using the following components:
A Study in Expressiveness: PRAM

Figure 6.2: Component model of a PRAM

1. Common memory
2. Local memory
3. Random Access Machine
4. Referee

The Referee is used to synchronize the substeps of the PRAM. Figure 6.2 shows a component model of the PRAM. The local and common memory components are instantiations of a
generic memory component. The common memory component fixes the identification on the value 0, whereas the local memory of RAM $i$ fixes the identification on $i$. This yields the structure model shown in figure 6.3.

![Figure 6.3: Structure of a PRAM](image)

### 6.4 RAM

The model in figure 6.4 shows that the first tasks of the RAM are initialization of the program counter (PC) and fetching of the first instruction. Fetching the next instruction and updating the program counter is abbreviated to a single *fetch* construct.

The three substeps of the PRAM influence the behavior of the individual RAMs heavily. The RAM starts with informing the *Referee* that it is ready to start reading from common memory by sending a *read* message. The model in figure 6.4 allows multiple reads. If the current instruction of the RAM (in the ins-register) is anything other than a read instruction, no operation is performed.

After reading from the common memory is complete, the RAM sends a *comp* message to *Referee* to indicate that it has finished reading, and waits for acknowledgement before commencing execution of the local computation. After the acknowledgement has been received, the RAM executes a sequence of instructions that do not access the common memory.

As soon as a read or write is encountered, the RAM sends a *write* message to the *Referee* to indicate that its computation is finished. The RAM can then perform a number of writes to the common memory, which completes a full step of the PRAM.

---

1. A PRAM that has the ability to perform an unlimited number of computations between successive accesses to the common memory, such as the one described in this chapter, is called an *ideal* PRAM.
As the goal of this section is to present a model of the structure and workings of the PRAM, we will not focus on the actual coding, decoding, and execution of the RAM operations such as add, multiply, load AC from local memory, store AC in global memory, conditional jump to a new program location, etc. It suffices to say that these instructions are rich enough to compute all computable functions [Sav97]. For the same reason, access to the local memory is not detailed, as it can be modeled analogously to accessing the common memory.

6.5 Memory

The memory component can be used both for the local storage space, and for the common storage space. Figure 6.5 shows that it provides access to stored data values through read and write services. The read service takes the address of a memory unit, and returns its value. The write service is used to set the value of a memory unit at a specific address. A
conflict can occur if multiple RAMs try to write to the same memory unit during the same PRAM step. The component presented in figure 6.5 conforms to the arbitrary model. It resolves conflicts between RAMs by allowing an arbitrary value to be written. In fact, all RAMs that try to write to the same memory unit succeed in doing so. However, only the last value is retained.

![Figure 6.5: Memory of the PRAM](image)

The creation of the memory units is skipped to concentrate attention on the essential features of the memory component. A memory component is identified by an ld, which is an integer. The local memory of RAM with number i is maintained by the memory component with ld=i. The memory with ld=0 implements the common storage area.

The CommExpr is used to construct a unique reference to a particular memory unit by combining the number of the RAM with the address. The read service retrieves the value that is stored at the address using the get service of the MemUnit, and returns it to the caller. Conversely, the write service calls the set service to assign a value to the MemUnit. The behavior model of the MemUnit can be found in figure 6.6.

### 6.6 Referee

The Referee models the synchronization of the substeps of the PRAM. Its behavior is shown in figure 6.7. First, each of the RAMs notifies the Referee that it is ready to read from the common memory. The synchronizer s1 ensures that no RAM can continue until all the
others have notified the referee as well. The first action of the referee after \( s_1 \) is to allow all RAMs to continue their program to read from the common memory.

The second substep follows the same pattern for the synchronization of the local computations. After a RAM had finished reading from the common memory, it notifies the Referee that it is ready to start its computation step. Synchronization of all RAMs is achieved by synchronizer \( s_2 \).

As soon as a RAM is finished with its computations, it notifies the Referee that it is ready to start writing to the common memory. Synchronizer \( s_3 \) suspends execution of the RAMs that want to start writing until all RAMs have finished their computation. Upon continuation, all RAMs have the opportunity to write to the common memory, which completes the PRAM step.

### 6.7 The Priority CRCW PRAM

It is possible to place restrictions on the access to the common memory during the first and third substeps of each PRAM step. If access to by more than one RAM to the same location is disallowed, access is exclusive. If this restriction does not apply, access is concurrent.

Four combinations of these classifications apply to reading and writing. The strongest restriction is placed on the Exclusive Read/Exclusive Write (EREW) PRAM, with successively weaker restrictions placed on the Concurrent Read/Exclusive Write (CREW) PRAM, the Exclusive Read/Concurrent Write (ERCW) PRAM, and the Concurrent Read/Concurrent Write (CRCW) PRAM. The PRAM described in this chapter is a CRCW PRAM.

When concurrent writing is allowed, conflicts can be resolved in one of the following ways:

1. The common model requires that all RAMs writing to the same location write the same value;
2. The *arbitrary* model allows an arbitrary value to be written;

3. The *priority* model writes the value that is written by the RAM with the lowest identification number into the common location.

It follows from the fact that any algorithm that is written for the common CRCW PRAM runs without changes on the arbitrary CRCW PRAM, and any program written for the arbitrary CRCW PRAM runs without changes on the priority CRCW PRAM, that the priority CRCW PRAM is the most powerful of the PRAM models. The memory component described in section 6.5 supports the *arbitrary* model.

To support the stronger *priority* scheme, the following changes must be made. The interface of *Memory* and *MemUnit* are extended to add the identifying number of the requesting RAM.
Figure 6.8: Memory unit that supports priority conflict resolution

to the write request. This enables the MemUnit to keep a record of the identification number of the RAM that requests the write service.

Furthermore, src and newVal attributes are added to MemUnit. The src attribute is used to keep track of the lowest number of the RAM that requested a write, while newVal holds the value it tries to write. Later writes are only performed if they are requested by a RAM with a lower number, as can be seen in figure 6.8. At the end of the write-to-common-memory substep of the PRAM, the Referee refreshes the common memory by requesting the refresh service of the common memory. The refresh sets the value of the src attribute of all memory units to a $N + 1$, by requesting refresh from all memory units. This ensures that the write requests are prioritized during each writing substep of the PRAM.

6.8 Conclusion

The exercise that is executed in this chapter shows that the techniques that are presented in this thesis can be used to describe a Parallel Random-Access Machine. Not only does the PRAM have sufficient expressive power to compute all computable functions, it also uses distributed control and localized data. Hence, our techniques possess sufficient expressive power.

The concepts that are used to model the PRAM are applied in a natural way. This indicates that the techniques presented in this thesis are indeed suitable for application domains that exhibit distributed control and localized data.

Furthermore, it has been shown that a straightforward model of the PRAM can be extended
to the strongest form of the PRAM, the Priority CRCW PRAM. This extension can be made without major changes to existing components.
Chapter 7

OO in Practice

7.1 Introduction

This chapter presents two applications of an object-oriented approach to two real-life situations. Both cases can be characterized as utilizing distributed control, local data, and multi-faceted presentation of data.

Section 7.2, which is based on [HO95a, HO95b, HO95c], describes the development of an object-oriented modeling technique that is tailored to the specific needs of the Dutch Ministry of Transport, Public Works and Water Management. It focuses on communication and interoperability between traffic systems, and shows how the more general modeling techniques of chapters 3 and 4 can be used as a basis for a special-purpose modeling technique. Suitability and comprehensibility are critical issues, which precludes the use of off-the-shelf techniques. The aggregation models show a kinship with the component structures presented in chapter 5. Development of the technique required approximately three man-months.

Section 7.3, based on [HVW97], is concerned with the development of an application structure for the management of scientific information for NIWI, an institute of the Royal Netherlands Academy of Arts and Sciences. The case focuses on the integration of systems and reuse of existing systems. It is fair to say that it treats systems as (very) large components. For instance, the database that contains data about relations of NIWI is a good example of a component that offers data to most of NIWI’s systems in different formats. The size of this project is approximately 8 man-months.

Conclusions are presented at the end of both cases. In addition, section 7.4 contains general observations.

7.2 A customized technique to model traffic systems

The case in this section was performed by the Software Engineering Research Centre for the Directorate-General for Public Works and Water Management Transport Research Centre
(AVV), of the Dutch Ministry of Transport, Public Works and Water Management. In short, the goal was to develop

Rules and guidelines for
the global description and specification of
the cooperation between and/or the integration of and/or the subdivision of
the current and future traffic systems (TSs)
from the perspective of a user/tenderer
to support an integral (migration) planning
in a form that is readable for non-IT specialists and
can be produced and maintained by third parties.

At the start of the project, the IT-situation could be described as unconnected islands of automation. Most traffic systems use their own, independent infrastructure (hardware, data communication, and data storage). The TSs are also more or less independently developed on a functional level. Very few TSs are conceived with the idea of open systems in mind.

In order to facilitate the communication between TSs, open systems need to be developed. This enables the gradual integration of independent systems into a coherent system. The ultimate goal is to reuse existing components to build new systems.

7.2.1 Project goal

The aim is to develop a customized modeling technique that caters for the specific needs of the Department of Transportation which can be used to describe the relations between TSs. First priority is the ability to describe the current situation. In this setting, a TS is considered to be a black box. The attention is focused on the interaction between TSs. At a later time, the technique must facilitate describing the division of TSs into components, stimulating reuse.

The models of the technique must be easily comprehensible, not only for experts, but also for traffic experts without specific IT-training. The technique should possess sufficient expressive power, and must support modeling on a conceptual level.

The most important objective of the technique is to provide descriptions of the interaction between TSs. It is important to be able to relate TSs in a single model for a thorough understanding of the relations and dependencies between TSs. TSs are related to each other on the basis of the data that a TS makes available to the outside world. In the context of this case, there will be no distinction between providing services and providing data, because both imply that one TS performs a task for another TS.

To describe the communication between two systems, it is sufficient to describe their interfaces and the protocol they use to exchange information. This implies that the internal workings of TSs are not central to the technique.
7.2.2 Customized technique

The models that comprise the customized technique reflect the intended target environment. Static relations between TSs are simple, and there is very little need to support complex constraints. Therefore, a simplified version of the structure models of chapter 3 is used.

7.2.2.1 Philosophy

The models strive to avoid implementation details by only describing which services are offered by a TS, omitting how a service might be executed. Furthermore, they are limited to a description of which system requests which services from which other systems. This guarantees that the technique supports the Service Decoupling Principle.

Comprehensibility is enhanced by limiting the number of available concepts, while making sure that no concepts that are necessary to model TSs on a conceptual level are left out. Furthermore, each model type deals with just a single perspective on TSs, avoiding the confusion that is possible if different perspectives are mixed in a single model.

Notationwise, graphical techniques are used to capture information. Graphical techniques offer a number of advantages over text-oriented techniques [TP91]:

- Graphical models are inherently two-dimensional, whereas textual descriptions are one-dimensional. This provides graphical techniques with an extra degree of freedom that can be used for presentational purposes.

- Graphical models are better geared toward describing the hierarchical structure of complex systems, and offer a more natural representation for concurrency.

- It is easier to study graphical representations in a selective manner, concentrating on essentials and filtering out less relevant information. A textual description must be read in a linear way, which carries the risk of overwhelming the reader with unwanted details.

7.2.2.2 Model integration

The technique uses a number of models. Every model uses its own concepts, and views the big picture from a separate perspective. The overview model describes how the different system types cooperate by providing a bird's eye view of the problem domain. This global image is detailed further in the aggregation models and in the communication models. Figure 7.1 shows how the models interlock to provide a complete description of the problem domain.

The composition of the systems in the overview model is described in the aggregation models. An aggregation model shows of which components a TS consists. The communication between systems is described in the communication models.
7.2.2.3 Overview model

The overview model is derived from the structure model of chapter 3. Its aim is to show the structure of the cooperation between TSs, independent of the actual implementation. To improve comprehensibility, two versions of the overview model are used:

1. global overview model
2. detailed overview model

The global overview model describes which systems communicate with which other systems, without attempting to classify that communication. An association between two systems in the global overview model means nothing more or less than that those two systems exchange information.

The detailed overview model provides an additional level of detail by distinguishing four types of communication:

1. Sending of control information
2. Sending of general data
3. Requesting of status information
4. Requesting of general data

In the first two cases, the sender has the initiative to initiate communication. In the last two cases, the receiver initiates communication. Initiative plays an important role in model integration. When integrating the communication models with behavior models, initiative provides the direction of communication. The component that has the initiative sends a message, whereas the component that does not have the initiative receives the message.
Notationwise, every type of communication uses its own type of arrow. The direction of the arrow indicates the direction of the flow of information. Initiative is shown by a dot. If the sender initiates communication, the dot appears at the source of the arrow. If the receiver initiates communication, the dot is drawn right behind the arrowhead.

![Diagram of a (simple) global overview model]

Figure 7.2: Example of a (simple) global overview model

Every arrow stands for one or more messages. A detailed description of these messages can be provided in the description of the individual systems. For every system is recorded which messages it can receive, what the parameters are, etc.

Figures 7.2 and 7.3 contain models of the following example:

1. **CTMS**\(^1\) switches **On-ramp control**\(^2\) on: **CTMS** has the initiative.
2. **CTMS** request a status report from **On-ramp control**: **CTMS** has the initiative.
3. **Traffic information** asks the **Fog detection** system for the current visibility figures. **Traffic information** has the initiative.
4. **Traffic jam detection** reports backed-up traffic to **Traffic information**: **Traffic jam detection** has the initiative.

### 7.2.2.4 Aggregation models

The aggregation models are used to depict the internal structure of systems: they “open” the black boxes. Aggregated systems are systems that consist of components. Figure 7.4 contains an example of an aggregation structure.

The arrows have a component as source and the composite as destination. They model a *consists-of* association, which is equivalent to a symmetric association for our purposes. From a dynamic point of view however, the consists-of association implies that all components can be addressed through the composite. The interface of the composite consist of

---

\(^1\) Central Traffic Management System  
\(^2\) On-ramp control systems vary the intensity of the stream of cars entering a highway, depending upon traffic conditions on the highway.
Figure 7.3: The detailed overview model of 7.2

Figure 7.4: Aggregation model of a fog detection system

The union of the interfaces of all components. A cardinality constraint can be associated with each consists-of association, indicating how many components make up the composite. The fog detection system in the example in figure 7.4 uses (among others) 1 to 4 visibility measuring devices.

Figure 7.5: Aggregation model of Traffic jam detection before promotion of matrix sign

The aggregation models can be integrated in the overview model because the technique adheres to the Model Integration Principle. Such an integrated presentation can be beneficial when integrating previously independent systems.
Sec. 7.2  A customized technique to model traffic systems

The communication between components of one system is not modeled. Their interfaces become relevant only if other systems start using them. In that case it is advisable to move the reused component to the overview model. This action is called promotion. After the promotion it immediately becomes clear which systems communicate with the “new” system, and if there are potential conflicts. A conflict might arise if multiple systems try to control the same system by sending it control information. Situations like this one can be detected automatically by indicating which TS appears as the destination of multiple control arrows.

Example 7.2.1

Consider the fog detection system of figure 7.4 which uses a warning sign. The reuse of the Warning sign (a matrix sign) of the traffic jam detection system (figure 7.5) yields the overview model of figure 7.6 and the adapted aggregation models of figure 7.7. The disappearance of the warning sign and the matrix sign in the two aggregation models is a clear signal that these display units are no longer exclusively part of a single system, but now operate as self-contained systems.

7.2.2.5 Communication model

The communication model is a further development of the interaction diagrams that were briefly discussed in chapters 1 and 2. It describes the types of communication that take place between systems, and which services are requested.

A communication model shows the interactions between a single system and its outside world. That outside world consists of all the systems that are connected with it in the overview model. Communication models are used to model cohesive scenarios. Examples of scenarios are:

1. Being started up by another system.

2. Performing a service, e.g. selecting an on-ramp control strategy, or displaying a text on a DRIP (Dynamic Route Information Panel).
Figure 7.7: Aggregation models after promotion of matrix sign

3. Coordinating a number of systems.

4. Shutting down a system.

The modeling of scenarios provides immediate feedback, and yields conformance to the Scenario Validation Principle.

The time axis of the communication models is vertical. The arrows indicate which message is sent to which system. Sending a message to a system is equivalent to requesting that the receiving system performs a service for the sending system. The arrow notation is identical to the one used in the detailed overview model.

**Example 7.2.2**

*Figure 7.8 shows an example of a scenario. It models the reaction of the CTMS to a beginning traffic jam. First of all, the CTMS verifies that the traffic jam detection system is not malfunctioning. It then asks the traffic jam detection system if traffic is gridlocked. Because that is the case, the On-ramp control system is requested to reduce the intensity of entering traffic, and switch to a suitable control strategy. Furthermore, the location and length of the traffic jam is displayed on a DRIP.*

**7.2.3 Guidelines**

To use the technique to full effect, attention must be paid to completeness and consistency. The remainder of this section presents a number of informal guidelines. Most of these guidelines can be formalized by expressing them using constraints,
Sec. 7.2

A customized technique to model traffic systems

Traffic jam On-ramp
detection CTMS control DRIP

status............................> •

traffic jam?

time status •  -C...................................

lower intensity strategy

status

traffic jam

Figure 7.8: Communication example: reaction to a beginning traffic jam

7.2.3.1 Completeness

The following are guidelines to obtain a complete specification:

1. All non-isolated systems in the overview model should be the focus of at least one communication model.

2. Every association in the overview model must have at least one counterpart in a communication model.

3. The set of all communication models that focus on a particular object should describe all communication that can be initiated by that object.

7.2.3.2 Consistency

1. If two systems are connected to each other in a communication model, then this must be reflected in the overview model:

   • The two systems must be connected in the global overview model.
   • A connection of the same type must appear in the detailed overview model.

2. A system that appears as a component in an aggregation model may not occur in the overview model.

3. No system present in the overview model can occur as a component in an aggregation model.

Legend:

Send data
Send control information
Request data
Request status information
4. If two systems are connected in the global overview model, then they must be connected by at least one arrow in the detailed overview model.

### 7.2.3.3 Comprehensibility

The use of a global overview model stimulates the analyst to avoid obscure drawing conventions. Drawing conventions can simplify the appearance of models, but requires that the users understand all the conventions. In the context of this project, drawing conventions would probably place too high a burden on the average user.

Every communication model is drawn with a focus on a single system. This makes it clear which system is the current centre of attention.

### 7.2.4 Way of working

The way of working depends strongly on the experience, personal tastes and preferences of the analyst. Hence, the way of working that is presented in the remainder of this section cannot be more than a starting point.

The approach is an iterative one. First a global picture is formed of the problem domain, which is detailed in a number of passes.

1. The goal of the first step is to arrive at a demarcation of the problem domain, preferably using an existing global overview model. If no global overview model exists, one should be drawn up.

2. Detail the structure of relevant systems in the global overview model using aggregation models.

3. Describe the interfaces of the systems that communicate with each other.

4. Model communication using communication models.

5. Use the communication models to provide additional detail to the overall structure described in the global overview model by making the detailed overview model. The detailed overview model can be used to verify correctness: does the control flow make sense, are there any conflicts between controlling systems, etc.

6. Repeat steps 1 through 5 until the design is finished.

### 7.2.5 Conclusion

The critical success factor for the specialized technique for the Ministry of Transport is comprehensibility. Easy comprehensibility was achieved by stripping undesired modeling concepts from common object-oriented techniques. This shows that, at least in this case, object-orientation delivers on its promise of reducing the gap between a problem domain and its models.
In general, the Ministry of Transport uses a traditional approach to system modeling. In practice, this approach has been unsatisfactory for describing interoperability, because it is hard to get an abstract overview of a collection of systems.

In contrast to the tradition approach, object-oriented approach benefits by adhering to the Locality Principle, which allows for a separation of concerns. This makes it easier to judge whether or not existing components can be reused.

Furthermore, by abstracting away from the actual implementation of traffic systems, support for the Service Decoupling Principle was achieved. This facilitates a more component-oriented way of working.

Experience has shown the technique’s worth in communication with the various parties that are involved, such as traffic experts and traffic system builders.

7.3 A system architecture for scientific information

NIWI, the Netherlands Institute for Scientific Information Services, is the result of a merger between six institutes that operate under the Royal Netherlands Academy of Arts and Sciences (KNAW).

The material in this chapter is based on [HVW97], a report that analyses the IT requirements of NIWI to facilitate the migration to a new IT structure. It focuses on:

- Integration of the information systems of the partners in the merger; most of the existing information systems cannot be used as a basis for the new system. Many of the existing systems were not designed with interoperability in mind.
- Interfacing with some existing systems is necessary.
- The information systems are designed with electronic information delivery in mind. However, they must also support the current way of working.
- The information systems must be adaptable and robust. They must take future developments into account.

The analysis aims to develop:

- A definition of the level of ambition for the development of the new information system.
- An overview of possible off-the-shelf products and available services.
- IT strategy considerations.
- An overview of possible scenarios to realize the new system.
- A recommendation for one of these scenarios.
This section limits discussion of the case to architectural issues.

The remainder of this section is structured as follows. Section 7.3.1 discusses the desired application architecture, which leads to the structure model of section 7.3.2. Section 7.3.3 discusses some dynamic aspects of the system. Using both structure and dynamics as input, the system is divided into a number of subsystems. This is done by grouping the objects into cohesive clusters, based on service requests.

### 7.3.1 Application architecture

NIWI can be examined from a number of perspectives. The most important ones are:

- **scientific discipline** e.g. biomedical sciences, Dutch language and literature, history, social sciences, ...

- **scientific information classification** e.g. journals, researchers, research activities, research results (“data sets”), books, institutes, ...

- **type of service** e.g. document delivery, subscription, project, clearing, intermediation, sale, lending, ...

- **communication medium** e.g. postal delivery, electronic mail, World Wide Web, FTP, fax, ...

- **type of relation** e.g. subscription holder, debtor, creditor, publisher, research information supplier, ...

A future-oriented system is capable of dealing with the addition of new disciplines, new products, new services, and new relation types.

To realize this, the NIWI application architecture builds the primary and supporting processes on top of a number of *core data collections* (CDC), as shown in figure 7.9. The CDCs represent a number of open databases, which allow different subsystems to store and retrieve information that is relevant to them. The three CDCs are:

1. Relations and Addresses
2. The collection of scientific information
3. Products and Services

These CDCs are described in section 7.3.1.1

The CDCs support three different process types within NIWI. They are:

1. Customer-oriented processes
2. Supplier-oriented processes

3. Supporting processes (not shown in figure 7.9)

Customer processes are described in section 7.3.1.2. Section 7.3.1.3 describes the acquisition processes.

7.3.1.1 Core data collections

The organization of NIWI is itself distributed over a number of locations. It is therefore important that the new system deals with this distribution by providing access to the CDCs to all locations. It is important for NIWI that the new system is able to present the information in the CDCs in different ways, according to the needs of the application that is requesting the information.

7.3.1.1.1 Relations and addresses

In short, relation is a term that indicates any person or organisation NIWI has (or intends to have) a commercial contact with. Dependencies between relations can be modeled using connections. A connection specifies how one relation relates to another. For example, a connection can be used to indicate that an organisation (a relation) is subdivided into a number of organisational units (possibly also relations).

A relation can be associated with different addresses, e.g. postal, visiting, or accounting. There must be a way to collect relations into groups, such as subscription holders, regular customers, creditors, or suppliers.
7.3.1.1.2 Scientific information collection

This CDC contains the information that is particular to the primary product of NIWI: scientific information. NIWI provides scientific information in different shapes and using different media. Scientific information is in part owned by NIWI: NIWI has an extended collection of books and periodicals. Furthermore, NIWI provides information about scientific information, (a) in the shape of catalogs which describe where books or journals can be found, and (b) by providing scientific information with a classification. The complex structure of this CDC is not detailed in this chapter.

7.3.1.1.3 Products and services catalogue

NIWI offers a large variety of services based on its large base of scientific information. The Products and Services catalog provides a market-oriented translation of the possible services in terms of prices, conditions, and options. It must be known of a service in which stage of processing it is, when its estimated time of completion is, to which type of scientific information it can be applied, etc. Section 7.3.2.1 details the structure of this CDC.

7.3.1.2 Customer processes

The task of customer processes is to handle service requests from clients of NIWI. These consist typically of requesting, delivery, reminding, and paying. Not all customer processes consist of all four subprocesses, e.g. because some services are performed free of charge. Furthermore, it is possible that a subprocess has different interpretations in different services. For instance, certain services require cash payment, while others may use credit. Furthermore, services may depend upon one another: the owner of a library card can borrow books without paying. Another example is the research database, which can be browsed at will by customers who (or whose organisation) has a subscription to its services.

7.3.1.3 Acquisition and maintenance processes

The objective of acquisition and maintenance is to acquire new information, classify it, and create new scientific information based upon existing information. Once acquired or created, the scientific information is made available to customers using a variety of services.

7.3.2 Structure model

The complete object model consists of approximately 100 classes. Therefore, the model is described in a number of consistent clusters:

- Relation and address
- Contact and address
- Finance
- Delivery
7.3.2.1 Delivery cluster

The delivery cluster of figure 7.10 describes the structure of the objects that are relevant for customer processes. A customer process is a process within NIWI that is executed to deal with a request that was made by a customer.

![Figure 7.10: Structure of the Delivery cluster](image)

At least the following information must be stored for a request:

- The relations that are involved in the request as requesting parties. It is possible that multiple relations act in different capacities in one request. E.g., one relation requests
a service, which is delivered to a second relation, while a third party foots the bill. In this case, all three relations are involved as requesting party.

- The relations that provide the service. Sometimes a request is rerouted to another information provider if NIWI does not have the resources to provide the service. The alternative supplier receives the request, and takes care of it. Another case in which a supplier other than NIWI occurs is in the mediation service, where NIWI acts as an intermediary between a customer and a supplier.

- The service is recorded. It provides a short description of the type of service that is requested. Examples are:
  
  1. document delivery
  2. taking out a subscription
  3. sales of a NIWI-product
  4. borrowing a book
  5. ...

The price and conditions of delivery are recorded for every service. Determining the price of certain services can be complicated, e.g. when a reduced price is used that depends on the type of customer (which is classified using the Relationgroup) and the volume of the information that is delivered. For example, the price of the service “copy journal paper” is $0.065 per page, if the number of copied pages lies in the range 1000–5000, the customer is a “large pharmaceutical company”, and delivery is by express-mail.

- The type of scientific information (SI-Item) that is requested.

### 7.3.2.2 Acquisitions cluster

The acquisitions cluster (see figure 7.11) contains the objects that pertain to the acquisition of SI-Items that NIWI supplies to its customers by means of services. The main focus of this cluster is on journal issues and books that are part of book series.

The central class is delivery request. A delivery request is made by a relation of NIWI. Delivery requests can be submitted by employees of NIWI, but may also be submitted by third parties, for whom NIWI performs acquisition.

The delivery request is submitted to a relation of NIWI. This relation acts as the supplier of the requested information. The delivery request pertains to a subscription or to a SI-Item directly. A subscription usually pertains to a journal or a book series. Aside from those, in principle it is possible to request delivery of any type of SI-Item, such as monographs, or files containing statistical data.
The result of requesting a delivery is the reception of a number of deliveries. Multiple bills can be associated with each delivery. It is also possible to associate a bill directly to a delivery request, e.g. when ordering a subscription\(^3\).

It is possible to record information that can be used to control the acquisition process, such as status of a request and expected date of delivery. This facilitates claiming of publications that have not been received.

Certain **SI-Items** are acquired through exchange agreements. For this to work, it is recorded which **SI-Items** (usually books and journals) are used for exchange purposes. For these items it is recorded which **relation** receives them.

### 7.3.3 Dynamics

This section aims to provide an illustration of the cooperation between some of the more important subsystems of NIWI. The scenarios of this section have been selected because they are

- crucial to NIWI

\(^3\)Subscriptions must be paid up front. Deliveries start after the money has been received by the publisher.
• complex, in the sense that they cross the boundaries between subsystems
• provide examples of distributed control

The remainder of this section discusses how cooperating subsystems handle requests for document delivery, how new scientific information is acquired, and presents an object behavior model for the document delivery subsystem.

7.3.3.1 Document delivery

The scenario in figure 7.12 uses a specialized document delivery subsystem to control the document delivery process. A specialized system is needed because NIWI handles approximately 1500 document delivery requests per day. The document delivery system uses the relation database for address information and for tracking of request statuses, the financial subsystem for billing, and the collection database for availability and location information.

The scenario can be divided into two large parts. The first part receives the request for a document delivery, verifies that it is a valid request, and adds the correct document to a pick list. At regular times, an employee collects the pick lists (which puts the documents in a convenient order) and retrieves the documents from the storage area.

The second part of the scenario consists of the actual duplication and distribution process. It starts when the document is put on a scanner, which (a) reads which document delivery request it is about to process via a bar code, and (b) scans the relevant pages of the...
document. The scanned results are handed over (in this case) to a fax, which transmits them to the customer.

Along the way, the status of the request is updated in the relations database, and the request is billed using the financial system.

### 7.3.3.2 Acquisitions

The responsibility of the acquisitions system is to support the acquisition and classification of scientific information. NIWI has a budget of approximately $1,000,000 to maintain its collection, and receives about 3300 journal issues per month.

The scenario that is presented in this section focuses on publications. However, acquisition and classification of other types of scientific information resembles this scenario closely.

![Acquisition scenario](image)

Figure 7.13: Acquisition scenario

The scenario in figure 7.13 shows how a request to add a Sl-item is processed by the acquisitions control subsystem. First a check is made if the requested item is already part of the collection, or if it is already being ordered. If it is not, then the budget is checked, and the ordering process can proceed by entering the formal classification in the national central catalog (GGC). The order is then sent to a supplier, and the item is entered in NIWI’s collection with the remark that it is being ordered.

If all goes well, the right publication will be received, along with an invoice. The invoice is checked, and if it is correct, paid. The acquisitions control system then performs the flagging service. This service is used to send copies of newly arriving documents automatically.
to customers who have flagged that document. The acquisitions system handles this by checking the status of the publication in the collection database, and generates a document delivery request for the document delivery system.

### 7.3.3.3 Behavior of document delivery system

The behavior model of figure 7.14 elaborates on the interaction diagram of figure 7.12, which specifies how a document delivery request is handled.

![Behavior diagram of document delivery system](image)  

**Figure 7.14: Behavior of document delivery system**

First, the verification action verifies the syntax and the completeness of the request. This is followed by a check to see if the requester is blacklisted, e.g. because of a bad financial record. Requests by blacklisted persons are automatically refused. If the relations database
confirms that the requester is not blacklisted, a search request is handed to the collection database.

If the request Sl-item is not in the collection of NIWI, NIWI can act as an intermediary for the information requester. If a relation has indicated that it would like NIWI to mediate if it does not have a particular Sl-item, an alternative supplier is selected, and the request is redirected. This mediation scenario is subject of an interaction diagram of its own, which is not included in this section.

If the requested item is found, its location and the details of the request are added to the pick list. As a result, someone will collect the item from storage, scan it, and send it to the information requester. This is however beyond the scope of the document delivery system.

Throughout the document delivery process, the status of the request that is being processed is updated in the relations database to allow customers to see what is happening to their request.

7.3.4 Conclusion

The selection of parts of the NIWI case exhibit much interaction between subsystems. The modeling of the dynamic aspects of such interactions can be made easier by supporting the Service Decoupling Principle. Interaction diagrams are useful in this respect, whereas object behavior models are less suitable.

The hard part of using an object-oriented approach is usually finding the objects. This case successfully used a simple clustering algorithm to identify candidate objects by clustering data elements and services.

Object behavior models were not extensively used in this case, because of the emphasis on system architecture, and the resulting emphasis on structural and communication aspects. However, a behavior modeling technique of a more declarative nature would probably have been useful. Such a technique should be the subject of further research.

7.4 General conclusion

Practical experience has shown that adherence to the Model Integration Principle helps in two areas. It makes collections of models more comprehensible, because it is clear how models can be fit together to obtain a comprehensive overview of a system. Built-in support for Model Integration is also useful in safeguarding the correctness of models.

The way of thinking that underlies the object-oriented approach really helps to tackle problems in domains that feature distributed control, local data, and multi-faceting. OO’s natural inclination towards support for the Locality Principle allows the analyst to concentrate his attention on abstract, global issues without having to worry about consequences of changes on a local level.

Chapter 1 argued that cooperation between systems is becoming more and more important. This is certainly true for the case studies presented in this chapter. While modeling these
interoperability aspects, the use of scenarios and use cases has been valuable, proving the value of support for the Scenario Validation Principle in an empirical way.
Appendix A

Mathematical Notation

In this appendix, the mathematical notation used in this thesis, as far as it is non-standard, is explained briefly.

A partial function \( f \) from \( A \) to \( B \) is defined by \( f : A \rightharpoonup B \). Formally, \( f \subseteq A \times B \) such that

\[
(a, b) \in f \land (a, c) \in f \implies b = c
\]

This property allows one to write \( f(a) = b \) instead of \( (a, b) \in f \).

The following abbreviations are used as notations for the fact that a partial function \( f \) is defined (\( \downarrow \)) or undefined (\( \uparrow \)) for a value \( a \):

\[
f(a)\downarrow = \exists b \in B \ [f(a) = b] \\
f(a)\uparrow = \neg f(a)\downarrow
\]

The functions \( \text{dom} \) and \( \text{ran} \) are respectively used to denote the domain and range of a (partial) function:

\[
\text{dom}(f) = \{ a \in A \mid f(a)\downarrow \} \\
\text{ran}(f) = \{ b \in B \mid \exists a \in A \ [f(a) = b] \}
\]

To avoid the frequent use of tuple brackets \( (\_,\_)\), a shorthand for denoting functions is used. For example, the function \( f \), defined by \( f = \{(p, a), (q, b)\} \), is denoted as \( \{ p : a, q : b \} \).

If \( f \) is a partial function, then \( f \oplus \{ a : b \} \) is also a partial function defined by:

\[
f \oplus \{ a : b \} = \{(a, b)\} \cup \{(x, y) \in f \mid x \neq a \}
\]

The function \( f \oplus \{ a : b \} \) therefore behaves the same as function \( f \) except that its value in \( a \) is \( b \).

If \( f \) is a partial function, \( f \ominus \{ a \} \) is also a partial function that omits \( a \) from the domain of \( f \). Formally it is defined by:

\[
f \ominus \{ a \} = \{ (x, y) \in f \mid x \neq a \}
\]
To avoid having to use many parenthesis as a result of repetitive function applications, the function composition notation may be applied. The composition $f \circ g$ is defined by:

$$f \circ g(x) = f(g(x))$$

Naturally, it is required that $\text{ran}(g) \subseteq \text{dom}(f)$.

The $i$-th element of a tuple $P = \langle a_1, \ldots, a_i, \ldots, a_n \rangle$, i.e. $a_i$, can be found by projection:

$$P_i = \langle a_1, \ldots, a_i, \ldots, a_n \rangle_i = a_i$$

Two tuples $P$ and $Q$ of the same length $n$ can be used to define a mapping $P \mapsto Q$:

$$P \mapsto Q = \{ P_i : Q_i \mid 1 \leq i \leq n \}$$

If $R : A \to B$ a relation and $a \in A$, then the relational image ($\langle a \rangle$) of $a$ is defined as follows:

$$R(\langle a \rangle) = \{ b \in B \mid R(a, b) \}$$
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk 80: The language and its implementation.* Addison-Wesley, Reading, Massachusetts, 1983.


148


<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher</th>
<th>Location(s)</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Som89]</td>
<td>I. Sommerville</td>
<td><em>Software Engineering</em></td>
<td>Addison-Wesley</td>
<td>Reading, Massachusetts</td>
<td>1989</td>
</tr>
<tr>
<td>[Str91]</td>
<td>B. Stroustrup</td>
<td><em>The C++ Programming Language</em></td>
<td>Addison-Wesley</td>
<td>Reading, Massachusetts</td>
<td>1991</td>
</tr>
</tbody>
</table>
Index

C, 80
D, 82
E, 79, 80
Follows, 66, 67, 73
II, 65, 82
IM, 75
Tail, 72, 84, 85
clt, 27, 47
AA, 98, 107

Conversation course, 99, 100, 102, 107
scenario, 99
topic, 98–100, 102

abstraction, 7
adaptability, 20
Advice Application, 98
aggregation, 23, 49, 50, 59, 95, 96, 105, 121, 123, 125
association, 4, 5, 27, 29, 37, 47, 48
communication, 105
consists-of, 59
symmetric, 32, 36, 38, 39, 42, 43, 47–50, 59, 64, 104, 125
uni-directional, 40
example, 41, 42
attribute, 4, 5, 42, 49, 50, 59, 65

behavior model, 140
black box, 5, 94, 109, 122, 125
capsule, 96
CASE tool, 2, 9
category theory, 25, 27
centralized control, 8
Chain of Responsibility pattern, 89
class, 5, 29
classes, 6

CMT, 92, 103
cold turkey migration, 1
collector, 61, 86–88
example, 87
communication, 53, 61, 66, 73–76, 83, 94
asynchronous, 61, 64, 66, 78
example, 78
synchronous, 61, 74
example, 76
communication expression, 66, 74–76, 88
communication function, 69, 85, 87
component, 5, 91–93, 95, 103
components, 9
Composite pattern, 47
composition, 23, 49, 94, 95
example, 97
comprehensibility, 2, 3, 6, 12, 18, 20, 121–123, 130, 141
concatenation, 95, 96, 105
conceptual, 2
concurrency
interobject, 63
intraobject, 63
conditional processes, 80
conformity rule, 38
constraint, 19, 20, 27, 30, 34, 43, 47, 48
base, 30
cardinality, 30–32, 126
example, 30
extensional uniqueness, 33, 43, 47
example, 34
total role, 32
example, 32
uniqueness, 32, 39, 47, 49, 50
control, 7
control structure, 7

153
CRC card, 16
CRC-cards, 110
create object, 61, 73, 82
CSCW, 9
CY, 14

data structure, 7
  global, 9
  local, 9
decision, 61, 64, 66, 80, 105
  example, 81
denotation, 26
dependency, 35, 36, 49
dependency relation, 35
destroy, 82
destroy object, 61
distributed control, 8, 11, 13, 17, 24, 90,
  99, 111, 118, 121, 138, 141
domain, 1–4, 6, 7, 9–11
  properties, 7
dynamic semantics, 34
  example, 36
encapsulation, 4, 5, 19, 23, 41, 98
capsulation function, 85
capsulation operator, 70, 71, 85, 87
ER, 38
ERP, 9
expressive power, 14, 111, 118
expressiveness, 2, 6, 12
extensibility, 53, 57
faceting, 7, 10
feature, 27, 29, 47–49
  bi-directional dependency, 29
dependency, 28, 29, 35, 39, 41, 42, 47, 49, 50
double dependency, 29, 49, 50
function, 28, 29, 39, 47, 50
inherence, 29, 46, 50
no-knowledge, 28, 29, 42, 48, 50
not-null, 28, 29, 39, 47, 50
features, 27
foreign key, 38
formal foundation, 2, 6, 12
generalization, 44, 49
granularity, 14, 21–23, 92, 109
guarded commands, 67
HOOD, 18
identification, 26
  weak, 26
identity, 5, 37, 38, 46, 47, 73–75, 104
inheritance, 20, 28, 35, 44, 45, 48–50, 58,
  93, 96, 104
  monotonic, 58
  multiple, 46
  pure, 58
initial state, 61, 66, 67, 72
instance, 5, 15, 28, 34, 35, 40, 43–46, 48,
  71
instantiation, 95, 96, 105
interaction diagram, 14, 56, 59, 61, 140
  example, 14
interaction model, 6, 59–60, 97
interface, 4, 5
interoperability, 121, 131, 142
intraobject concurrency, 83
join, 31
  natural, 31
KISS, 23
localized data, 9, 11, 13, 24, 99, 111, 118,
  121, 141
maintainability, 20
MemUnit, 115, 117, 118
messages, 61
metamorphism, 44
method, 1–6
  data-oriented approach, 3
  formal, 3
  formal approach, 3
  natural language approach, 3
  object-oriented approach, 3
  process-oriented approach, 3
structured approach, 3, 6
methodology jungle, 2, 6, 11, 37
Model-View-Controller, see MVC
multi-faceting, 10, 11, 13, 16, 18, 24, 90, 99, 121, 141
multi-valued property, 43
MVC, 7, 9, 10
NIAM, 38, 44
nondeterministic behavior, 63
object, 5
  control, 7
  creation, 65
  data, 7
  destruction, 65
  entity, 7
  interface, 7
  state, 61, 62
  value, 5
object behavior model, 6, 15, 19, 56, 60–62, 65–67, 71–89
  example, 61
  semantics, 71
  syntax, 65
object creation, 67
object identifier, 26
object interaction diagram, 60
object model, 6, 25
Objectory, 3, 14, 47, 50–51, 53–62, 66
  acquaintance association, 58
  communication association, 59
  control object, 57
  entity object, 57
  instance association, 58
  interaction diagram, 61
  interface object, 57
objects, 4
OBM, see object behavior model
Observer pattern, 61
OID, 26, 27, 66, 67
OMT, 3, 14, 16, 19, 23, 37
OO, 4, 6
OOA, 3
OODA, 3
OOOSD, 16
OOSD, 16
operation, 4, 5
OSA, 16, 18, 23, 96
perform task, 61, 79
population, 27, 34, 35, 64
populations, 28
PRAM, 111–113, 115, 118
  classification, 116
  component model, 113
  ideal, 113
  memory, 114
  presentation, 7
previous state, 61, 62, 71, 73
principle
  Execution Resilience, 13, 17, 18, 53
  Locality, 19, 30, 92, 110, 131, 141
  Model Integration, 19, 60, 105, 110, 123, 126, 141
  Rightsizing, 21, 23, 92, 93, 110
  Scenario Validation, 13–15, 83, 110, 128, 142
  Service Decoupling, 16, 90, 92, 110, 123, 131, 141
Process Algebra, 15, 54, 65, 68–71, 83
PSM, 38
RAM, 111, 113, 115–118
  behavior model, 113
  component model, 111
RDD, 3, 15–17, 19, 23
receive, 61, 74, 78
receive message, 75
receive signal, 78
Referee, 112, 113, 115, 116, 118
  referential integrity, 38, 41–43, 82
reational image, 33
request, see service
Responsibility-Driven Design, see RDD
return from message, 61, 76
return from signal, 87
reuse, 6, 12, 53
rewrite operator, 71, 73, 79, 80
ex ample, 77
robustness, 53, 57, 131
scaleability, 21–23
scenario, 13, 14
send, 61, 74, 78
send message, 74
send signal, 78
service, 4, 6, 56, 59, 61, 66, 71, 74, 78, 86, 94, 103, 106
 provider, 66, 75, 86–88
service request, see service
set, 46
signals, 61
silver bullet, 3
Simula, 4
Smalltalk, 4
splitpoint, 84
 example, 84
state, 61, 72
state graph, 28
state operator, 70, 71, 74–76, 79, 80, 82, 83, 87
 example, 77
state transition diagram, 16, 61
STD, 16
 example, 16
structure model, 6
subtyping, 44
suitability, 3, 6, 7, 12, 13, 22, 121
surrogate, 26
synchronizer, 61, 84, 85, 88, 115, 116
 example, 86
system model, 6
system state, 64, 65, 71, 75, 76, 79, 80, 82, 83

technique, 2
techniques, 2
trace, 14, 15, 60, 83
 example, 83
Turing machine, 111
type graph, 27
example, 28
UML, 14, 16, 18, 19, 47–50
Unified Modeling Language, see UML
use case, 14, 18, 19, 55, 56, 60, 61, 142
 example, 55
validation, 2, 14, 15, 83, 90
value object, 43
value objects, 27, 59
values, 5
verification, 2, 90
workflow, 8, 9
Author Index

Apers, P.M.G. 6, 58, 145
Arnold, K. 16, 145
Arnold, P. 26, 146
Atkinson, M. 25, 145
Avison, D.E. 2–3, 145
Baeten, J.C.M. 54, 65, 146
Balsters, H. 6, 58, 145
Bancilhon, F. 6, 25, 145
Bapat, Subodh 21, 43, 58, 96, 145
Barros, A.P. 13, 146
Bassett, P.G. 92, 145
Benyon, D. 3, 146
Bodoff, S. 26, 146
Bommel, P. van 38, 146
Booch, G. 3, 14, 25, 48, 146
Brooks Jr, F.P. 3, 146
By, R.A de 6, 58, 145
Champeaux, D. de 4, 16, 25, 37, 40, 42–44, 74, 146
Chen, P.P. 38, 146
Christerson, M. 3, 7, 14, 16, 19, 25, 38, 40, 42–43, 53, 150
Coad, P. 3, 38, 146
Cockton, G. 7, 10–11, 147
Coleman, D. 26, 146
Cook, S. 43, 146
Dahl, O.J. 4, 147
Daniels, J. 43, 146
Davis, A.M. 2, 26, 147
DeWitt, D. 25, 145
Dedene, G. 54, 61, 147
Delatte, B. 18, 147
Delobel, C. 6, 145
Dijkstra, E.W. 67, 147
Dittrich, K. 25, 145
Dollin, C. 26, 146
Elmasri, R. 26, 147
Embley, D.W. 2–4, 6, 16, 25, 38, 40, 42–44, 54, 63, 83, 96, 147
Emde Boas, P. van 147
Faure, P. 4, 16, 25, 37, 40, 42–44, 74, 146
Fitzgerald, G. 2–3, 145
Frederiks, P.J.M. 15, 47, 147–148
Gamma, E. 8, 47, 61, 89, 147
Gilchrist, H. 26, 146
Glabbeek, R.J. van 68, 148
Goldberg, A. 4, 6–7, 147
Goldkuhl, G. 93, 150
Gosling, J. 16, 145
Gram, C. 7, 10–11, 147
Griethuysen, J.J. van 2, 147
Gyssens, M. 6, 27, 152
Halpin, T.A. 30, 38, 44, 148
Hammer, M. 46, 148
Hayes, F. 26, 146
Heitz, M. 18, 147
Helm, R. 8, 47, 61, 89, 147
Henderson-Sellers, B. 2, 148
Hubbers, J.W.G.M. 12, 15, 19, 42, 97, 121, 131, 148–149, 151
Hutt, Andrew T.F. 25, 149

Iivari, J. 149

Jackson, M. 7, 150
Jackson, M.A. 3, 149
Jackson, R.B. 6, 54, 63, 83, 147
Jacobson, I. 2–3, 7, 14, 16, 19, 25, 38, 40, 42–43, 48, 53, 146, 149–150
Jensen, K. 16, 150
Jeremaes, P. 26, 146
Johnson, E. 8, 47, 61, 89, 147
Jones, C.B. 3, 150
Jonge, W. de 26–27, 152
Jonsson, M. 3, 7, 14, 16, 19, 25, 38, 40, 42–43, 53, 150
Kanellakis, P. 6, 145
Kaschek, R. 28, 150
Kernighan, Brian W. 16, 150
Kilov, H. 25, 37–38, 42, 44, 150
Kolp, M. 49, 150
Krasner, G.E. 7, 150
Kristen, G. 3, 23, 150
Kurtz, B.D. 2–4, 16, 25, 38, 40, 42–44, 96, 147
Lang, N. 42, 151
Laundry, M. 3, 151
Law, D. 3, 150
Lea, D. 4, 16, 25, 37, 40, 42–44, 74, 146
Lippe, E. 25–27, 47, 148
Longworth, G. 3, 150
Lundeberg, M. 93, 150
Maier, D. 25, 145
Malouin, J.L. 3, 151
Mandrioli, D. 92, 151
Marca, D.A. 93, 151
Martin, J. 3, 150–151
McGowan, C.L. 93, 151
McLeod, D. 46, 148
Mellor, S.J. 25, 151
Meyer, B. 4, 16, 42, 53, 92, 150–151
Muller, J.F. 18, 147
Myhrhaug, B. 4, 147
Navathe, S.B. 26, 147
Nieuwland, E.R. 85, 148
Nilsson, A. 93, 150
Nygaard, K. 4, 147
Odell, J. 151
Oosterom, N.E. van 12, 97, 121, 148–149, 151
Overgaard, P. van 3, 7, 14, 16, 19, 25, 38, 40, 42–43, 53, 150
Pirotte, A. 49, 150
Pong, L. 123, 152
Pope, S.T. 7, 150
Proper, H.A. 13, 146
Ritchie, Dennis M. 16, 150
Robson, D. 4, 7, 147
Ross, J. 25, 37–38, 42, 44, 150
Rubin, K.S. 6, 147
Salus, P.H. 91, 151
Savage, John E. 12, 111, 114, 151
Shlaer, S. 25, 42, 151
Siebes, A. 27, 151
Skidmore, S. 3, 146
Snoeck, M. 54, 61, 147
Sol, H.G. 1, 3, 151
Sommerville, I. 1, 152
Spivey, J.M. 3, 152
Stroustrup, B. 4, 16, 152
Szyperski, C. 92, 152
Tse, T.H. 123, 152
Tuijn, C. 6, 27, 152
Vaandrager, F.W. 54, 71, 152
Verhoef, T.F. 12, 92, 121, 131, 149, 152
Vlissides, J. 8, 47, 61, 89, 147
Vreeze, C.C. de 6, 58, 145

Weide, Th.P. van der 15, 25–27, 38, 43–44, 92, 146–149, 152
Weijland, W.P. 54, 65, 68, 146, 148
Wiener, L. 3, 15, 37, 42, 44, 110, 152
Wieringa, R.J. 26–27, 54, 152
Wijers, G.M. 12, 121, 131, 149
Wilkerson, B. 3, 15, 37, 42, 44, 110, 152
Wirfs-Brock, R.J. 3, 15, 37, 42, 44, 110, 152
Wirth, N. 16, 150, 152
Woodfield, S.N. 2–4, 6, 16, 25, 38, 40, 42–44, 54, 63, 83, 96, 147
Yourdon, E. 3, 38, 146, 152
Zdonik, S.B. 25, 145
Samenvatting

De wereld van de Informatietechnologie (IT) is complex en dynamisch. Systemen moeten kunnen werken met veel verschillende soorten informatie, en moeten veel verschillende functies kunnen uitvoeren. Systemen in het bank- en verzekeringwezen bestaan uit duizenden programma’s. Het ontwerpen van een dergelijk systeem is een lastige taak. De dynamische aard van IT is het gevolg van de steeds veranderende applicatiedomeinen. Gebruikers vragen om steeds complexere applicaties om nieuwe problemen op te lossen, en willen steeds meer informatie.

Om het ontwerptraject te ondersteunen zijn methoden en technieken ontwikkeld. Het doel van deze methoden en technieken is het verhogen van de productiviteit van de ontwerpers en het verbeteren van de kwaliteit van de ontworpen systemen. Helaas is er in de loop van enige tientallen jaren een overvloed aan methoden en technieken geïntroduceerd. Dit heeft geleid tot een situatie die vaak wordt aangeduid met de term methodology jungle, of methoden-jungle.

Vanwege het bestaan van deze jungle is het van belang om over criteria te beschikken die behulpzaam zijn bij het beoordelen van de sterke en zwakke punten van methoden. Voor analysemethoden werden zulke criteria ontwikkeld in [Hof93]. Deze criteria beslaan de gebieden conceptualiteit, expressiviteit, formele onderbouwing, toegankelijkheid en geschiktheid voor het beoogde toepassingsgebied.

Een recente toevoeging aan de methodology jungle wordt gevormd door de verscheidenheid aan object-georiënteerde (OO) methoden. Hoewel OO misschien niet het definitieve antwoord is op de problemen die inherent zijn aan softwareontwikkeling, biedt OO wel een aantal gewenste eigenschappen die de kwaliteit van het software-ontwikkelproces kunnen verbeteren.

Dit proefschrift richt zich met name op het criterium geschiktheid. Hoofdstuk 1 introduceert een karakterisatie van toepassingsgebieden aan de hand van drie eigenschappen: besturingsstructuur (control structure), gegevensstructuur en gefaceteerdheid (faceting). De mogelijke besturingsstructuren variëren van gecentraliseerd tot gedistribueerd. In een applicatie met een gecentraliseerde besturingsstructuur is er één subsysteem (of object) dat het gedrag van het systeem als geheel bestuurt.

De gegevensstructuur geeft aan hoe de gegevens die binnen een applicatiedomein gebruikt worden geparitioneerd zijn. De uiterste waarden zijn een globale gegevensstructuur (er is
Samenvatting
geen partitionering) en vele lokale gegevensstructuren (er wordt veel gebruik gemaakt van
gescheen).
Gefacetteerdheid verwijst naar de manier waarop gegevens naar andere subsystemen gepre
senteerd worden. Een systeem dat zich onderscheidt door een hoge mate van gefacetteerd
heid bevat veel componenten die hun gegevens op veel verschillende manieren beschikbaar
stellen aan de buitenwereld. Een systeem dat geen gebruik maakt van gefacetteerdheid biedt
gegevens slechts op een enkele manier aan.
Dit proefschrift richt zich op toepassingsgebieden die gekarakteriseerd kunnen worden door
een veelheid aan faceten, een gedistribueerde besturingssystemen en lokale gegevensstruc
turen. Het doel is om orde in de OO methodenjungle te scheppen. Dit wordt gedaan door
een aantal OO analysemethoden die hun nut in de praktijk bewezen hebben formeel te
analyseren en uit te breiden.
Hoofdstuk 2 van dit proefschrift richt zich op de geschiktheid van een methode of techniek
voor het specifieke toepassingsgebied waar dit proefschrift zich tot beperkt. Het biedt een
aantal principes die gebruikt kunnen worden om methoden/technieken te beoordelen op
hun geschiktheid. De principes zijn:

**Scenario Validation principe** Een techniek moet een expliciete notie van scenario aan-
bieden om het valideren van modellen te ondersteunen.

**Service Decoupling principe** Een techniek moet de ontkoppeling van serviceverzoeken
en hun definitie ondersteunen.

**Execution Resilience principe** Een techniek moet het afhandelen van operationele fou-
ten ondersteunen om een veerkrachtige uitvoering van het conceptuele model te be-
werkstelligen.

**Model Integration principe** Een techniek moet de integratie van individuele modellen
ondersteunen door een expliciete afbeelding van individuele modellen naar een samen-
hangend ontwerp aan te geven.

**Locality principe** Een techniek moet het modelleren van eigenschappen op locaal niveau
actief ondersteunen.

**Rightsizing principe** Een techniek moet modelleerconcepten aanbieden die het model-
leren op verschillende schaalgröottes en op verschillende niveau's van granulariteit
faciliteren.

Hoewel niet geclaimd kan worden dat deze verzameling principes compleet is, kunnen ze
desondanks een belangrijke bijdrage leveren in de ontwikkeling en toepassing van object-
georiënteerde methoden en technieken. De principes kunnen gebruikt worden om de vo-
ordelen van verschillende modelleeraanpakken kwalitatief te beoordelen.
Hoofdstuk 3 richt zich op een van de kernmodellen van OO methoden: het structuur-
model (ook wel objectmodel of OO datamodel genoemd). Er is geen consensus over wat
een structuurmodel is. Soms staan concepten met dezelfde naam voor fundamenteel verschillende ideeën, terwijl in andere gevallen concepten met verschillende namen inhoudelijk sterk op elkaar lijken. Dit proefschrift analyseert de concepten die bestaande technieken aanbieden en komt op basis daarvan tot een klein aantal fundamentele begrippen (*features en *constraints). De concepten die door verschillende technieken aangeboden worden kunnen worden gekarakteriseerd door deze begrippen te combineren. De formele semantiek van de features en constraints is geïnspireerd door een op categorie-theorie gebaseerde aanpak.


Componenten zijn het onderwerp van hoofdstuk 5. Het vertrekpunt ligt bij de Component Modeling Technique (CMT). CMT heeft tot doel om componenten precies te kunnen beschrijven. Dit wordt gedaan door de *functionaliteit te modelleren, zonder aannames te doen over de manier waarop die functionaliteit geïmplementeerd gaat worden. CMT’s sterke punt is de flexibele manier waarop componenten samengesteld kunnen worden op basis van al bestaande componenten. Dit wordt geïllustreerd aan de hand van een praktische case, namelijk een front-office applicatie uit de bank- en verzekeringswereld. Om ook de *interne functionaliteit van een component te modelleren wordt CMT uitgebreid door integratie met structuurmodellen en gedragsmodellen. Ook deze uitbreidingen worden geïllustreerd aan de hand van de front-office applicatie.

De expressiviteit van de technieken die in dit proefschrift beschreven zijn wordt onderzocht in hoofdstuk 6. Dit wordt gedaan door middel van de Parallel Random Access Machine (PRAM), een abstract programmeermodel dat alle berekenbare functies aankan. De PRAM kan gekarakteriseerd worden als een applicatie die intensief gebruik maakt van locale data en een gedistribueerde besturingsstructuur. Er is gekozen voor het modelleren van een PRAM in plaats van een Turing-machine, omdat de PRAM beter bij het probleemgebied aansluit. De PRAM blijkt goed te modelleren met de gepresenteerde technieken. In eerste instantie wordt een eenvoudige variant gemodelleerd. Dit model kan echter makkelijk aangepast worden om de krachtigste vorm van de PRAM te modelleren.

De in dit proefschrift ontwikkelde technieken worden empirisch gevalideerd door ze toe te passen in twee praktische case studies (hoofdstuk 7). Beide cases gebruiken een gedistribueerde besturingsstructuur, locale data en meerde faceten. De eerste case, uitgevoerd in opdracht van het ministerie van verkeer en waterstaat, beschrijft een gespecialiseerde
ontwerptechniek die toegesneden is op het modelleren van de samenwerking tussen wegverkeerssystemen. De tweede praktijktoepassing is uitgevoerd in opdracht van het Nederlands instituut voor wetenschappelijke informatiediensten (NIWI). Het NIWI is het resultaat van een fusie tussen zes instituten van de Koninklijke Nederlandse academie van wetenschappen KNAW). Aandachtspunten bij deze migratie naar een nieuwe informatietechnologieinfrastructuur zijn de integratie van de bestaande systemen van de fusiepartners en de mogelijkheid gebruik te maken van elektronische informatielevering.

Op basis van de opgedane praktische ervaringen kan geconcludeerd worden dat het voldoen aan het Model Integration principe helpt op twee gebieden. Het maakt de verzameling modellen beter begrijpelijk, en bewaakt de consistentie tussen modellen. In hoofdstuk 1 wordt gesteld dat samenwerking tussen systemen steeds belangrijker wordt. Dit is zeker waar voor de cases in hoofdstuk 7. Het belang van het Scenario Validation principe is in de praktijk groot gebleken. Ook het Locality principe bleek in de praktijk grote waarde te hebben.
Curriculum Vitae
