Faculteit der Wiskunde en Informatica

# A Method for Building and Evaluating Formal Specifications of Object-Oriented Conceptual Models of Database Systems

R.J. Wieringa

Rapportnr. IR-340
ongewijzigde herdruk
januari 1995

*vrije* Universiteit *amsterdam*

# A Method for Building and Evaluating Formal Specifications of Object-Oriented Conceptual Models of Database Systems (MCM)[1]

R.J. Wieringa

Faculty of Mathematics and Computer Science
Free University
De Boelelaan 1081a
1081 HV Amsterdam
roelw@cs.vu.nl

December 15, 1993

© R.J. Wieringa

[1]This research is partially supported by Esprit Basic Research Action IS-CORE (working group 6701).

## Abstract

This report describes a method called MCM (Method for Conceptual Modeling) for building and evaluating formal specifications of object-oriented models of database system behavior. An important aim of MCM is to bridge the gap between formal specification and informal understanding. Building a MCM model is a process that moves from the informal to the formal, evaluating the model is a process that moves back from the formal to the informal.

First, a general framework for information system development methods is given, that is used to indicate which elements are needed to build a particular information system development method. In general, the following elements are needed (see figure 0.1)

1. **Requirements determination methods** that can be used to determine the information needs of the environment, and to find functional and nonfunctional requirements specifications.
2. **Conceptual modeling methods** that can be used to elaborate the statement of functional requirements into a formal specification of observable system behavior.
3. **Implementation methods** that can be used to transform the conceptual model specification into an implementation within the constraints indicated by the nonfunctional requirements.
4. **Project management methods** that can be used to manage the development process in the presence of limited resources and a potentially disturbing environment.

MCM is a conceptual modeling method, and must therefore in any information system development project be supplemented with three other kinds of methods.

MCM contains three kinds of methods (figure 0.1):

1. **Observation methods** to find relevant data about the required database system.
2. **Induction methods** that allow one to go from a finite set of data about required system behavior to a conceptual model that represents all of this behavior.
3. **Evaluation methods** that allow one to test the quality of a specification of a conceptual model.

In this report, I concentrate on induction and evaluation methods and merely make a list of relevant observation methods. The induction methods listed in figure 0.1 are not exhaustive. MCM can be viewed as a framework within which methods and techniques for conceptual modeling can be plugged. Some of these methods and techniques are mentioned in this report but not elaborated.

There are three kinds of evaluation methods, that deal with the *validity* of the conceptual model, the *utility* of the specified behavior, and the *quality* of the use that is made of the available modeling constructs. Prototyping and animation are briefly discussed as evaluation methods. The quality checks, however, are listed exhaustively.

The result of following MCM is a conceptual model. In the philosophy of MCM, a conceptual model consists of three components (see figure 0.2):

1. The **UoD model** is a model of the part of reality represented by the database system.
2. The **DBS model** represents DBS behavior, such as the queries to be asked from the DBS, the user interface, the contents and layout of reports produced by the DBS, etc.
3. A model of the **boundary** between the DBS and the UoD. This is a list of all possible transactions that the DBS can engage in, plus the function that this behavior has for the user of the DBS.

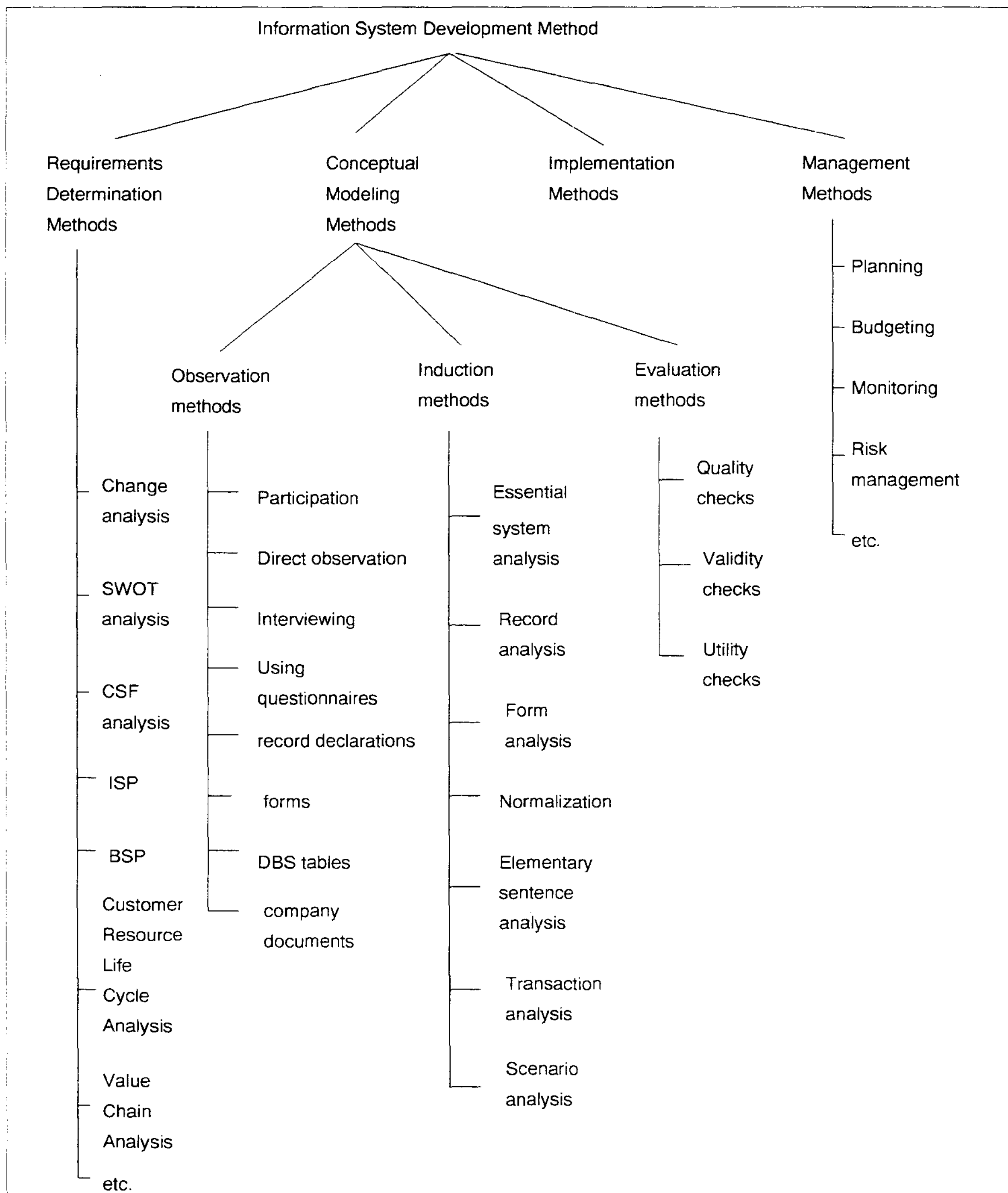



Figure 0.1: The components of an information system development method. MCM is a method for conceptual modeling only and must be supplemented with methods of the other three kinds.

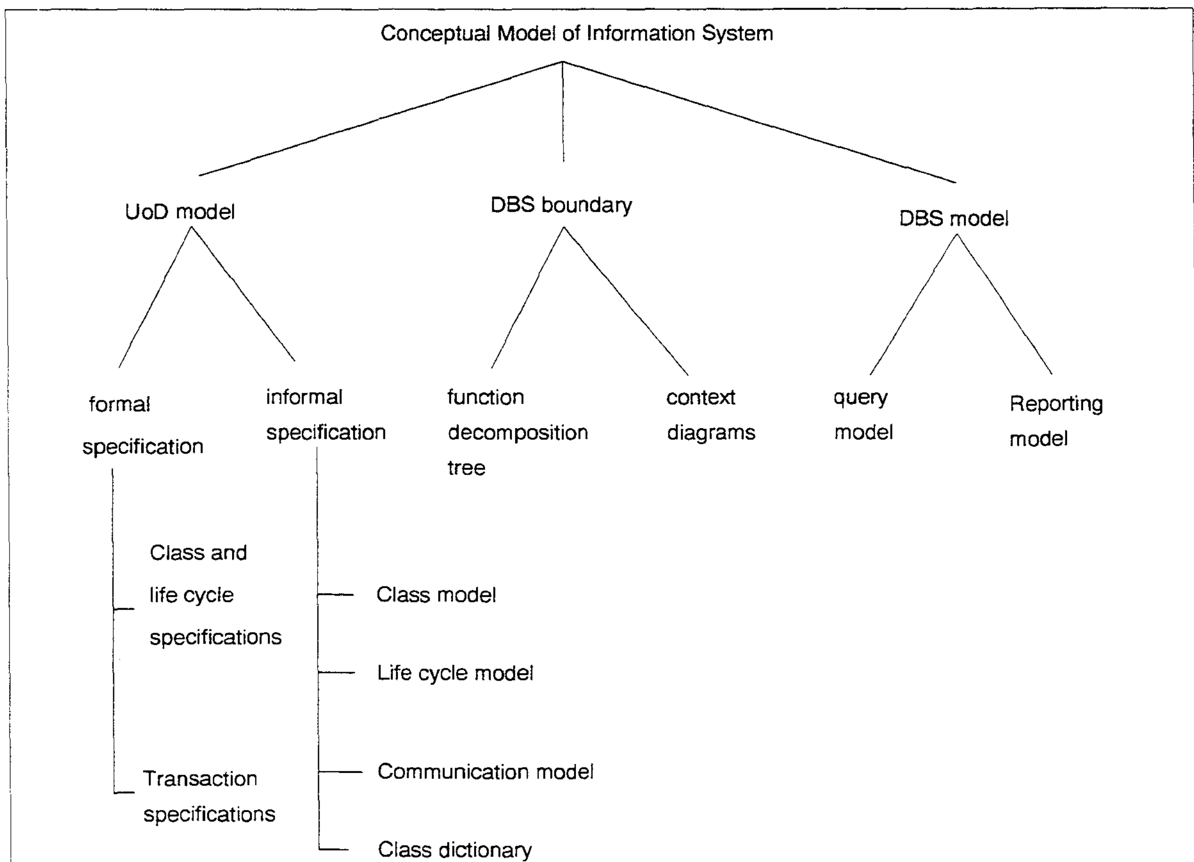



Figure 0.2: The components of a conceptual model according to MCM. The method can be used to find a model of the DBS boundary and of the UoD model. LCM can be used to write a formal specification of a UoD model.

MCM is concerned only with the models of the UoD and of the DBS boundary. These are decomposed as shown in figure 0.2.

Note that there is a formal and an informal part of the UoD model. The formal part is intended to be specified in a language called LCM (Language for Conceptual Modeling)[1], but the intention is that other languages could be used as well, such as Oblog, TROLL, TROLL-light or FOOPS. MCM is designed for use with LCM, but the dependence on LCM has been isolated in a single section of chapter 7 (The Structure of Specifications of UoD Models).

The methods and techniques described in this report are gathered from a large variety of sources; none of them are new, although they have not been integrated into a single method before. Sources for all method components are duly noted in this report. Two companion reports describe the syntax of LCM [31] and an application of MCM and LCM on a case study [134].

[1]The name "LCM" replaces "CMSL" (Conceptual Model Specification Language). The reason for the replacement is that "CMSL" is hard to pronounce, and this difficulty is mainly due to the "S" in "CMSL". Dropping the "S" gives "CML", which is the name of a language developed by Bob Wielinga at the University of Amsterdam. We thought it opportune to choose the name LCM instead. This has the consequence that MCM is the natural choice of a name for the method described in this report.

# Contents

**1 Introduction** **1**
1.1 MCM and formal specification . . . . . 1
1.2 Progress and reuse in development methods . . . . . 1
1.3 The contingency approach and method engineering . . . . . 2
1.4 Structure of the report . . . . . 3

**2 Information System Development** **5**
2.1 Three aspects of systems . . . . . 5
2.2 Rational choice . . . . . 5
2.3 The regulatory cycle . . . . . 6
2.4 Method engineering . . . . . 8
2.5 The structure of a requirements specification . . . . . 9

**3 Conceptual Modeling** **11**
3.1 The empirical cycle of conceptual modeling . . . . . 11
3.2 Observation methods . . . . . 13
3.3 Induction methods . . . . . 14
3.4 Evaluation methods . . . . . 17

**4 The Behavior of Data Processing Systems** **18**
4.1 Reactive systems . . . . . 18
4.2 Atomicity of transactions and events . . . . . 19
4.3 Transaction atomicity and interface technology . . . . . 20
4.4 A classification of DBS transactions . . . . . 21
4.4.1 Transactions about the UoD . . . . . 22
4.4.2 Transactions about the DBS . . . . . 23
4.5 The structure of information system models . . . . . 24
4.5.1 The UoD model . . . . . 24
4.5.2 The DBS model . . . . . 25
4.5.3 Comparison with JSD . . . . . 25

**5 Representing System Boundaries** **26**
5.1 The context diagram . . . . . 26
5.2 The function decomposition tree . . . . . 27
5.2.1 Function decomposition of organizations . . . . . 27
5.2.2 Function decomposition of DBSs . . . . . 30
5.2.3 Quality checks for function decomposition trees . . . . . 33
5.3 Combining function decomposition trees and context diagrams . . . . . 34

**6 The Structure of UoD Models** **35**
6.1 Objects . . . 36
6.1.1 Object identity and existence . . . 36
6.1.2 Objects and values . . . 37
6.1.3 Classes and types . . . 38
6.1.4 Attributes and predicates . . . 39
6.1.5 Identifiers, surrogates and keys . . . 40
6.1.6 Quality checks for objects and values . . . 43
6.2 Relationships . . . 44
6.2.1 Identification and existence . . . 44
6.2.2 Cardinality constraints and many-one relationships . . . 44
6.2.3 Quality checks for relationships versus objects . . . 48
6.3 Taxonomic structures . . . 49
6.3.1 Static subclasses and inheritance . . . 49
6.3.2 Dynamic subclasses and inheritance . . . 52
6.3.3 Role-playing and delegation . . . 53
6.3.4 Quality checks: Classification principles . . . 56
6.3.5 Quality checks: Static versus dynamic specializations . . . 57
6.3.6 Quality checks: Role playing versus specialization . . . 58
6.4 The instantiation relationship . . . 59
6.5 Events . . . 60
6.5.1 Requirements for events . . . 61
6.5.2 Inheritance and overloading . . . 62
6.5.3 Creation and deletion events . . . 63
6.5.4 Quality checks for events . . . 63
6.6 Life cycles . . . 63
6.6.1 Process operators . . . 63
6.6.2 Nondeterministic state and and bisimulation . . . 65
6.6.3 Dialogs and life cycles . . . 67
6.6.4 Life cycle inheritance . . . 70
6.6.5 Quality checks for life cycles . . . 70
6.7 Communication . . . 71
6.7.1 Transaction decomposition . . . 71
6.7.2 Quality checks for communications . . . 75
6.8 Integrity constraints . . . 75
6.8.1 DBS constraints and UoD constraints . . . 75
6.8.2 Necessary truths . . . 76
6.8.3 Treating UoD norms as DBS constraints . . . 77
6.8.4 Dynamic and static constraints . . . 78
6.8.5 Existence constraints . . . 78

**7 The Structure of Specifications of UoD models** **80**
7.1 Informal specification . . . 80
7.2 Formal specification in LCM . . . 85
7.3 Relations between different parts of the specification . . . 90

**8 An Induction Method for Finding UoD Models** **93**
8.1 Outline of an induction method . . . . . 93
8.2 Making a function decomposition tree . . . . . 94
8.3 Transaction analysis . . . . . 96
8.3.1 Object analysis of transactions . . . . . 96
8.3.2 Elementary sentence analysis of transactions . . . . . 99
8.3.3 Transaction decomposition . . . . . 99
8.4 Scenario analysis . . . . . 102
8.4.1 Narrative scenario analysis . . . . . 102
8.4.2 Event trace analysis . . . . . 105

**9 Evaluation Methods for DBS Models** **108**
9.1 Validation methods . . . . . 108
9.2 Utility checks . . . . . 109

**10 Summary and conclusions** **110**
10.1 Method engineering and contingency . . . . . 111
10.2 Network modeling . . . . . 111
10.3 Real time and deontic logic . . . . . 111
10.4 Historical DBS modeling . . . . . 111
10.5 Animation . . . . . 112
10.6 Metamodeling . . . . . 112
10.7 Query modeling . . . . . 112
10.8 User interface modeling . . . . . 112
10.9 Modeling user procedures . . . . . 113

**A Summary of the Method** **120**

# Chapter 1

# Introduction

## 1.1 MCM and formal specification

This report describes a method that is intended to bridge the gap between informal requirements on system behavior and formal specification of that behavior. Since the method takes the developer from informal requirements to formal specifications, it can itself not be formal [127]. However, an effort has been made to make it as precise as possible — but not more precise than that.

The method presented in this report is called MCM (Conceptual Modeling Method) and is designed to be used with a formal specification language called LCM (Conceptual Modeling Language). Two companion reports define the syntax of LCM and give a nontrivial case study of an application of LCM using MCM [31, 137]. An application of LCM on a non-trivial portion of a student administration system is given by Dehne [26]. A comparison of LCM with OMT [99], using the example of an Automated Teller Machine and the ISO car registration example [42], is given by Drenth and Paulides [29].

The relationship between MCM and LCM is that MCM provides a set of tasks that, if executed, leads to a formal specification of required system behavior in LCM. Since LCM is in the same family of languages as Oblog [22], FOOPS [40], TROLL [52, 45], and TROLL*light* [21], it is reasonable to assume that MCM, or parts of it, can be used to develop specifications in those languages as well. It would be interesting to experiment with yet other specification languages to see if MCM, or elements of it, could be used to produce specifications in those languages.

## 1.2 Progress and reuse in development methods

Although MCM is a new method, nothing about it is new: MCM is composed of components of existing methods, like Jackson System Development (JSD) [49], Entity-Relationship (ER) modeling [17, 8], Information Engineering (IE) [74], Structured Analysis (SA) [27, 79, 122, 123, 124] semantic modeling methods [47, 88], and from other sources, such as important insights from philosophical logic in the nature of taxonomies [93]. It will be clear from reading this report that the JSD method has had the greatest formative influence on MCM. Naturally, in order to adapt the various elements from these diverse sources to the context of a single, coherent method for conceptual modeling, they underwent some changes. Nevertheless, the sources of the components that go into MCM are clearly visible. They will be acknowledged at the point where they are used.

This reuse of existing components to build MCM has a number of reasons. First, in the past thirty years, a large number of methods for developing and modeling information systems have been designed and used, and it it is simply not true that all these methods are bad. It is my belief that every method contains at least one good idea, and that a small number of methods contain many good

ideas. However, it is a fact that no method contains *only* good ideas. This means that there is room for progress in development and modeling methods. But progress does not consist in the proposal of a revolutionary new method that differs from all that went before; progress consists of patiently studying the achievements of the past and trying to bring these achievements a few steps further. To do this, we must try to understand what is useful in the existing methods and incorporate that in our new methods. This is the kind of progress that exists in mathematics and in the empirical sciences, and this is the kind of progress that I would like to see in the field of method studies — that is, in methodology[1].

A second reason for incorporating elements from older methods is that this eases the transition from existing and proven methods to methods like MCM. For one thing, to the extent that MCM contains elements already known to practitioners from their current way of working, they will have few problems using the very same elements in a new method. In addition, the study of existing methods in order to filter out useful elements involves the creation of a framework in which different methods can be compared and analyzed. Such a framework is not only useful for the study of methods, but also for explaining their difference and hence for easing the transition from old to new methods.

## 1.3 The contingency approach and method engineering

I mentioned above that no method contains *only* good ideas, and MCM is no exception to this. It has been remarked by several authors that there is no universal method, that can be used in all circumstances [70]. A method for modeling and/or development is good if it is useful, and the utility of any method depends upon the context of its use. There are so many different kinds of contexts that it is an illusion to think that there could be one method that is useful in all of them. The differences between transaction processing systems, management information systems, decision support systems, expert systems, real-time systems, embedded systems and industrial control systems is just too great to expect one method to be useful for all these application contexts.

MCM is intended for use administrative applications that run on a single machine as well as for embedded systems that interact with other machines. The method is not fit for all possible situations. However, extensions are planned for real-time and control applications that run in a distributed or heterogeneous environment, such as integrated administrative and –control applications and EDI networks.

In the **contingency approach** to system development, characteristics of the application are used to choose between different available development methods [25, 85, 109]. MCM assumes a **method engineering** approach, which goes one step further: instead of choosing between methods, it allows

[1]It is well-known that physics is claimed to go through periods of normal science, in which physical knowledge accumulates, as well as revolutionary periods, in which the old paradigms are replaced by new ones and in which there is such a shift in meaning of theoretical terms that one must speak of the replacement of one body of knowledge by another, rather than of a simple accumulation of knowledge [32, 62, 63]. Although there is no agreement among philosophers of science as to the nature of meaning- and paradigm shifts, there is undoubtedly a truth in this view of the scientific enterprise. However, this does not invalidate the claim that our knowledge of physics has progressed since Greek science; the debate is about the nature and mechanics of this progress, not about whether there has been progress at all. More importantly, an important part of this progress has always consisted in testing the achievements of the past and of trying to improve upon this step by step. At no point in history, scientists have sat together to consciously devise a revolution, nor have they at any moment progressed by ignoring the past. Rather, in their effort to apply the insights of the past and to understand its limitations, they discovered new theories, that turned out, in retrospect, to trigger revolutions in scientific knowledge.

In the same way, we should not try to consciously break away from the insights of structured development, functional decomposition, systems engineering, and a host of other methodological insights from the last thirty years. Such attempts at revolution run the danger that we forget all about past achievements and that we set back the clock every ten years. For those who take the trouble to read the literature on structured development from the 1970s, it is striking how the insights that were revolutionary then are now sold as part of the revolution of object-orientation. Thus, instead of running after the latest fashion, we should study the past and build upon its achievements.

one to choose between components of methods [126] so as to put them into a customized modeling method. This requires a framework into which the chosen method components can be put. This report presents two related frameworks. Chapter 2 gives a global framework for the engineering of information system development methods and chapter 3 gives a general framework for the engineering of conceptual modeling methods. The modeling framework is actually a part of the development framework; MCM is the result of filling in this modeling framework in a particular way. The frameworks are the result of a detailed study of a number of information system development methods, which I present elsewhere [133].

The approach taken in this report to method engineering is that each particular information system development method must be composed of four kinds of methods: requirements determination methods, conceptual modeling methods, implementation methods and project management methods. MCM is merely a conceptual modeling method; it must therefore be supplemented with methods of the other three kinds. In addition, as will be made clear in this report, MCM itself is a flexible framework that consists of components taken from existing methods. These components can be replaced or extended by adding yet other components of other methods.

A consequence of this method engineering point of view is that it is not rigidly prescribed which tasks to perform in MCM, nor in which sequence tasks are to be performed. The user of MCM can choose to omit tasks, add tasks, perform them in an order different from the one they are presented in, etc. Which choices are made in a particular system development project is a matter of project management. As stated above, MCM is one example of how to fill the framework for modeling presented in chapter 3 with particular tasks. The user of this report is free to fill the modeling framework with different tasks.

## 1.4 Structure of the report

Chapters 2 and 3 give frameworks for information system development and conceptual modeling, respectively. MCM is a conceptual modeling method that results from filling in the framework of chapter 3, for conceptual modeling, in one particular way.

Chapters 4 and 5 provide a motivation for filling in this framework in this way. First, chapter 4 (The Behavior of Data Processing Systems) looks at the nature of DBS transactions and distinguishes transactions that are about the universe of discourse (UoD) from transactions that are about the DBS. This leads to the identification of the UoD model as the most important part of a DBS model. Second, chapter 5 (Representing DBS Boundaries), discusses function decomposition trees and context diagrams as means of representing system boundaries.

Chapters 6 to 9 then describe the MCM method. In chapter 6, the structure of the UoD model is described by discussing the nature and representation of objects, relationships, taxonomic structures, the instantiation relationship, events, life cycles, communication and integrity constraints. All of this is done in a language-independent way.

Chapter 7 discusses the structure of *specifications* of UoD models. Specifications in MCM consist of an informal and a formal part, that have precisely defined relationships. The formal specification discussed in chapter 7 is done in LCM, but the intention is that other languages could be used.

Chapter 8 describes induction methods for discovering UoD models from a finite set of observations of UoD behavior. After an outline of the method, we look at ways to build function decomposition trees to represent system boundaries, and discuss *transaction analysis* to discover objects, events and communications, and *scenario analysis* to discover life cycles. There are other induction methods that can be used within the conceptual modeling framework assumed by MCM, such as *record analysis*, *form analysis*, and *normalization*. These are briefly mentioned when the framework for conceptual modeling methods is discussed in chapter 3, but they are not discussed further in chapter 8.

Chapter 9 discusses evaluation methods for conceptual models. Prototyping is discussed as a

method to test the *utility* of a conceptual model and animation is discussed as a method to test the *validity* (i.e. truth value) of a conceptual model. There is a third group of evaluation methods, called *quality checks*, that test whether good use has been made of the available modeling constructs. These checks are discussed in chapter 5 for function decomposition trees and in chapter 6 for the UoD modeling constructs.

Chapter 10 concludes the report by summarizing the results of this study and by listing topics for further research.

The frameworks for information system development and conceptual modeling are combined in appendix A. In addition, possible components to use at different places in this combined framework are indicated, together with places in the literature where these components are described. The appendix at the end of the report summarizes all methods discussed in this report. An index of key terms is given.

# Chapter 2

# Information System Development

## 2.1 Three aspects of systems

We are interested in system development, where a **system** is defined to be any observable part of the world. This excludes unobservable entities like numbers and truth values –but it includes the visible symbols used to represent those abstract entities. Every system has a **behavior**, which is the sum total of all possible observations that can be made of the system, and an **implementation**, which is the way in which a system is realized in subsystems. A system observation is in this report a communication between observer and system, i.e. it is an event in which two systems, the observer and the observed, interact.

All systems have a behavior and an implementation, but tools made by man have (or should have) in addition a **function**, which is the service that the system is designed to deliver to its environment. The function of a tool provides the reason why the tool exists, i.e. why resources are spent to build it. The environment for whom the tool must have a function is the system **user**, who expects to have a benefit of interacting with the tool.

**Development** is a process of trying to bring the world in a better state than it currently is. In the special case of **system development**, this is done by delivering a system. Since the delivered system has (or should have) a function, which is realized by system behavior, which in turn is realized by the implemented

1. the identification of a desired system *function*,
2. the specification of a *behavior* that realizes this function, and
3. the *implementation* of a system that displays this behavior.

This list of three tasks gives us the basic structure according to which to analyze system development methods. In the analysis of any system development method, we should look for these three tasks.

## 2.2 Rational choice

The distinction between function, behavior and structure is a tool that can be used to analyze development methods. A second analytical tool that can be used to analyze system development methods is the logical structure of the **rational choice process**. This process has the following structure:

1. Identify the choice to be made.

2. Analyze the current situation, including the need to make a choice.
3. Generate possible alternatives for action.
4. Evaluate the possible alternatives.
5. Choose an alternative.

The rational choice process in requirements determination is well-known from decision theory [110, 111], artificial intelligence (where it is known under the name of *means-end analysis* [112]), and management science [82]. We can use the rational choice process to analyze system development methods because in system development, choices must be made. We can therefore investigate to what extent the method prescribes these choices to be made in a rational way.

## 2.3 The regulatory cycle

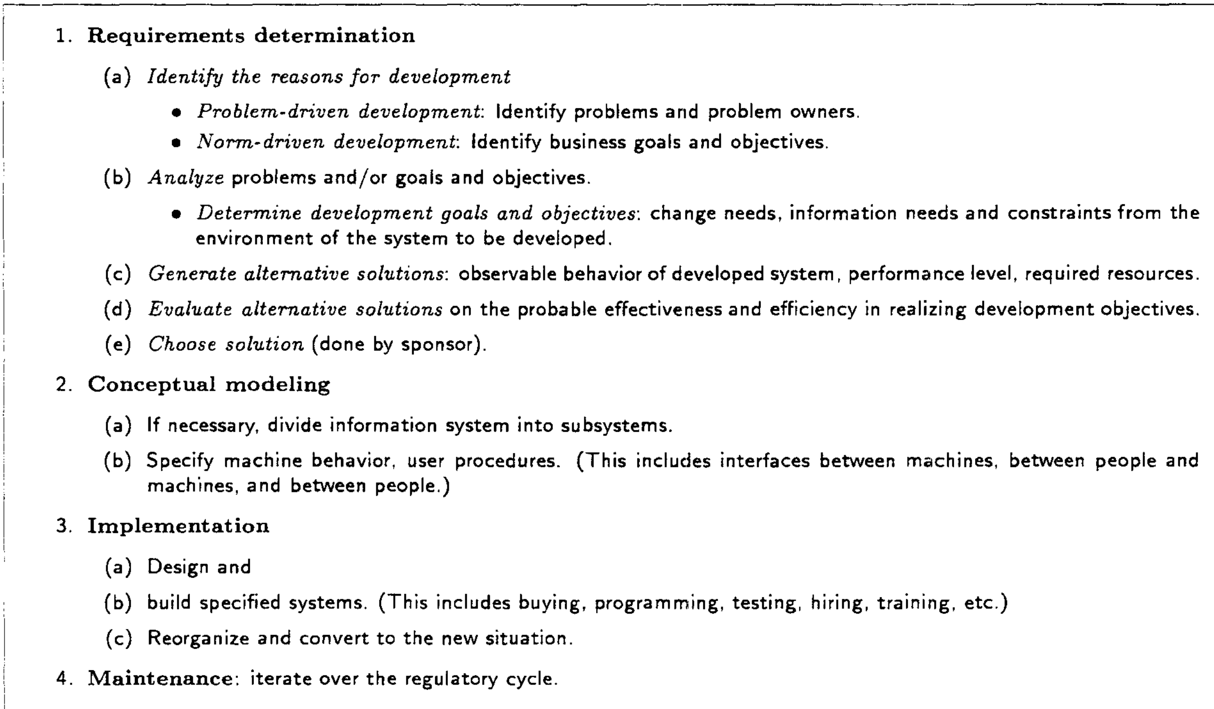
We saw above that system development consists of three tasks, the identification of system function, the specification of required behavior, and implementation. Nothing was said above about the order in which these tasks should be performed; it was merely stated that these tasks must be performed in some way during system development. I now make two **rationality assumptions**, which will allow us to use the analytical tools sketched above to build a framework for system development methods. The rationality assumptions are as follows:

1. The development process can use *unlimited resources* and
2. it takes place in a *perfect environment.*

Under these assumptions, we can assume that the development process actually consists of the three tasks in the order listed above, and that any choice to be made in this process is made according to the rational choice model. If the first development task, determination of system function, is made according to the rational choice model, we get the following idealized development method:

1. **Requirements determination**: Identify the needs of the environment.
   (a) Identify the reasons for development.
   (b) Analyze the reasons for development.
   (c) Generate possible solutions.
   (d) Evaluate the solutions.
   (e) Choose a solution.
2. **Conceptual modeling**: specify required system behavior.
3. **Implementation** of the required behavior.

This idealized method is called the **regulatory cycle** of system development. It is very general, because it does not mention at all what kind of system would be developed. The regulatory cycle is the underlying structure of many development methods. It is for example the logic of therapeutic action in psychology, medicine, and politics [117], management consultancy [101], and systems engineering [43]. It is also the underlying structure of a number of information system development methods, including ISAC [68], Information Engineering [75, 76, 77], ETHICS [83], SSADM [28] and Structured Analysis/Structured Design (SA/SD) [27, 36, 79, 122, 123, 124, 140]. Figure 2.1 shows what we get if we specialize the method with the idea that computerized information systems are developed. The regulatory cycle of information system development in figure 2.1 has been arrived at by means of a study of a number of the methods mentioned above.



1. **Requirements determination**
   (a) *Identify the reasons for development*
   - *Problem-driven development*: Identify problems and problem owners.
   - *Norm-driven development*: Identify business goals and objectives.

   (b) *Analyze* problems and/or goals and objectives.
   - *Determine development goals and objectives*: change needs, information needs and constraints from the environment of the system to be developed.

   (c) *Generate alternative solutions*: observable behavior of developed system, performance level, required resources.
   (d) *Evaluate alternative solutions* on the probable effectiveness and efficiency in realizing development objectives.
   (e) *Choose solution* (done by sponsor).
2. **Conceptual modeling**
   (a) If necessary, divide information system into subsystems.
   (b) Specify machine behavior, user procedures. (This includes interfaces between machines, between people and machines, and between people.)
3. **Implementation**
   (a) Design and
   (b) build specified systems. (This includes buying, programming, testing, hiring, training, etc.)
   (c) Reorganize and convert to the new situation.
4. **Maintenance**: iterate over the regulatory cycle.

Figure 2.1: The regulatory cycle of information system development. This is a skeleton of an information system development method, which should be filled in at the appropriate places with methods and techniques for problem analysis, information requirements analysis, conceptual modeling, implementation, etc. It must also be combined with methods and techniques for the management of the development process.

## 2.4 Method engineering

The regulatory cycle is not a development method, because it makes unrealistic rationality assumptions and because it does not give enough advice to carry out a development project, even if the rationality assumptions were satisfied. Rather than being a development method, it is a *framework for method engineering*. This framework can be used by supplementing it with four kinds of methods.

1. *Requirements determination methods* to identify relevant problems, interest groups, business strategies, to analyze information needs, etc. Important methods in this context are the analysis of strengths, weaknesses, opportunities and threats (SWOT), Rockart's critical success factor analysis [96], King's strategy set transformation [60], customer resource life cycle analysis as proposed by Ives and Learmont [48], Porter's value chain analysis [90, 91], Wiseman's strategic option generator [139], etc. These methods analyze the current situation on its potential for new strategy, which could lead to the identification of the need for information system (re)development. Other methods concentrate on problem-solving, such as ISAC change analysis [68] and the first part of ETHICS [83]. In the case of ill-defined problems, Checkland's Soft Systems Methodology (SSM) [16, 15] could be used to find out what it is one wants to be changed. Pava's version of socio-technical analysis can be used to determine managerial information needs in a way that balances social and technological factors [87].

2. *Conceptual modeling methods* for the specification of required system behavior. This is the main subject of this report. The kind of methods used here will influence some of the methods to be used in requirements determination (e.g. information needs analysis) and implementation. Examples of conceptual modeling methods for information systems are entity-relationship (ER) modeling [17, 8], NIAM [86], part of Jackson System Development [49] (JSD), the part of Structured Analysis (SA) that deals with modeling the required system [27, 79, 122, 123, 124], and the modeling parts of object-oriented methods such as Object Modeling Technique (OMT) [99], Objectory [50], and the methods of Coad and Yourdon [19] and Shlaer and Mellor [107, 108]. These methods use a variety of modeling structures and cannot be combined at will. The problem for method engineering is to study the relationships between these structures, indicate where they can be combined, and where they are incompatible. MCM arose from such a study of conceptual modeling methods [133].

3. *Implementation methods.* This includes methods for implementing the manual part of the system (if any) and the automated part (if any). Implementation can be done manually (using programmers) or automatically (using program generators), and in both cases we have a choice of coding everything from scratch or adapting parametrized software. If manual implementation is chosen, then, as said above, the conceptual modeling method (structured or object-oriented) will influence the choice of implementation method.

4. Last but not least, methods and techniques for the *management of the development process* are needed. The regulatory cycle assumes unbounded resources and a perfect environment, so it must be supplemented with development management methods, that deal with finite resources and a possibly disturbing environment of the development process. Examples of management methods are methods for planning, budgeting, monitoring, staffing, reporting, risk analysis, cost/benefit analysis, etc. etc. For the most part, these methods are independent from the kind of system being developed as well as from the other three kinds of methods with which they are combined. However, some peculiarities arise from the fact that computerized information systems are developed. For example, the precedence ordering between software development tasks is often not as rigid as it is for other kinds of development projects. Useful introductions to project management in the context of information systems development and software engineering

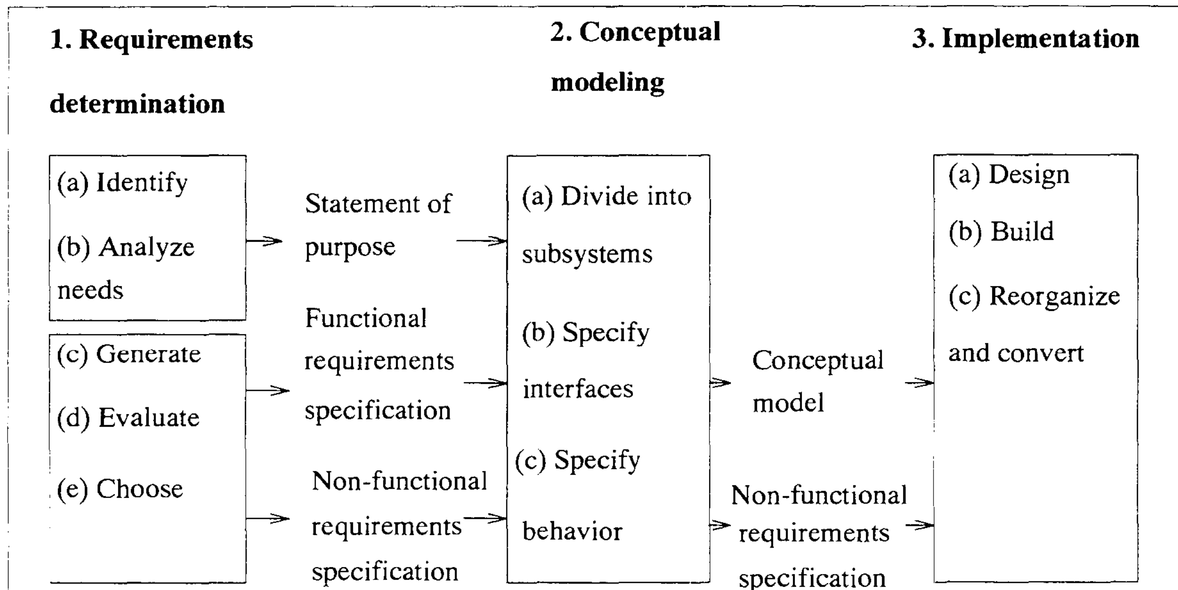



Figure 2.2: Conceptual modeling is the transformation of the functional requirements specification into a precise specification of observable system behavior. Usually, this leads to an update of the non-functional requirements, because they can be made more precise by stating them in terms of the conceptual model. In a rational development process, the statement of purpose is not changed during conceptual modeling.

are given by Rook [97], Suthwell [114], a collection of papers edited by Thayer [119], and a special issue of the *Communications of the ACM* of October 1993.

One important decision for development management is the choice of a path through the regulatory cycle. Examples of paths through the cycle are linear development, iterative development, throw-away prototyping, evolutionary prototyping, spiral development, etc. Descriptions of these methods in the context of software development can be found, among others, in a collection of papers edited by Agresti [2]. An explanation of how these paths can be mapped to the regulatory cycle is given by me elsewhere [133].

In this report, we are concerned with conceptual modeling only.

## 2.5 The structure of a requirements specification

Figure 2.2 shows the products of the different tasks in the regulatory cycle and what they are used for. For us it is important to note that a **requirements specification** consists of three parts:

1. A **statement of the purpose** of the required system. This describes the need in the environment that is to be met by the system.
2. A **functional requirements specification** that describes the behavior required of the system if it is to fulfill its purpose effectively.
3. A **non-functional requirements specification** that gives constraints on the implementation of the system that improve the usefulness of the system.

Non-functional requirements contain performance requirements, standardization requirements, quality criteria, etc. The output of requirements determination is informal and is worked out further during conceptual modeling. In particular, the functional requirements specification is worked out to a precise conceptual model of the observable behavior of the required system, that can serve as input to the implementation task.

A **conceptual model** of system behavior is a model of the behavior of the system, independent of any implementation technology. McMenamin and Palmer [79] call this an **essential model** of the system The conceptual model of a system represents the essence of what the system must do to achieve its function, regardless of the implementation used. Conceptual models may depend upon implementation technology used in the environment of the system, but do not depend upon the implementation technology used for the system itself.

During conceptual modeling, the non-functional requirements may also be updated. This is because a more precise description of the behavior of the system is available in the form of a conceptual model specification. This facilitates a more precise specification of constraints on system implementation. For example, we can now state precisely what the volume of data stored about a particular class of objects is, what the frequency of updates is for a class of objects, how fast the response must be to particular queries that can now be stated formally, etc.

# Chapter 3

# Conceptual Modeling

**Conceptual modeling** is a process of trying to find a true representation of the world. Where development is a normative activity, because it is a process of trying to make a *better* world, conceptual modeling is a descriptive activity, because it is a process of *describing* the world. What is described may however be the current or a desired state of the world. If we model a desired state of the world, then this modeling activity is intertwined with development activity, but both activities are still distinct, because in this case, modeling is the second stage of a rational development process. The crucial development question is whether we improved the world in the best (or an optimal) way; the crucial modeling question is whether we described the desired improvements correctly.

In the context of information system development, conceptual modeling is the representation of the behavior required of an information system. An **information system** consists of three kinds of systems:

1. Subsystems of an organization, including organization units and/or people.
2. A **database system** (DBS) that contains data about a part of the world called the **universe of discourse** (UoD) of the DBS. The database system may itself consist of a number of systems, implemented on the same or on different computers, connected through a network.
3. A **user interface** that connects the mechanical subsystems with the human subsystems.

In this report, we are concerned with the DBS of an information system, and we will assume that this is a computerized system. The organizational subsystem and user interfaces of an information system will be ignored.

In addition to information systems, we are also interested in **embedded database systems**, such as production control systems, cruise control systems, etc. An embedded database system is a data processing system whose users are themselves machines; that is, it provides a service to other machines. Embedded systems are always computerized, often have less emphasis on data storage and more on controlling the environment in real time. MCM is intended for modeling DBSs in the context of information system development as well as for modeling embedded systems.

## 3.1 The empirical cycle of conceptual modeling

Just as the regulatory cycle is the only ideal method for development, so the **empirical cycle** is the only ideal method for conceptual modeling. The empirical cycle of conceptual modeling of required information system behavior is shown in figure 3.1. The empirical cycle is known from the philosophy of science [54, 84], where evaluation is usually split into *deduction* of testable consequences from the

1. **Observation.** Using the requirements specification as guideline, collect instances and descriptions of relevant DBSs and of the represented UoD.
2. **Induction** of a model. Depending upon the kind of observations made and the kind of model sought, do one or more of the following:
   - elementary sentence analysis,
   - some form of transaction analysis,
   - some form of scenario analysis,
   - some form of record analysis,
   - some form of form analysis,
   - normalization of existing relation definitions,
   - some form of essential system analysis.
3. **Evaluation** of the model.
   - The *quality checks* that can be performed depend upon the kind of model structures.
   - The *validation* checks consist of some form of animation of the model.
   - The *utility checks* consist of some form of prototyping the required system functions.
4. **Improve** the model by iterating over the empirical cycle.

Figure 3.1: The empirical cycle of conceptual modeling. The induction and evaluation methods are treated in chapters 7 and 8.

model and *testing* the hypothesis by doing an experiment in which the testable consequences will be confirmed or falsified. In the empirical cycle of figure 3.1, deduction and experimental testing is called **validation**. The reasons for the presence of the other two evaluation methods, utlity and quality checks, are explained in section 3.4.

The empirical cycle has the structure of a rational choice process, as can be seen if we present it in a slightly different way:

1. Identify the reasons for discovery (*observation* of unexplained phenomena).
2. Analyze the observations of the unexplained phenomena (*induction* of a model).
3. Generate alternative hypotheses that explain the phenomena (*induction* of a model).
4. Evaluate the hypotheses (*Evaluation* of the hypotheses).
5. Choose a hypothesis.

The similarity between the regulatory and the empirical cycles is that in both, we are involved in a rational choice process in which we search for a solution. The difference between the two is that in the regulatory cycle, we are searching for an action that will improve the world, whereas in the empirical cycle we are searching to improve our knowledge of the world (including our knowledge of the desired states of the world). In the context of information system development, we try to model required system behavior, without trying to change the requirements.

The empirical cycle is present in a number of modeling methods, like NIAM [86], Entity-Relationship (ER) modeling [17, 8], the data flow modeling part of SA [27, 79, 122, 123, 124, 140] and Jackson System Development (JSD) [49]. However, the presence of the empirical cycle in these methods is very slight. In NIAM it consists of a recipe for deriving a NIAM model from elementary observation sentences and for validating this model with domain specialists by means of population diagrams. In

| | Instances | Types |
|---|---|---|
| DBS | The current DBS | Record declarations, forms, DBS tables |
| UoD | Situation descriptions, direct observation, participant observation | Interviews, questionnaires, requirements specification, company documents |

Figure 3.2: Four different sources of data to be used for conceptual modeling.

SA, it consists of deriving an essential system model from observations of the current system, deriving from this a model of the required system, and validating the required system model with domain specialists by means of walkthroughs. However, these global methodological guidelines take only a small part of textbooks on these methods. In textbooks on ER modeling and JSD, the empirical cycle plays an even more modest role.

Instead of concentrating on the empirical cycle, descriptions of conceptual modeling methods concentrate on the *structures* to be found in conceptual models. For example, NIAM provides facts as basic model structures, ER provides entities and relationships as basic model structures, structured analysis use data flows, data transformations and data stores as basic structures, and JSD uses such structuring primitives as entities, life cycles and communication.

Thus, finding a common ground for the different *methods* of conceptual modeling is not difficult: this is the empirical cycle. Finding a common ground for the *structures* of different conceptual models is much harder. The problem for method engineering is to find relationships between these structures and indicate to which extent the structures can be combined in one model. I study these relationships elsewhere [133]. In this report, I use the results of that study to design MCM. In the rest of this chapter, I give a survey of the three major tasks in the empirical cycle of conceptual modeling: observation, induction, and evaluation.

## 3.2 Observation methods

An **observation method** is a systematic way of gathering data as a preparation for the induction task. There are two kinds of data that can be gathered, data about *types* and data about *instances*, and there are two kinds of sources where to look for this data, the *DBS* and the *UoD*. This gives four kinds of sources of data, shown in figure 3.2. This table is derived from a similar table given by Rock-Evans [95].

### Observation of instances

First of all, we can look in the UoD for particular instances of situations that must be registered by the DBS. We can look for these instances in some relevant DBS that currently exists and is accessible, or we can look for these instances in the UoD. For example, we can analyze the current DBS, as is done in essential system analysis [79], or we can perform some jobs in the organization for which an information system is to be developed.

### Observation of types

Even when, at the instance level, we gather data about individual occurrences, particular situations and observable instances of entities and events, it is impossible to ignore the type level. People cannot observe anything without classifying it. It is impossible to look at a building without realizing that one is looking at a house, a factory building, a shop, o at least at a building. If you are not able to give any general name to what you look at, then you are not *looking* at a particular thing at all: all

you do is direct your gaze in the direction of something that would be described by others as a house, or a factory building, etc. We can see this also in descriptions of particular situations: If we describe what goes on in a particular situation, then this description is full of general names like "person", "document", "member" "loan", "clerk", "counter" etc. These general names are names of concepts and they are candidates for type names in a conceptual model of the situation.

What is meant by observations at the *instance level* is not that no type names are used in descriptions of these situations, but that particular situations are described. In observations made at the *type level*, on the other hand, no particular situations are described but general statements are collected, that describe the UoD in general terms, or that describe the information requirements of problem owners and DBS users, or describe rules, procedures, regulations, laws, in general terms that may be relevant for the modeler. Here are some examples of observations that can be made at the type level.

- At the DBS type level the *forms* in use in a manual DBS can give important clues about what types of things exist in the UoD and about what their relationships are. *Record declarations* and *table definitions* of automatized DBSs are also a useful source of data for discovering relevant entity types.
- At the UoD type level, the modeler can conduct *interviews* and collect *questionnaires* about the UoD. The data gathered this way does not describe particular situations, but contains the knowledge of domain specialists in general terms. There are also innumerable *company documents*, such as marketing leaflets, memos, quarterly reports, mission statements, reports of meetings, etc. that can be used as a source of data. Foremost among these is of course the *requirements specification* that came out of the requirements determination task.

Observation methods are not treated in this report. A good introduction to methods for interviewing, using questionnaires, direct observation and data sampling for information analysis is given by Kendall and Kendall [55] and by FitzGerald and FitzGerald [33].

## 3.3 Induction methods

An **induction method** is a method to build a model of a system based on a finite number of observations of the system. The input to the iduction process is a finite set of observations, and the output is a model that makes a statement about an infinite number of possible states of reality. The **inductive jump** from a finite set of observations to a model that accounts for infinitely many possible observations can never be justified fully. The resulting model will always contain an element of hypothesis.

The induction methods that can be used in conceptual modeling depend the data used as input as well as upon the modeling structures produced as output. Figure 3.3 gives a schematic overview of the possibilities and gives some examples, to be discussed below. Figure 3.3 distinguishes file-oriented– from object-oriented conceptual models. A **file-oriented conceptual model** represents a system as a collection of files upon which programs act. Files have memory but no activity, and programs have activity but only limited (short-term) memory. An **object-oriented conceptual model** represents a system as a collection of objects that have local state and behavior, and that communicate with each other. MCM is a method that leads to object-oriented conceptual models.

### Induction from DBS instance observations

There are two induction methods that start from observations of the current DBS.

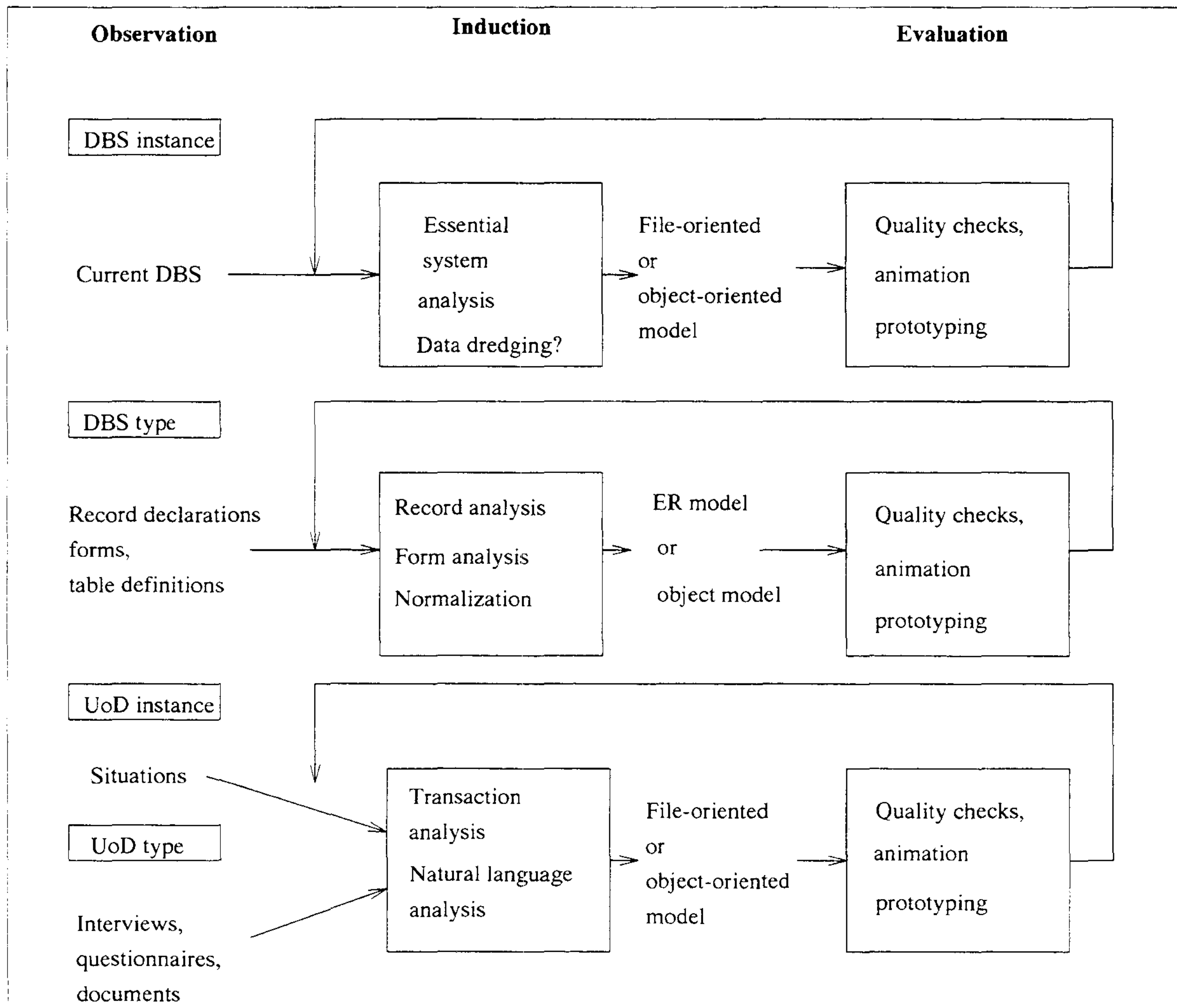



Figure 3.3: Different induction methods are geared to different observational data and/or different kinds of models.

- **Essential system analysis** is a modeling method that expects observations of the current DBS as input and leads to an essential model (i.e. a conceptual model) of that DBS as output. An essential systems analysis method for data flow models has been described elaborately by McMenamin and Palmer [79] as a method for Structured Analysis. Although I am not aware of any essential systems analysis method that produces object-oriented models, there is no reason why this approach should be restricted to producing data flow models (the *file-oriented models* of figure 3.3).

- Figure 3.3 also mentions **data dredging** as a possible induction method to find a model of a current database system. Data dredging is the analysis of a large collection of data in the hope of finding some unsuspected pattern, i.e. a model of the data. This is still a hypothetical possibility.

Essential systems analysis and data dredging are ignored in this report.

### Induction from DBS type observations

There are a number of methods to produce ER models from observations at the DBS type level.

- **Record analysis** is the analysis of record declarations to yield possibly relevant entity- or relationship types and possibly relevant attributes. Record analysis is described as a method to find ER models by Rock-Evans [95, pages 42–43], and by Batini, Ceri and Navathe [8, pages 101–109].

- **Form analysis** is the analysis of two-dimensional forms, either made of paper or screen-based, to yield possibly relevant entity- or relationship types and possibly relevant attributes. Form analysis is described as a method to find ER models by Rock-Evans [95, pages 37–41], and Batini, Ceri and Navathe [8, pages 92–100].

- **Normalization** is a method to transform a collection of attributes into a set of relation schemas such that each relation schema consists of attributes that depend only upon the key and upon all of the key. This simple dependence structure *may* be caused by the fact that each relation represents one object class and normalization is therefore a heuristic to find object classes in an object-oriented method. Normalization is described from a practical point of view by Rock-Evans [95, pages 60–72] and from a theoretical point of view in textbooks on database management systems, e.g. Date [24] or Ullman [120].

These methods are geared towards analyzing *existing* systems and are therefore part of **reverse engineering** (making a conceptual model of a currently existing system, with a view to reimplementing it in a better way). Again, although these methods have been proposed for ER models, I see no reason why they could not be applied in the search for object-oriented models. This is especially so since we can view the class model, which is part of of object-oriented conceptual model produced by following `MCM`, as an extension of an ER model. I will ignore the methods listed above in this report. However, they are examples of methods that could be added to `MCM`. The appendix of this report indicates the task in `MCM` where they could be used.

### Induction from UoD observations (instances and types)

The following induction methods start from observations in the UoD, at the instance or type level.

- **Elementary sentence analysis** takes a text containing a requirements specification as input and, very roughly, takes nouns to be candidate object types and verbs to be candidate relationship types. Again, it can be used as induction method for object-oriented models. Elementary

sentence analysis is described by Chen [18], Rock-Evans [95, pages 44–53], and Batini, Ceri and Navathe [8, pages 86–90] as a procedure to derive ER models. Nijssen and Halpin show how the NIAM modeling method is based upon the analysis of elementary sentences [86]. Abbott [1] and Saeki et al. [100] show how elementary sentence analysis can be used as a procedure to develop software. In chapter 8, it is shown how elementary sentence analysis can be used in `MCM`.

- **Transaction analysis** takes as input some kind of list of all desired DBS transactions. It is described in chapter 8 (An Induction Method for Finding UoD Models). It is proposed as a method to find ER models within Structured Analysis by Flavin [34]. The event-oriented view essential to transaction analysis can also be found in SA for real-time systems [122, 123, 124] and modern structured analysis [140]. Transaction analysis is part of `MCM`.
- **Scenario analysis** is a method to find a model of the behavior of a system as it is composed of elementary transactions. For example, the behavior of an automated teller machine can be found by analyzing use scenarios in which the ATM interacts with a user. Scenario analysis is used in real-time system modeling [46, 123], JSD [49], in OMT [99] and the Shlaer/Mellor method [108]. It is part of `MCM` (chapter 8).

The above three methods are discussed in chapter 8. They are "official tasks" within `MCM`. It is however possible to use other methods within `MCM`.

## 3.4 Evaluation methods

**Evaluation** of a DBS model consists of deducing observable consequences from it and testing these consequences against reality, to see if the model is correct. In this narrow sense, evaluation is the same thing as *validation*, discussed below. In the context of conceptual modeling, we add two other elements to evaluation, viz. checking the *quality* of our use of the available modeling constructs and checking the *utility* of the model with respect to the required DBS functions.

1. A **quality check** is a check on the appropriate use of the modeling constructs in our model. Quality checks are performed during induction, but they formally belong to the evaluation task. Which quality checks there are depends upon the kind of model structures we use. Chapter 5 (Representing DBS Boundaries) gives quality checks for function decomposition trees and chapter 6 (The Structure of UoD Models) gives the quality checks that can be used for the structures present in UoD models of `MCM`.
2. If the modeled system would exist, **validation** would consist of deducing observable consequences from the specification, and setting up an experiment to see whether this behavior actually is displayed by the system. If the behavior is not displayed by the system, the model is falsified. Since the system does not yet exist, we must resort to other methods. These are discussed in chapter 9 (Evaluation Methods for DBS Models).
3. **Utility checks** are ways to check whether the DBS described by the model has all the functions required of it. We must for example be sure that all required queries can be answered by the system, that in embedded system all control signals are issued as required, etc. Utility checks are also treated in chapter 9.

The three aspects on which a model is tested correspond to the classical triad of Beauty, Truth and Goodness. Improvement of a model so that it scores better on quality checks usually leads to models that are simpler and, in a sense, more beautiful. Improving the validity of a model increases the truth-value of the model, and improving its utility increases its positive value, i.e. its "goodness" for the user.

# Chapter 4

# The Behavior of Data Processing Systems

The previous two chapters give a framework for methods for information system development and, within that, for conceptual modeling. The conceptual modeling framework is the empirical cycle known from the philosophy of empirical sciences. `MCM` is a particular way to fill in this framework. In order to motivate the way in which this is done, I show in this chapter that a conceptual model of an information system can be decomposed into a *UoD model* and a **DBS model**. The UoD model is the core of the DBS model, and finding it is the core of `MCM`.

I start with pointing out that data processing systems are reactive systems, which process transactions in a way that depends upon their internal state. Next, I discuss the concept of atomicity of transactions. Finally, I show that transactions of database systems are special in the sense that they have a meaning. This has the consequence that DBS transactions can be partitioned into two groups, those that are about the DBS itself and those that are about the universe of discourse of the DBS. It is this second group of transactions that is central to DBSs and that must be central to any conceptual modeling method.

## 4.1 Reactive systems

Any system receives input events and produces output events.

- A **functional system** is a system whose reaction to an input does not depend upon previously received inputs. Thus, once a functional system produced an output, it has no memory of any inputs that it received. A functional system behaves like a mathematical function. (This use of the word "function" has nothing to do with function as service delivered by a system to its user.)
- A **reactive system** is a system whose reaction to input depends upon its current state. Thus, a reactive system has a memory of at least some past inputs.

The term "reactive system" was coined by Manna and Pnueli [71]. McMenamin and Palmer [79] mean the same thing with the term "planned response systems".

Examples of reactive systems are database systems, operating systems, text editors, etc. Examples of functional systems are the functional primitives of a data flow diagram, Pascal programs without file parameters, mathematical routines, etc. Note that we now use "function" in two meanings of the word: function as *service* for the environment and function as *mathematical function*. The service

that a functional system provides to its environment is always to compute a mathematical function. The service that a reactive system provides to its environment is always to provide an "intelligent" reaction, where the intelligence consists of producing a reaction that depends upon more than just the current input, because it also includes a dependence upon previously received input.

## 4.2 Atomicity of transactions and events

Intuitively, an **event** is something that can occur at a moment in time. What this means exactly is left undefined here. Something will be said on the nature of events in chapter 6, where events will be defined as relations on system states.

An event in which a system $S$ interacts with its environment is called a **transaction** of $S$. Transactions are the smallest units of change of a system. All transactions are events, but there may be events that are internal or external to the system, and hence are not transactions.

We regard events (and hence transactions) as atomic happenings in the world, in two meanings of this word:

1. **Atomicity**: An event occurrence contains no intermediary states.
2. **Discreteness**: An event has no duration.

We assume that time is discrete, i.e. that the set of all possible time points is isomorphic to the set of natural numbers. The discreteness property of events can then also be formulates as follows:

- Each event occurrence takes one moment of time.

Whether one moment (the tick from moment $n$ to moment $n+1$) is viewed as having *no* duration or a duration of exactly 1 tick, is currently not important in MCM. We may have to eliminate this sloppiness in a future version of MCM, when real-time issues are incorporated.

The atomicity property of events means that events have no intermediary state. This means, among others, that *walking* is not an event but a state, but that *start_walking* and *stop_walking*, by contrast, are events. Another implication is that in MCM, we will not represent output events as occurring concurrently with the input events that trigger them. The reason for this is simply that output is *caused* by input, and there must always be a sequential relation between the occurrence of a cause and the occurrence of an effect. The occurrence of a cause always precedes the occurrence of an effect.

To illustrate, suppose $M$ is a machine with a button that acts as a toggle for a light bulb attached to $M$. An attempt to press the button is an input event which we will call *press*. If $M$ reacts to this by switching the light on or off ($switch_on$ or $switch_off$), then this output always *follows* the input that triggered is. Assuming that the bulb is initially off, the behavior of $M$ could be represented by the trace

$$press\ .\ switch_on\ .\ press\ .\ switch_off\ .\ \ldots$$

The dot represents sequential execution.

The reaction of a machine to input may be empty. If the bulb is broken, for example, then the response of the machine to input events is the empty event $\epsilon$ and its behavior can be represented by the trace

$$press\ .\ \epsilon\ .\ press\ .\ \epsilon\ .\ \ldots$$

The $\epsilon$ can be removed from this trace without altering the meaning of this description of system behavior.

Whether a trace of events is the best way to specify system behavior is not the point here. The point to remember about the example is that the reaction to an event always *follows* the event that caused it and never occurs simultaneously with its cause.

## 4.3 Transaction atomicity and interface technology

The atomicity of transactions is an abstraction that must be realized by the interface technology of the modeled system. The following discussion of the role of interface technology in the atomicity of transactions is based on a discussion of events in real-time structured analysis (SA/RT) given by Ward and Mellor [123, pages 30–35]. (There are some differences in approach between `MCM` and SA/RT concerning the nature of events, but with suitable changes, what SA/RT has to say about events is relevant for `MCM`.)

Each transaction of a system $S$ is an interaction between $S$ and its environment. Transactions may occur at the initiative of the environment of $S$ or on the initiative of $S$ itself. Whatever the case, the transaction must be implemented in a piece of technology called **interface technology**. Interface technology has two characteristics:

- Interface technology allows the system to engage in transactions with the environment. If the transaction is an input transaction, interface technology allows the system to recognize that an external event occurred to which it must respond. If it is an output transaction, interface technology allows the system to inform the environment that an output is produced.
- Interface technology is not part of the implementation of the system. Rather, it shields the system from its environment just as much as it connects the system with its environment. Interface technology often cannot be chosen by the system developers, but is imposed upon them by the environment. The concept of implementation-independence does not extend to interfaces: the essential (i.e. implementation-independent) system model must incorporate structures that result from the choice of interface technology. The implementation of a system is that part of the realization of the system that can be chosen by the developer, and interface technology is not part of this.

If we look close at interface technology, we break through the atomicity of transactions to a lower level of detail to see how transactions are realized. Here are some examples:

- A *borrow* transaction between a library member and the library itself is realized at the circulation desk by means of a dialog in which the member asks for a document, a circulation desk clerk checks the availability of the document in a database system, and then either notifies the member that the document is not available or else goes to the storeroom to fetch the document, updates the database system with the *borrow* event, and hands over the document to the customer. This entire process realizes a single atomic transaction, *borrow*.
- A management information system connected to a transaction processing system (TPS) may require aggregate data about a day's transactions from the TPS every midnight. Production of this data is itself a transaction in which the TPS is engaged, and it is triggered by another transaction, which we may call *time to produce aggregate report*. This second transaction is just the passing of a significant moment in time, called a **temporal event**. (A better term would be temporal *transaction*, but I follow established terminology here.) Interface technology that allows the system to recognize the occurrence of a temporal event includes a system clock and software that compares every tick of this clock with a set of significant time points. Event *recognition* is a process that takes place in the system itself, using the system clock and some software. The *recognized event* however takes place outside the system: the passage of time is not an internal system process. And though the passage of time is a continuous process, the recognized temporal event is discrete. The recognized event, *time to produce aggregate report*, triggers an output transaction in which a report is produced. The temporal event and the output transaction are both atomic.

- A system that cuts sheets of metal transported over a conveyor belt must recognize the points at which the sheet must be cut. It has a *cut* transaction that is triggered by another transaction, which we may call *the place of the sheet where it must be cut is positioned under the cutter.* We can call this a **spatial event**, because it consists of a certain object being in a certain position. When this spatial event occurs, there may be nothing visible in the sheet, which is a continuous flat surface. Interface technology required to recognize this spatial event includes a sensor that measures the speed of transport of the conveyor belt and a synchronization mechanism that measures the moment when the sheet starts being transported by the belt. Again, event recognition is a process that takes place inside the system, but the recognized event occurs outside the system. And just as for temporal events, there is a continuous process (passage of the sheet on the conveyor belt), which underlies the discrete spatial event.
- A cruise control system for a car may have a transaction "maximum speed reached". The car engages in a continuous change of speed, which may go from 0 to the maximum, but only certain discrete events in this continuous change are considered to be system transactions of the cruise control system.
- Registration of a withdrawal of stock from a store room is a transaction, that may trigger a second transaction which reports to the DBS user that stock is now below a certain level and must be replenished. Recognition that this second transaction ought to take place is an internal system process, but the recognized transaction occurs outside the system (in the store room). In fact, we have two external transactions, which we may call *withdraw stock* and *stock falls below a critical level.* The second transaction has as effect the creation of an obligation on an agent in the UoD to reorder stock. Both occur outside the computer and both can be registered by the computer. The *withdraw stock* transaction sometimes occurs on its own and sometimes occurs simultaneously with the *stock falls below a critical level* transaction. The recognition mechanism for the second transaction is internal to the computer. This recognition mechanism is often called a *trigger*; at the conceptual level, they are modeled in JSD as *interactive functions.*

In all these examples, we may be able to observe quite a lot of events (the *borrow* example) or we may observe nothing in particular (the temporal and spatial event examples), or we may observe no discrete events at all. It is interface technology that allows us to identify the relevant transaction at the relevant level of abstraction.

As Ward and Mellor observe, it is not necessary and often not even possible to fix interface technology in advance of system development [123, page 16]. In these case, we must identify relevant system transactions on a conceptual level and proceed from there. These transactions can be used to find a suitable interface technology, without affecting further analysis and implementation of the system.

## 4.4 A classification of DBS transactions

It is customary to classify the transactions of any system into **input transactions**, which are transactions initiated by the environment of the system, and **output transactions**, which are transactions initiated by the system itself. For DBS modeling it is more helpful to apply another distinction, based upon the *meaning* of DBS transactions. The distinguishing feature of DBSs is that DBS transactions have a *meaning*, that goes beyond the physical occurrence of the transactions. This meaning is assigned to them by people. Thus, DBS contains data about a part of the world called the **universe of discourse** (UoD). This causes a *duplication* to occur between the UoD and the DBS:

- For each relevant *entity* in the UoD, the DBS contains a *surrogate* for that entity;

- for each relevant *attribute* of the entity the surrogate has a corresponding attribute;
- and for each relevant *event* in the life of the entity, the surrogate contains a corresponding *update*.

The term UoD gained prominence in the 1982 ISO report on Conceptual Schemas [42]. Surrogates were introduced by Codd in RM/T [20], but they have an older ancestry [44].

We now have two important features of DBSs: they are reactive systems and their transactions are meaningful. Reactiveness is a characteristic that DBSs share with other kinds of systems. The fact that DBS transactions have a meaning, on the other hand, is the distinguishing feature of DBSs. The meaningfulness property has the consequence that DBS transactions can be partitioned into two groups according to their meaning: those that are about the UoD and those that are about the DBS itself. We look at these two groups in the following two subsections.

### 4.4.1 Transactions about the UoD

To realize the meaningfulness of DBS transactions, there is a fundamental class of input transactions of a DBS:

- **Registration transactions** communicate the occurrence of an event in the life of a UoD entity to the surrogate that represents the entity.

A registration transaction always occurs at the initiative of the UoD, possibly with an intermediary who observes an event occurrence in the UoD and communicates this to the DBS. For example, a circulation desk clerk of a library acts as an intermediary between the UoD event *borrow* and the registration event *borrow* (i.e. the clerk is part of interface technology).

There is usually a *delay* between the registered event occurrence and the registration event occurrence. For example, the *borrow* event in the UoD precedes the registration of the *borrow* event by the DBS. Note that the registered event may be the formation of the intention to do something in the future. For example, a student may register for a test or a manufacturer may register a car to be produced in the future. In these cases, what is registered is not a future event (doing a test or producing a car) but the formation of the intention or commitment to perform a future event. The registered event still precedes the registration of this event.

Registration transactions are events with a *meaning*; their meaning is that a UoD event occurred. This is the fundamental "aboutness" of DBSs, because it says that DBS states and events are *about* UoD states and events. We just saw that the aboutness may be repeated, for example when the registered event is the formation of a commitment to do a test in the future. The registered event then has itself a meaning, a contents, which is *about* something that may or may not be realized in the future. There is no theoretical limit to the depth of this nesting of "aboutness".

A DBS that engages only in registration behavior, has no use. In order to have a *function* for its environment, a DBS must engage in other behavior. Keeping in mind that a DBS is a *reactive system*, any behavior of a DBS can only be a transaction initiated by the environment of the system, or a reaction to such a transaction. Now, the environment of the DBS consists of two things:

1. *Other systems.* These are users, other DBSs (for example in an EDI network), or other kinds of machines (in an embedded system). Of these, we regard machines and DBSs as planned response systems and users as entities who can initiate events themselves (e.g. form a commitment to do a test). Systems in the environment of the DBS may be the source of registration transactions, but they may also issue queries and commands to the DBS.
2. *Time.* Both the DBSs and all systems with which it interacts are in time. (It is curious that we can just as well say that time passes in the UoD and DBS as that we can say that the DBS and UoD exist in time.) We assume for simplicity that there is one globally accessible *clock* that always shows the correct time. This clock is the source of transactions called temporal events.

Looking first at the interaction between the DBS and time, we find that an important class of transactions is the following:

- **Temporal events** are significant moments in time. Examples are "midnight", "end-of-month" etc., which for the database system may have the significance "time to report on yesterday's transactions", "time to pay salaries", "start salary calculations" etc.. A temporal event is *about* the UoD, because time passes (also) in the UoD.

We saw earlier that the *recognition* of a temporal event may be an internal DBS process, but that the temporal event itself takes place outside the control of the DBS and must be regarded as an input event to which the DBS must react.

Looking at the interaction of the DBS with other systems in its environment, we find that two other important DBS transactions about the UoD are control transactions:

- A DBS can perform **directive control** transactions, in which it tells another system to do something. Examples are requesting for a password, requesting for an input value, sending a control signal to an elevator system, etc. These transactions may occur as response to a registration transaction, a command, a temporal event, a query, or another input transaction. Control transactions are about the UoD, for they tell a system in the UoD of the DBS to do something. The UoD includes that part of the world about which data is registered and to which commands are sent.

- A DBS can perform **declarative control** transactions, in which it declares the state of the UoD to have changed. For example, a bank computer can declare the credit limit of a customer to have changed. It uses an algorithm for this that looks at the balance history, and this algorithm is triggered by a registration transaction (e.g. depositing or withdrawing money) or by a temporal event (e.g. the end of the month). Declarative control transactions are always about the UoD.

Control transactions are output transactions of the DBS. Declarative transactions combine features of registration and control, because they *represent* a state change in the UoD, but at the same time they are the *reason* why this state change occurs. There are two important practical differences between declarative transactions and registration transactions.

1. A registration transactions always *follows* the registered event with a non-zero delay, whereas a declarative transaction is always *simultaneous* with the event it registers.

2. A registration transaction occurs because its corresponding UoD event occurred, whereas for a declarative transaction, its corresponding UoD event (the *declared event*) occurs because the declarative transaction occurs.

Both characteristics imply that there cannot be a *causal* relationship between the declared event and the its declarative transaction. The relationship between cause and effect is sequential. Because the speed of light is finite, a cause cannot occur simultaneously with its effect. The relationship between declarative transactions and the events they declare is not causal, but *conceptual*: it is defined by a conceptual model shared by the participants of the social situation. If they agree that the declared event takes place, it takes place; if they decide that it does not take place, then nothing happened. This means that the declaration and the declared event can occur simultaneously. The structure of this kind of magic is analyzed by Searle [105, 106, 104], Kimbrough, Lee and Ness [59, 58] and Wieringa [129, 133].

### 4.4.2 Transactions about the DBS

Turning now to transactions about the DBS itself, we find the following obvious kinds:

- **Queries** are questions asked about the state of the DBSs. They are *about* the DBS, but because the DBS is supposed to correctly represent the state of the UoD, they are, indirectly, also about the UoD. Queries are input transactions of the DBS.

- **Reporting** about the state of the DBS. Reports may be produced as a response to a query, but also as response to a temporal event (periodic reporting about the number of users logged in) or as a response to a registration event (confirmation of the update). Reporting is an output transaction of the DBS.

Modeling query and report transactions involves modeling the user interface, dialog structure, report layout etc.

It is tempting to classify **administrative transactions** under the same heading as the two above. Administrative transactions register the fact that certain DBS transactions have occurred, and they are thus certainly about the DBS. Administrative data is data about the DBS itself. Any transaction may cause a state change in administrative data. For example, a query can cause an update to the queried surrogates which records how often these objects were queried.

Administrative transactions are not input or output transactions of the DBS, but are *registration transaction* of a second DBS, which maintains data about the first DBS. We call this second DBS a **meta-DBS** of the first DBS. Administrative transactions are, by definition, about a DBS. However, the situation is simply that the UoD of a meta-DBS is itself a(nother) DBS. Administrative transactions should not be classified in the same category as querying and reporting.

To fulfill its task, the meta-DBS contains a *data dictionary* and a *data directory*, a *user directory*, etc. DBS transactions may cause registration events for the meta-DBS. The meta-DBS may be integrated with the DBS it represents, but logically, it is separate from it. Study of the behavior, function and structure of a meta-DBS presupposes clarity about the structure and behavior of DBSs. In the sequel, I will therefore ignore the meta-DBS in the rest of this report. This means that administrative transactions are left out of consideration.

## 4.5 The structure of information system models

### 4.5.1 The UoD model

The above analysis of DBS behavior gives support to the claim the an information system model must be divided into two parts, a UoD model, which represents transactions that are about the UoD, and a DBS model, which represents transactions about the DBS. The **UoD model** represents at least the following kinds of transactions:

- Registration transactions.

- Directive control transactions.

- Declarative control transactions.

- Temporal events.

We will take an *object-oriented* view of the UoD model and assign every transaction except temporal ones to one or more objects. If the transaction is assigned to more than one object, it will be called a *communication.*

*Temporal events* play a different role than the other transactions, for time is not an object (although clocks are). Rather, objects are "in" time and each event occurrence in the life of an object takes place at a particular time. The issue of real time and temporal events has not yet been elaborated in LCM and it does not yet play a large role in MCM.

The specification of DBS transactions includes a specification of their logic and their meaning, as well as of other properties, such as appearance (e.g. user interface), response time, error handling, etc. MCM concentrates only on the logic and meaning of DBS transactions. This is done by specifying a conceptual model of the UoD in terms of which the transactions can be interpreted. The structure of UoD models is treated in chapter 6.

### 4.5.2 The DBS model

The second part of the information system model is the **DBS model**. The DBS model represents the following transactions about the DBS:

- Query transactions.
- Reporting transactions.

Just as for other DBS transactions, query and report transactions have not only a logic and a meaning, but have other properties, such as their appearance, response time, etc. Formal query languages only deal with their logic and meaning.

I take a *declarative* view of DBS queries, in which they are specified by stating what information is needed, without saying how it must be computed. (This use of "declarative" should not be confused with the use of the same word in "declarative transactions".) This declarative view of queries contrasts with the object-oriented view taken of the UoD model.

Within the declarative approach to queries, there are still many options, but I assume that a deductive query specification language is used. LCM is a deductive model specification language that is intended to be used for specifying DBSs, but it is reasonable to assume that it can also be used to specify queries. Whether or not LCM is fit as a query language is still a topic for future research. In the rest of this report, modeling queries and reports is ignored.

### 4.5.3 Comparison with JSD

The partitioning of the DBS model into a UoD model and a DBS model is inspired by the partitioning of the model in a UoD model and a DBS model in JSD [49], but differs from it. In JSD, the UoD model is a model of registration transactions only, and the DBS model is a model of DBS *functions*. The DBS model is a specification of input functions, including timing and error processing, of output functions, including access paths and report formats, and of interactive functions. In MCM, the DBS model consists of a query model only, i.e. it is a model of only those JSD output functions that produce a report about the DBS state. Interactive functions and those output functions that perform control over the environment are transactions about the UoD, and they are therefore modeled as part of the UoD model.

The declarative view of queries taken in MCM also differs from the view of queries in JSD. In JSD, query answering is specified *imperatively* by means of *output functions*. An output function is a process with a local state that is updated by events. From an object-oriented point of view, each output function is an object with a local state and behavior, that interacts with other objects in the system. These other objects may be surrogates for UoD objects or they may be other function processes.

Now, a model that treats both surrogates and function processes as objects with a local state, local updates and message-passing capability tends to get complex. By specifying queries declaratively, we simplify the conceptual model of DBS behavior, because it eliminates a large number of processes, and we simplify the specification of queries, because we can concentrate on the result of the query only and ignore how this result must be produced.

# Chapter 5

# Representing System Boundaries

MCM is a *transaction-oriented* modeling method because it starts with a list of all possible transactions that the DBS is required to engage in. The list of possible transactions represents the **boundary** between the DBS and its environment. From this list, an object-oriented UoD model will be built. In this chapter, we discuss two ways to represent the DBS boundary, the context diagram and the function decomposition tree.

## 5.1 The context diagram

A simple way to represent the system boundary is the *context diagram* [27, 122, 140]. This shows for each system in the environment of the DBS what the transactions of the DBS with that system are. The conventions used in MCM for context diagrams are the following (figure 5.1):

- The system of interest is shown as a circle. The *type* of the system (e.g. "S") is written inside the circle, and a name of this individual system (e.g. "s") declared as variable of this type.
- Systems with which it interacts are shown as rectangles. Again, the type is written inside the rectangle and an arbitrary instance of that type is named by a variable declared in the rectangle to be of that type. One system type may be drawn several times in one context diagram.
- Each transaction is a communication between the system of interest and one or more external systems, represented by a hyperedge connecting the communicating systems, labeled with the name of the transaction. Among the communicating systems, there is at most one that initiates the communication. If there is such a system, its name is prefixed to the name of the transaction.

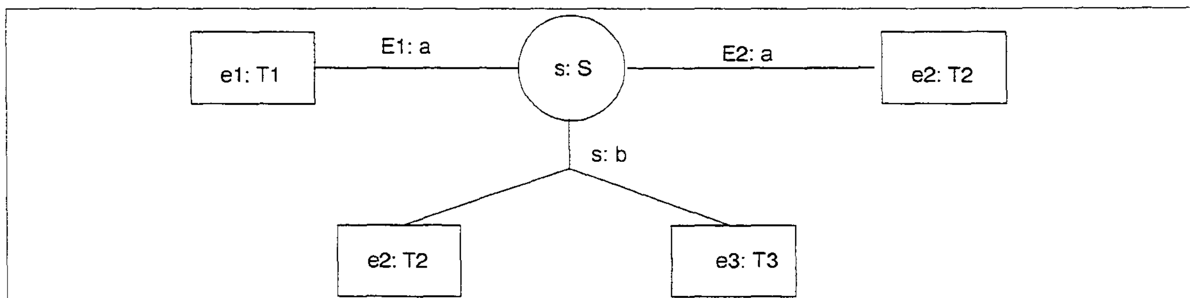



Figure 5.1: Conventions for context diagrams in MCM.

Figure 5.1 shows a system s which can receive transaction a from external systems of type T1 or T2 and can initiate a transaction b, received by systems of type T2 and T3. Figure 5.2 shows a number of context diagrams for the circulation desk of the library.

Context diagrams must be drawn whenever we want to make clear in which network of communications the system of interest is embedded. In EDI modeling and in embedded system modeling, this is of crucial importance. Often, in these systems, there is not one particular system of interest. Future extensions of MCM will deal with this situation more precisely.

Note that context diagrams only represent transactions caused or received by *systems* in the environment of the modeled system. Since time is not a system, temporal transactions can, strictly spoken, not be represented on a context diagram. We circumvent this minor problem by representing an external system called $CLOCK$ and using it as source for all temporal transactions. $CLOCK$ does not receive any output from the system. Figure 5.3 gives an example of this.

Context diagrams are taken from Structured Analysis, although there are important differences:

- In Structured Analysis, the system of interest is connected to systems in its environment by means of arrows.
- In Structured Analysis, each arrow in the context diagram represents data flow, whereas in MCM, each arrow represents a transaction. One transaction can cause many data flows, so that context diagrams used in MCM tend to have less arrows in them than context diagrams used in SA.
- In Structured Analysis, the rectangles usually represent system *types*, although this is is left unclear and one context diagram may represent individual systems as well as system types. In MCM, the rectanlges always represent individuals (although they may be arbitrary individuals of a type).
- We use the special external system $CLOCK$ to show temporal transactions on a context diagram.

## 5.2 The function decomposition tree

A **function decomposition tree** represents the set of transactions of a system by a tree, in which the leaves represent transactions and the root represent the *function* that total system behavior has for the environment of the system. The technique of using function decomposition trees to represent system boundaries has been taken from Information Engineering.

Function decomposition trees typically show all transactions of a DBS, including temporal events and queries. We will see examples of this in a moment.

### 5.2.1 Function decomposition of organizations

Function decomposition trees can be drawn for any system that has a function for its environment, including organizations. Figure 5.4 shows part of a function decomposition tree of a library. The leaves of this tree must be further decomposed, until we reach such atomic transactions as *lend*, *return*, *buy*, *lose*, etc.

The root of a function decomposition tree of an organization represents the **mission** of the organization. This is the highest-level statement of the function of the organization, and it states the reason why the organization exists at all. The mission of the library could, for example, be

> to make documents available to its members.

Mission statements should be concise.

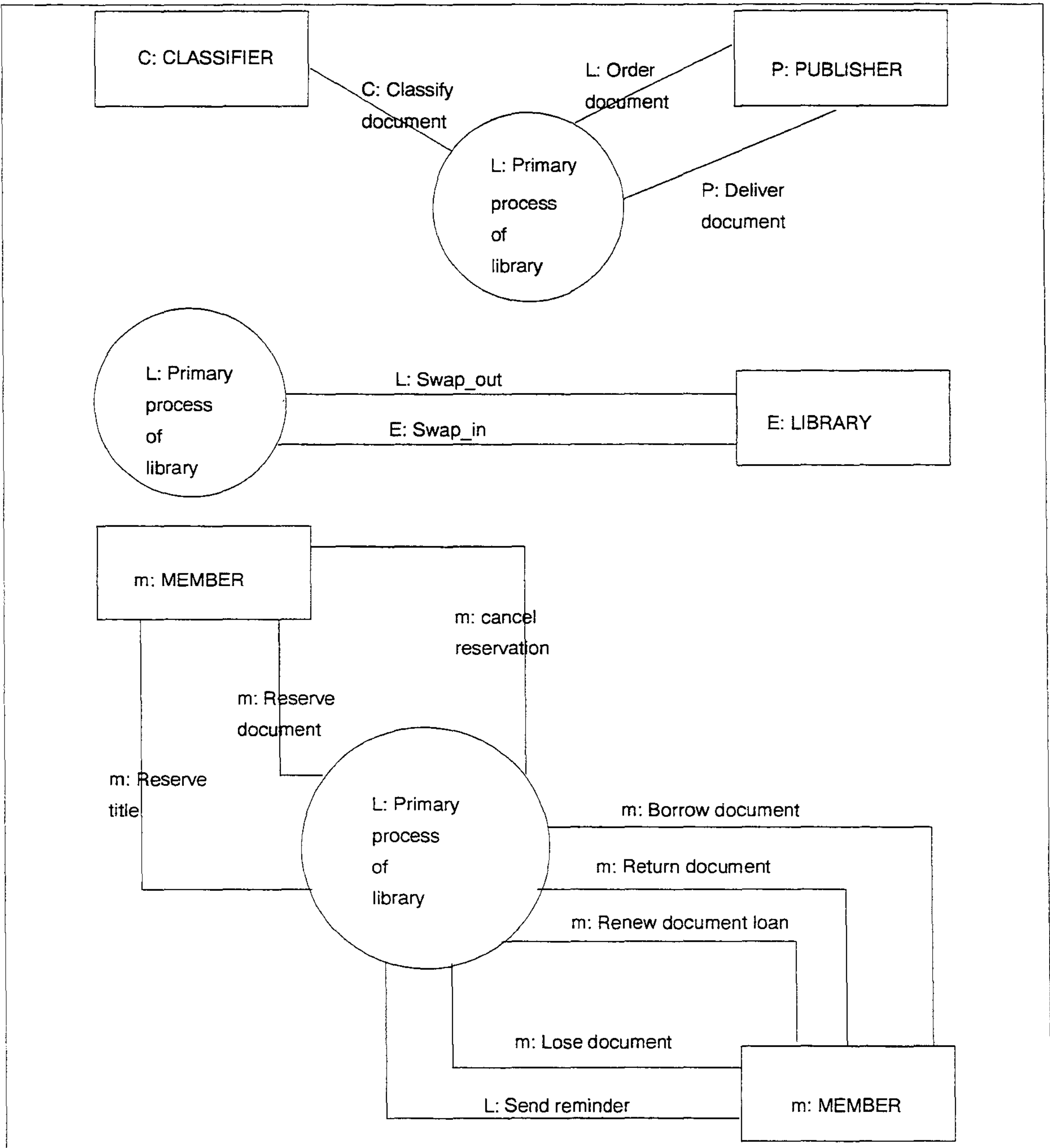



Figure 5.2: Representation of a system boundary by several context diagrams.

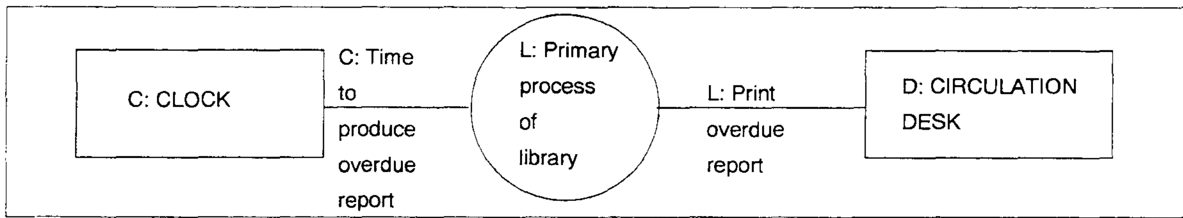



Figure 5.3: Representation of temporal events by adding a clock as external system to a context diagram.

Library Mission

- Management
  - Consultation of Member Representatives
  - Budget Planning
  - Consultation with University Board
  - Strategic Planning
  - External Reporting
- Primary Process
  - Document Acquisition
  - Document Circulation
  - Inter-library Traffic
- Finance
  - Budgeting
  - Payments
  - Auditing
- Support
  - Member Services
  - Document Preservation

Figure 5.4: Part of a function decomposition tree of the library.

An important advantage of function decomposition trees is that they relate system behavior to system function. This means that they are a good way to find the a model of the *essential behavior* of the system. (The concept of essential behavior was introduced by McMenamin and Palmer [79].) Anything in the behavior of the system that cannot be related to the function of the system is not essential; it is an artifact of the current implementation of the system.

A second important advantage of function decomposition trees is that they can be understood by all relevant parties. That is, they are very informative representation techniques. Consider the the mission statement of an organization.. This is just the root of a "tree", so it is the simplest function "decomposition tree" possible. Even this degenerate tree has a high information value, because it excludes zillions of possible behaviors from consideration in a language understandable for everyone. The mission statement of the library implies for example that the library will not engage in buying or selling books for a profit.

By elaborating the mission statement of an organization in a function decomposition tree, we make the contents of the mission statement more precise and we also show what the core business of the organization is. This increases the understanding that all parties have of the system. Note that a function decomposition tree of a system does *not* show the way the system is decomposed into subsystems. Function decomposition trees are resistant against change of implementation: they show the essential activities of the system, i.e. those activities needed to perform its function, and nothing else.

### 5.2.2 Function decomposition of DBSs

We can discover which transactions a DBS must be able to perform by making a function decomposition tree whose leaves are the required DBS transactions. This tree is a decomposition of a particular function to be realized for the environment — for example for an organization — into atomic transactions. The function decomposition tree of a DBS represents the behavior of a DBS because it lists all possible DBS transactions at it leaves. In addition, it represents the function that these transactions have for the environment, because the transactions are grouped into functions. We call the immediate parents of the transactions in the function decomposition tree **services** in `MCM`. The immediate parents of a function decomposition tree are called **domains**.indexDomain!in a function decomposition tree—bb Continuing this grouping, we reach the node of the tree, which represents the function of the DBS. The node of the function decomposition tree must be a concise version of the statement of purpose of the DBS.

Figure 5.5 shows a decomposition of the primary process of the library down to the level of transactions. All transactions in this tree are transactions of the library, i.e. atomic interactions between the library and its environment. The library environment consists of members, publishers, documents that are ordered, etc. The services in figure 5.5 are Document Acquisition, Document Circulation, and Inter-library Traffic.

Suppose we want to have a DBS whose function is

> to maintain an administration of all transactions at the circulation desk and answer all queries about these transactions.

Then each Document Circulation transaction in figure 5.5 corresponds to a registration transaction of this DBS. A function decomposition tree of this DBS is shown in figure 5.6. The decomposition tree shows a clear distinction between transactions with a meaning in the UoD (those in the Document Circulation service) and transactions with a meaning in the DBS (querying and reporting). We will see in chapter 6 (The Structure of UoD Models) that UoD-oriented transactions are decomposed as synchronous occurrences of local events in the life of different objects.

Note that using function decomposition trees to show how the transactions of a DBS are related to the function of the DBS differs radically from the *functional decomposition strategy* of Structured

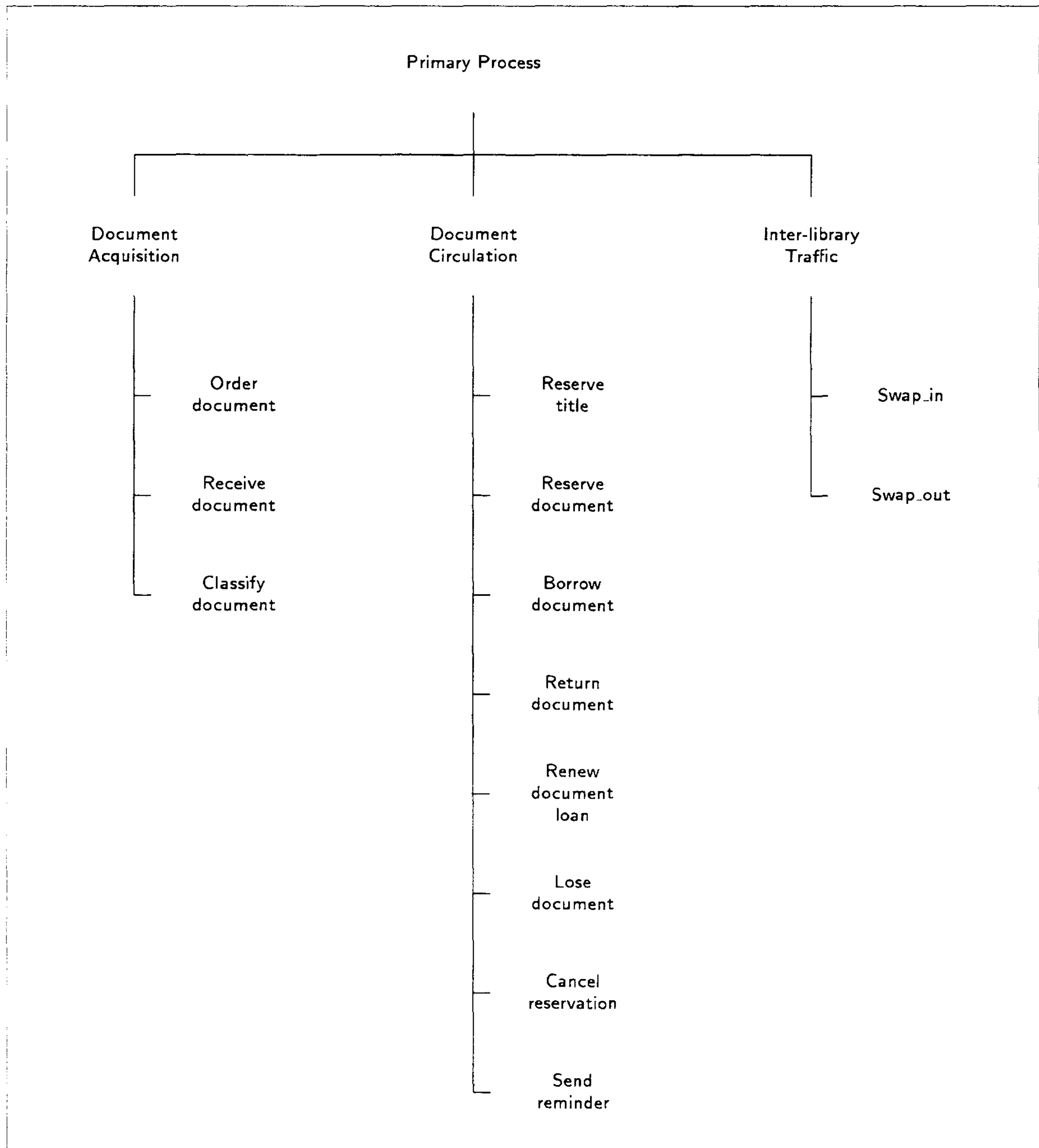



Figure 5.5: Decomposition of the primary process of a library to the level of transactions.

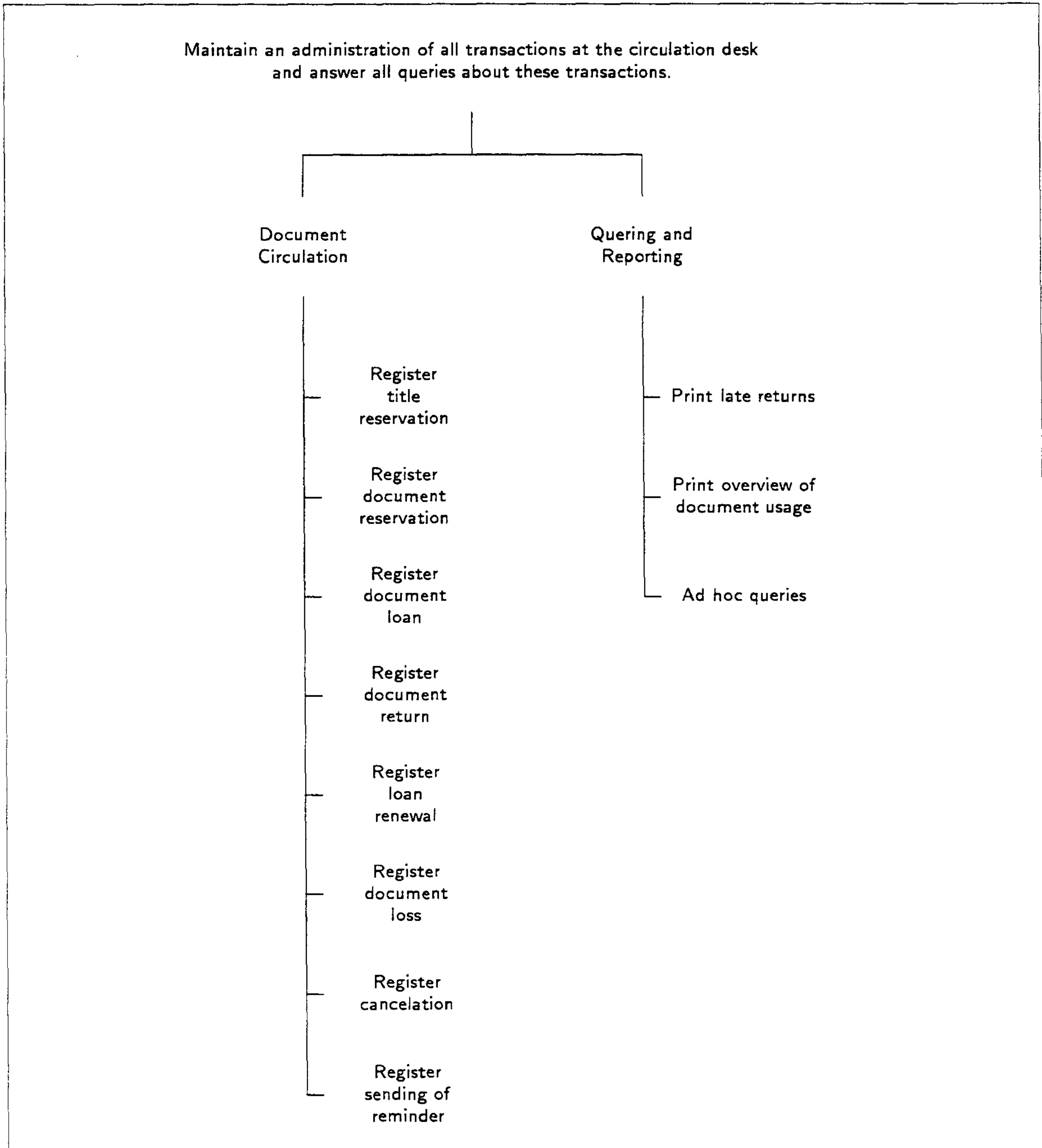



Figure 5.6: Function decomposition tree of a DBS for the circulation desk.

- **No mechanism**. A function decomposition tree does not show mechanism.
- **no job roles**. A function decomposition is independent from the jobs people do.
- **No organization structure**. A function decomposition tree is independent from the organization structure.
- **No sequence**. A function decomposition tree does not show sequences of activities.

Figure 5.7: Quality checks for function decomposition trees.

Analysis. If we would follow a functional decomposition approach to DBS modeling, we would start with DBS transactions (external events to which the system must respond) and decompose the way in which the system responds to these transactions into data transformations. This strategy *starts* from required DBS transactions. By contrast, our function decomposition *stops* when we reach the level of transactions.

In the circulation desk DBS, the distinction between registered event (which occurs in the UoD) and registration event (which is a DBS transaction) may be blurred. For example, library procedures may state that reserving a title takes place only when the DBS *registers* a title reservation. In other words, according to library procedures, a title is reserved, not when a library member requests it, but when the DBS executes a Register Title Reservation event. But this means that this is a *declarative control transaction* and not a registration transaction.

In other transactions, this confusion cannot arise. For example, the Lose document event in the UoD occurs before this fact is reported to the DBS in a Register document loss transaction. Nevertheless, even in this case, confusion may arise about whether we are talking about events in the life of the DBS or events in the life of an object in the UoD. This is because we tend to use the names for the UoD events as names for the DBS transactions. In that case, the names for the circulation desk transactions in the function decomposition tree of the DBS (used in figure 5.6) are the same as the names for the library transactions at the circulation desk (used in figure 5.5). Such a usage in turn strengthens the impression that all DBS transactions are *declarative*, i.e. that the registered event only occurs when its occurrence is registered in the DBS.

### 5.2.3 Quality checks for function decomposition trees

The checks in this subsection are taken from a practical introduction to function decomposition given by Barker and Longman [7].

A function decomposition tree represents only two things, *what* happens in the represented tool and *why* it happens. Each node of the tree represents a function of the tool, that is, a piece of useful behavior. A node $N$ itself represents *what* the behavior is, its parent represents the reason *why* the behavior is useful. Conversely, the daughters of a node $N$ represent *all* behavior necessary to achieve the behavior associated with $N$. There is no implication of sequence or mechanism in the list of daughters. More in particular, we have the following quality criteria for function decomposition trees (see figure 5.7).

- A function decomposition tree does not represent *how* behavior is performed. Activities in the tree may be performed manually, mechanically, electronically or mentally. This is not visible in the tree.
- A function decomposition tree does not represent *who* performs the activities. Nodes in the tree are not job descriptions. What any one person does may be scattered all over the tree.

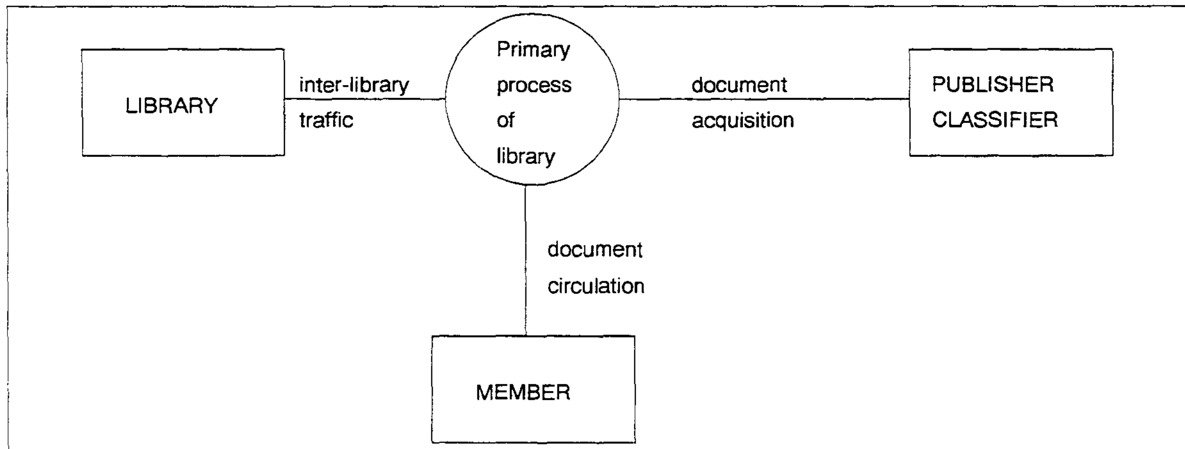



Figure 5.8: Context diagram of the primary process of the library.

- A function decomposition tree does not show *where* activities are performed. It does not show a breakdown of the organization into departments. It merely shows what must be done to achieve the mission of the organization.
- A function decomposition tree does not show *when* activities are performed. To find out which activities are the daughters of a node, one may mentally follow a sequence of activities. However, this is merely a heuristic though and the list of daughters of a node does not imply anything about the sequencing of activities.

## 5.3 Combining function decomposition trees and context diagrams

It is useful to combine function decomposition trees and context diagrams. Each context diagram should then show all transactions belonging to one node in the tree. For example, the context diagrams in figure 5.2 correspond with the three services in the function decomposition tree for the primary process of the library in figure 5.5.

We can also associate a context diagram with a higher-level node in a function decomposition tree. In order not to clutter up the diagram, we may then represent all transactions of this node (i.e. the leaves reachable from the node) by one line. For example, figure 5.8 shows a context diagram that corresponds with the primary process of the library (shown in figure 5.4). The lines in this context diagram do not represent transactions but activities, which have no (unique) initiator. In addition, each activity may involve several external systems, without implying anything about involving them conjunctively or disjunctively. All external systems that are involved in at least one transaction of the activity are written inside the external entity box. This box itself does not represent one type of external system, but is merely a visual grouping of several types of external systems. One external system type may appear in several boxes.

# Chapter 6

# The Structure of UoD Models

The UoD model represents the meaning of the DBS transactions that are about the UoD. In MCM, we specify each DBS transaction that is about the UoD as a set of one or more events. All events in a transaction occur synchronously and each occurs in the life of a different object. Note that some transactions may consist of only a single local event.

Thus, the UoD itself is thus viewed as a set of communicating objects. At each moment, the UoD is in a state that is determined by

1. the set of existing objects and
2. the state of each existing object.

A state transition of the UoD consists of

1. creation or deletion of objects and/or
2. a set of events that occur in the life of a set of objects.

Since the UoD is modeled as a set of communicating objects, in any state transition a number of communications may occur synchronously; each of these communications is itself the synchronous occurrence of a number of local events. For example, in the UoD of a library, there may occur at one particular moment a *borrow* event, that is a communication in which one $MEMBER$, one $DOCUMENT$ and one $LOAN$ object participate. Each of these participants then perform as local event, and the three local events jointly form the *borrow* event.

A state transition can also contain local events that do not participate in a communication at all. As another example, a $MEMBER$ may perform a *change_address* event in isolation, without participating through this event in a communication. The *borrow* and *change_address* events may occur synchronously, without this being a communication.

In MCM we make the simplifying assumption that each registration transaction consists only of one event, which may be a local event in the life of an object or a communication event shared by different objects. This simplification can do no harm, because a time-sharing implementation of a DBS can take care of the registration of multiple local- and communication events that occur simultaneously.

We view the DBS as an *observer* of the UoD. Each observation of the UoD made by the DBS is an observation of one local event in the UoD, or a communication between objects in the UoD. Such an observation is modeled as a registration transaction.

The DBS is not only an observer of the UoD, it is also an *actor* that may send directive or declarative control transactions to the UoD. Just as registration transactions, control transactions may be local events in the life of an object, or they may be communications shared by several objects.

A special case is the occurrence of a temporal event. Temporal events do not occur in the life of an object —time is not an object— and they require real-time techniques to be modeled. Temporal events do not yet play an important role in MCM, but a future extension will pay more attention to them.

Summing up, we will model registration and control transactions, which consist of a set of one or more events local to one or more objects. From this it should be clear that the communication model of UoD objects is the heart of the UoD model. We saw earlier already that in MCM the UoD model itself is the heart of the information system model.

In this chapter, we look at the structure of the UoD model in detail. In particular, we will look at objects and their classification, at object life cycles, and at object communication. For each modeling structure, we give a number of quality checks, that can be used to check whether good use has been made of the structure. In the next chapter, we show how all these components are drawn together in a specification of the UoD model.

## 6.1 Objects

In this report, we are only concerned with objects *in the UoD* and not with objects in the DBS. This is not a serious limitation, for I make no assumptions about what the UoD is. If we are making a meta-DBS of a DBS, then our UoD is the represented DBS and all our objects exist in that DBS. Similarly, if we are implementing a DBS specification, then many of the objects we are interested in exist in a computer, e.g. as areas on disk, as a window on a screen, etc. To keep the explanation simple in what follows, we will take UoDs outside DBSs and computers as examples.

### 6.1.1 Object identity and existence

An **object** is

1. any observable part of the world that
2. exists in a possible state of the UoD and
3. has an identity that is unique and rigid.

These three characteristics have some resemblance to the JSD entity criteria, but are not identical to it [49, page 66]. The first property eliminates abstract entities like numbers and truth values from being objects. Included are, for example, persons, trees, houses, factories, committees and employees. What is *observable* depends upon the observational powers of the observer, and an important part of this is determined by the knowledge that the observer has of the UoD. It requires normal human perceptional powers to be able to observe a tree, but it requires considerable background knowledge, partly determined by local culture, to observe a committee or a company. When we observe a company, all we see in a physical sense is people talking with each other, buildings, rooms and a central heating system. The observation of managers, secretaries, documents and library members requires knowledge about the social system in which these objects exist[1].

[1]This knowledge is of such a nature that if one has it, then one is part of the social system about which the knowledge is, or at least one is at least able to take part in it. Social knowledge merges observation with participation. This dual nature of social knowledge is studied in detail by Giddens [37], who uses insights from Schuetz [102]. On a more practical level, this dual nature of social knowledge is the principle behind the debate about whether requirements determination or conceptual modeling should be performed by domain specialists, who acquired their knowledge of the developed system by participating in it, or by "informatics specialists", who acquired their knowledge of the developed system by observing it. The answer is probably: both are needed, but in order to function well, both must acquire the other's knowledge.

The second property of objects means that the object need not exist now, but may have existed in the past, or may possibly exist in the future, etc. For each object there is at least one possible state of the UoD which the object exists, even if that state has never come to pass and will never be reached from the current state.

**Existence** is in this context synonymous with "existence for the database system" and this is in turn synonymous with "being relevant for the database system". During its existence for the database system, the database system must register events that occur in the life of the object and the DBS may send control events to the object; this is part of the meaning of "relevant for the DBS". However, it may even happen that a surrogate for an object exists in the DBS before the UoD object itself exists, or after the UoD object has ceased to exist. For example, a surrogate for a car may be created before the car is actually manufactured; and a surrogate for a document may be retained even after the document itself is lost or destroyed. In these cases, the object in the UoD is relevant for the DBS even before its existence can be registered or even after control signals can be sent to it.

The guideline to determine whether an object exists, and for whom it exists, is to ask whether it can be interacted with, and by whom. Let us fix the meaning of **existence in the UoD** by saying that an object exists in the UoD if

- the DBS can register events in the life of the object or
- can send control events to it.

Similarly, a surrogate (or any other object in the DBS itself) **exists in the DBS** exists if

- other surrogates can interact with it (e.g. through communication events), or
- the DBS can answer queries about the state of the surrogate, or
- the DBS can update the state of the surrogate with a registration event, or
- the surrogate can cause a report or a control event to be output by the DBS.

The third property of objects means that each object has a unique and rigid **identity**.

- The identity must be *unique*, which means that its identity is what distinguishes an object from all other possible objects. This means that object identity is a measure of *difference*. Two object observations are observations of different objects if the identities of the observed objects differ.
- The identity must also be *rigid*, which means that its identity is what remains the same under all possible changes of state of the object. This means that object identity is also a measure of *equality*. Two object observations are observations of the same object if the identity of the observed objects is the same.

These two identity properties are discussed by Gabbay and Moravcsik [35].

If an object would not have identity, then we could for example have objects that merge or split. This would cause problems with object identification that we want to avoid. More on object identity follows, when we discuss identifiers and keys.

### 6.1.2 Objects and values

A *system* has been defined earlier as any observable part of the world. Since objects are also observable parts of the world, they are systems. As stated earlier, this excludes unobservable entities like numbers, truth values, and even characters and strings from being objects. These abstract entities are called **values**. We may represent these abstract entities by visible symbols, but there is a difference between the symbol and what it represents. For example, we may write down the letter "a" in many different

typefonts and many different typesizes. All these different symbols represent *the* letter "a". Though there are many symbols that represent "a", there is only one letter "a".

Objects are observable, and an observation is an event in which the observed object communicates something to the observer. Objects are therefore necessarily dynamic. They have a state, can have local events and are "in" time. This means that the concept of existence is applicable to objects: at any moment, they either do or do not exist. By contrast, values do not have a state, have no local events, are "outside" time and the concept of existence is not applicable to them. It makes no sense to say of values that they do or do not exist, and there are no creation or deletion events for values. (There are of course creation and deletion events of symbols that represent values.) Objects and values both have properties; the properties of objects cannot be derived logically but must be observed, the properties of values must be derived logically and cannot be observed. I discuss the distinction between objects and values elsewhere [131]. An early treatment of the distinction is given by McLennan [69]

### 6.1.3 Classes and types

A **class** is a set of possible systems. These systems may be objects, relationships, roles, or classes, and so in MCM we distinguish object classes, relationship classes, role classes, and metaclasses. In the future, we may recognize still other kinds of classes. To keep the following definitions general, I state them in terms of systems. For concreteness, you may replace the word "system" in the definitions by "object", "relationship" or "role". For each class, we can distinguish three important sets.

1. The **extension** of a class is the set of all possible and actual systems in the class. The extension is therefore just the class itself, but I talk of "the extension of a class" to emphasize that I mean this set. The elements in the extension are called the **instances** of the class. The extension of a class $C$ is written as $ext(C)$.

2. In any state of the world, the **existence set** of a class is the set of class instances that exist in that state. The extension of the class is always the same set of instances, independently from the state of the world, but the existence set varies with the state of the world. The existence set of class $C$ in state $\sigma$ is written as $ext_{\sigma}(C)$.

3. The **intension** of a class is the set of all properties shared by all class instances. The intension of class $C$ is written as $int(C)$.

It is also useful to have the concept of a type. The following definitions remain at the informal level but have a backing in the theory of abstract datatypes.

- A **type** is a set $E$ of entities together with a set $P$ of properties shared by these entities. We call $E$ the **extension** of the type and $P$ the **intension** of the type.

For example, a pair $\langle C, I\rangle$ where $C$ is a class and $I$ the intension of the class, is a type. However, the extension of a type need not exist of objects but can also consist of values. For example, a pair $\langle V, P\rangle$ where $V$ is a set of values and $P$ the set of properties shared by those values is a type. Consequently, we can speak of an object type, a value type, etc.

There are other definitions of what classes and types are. For example, in SmallTalk [41] classes are defined as *implementations* of objects, and in Galileo [4] a class is what is called *class existence set* in MCM and a type is what is here called *class intension.* The above definition is inspired on the algebraic theory of abstract data types, in which a data type is a collection of sets together with operations on tbose sets. What is called a *value type* in LCM is usually called a *data type* in other languages. The term "data type" is not used in MCM because data are symbols stored in computer memory; in the terminology of MCM, data are objects because they are observable parts of the world.

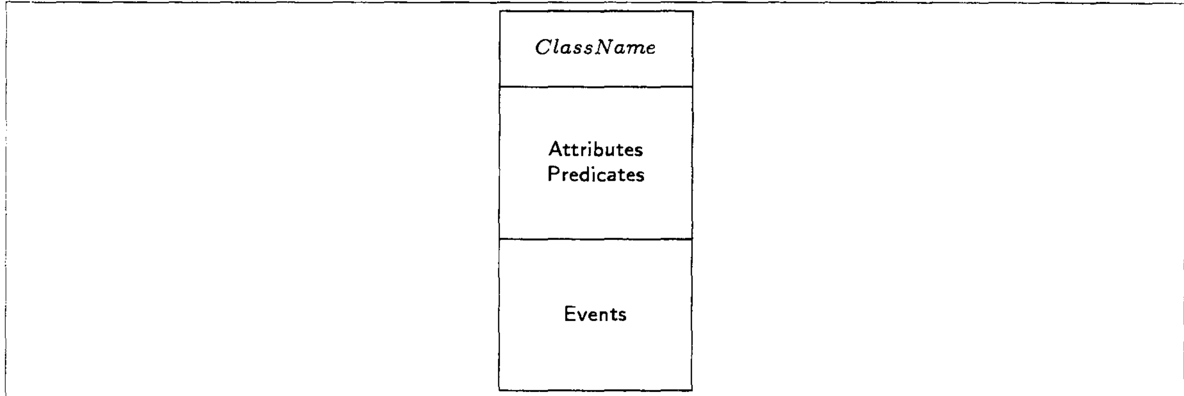



Figure 6.1: Graphical representation of classes.

Classes are represented graphically in MCM by rectangles, in which we write the class names, and attributes and predicates (if any) and events (if any) that are applicable to the class instances. Figure 6.1 shows the layout of the class boxes. Attributes and predicates show the structure of the state space of class instances, events show the structure of state transitions of class instances. What attributes, predicates and events are, is explained later in this chapter.

A diagram containing class boxes is called an **class diagram**. The class diagram is the most important part of the UoD model. It is a graphical declaration of which classes there are, what their relationships are, and what the attributes, predicates and events are that apply to their instances. The graphical convention for object classes used in MCM is derived from the convention used by Coad and Yourdon [19]. For a survey of possible class diagram notations, see Edwards and Henderson-Sellers [30].

### 6.1.4 Attributes and predicates

An **attribute** is a type of non-disturbing observation that can be made of an object. A particular observation of this type is an **attribute value**. For example, *name* and *age* can be modeled to be attributes of *PERSON* instances. This means that we can make *name* and *age* observations of *PERSON*s, i.e. *name* and *age* are types of observations that we can make of *PERSON*s. Moreover, these are non-disturbing observations, for they don't change the state of the observed *PERSON*s. Particular observations of *name* and *age* could be represented by "John" and 12.

Each attribute has a value type from which its possible values are taken, called its **codomain**. The **domain** of an attribute is the class to whose instances it applies. At any moment, an attribute is a function from the existence set of its domain (in)to its codomain. Thus, in state $\sigma$, we have $age : ext_\sigma(PERSON) \rightarrow NATURAL$.

Attributes may be *overloaded*, which means that we can have two attributes with the same name but different domains. The codomains of the two attributes may or may not be the same. For example, we can have a *name* attribute of *COW* and of *PERSON*, both with the codomain *STRING* or else one with codomain *NATURAL* and one with codomain *STRING*.

A **predicate** is a property applicable to class instances. All predicates are unary, because they are local properties of class instances. Predicate names may be overloaded.

Figure 6.2 gives a class diagram containing a single class, called *PERSON*, with some attributes. *Junior* is a predicate. No events are declared in the diagram. I follow the convention to write attribute names in lower case letters, predicates names in lower case starting with one upper case letter, and

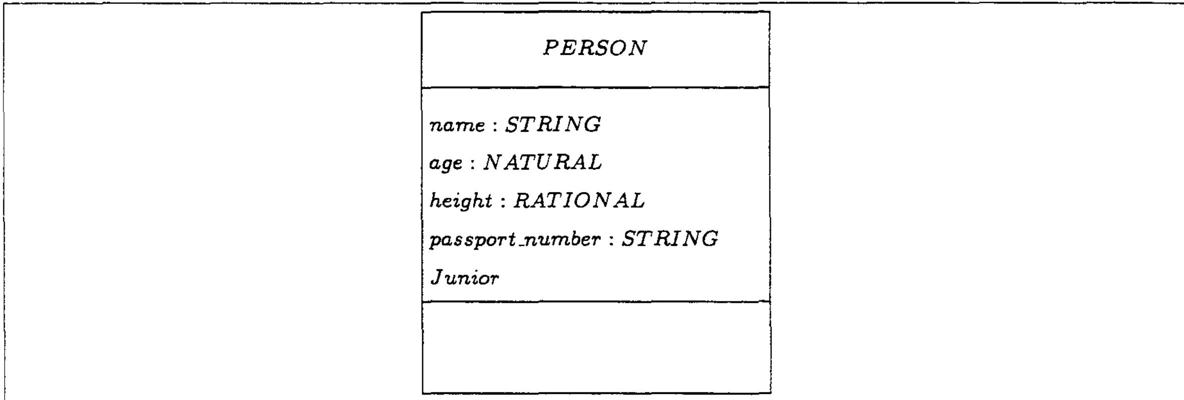



Figure 6.2: Class diagram of a *PERSON* class.

type names in upper case letters. In this convention, names are case-sensitive, i.e. "$a$" and "$A$" are two different characters.

Note that we could have defined a Boolean attribute *junior* : *BOOLEAN*. The two possible attribute values for *junior* are *true* and *false*. This is different from defining a predicate *Junior*, because the value type *Boolean* differs from the type of truth values that is assumed by predicates. The difference is of a computational nature: in each state $\sigma$ of the world, *junior* is a function $ext_\sigma(PERSON) \to BOOLEAN$ whereas *Junior* is a set of existing *PERSON*s (namely those persons that are *Junior* in state $\sigma$). Computationally, we can do different things with functions than we can do with sets. The precise nature of this difference depends upon the formal specification language used in conjunction with MCM.

### 6.1.5 Identifiers, surrogates and keys

The material in this subsection is based on a report by Wieringa and De Jonge [135]. Other material on identification can be found in Khoshafian and Copeland [57], Codd [20], Hall, Owlett and Todd [44] and Kent [56].

To make the concept of object identifier (oid) more precise, we assume a set $P \subseteq$ of all possible proper names and a set $O$ of all possible objects and we assume that these sets do not change. The sets of names and objects that actually *exist* in a state of the world are subsets of $P$ and $O$, respectively. A **naming relation** is a subset of $P \times O$. The elements of a naming relation are assignments of proper names to objects. Naming relations may change during a state transition of the world.

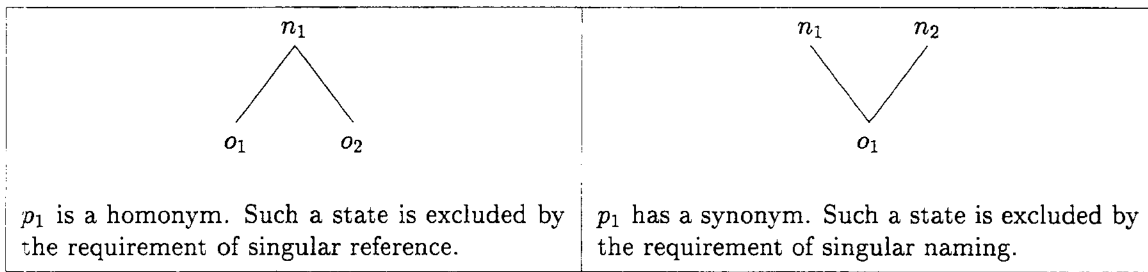
A **naming scheme** is a function that assigns to every possible state $\sigma$ of the world a naming relation $N_\sigma$. We define the domain and range of a naming relation as follows:

$dom(N_\sigma) = \{p \mid \exists o \in O \ : \ \langle p, o\rangle \in N_\sigma\}$ and
$range(N_\sigma) = \{o \mid \exists p \in P \ : \ \langle p, o\rangle \in N_\sigma\}$

We call $P$ the **namespace** of $N$ and $O$ the **objectspace** of $N$. In any state $\sigma$ of the world, there may be names in $P$ that are not assigned by $N_\sigma$ to any object in $O$, and that there may be objects in $O$ that are not assigned a name by $N_\sigma$.

Note that there may be named objects (i.e. in $range(N_\sigma)$) that do not exist in $\sigma$. For example, we may give a car a name (e.g. a serial number) before it is produced and we may continue using this name as a proper name for this car, even after the car is demolished.

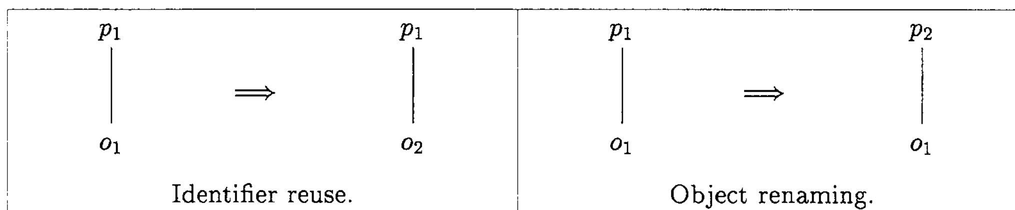
A naming scheme is called an **object identification scheme** if it satisfies the following three oid requirements.




$p_1$ is a homonym. Such a state is excluded by the requirement of singular reference.

$p_1$ has a synonym. Such a state is excluded by the requirement of singular naming.

Figure 6.3: The situations excluded by the two singularity requirements.




Identifier reuse.

Object renaming.

Figure 6.4: Two state transitions allowed by the singularity requirements but excluded by monotonic designation.

- **Singular reference**. A naming relation $N$ satisfies the singular reference requirement if in each possible state $\sigma$ of the world, $N$ is a function $dom(N_\sigma) \rightarrow O$.
- **Singular naming**. A naming relation $N$ satisfies the singular naming requirement if in each possible state $\sigma$ of the world, $N^{-1}$ is a function in $range(N_\sigma) \rightarrow P$.
- **Monotonic designation**. In each pair of successive states $\sigma_1$ and $\sigma_2$ of the world, $N_{\sigma_1} \subseteq N_{\sigma_2}$.

Figure 6.3 illustrates the violations of the singularity requirements.

Figure 6.4 shows two state transitions that are allowed by the singularity requirements but which are excluded by monotonic designation. The requirement of monotonic designation is equivalent to each of the following two requirements:

- **Rigid reference** After each state transition of the world, each proper name remains referring to at least the same object(s) as before.
- **Rigid naming**: After each state transition of the world, each object remains named by at least the same proper name(s) as before.

An example of an identification scheme is the employee numbering scheme in a company. The namespace of this scheme could be the set of all natural numbers, and the objectspace could be the set of all possible employees of the company. Objects can have identifiers because objects have, by definition, identity.

The three identifier requirements are very hard to ensure in practice, because people can commit fraud, mistakes are made in the assignment of identifiers, etc. What happens in practice that for each type of object identifier, there is a *generator* that generates a fresh identifier when it is needed, unused before, and a *distributor* of identifiers, who assigns identifiers to systems (objects, relationships, roles,

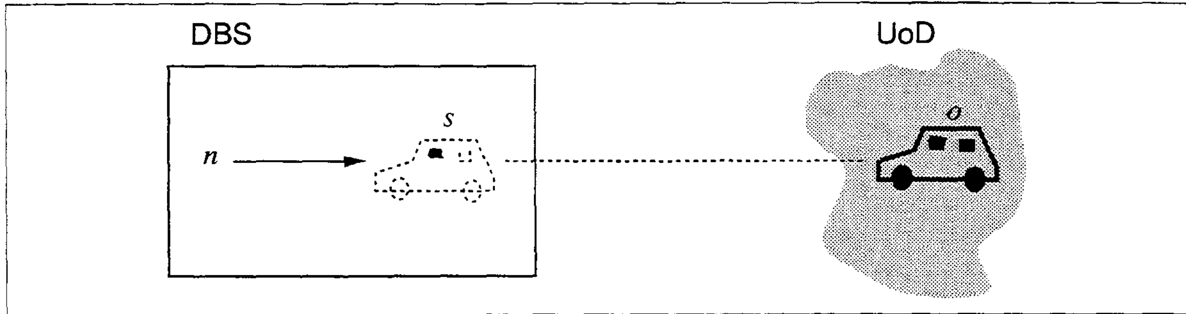



Figure 6.5: An external (UoD) object $o$ is represented by an internal (DBS) object $s$, which is identified by an internal identifier $n$. The internal object $s$ is a surrogate for the external object $o$.

etc.) that enter an existence set. The generator can make mistakes by generating an identifier that has already been used, and the distributor of identifiers can make mistakes by assigning an identifier to systems that already have one. In addition, people can commit fraud etc., which may cause one object to have two identifiers (violating name uniqueness), one identifier to name two objects (violating singularity) or a naming relationship between identifiers and objects to change (violating monotonic designation). We are not concerned here with mistakes, fraud etc. connected to identifiers, nor with issues of privacy that come up if we use global identifiers for people. We make some remarks about this are elsewhere [135].

With respect to any DBS, we distinguish internal identifiers from external identifiers. An **internal identifier** of a DBS is an identifier that is invisible to users of the DBS. Internal identifiers are always generated by the DBS itself. The namespace of internal identifiers is unknown to DBS users. We will see below that the domain of internal identifiers may be any set of objects internal or external to the DBS. An **external identifier** of a DBS is an identifier visible to users of a DBS. External identifiers may very well be generated by the DBS, but more often, they are generated by an organization such as a company (employee number), a government agency (social security number), etc. The namespace of external identifiers is known to users of the DBS and the objectspace of external identifiers is usually some set of objects in the real world.

Whether an identifier is internal or external, it must be assigned to a system of which it will be a proper name for the rest of time. We don't care how this is done. Internal identifiers may be assigned to objects in the UoD by a database administrator or an operator of the DBS. External identifiers are assigned to objects by people or by organizations, and the result of this assignment may be communicated to the DBS later.

For example, the DBS may generate a fresh number, invisible to the DBS user, but notify the user that it has generated one. The user may then decide to assign this identifier to an object in the UoD, e.g. a car, and act accordingly, e.g. fill in attribute values for this object. An internal identifier used this way identifies an internal DBS object that itself is a **surrogate** for an external object in the UoD (see figure 6.5).

When we have a class, say $PERSON$, then in a UoD model the class instances are actual or possible persons. An attribute like $age$ is a function $ext_\sigma(PERSON) \rightarrow NATURAL$ that, in each state of the world, accepts an existing element of $PERSON$ and delivers an element of $NATURAL$. If $age$ is interpreted in a DBS model, it is still a function $ext_\sigma(PERSON) \rightarrow NATURAL$, but the instances of $ext_\sigma(PERSON)$ are now internal objects, which are surrogates for real persons. It is these surrogates that are identified by internal person identifiers.

In the formal semanntics of LCM specifications, we make no distinction between the instances of a class $C$ and their internal identifiers. Intuitively, though, the picture in figure 6.5 should be kept in

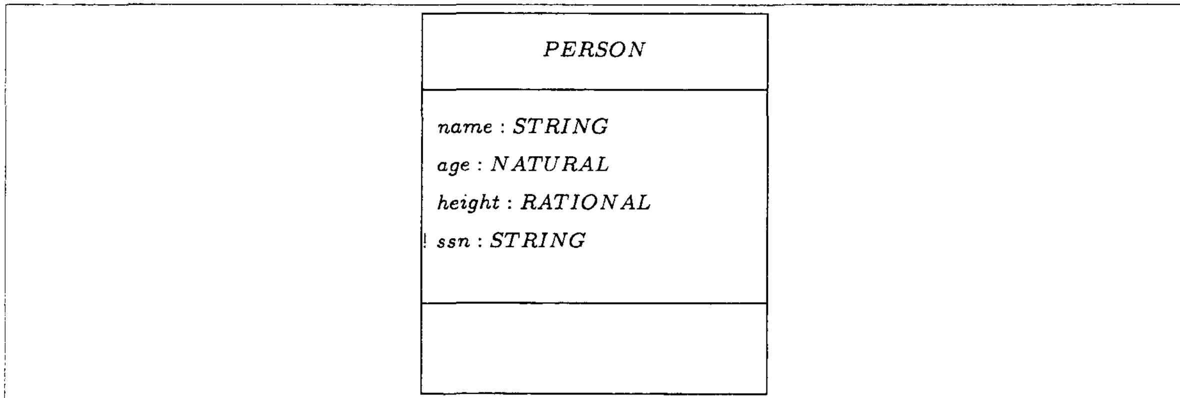



Figure 6.6: Class diagram of a *PERSON* class with an external identifier indicated.

- **Observability**. Objects are observable, values are not.
- **State**. Objects have a state, values have not.
- **Identity**. Objects have an identity independent from their state, values *are* their identity.
- **History**. Objects have a history and values have not.
- **Interaction**. Objects can interact with other objects, values cannot interact with anything.

Figure 6.7: Quality checks to distinguish objects from values. These can be used to determine whether good use has been made of the modeling constructs.

mind.

In addition to having internal identifiers, we may want to define an external identifier for objects, such as *ssn* for *PERSON*. This can be indicated in a class diagram by prefixing the name of the attribute whose value is an identifier with an exclamation mark (an inverted "i", which looks like the "i" of "identifier"), as shown in figure 6.6.

A **key** of class $C$ is a set of one or more attributes declared for instances of $C$ such that, in each state of the world, the existing instances of $C$ have a combination of values for these attributes that is unique among existing objects of $C$. For example, suppose we have an object class *ADDRESS* with attributes *city*, *street*, *nr* and *zip*, then (in The Netherlands), the set $\{zip, nr\}$ is a key of *ADDRESS*. Keys are different from identifiers, for

- they are not unique among all *possible* objects, but among all existing instances of one class, and
- they are only required to be unique in each possible state of the world, not across all possible states.

We analyze differences between keys, surrogates and identifiers in detail elsewhere [135]. Keys are mentioned here for nostalgic reasons. LCM contains a facility to define keys.

### 6.1.6 Quality checks for objects and values

Figure 6.7 gives a number of checks that can be used to distinguish objects from values. The checks all follow from the difference between observability and unobservability. In a particular UoD model, it

is often a modeling decision to represent a particular entity as a value or as an object, but we can use the quality checks of figure 6.7 to find out whether we made the right decision. To illustrate, consider the problem whether to model an address of a person as a structured value, e.g. as a string, or as an object. The list of observability checks says the following about the decision.

Viewed as a string, the address is an abstract, unobservable entity. Any properties that the address has, it has as a value. For example, it has a length (number of characters) and consists of certain characters in a character set. In addition, viewed as a value, the address has no state. Viewed as an object, on the other hand, the address has a state, which consists of, for example, a city, a street, a number and a zip code. In addition, the address also has an identity. This means that the address could be in a different state (have a different city, zip code, street name etc.) without losing its identity. That allows changes of city name, street name, house numbering scheme or even zip code without changing anyone's address. Such changes would not be not possible if the address were modeled as a value. In that case, there would be no changes of address state but simply replacements of one string by another. Changes in zip code or house numbering scheme would be indistinguishable from the case where people move from one address to another. Finally, as an object, an address has at any moment a history of such state changes, and each of these changes involves an interaction with another object, such as a government agency. This history and this interaction cannot be modeled when the address is modeled as a value — unless this value is somehow coupled to an identifier, which would turn the value into a representation of the state of the object.

## 6.2 Relationships

### 6.2.1 Identification and existence

A **relationship** is a tuple of systems, called the **components** of the relationship. Because objects and relationships are systems, we can have relationships between objects and/or relationships (although the second possibility is uncommon). The concepts of class attribute and predicate are general enough to be applicable to relationships, so we can talk about a **relationship class** and **relationship attribute**, **relationship predicate**.

In a DBS, a relationship *surrogate* is a tuple of internal objects that represents a relationship in the UoD. A **relationship identifier** is the tuple of identifiers of the components of the relationship. A relationship identifier is therefore structured, i.e. it exists of a tuple of identifiers, one for each component of the relationship. This is represented in a class diagram by drawing fat arrows from the relationship class to the component classes (figure 6.8). These arrows represent projection functions from the extension of the relationship class to the extension of the component classes. Each arrow is named by the role that the component plays in the relationship.

For example, figure 6.9 shows the $LOAN$ relationship class between $DOCUMENT$ and $MEMBER$. Relationships may relate elements of the same domain, as shown in figure 6.10. This is often called a *recursive relationship*.

Relationships are subject to the **component existence constraint**, which says that the relationship cannot exist if its components do not exist. The dependence is one-sided, for in general, the components can exist without any existing relationship among them. For example, a library member and a document can exist without standing in a $LOAN$ relationship, but a $LOAN$ instance between member $m$ and document $d$ can only exist if $m$ and $d$ exist.

### 6.2.2 Cardinality constraints and many-one relationships

Constraints are discussed more fully in section 6.8, but an important class of constraints must be discussed here. In general, a **constraint** on a DBS is a norm that an implementation of the DBS

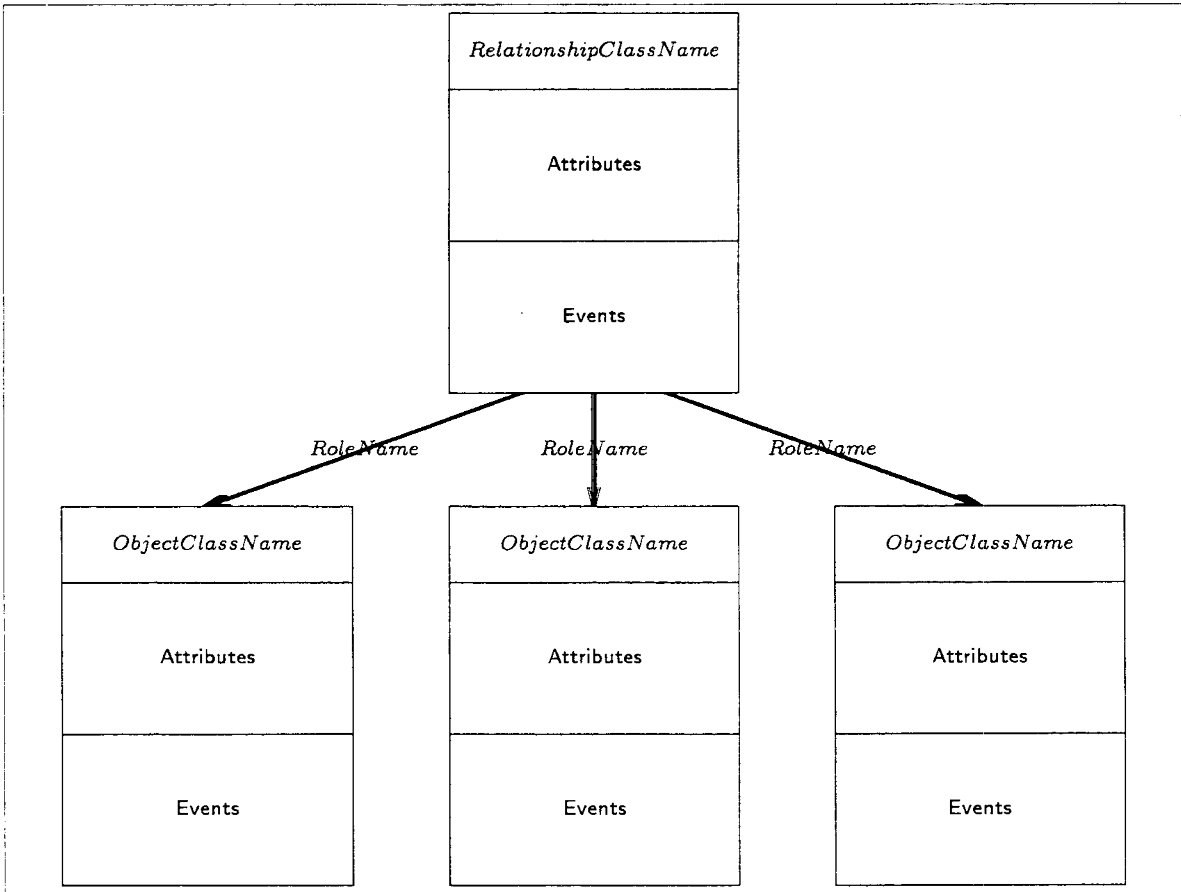



Figure 6.8: Class diagram of a relationship. The fat arrows are projection functions from the extension of the relationship class to the extension of its components.

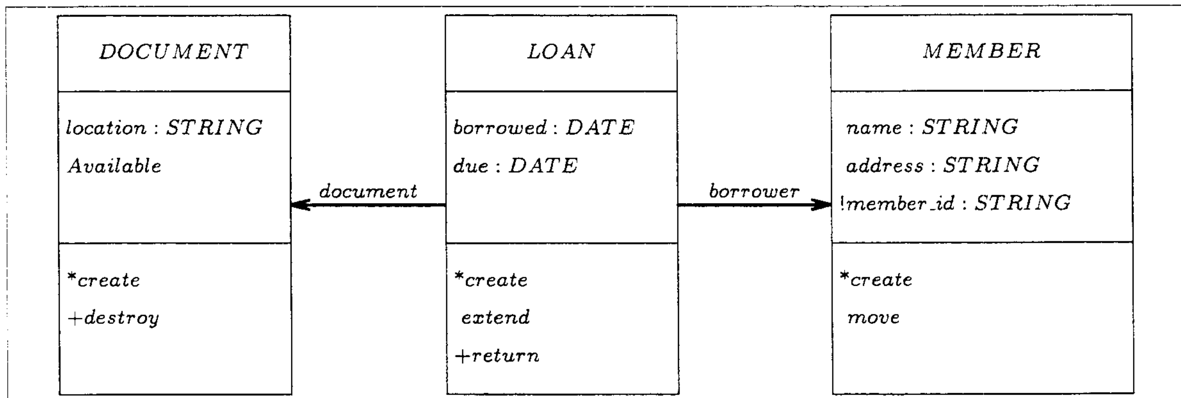



Figure 6.9: A relationship class. The * and + symbols indicate creation and deletion events, respectively.

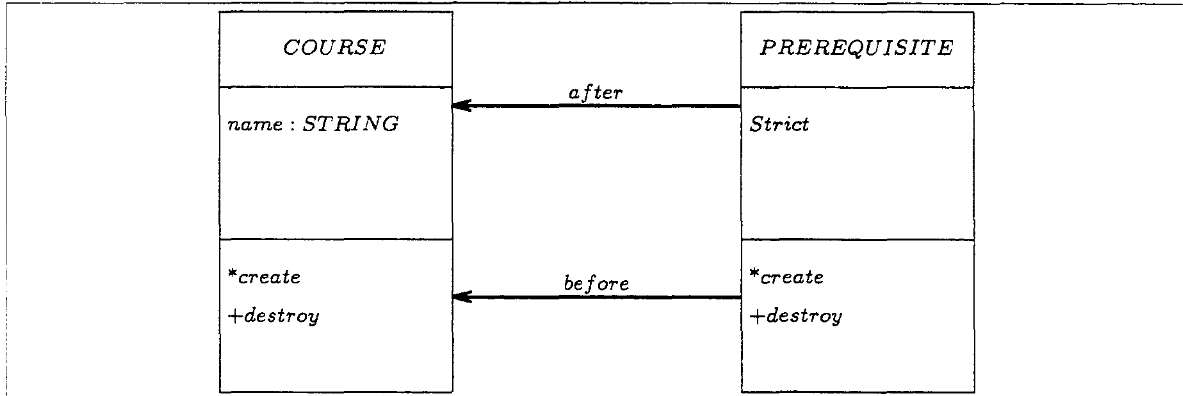



Figure 6.10: A "recursive" relationship class.

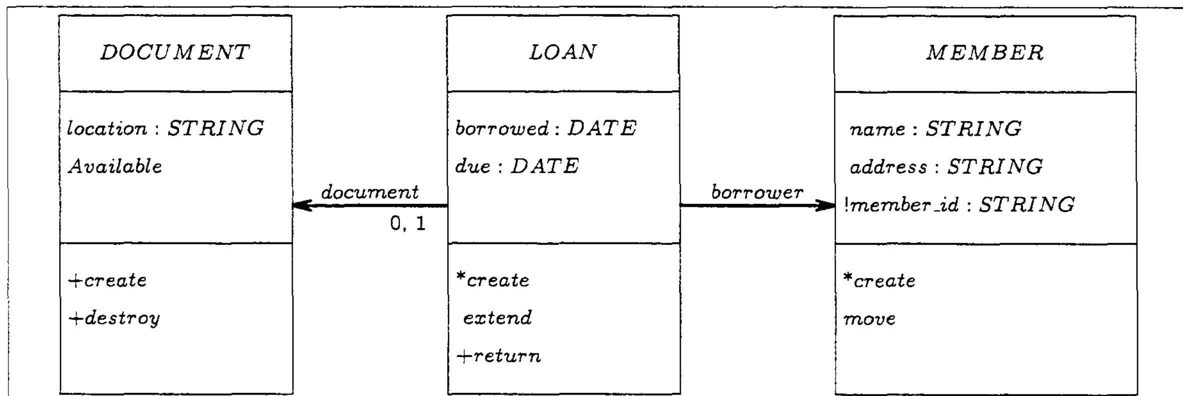



Figure 6.11: Representation of cardinality constraints. Each existing *DOCUMENT* is related by *document* to 0 or 1 existing *LOAN*s and each existing *MEMBER* is related by *borrower* to any number (including 0) of existing *LOAN*s.

should fulfill. A **cardinality constraint** is a norm that restricts the number of instances one can find if one traverses an arrow in a class diagram in reverse direction (from the arrow head to the arrow tail). Cardinality constraints are represented in a class diagram by writing a set of natural numbers at the tail of an arrow (figure 6.11). In general, a cardinality constraint is always represented by a set $c$ of natural numbers, which is written at the tail of an arrow. If there is an arrow labeled $C_1 \xrightarrow{a} C_2$ then the meaning of the cardinality constraint $c$ at the root of this arrow is as follows:

> In each possible state of the UoD, for each existing instance $o$ of $C_2$ there are $n$ existing instances of $C_1$ mapped by $a$ to $o$, where $n \in c$.

If we write $a_\sigma$ for the interpretation of attribute $a$ in state $\sigma$

$$a_\sigma : ext_\sigma(C_1) \rightarrow ext_\sigma(C_2),$$

then the cardinality constraint $c$ at the tail of $a$ says that for all $o \in ext_\sigma(C_2)$,

$$\mid ext_\sigma(C_1) \cap a_\sigma^{-1}(o) \mid \in c.$$

There are a number of frequently occurring cardinality constraints, that are not represented as a set:

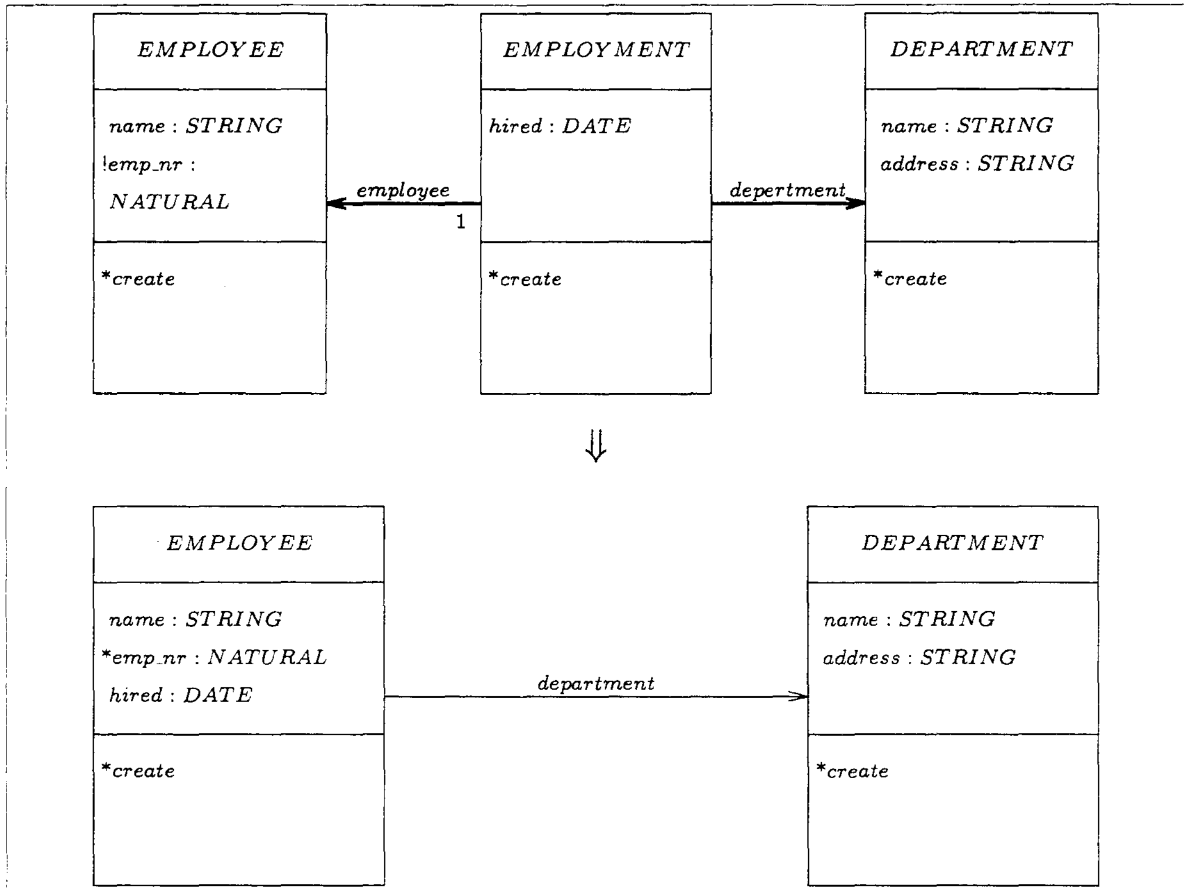



Figure 6.12: Representation of a many-one relationship as a thin arrow.

| Cardinality constraint | Represented as |
|---|---|
| { 0, 1 } | 0, 1 |
| { 1 } | 1 |
| {1, 2, 3, ...} | ≥ 1 |
| {0, 1, 2, 3, ...} | (Omitted from the diagram) |
| $\{n_1, \ldots, n_2\}$ | $[n_1, n_2]$ |

For example, figure 6.11 contains the following cardinality constraints:

- In each state of the world, each existing *DOCUMENT* instance is related to at most one *LOAN* instance.
- In each state of the world, each existing *MEMBER* is related to arbitrarily many (including possibly 0) existing *LOAN* instances.

A binary relationship of which one cardinality is 1, is called a **many-one** relationship. It can be represented as an arrow, as shown in figure 6.12. This involves moving attributes from the relationship to the component on the one side of the relationship. This is a small price to pay for the visual simplicity achieved by replacing the relationship box by an arrow.

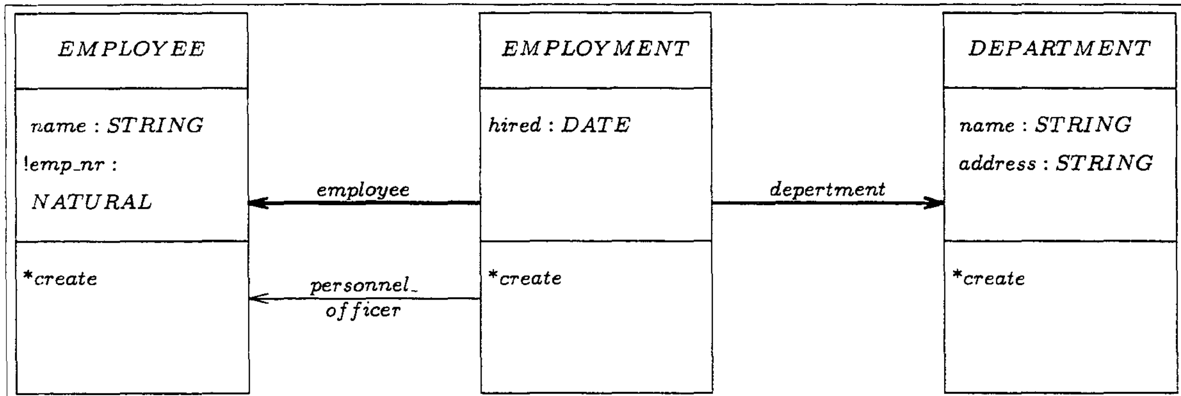



Figure 6.13: A relationship class may be related by a thin many-one arrow to another class.

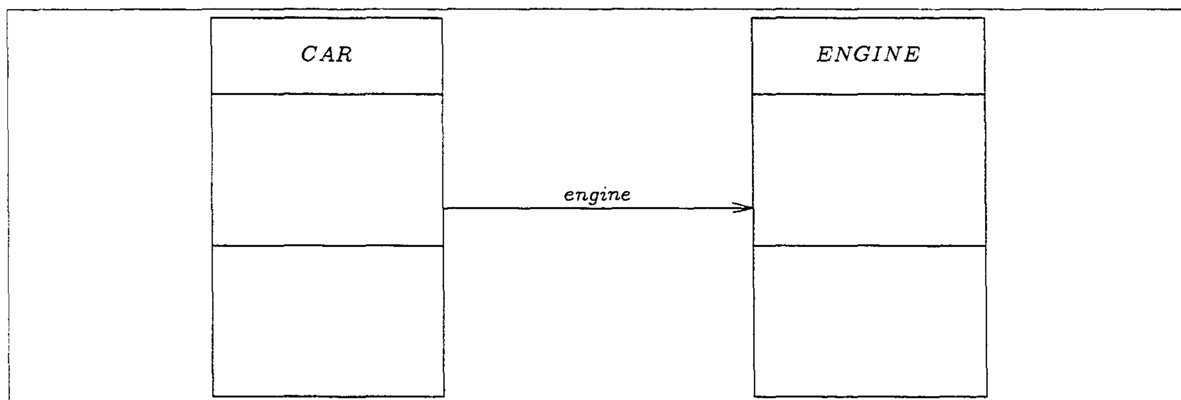



Figure 6.14: Representation of a part-whole relationship. The arrow is thin because a *CAR* can change *ENGINE*s without changing its identity.

Note that the resulting arrow is thin. This is meant to indicate that there is no dependent identity from *EMPLOYEE* to *DEPARTMENT*. We can have models in which a relationship class is itself related by a thin many-one arrow to another class, as shown in figure 6.13. The figure shows an *EMPLOYMENT* relationship for which a personnel officer is responsible. The personnel officer can be replaced without affecting the identity of the *EMPLOYMENT* relationship, so it is not a component of the *EMPLOYMENT* tuple.

Many-one arrows can be used to represent part-whole relationships between components and aggregates, as shown in figure 6.14. *CAR* is often called an **aggregate** class. By using many-one arrows, no special constructs are needed for the representation of aggregates.

### 6.2.3 Quality checks for relationships versus objects

The only difference between objects and relationships is that relationships have a structured identity and objects have an unstructured identity. The identity of a relationship is a tuple of the identity of its components. This gives us the two quality checks in figure 6.15.

To give examples, consider the following object classes.

- **Dependent relationship identity**. The identity of a relationship is the tuple of the identities of its components.
- **Independent entity identity**. Objects have their own identity, that is not composed of another identity.

Figure 6.15: Identity checks to distinguish ER entities from relationships.

| Relationship class | Components |
|---|---|
| *LOAN* | Member, document |
| *WRITE_PERMISSION* | User, file |
| *PART_OF* | Car, engine |
| *MEMBERSHIP* | Student, student society |
| Object class | Components |
| *DEPARTMENT* | Department members |
| *CAR* | Engine, wheels, ... |
| *SCHOOL* | Faculties |

The instances of *all* the classes in the left-hand column of the table can be viewed as having *components* in some sense of the word. The difference is that instances of the first four types are *identified* by means of their components, whereas instances of the last three types are identified independently from the identity of their components. A car does not become another car when parts are replaced, a school keeps its identity when faculties are added or deleted, and a department does not become another department when members leave it. A *LOAN* relationship, on the other hand, is destroyed and replaced by another *LOAN* relationship if we "replace" one of its components by different component.

## 6.3 Taxonomic structures

### 6.3.1 Static subclasses and inheritance

We say that $C_1$ is a **subclass** of $C_2$ if the extension of $C_1$ is a subset of the extension of $C_2$. We then write $C_1 \xrightarrow{is_a} C_2$. If we write $ext(C)$ for the extension of $C$, we have

$$C_1 \xrightarrow{is_a} C_2 \Leftrightarrow ext(C_1) \subseteq ext(C_2).$$

In MCM, we require a subclass always to be element of a partition of its immediate superclass, and represent this as shown in figure 6.16. We say that $C_1, \dots, C_n$ is a **static partition** of $C_0$ if

$$\bigcup_{i=1,\dots n} ext(C_i) = ext(C_0),$$

$$ext(C_i) \cap ext(C_j) = \emptyset \text{ for } i \neq j.$$

The meaning of the word "static" will become clear in the next subsection. A partition of a class in subclasses can have any finite number of elements, as long as it contains at least two elements. We may have any finite number of partitions per class.

There is an invisible cardinality constraint in the diagram. Each subclass is related by an *is_a* relationship class to its immediate superclass. The cardinality that is invisibly present at the start of the *is_a* arrow from subclass to superclas is $\{0, 1\}$. This invisible constraint expresses the fact that each existing instance of the superclass is related to at most one instance of a subclass. The fact that

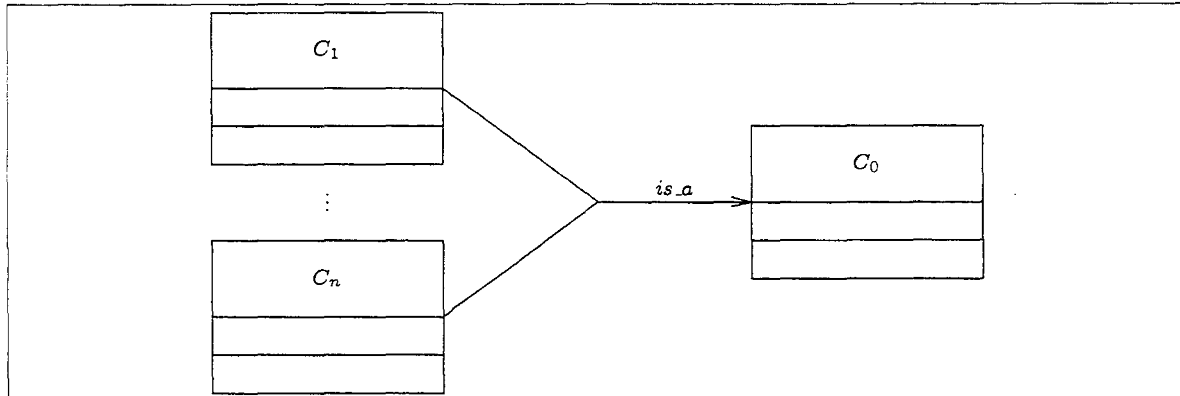



Figure 6.16: *is_a* is a relationship class that relates all subclasses in a partition to their immediate superclass.

each existing instance of the superclass is *identical* (has the same identifier as) at most one instance of a subclass, is expressed by the *is_a* label of the arrow.

Figure 6.17 shows a class with two partitions. Each pair of partitions of a class creates a number of intersection classes, so the number of intersection classes tends to grow exponentially in the number of partitions per class. For example, in figure 6.17 we have intersection classes

$CAR * DIESEL$,
$CAR * GAS$,
$TRUCK * DIESEL$,
$TRUCK * GAS$,
etc.

These intersection classes are not shown in the diagram, but they are considered to be part of the model. The number of subclasses grows even bigger when we partition subclasses into smaller subclasses, etc. All intersection classes that can be formed this way are considered to be part of the model.

The *is_a* arrows in the diagram form a binary relation on classes (those visible in the diagram and those implied by all possible intersections), and we require the reflexive transitive closure of this relation to be a partial order on classes.

All classes can be partitioned into subclasses, including relationship classes. Figure 6.18 shows a relationship class partitioned into two subclasses. As before, this means, among others, that

$ext(TEMPORARY) \subseteq ext(EMPLOYMENT)$ and
$ext(PERMANENT) \subseteq ext(EMPLOYMEN)T$.

The identifiers of $TEMPORARY$ and $PERMANENT$ relationships consist of exactly as many components as the identifiers of $EMPLOYMENT$ relationships (they *are* $EMPLOYMENT$ relationships), but it is possible that $TEMPORARY$ and $PERMANENT$ relationships have more attributes, or otherwise have more properties, than $EMPLOYMENT$ relationships have in general. A relationship is not specialized by adding more components to its identifiers, but adding more properties: more attributes, predicates, events or constraints. This agrees with the well-known Cardelli-like record specialization mechanism [13], except that Cardelli's records have no identifier. This means that Cardelli's records are structured values instead of objects in a state.

When $C_1 \xrightarrow{is_a} C_2$ and $o$ is an instance of $C_1$, then $o$ is an instance of at least two classes, $C_1$ and $C_2$. Remember that the intension of a class is the set of all properties shared by all its instances.

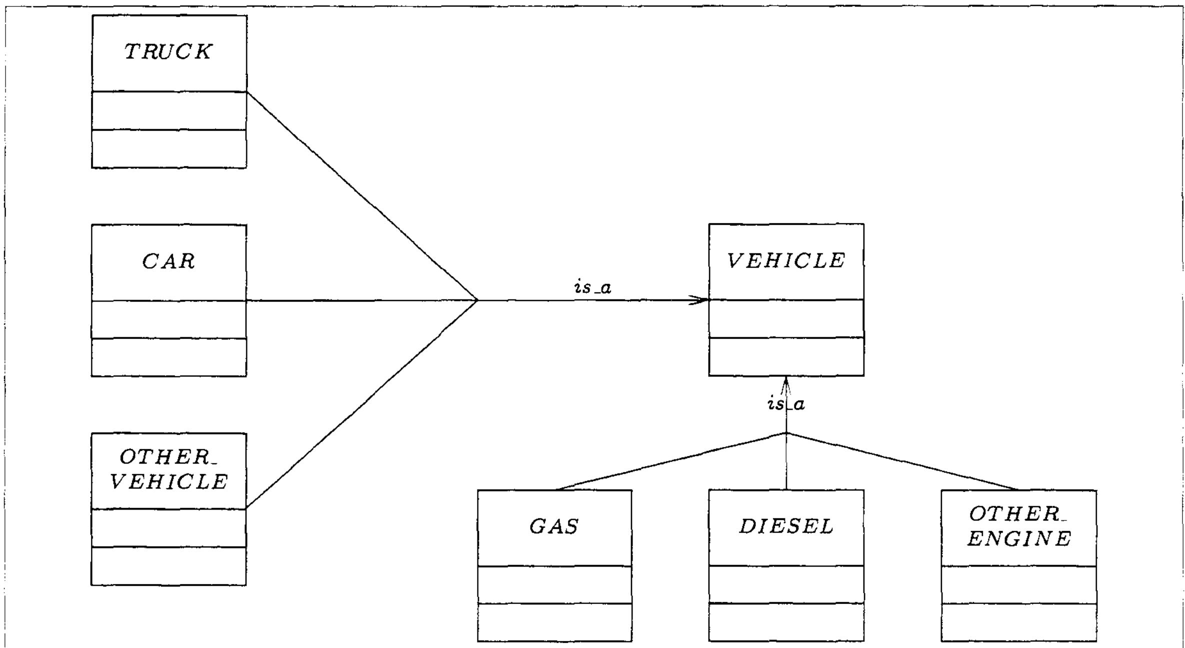



Figure 6.17: Two partitions of one class.

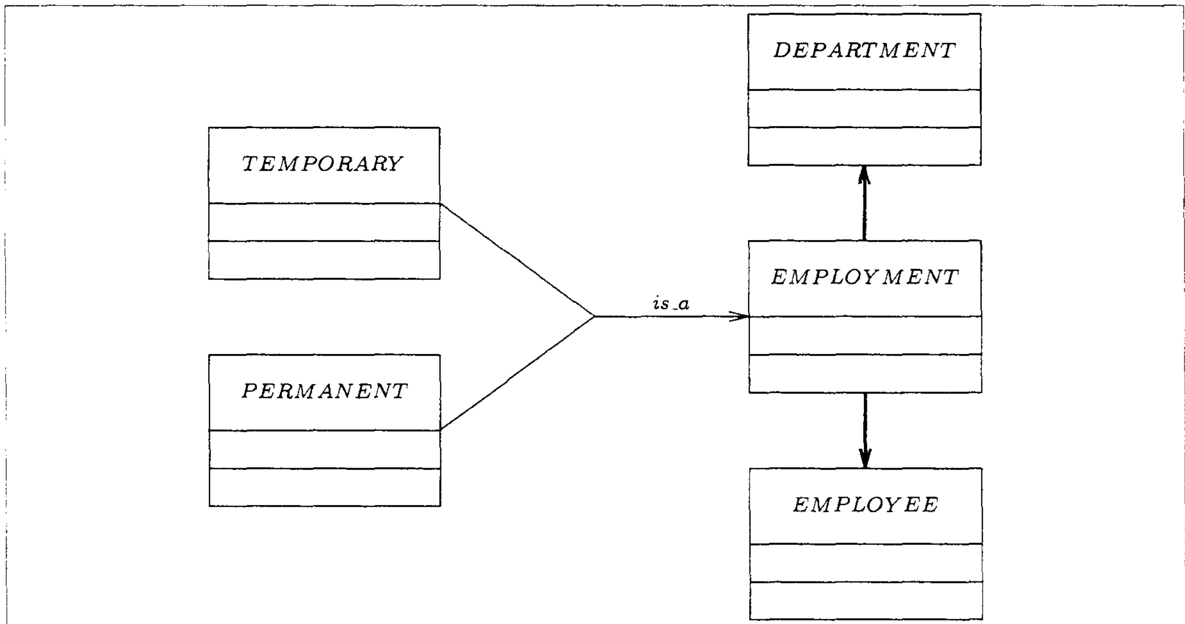



Figure 6.18: A partition of a relationship class.

Then if $o$ is an instance of $C_1$, it has all the properties in the intension of $C_2$ as well as those in the intension of $C_1$. This is called **inheritance** of properties. We can make this a bit more explicit by using the notation $int(C)$ for the intension of $C$. Then

$$C_1 \xrightarrow{is_a} C_2 \Rightarrow int(C_2) \subseteq int(C_1).$$

The subset relationship between intensions is the reverse of the subset relationship between extensions.

The *is_a* relationship was introduced in data modeling by Smith and Smith in 1977 [113] and is used widely in the semantic modeling approach [47, 88]. It is also central in the object-oriented modeling approach.

### 6.3.2 Dynamic subclasses and inheritance

The subclass relation defined so far is *static*, because an object can never move from one subclass to another. We need in addition a *dynamic* subclass relation, that allows objects to migrate from one subclass to another. For example, a person may become a student and after a while ceases to be a student again.

To define dynamic subclasses, let $C$ be a class, $\sigma$ be a state of the world, and $ext_\sigma(C)$ be the existence set of $C$ in $\sigma$. We say that $C_1, \ldots, C_n$ is a **dynamic partition** of $C_0$ if in each possible state $\sigma$ of the world, we have

$$\bigcup_{i=1,\ldots n} ext_\sigma(C_i) = ext_\sigma(C_0),$$

$$ext_\sigma(C_i) \cap ext_\sigma(C_j) = \emptyset \text{ for } i \neq j$$

and there are different states $\sigma_1$ and $\sigma_2$ such that

$$ext_{\sigma_1}(C_i) \cap ext_{\sigma_2}(C_j) \neq \emptyset \text{ for some } i \text{ and } j \text{ with } i \neq j.$$

The first two requirements mean that in each state of the world, dynamic subclasses partition their immediate superclass, and the third requirement means that there are at least two different states of the world such that if the world changes from one of these states into the other, at least one object moves from one subclass to another.

A subclass is called **dynamic** if it is possible that its existence set changes without any change in the existence set of a superclass. For example, it is possible that the existence set of $STUDENT$ changes but that the existence set of $PERSON$ does not change (because a person becomes a student or ceases to be a student). In that case, $STUDENT$ is a dynamic subclass of $PERSON$.

Theoretically, it is possible to define a static partition of dynamic subclass. In MCM, this is prohibited. This simplifies the models considerably without excluding essential modeling capabilities.

If $C_1$ is a dynamic subclass of $C_2$, we write $C_1 \overset{is_a}{\dashrightarrow} C_2$. Dynamic partitions are represented by dashed *is_a* arrows, as illustrated in figure 6.19. The dashed arrows mean that in each state of the world, the existence set of $VEHICLE$ is partitioned by the existence sets of $ACTIVE$ and $WRECK$, so that each $VEHICLE$ is either $ACTIVE$ or a $WRECK$. However, in a state transition, a $VEHICLE$ may move from one of these subclasses to the other. (If there is a constraint that it can only move from $ACTIVE$ to $WRECK$ and not from $WRECK$ to $ACTIVE$, then this must be represented in the life cycle of a $VEHICLE$ class.) Note that by inheritance, each subclass of $VEHICLE$ is also partitioned into an $ACTIVE$ and a $WRECK$ subclass. This gives dynamic intersection classes like the following:

$ACTIVE * CAR$,
$ACTIVE * TRUCK$,
$WRECK * CAR$, and
$WRECK * TRUCK$.

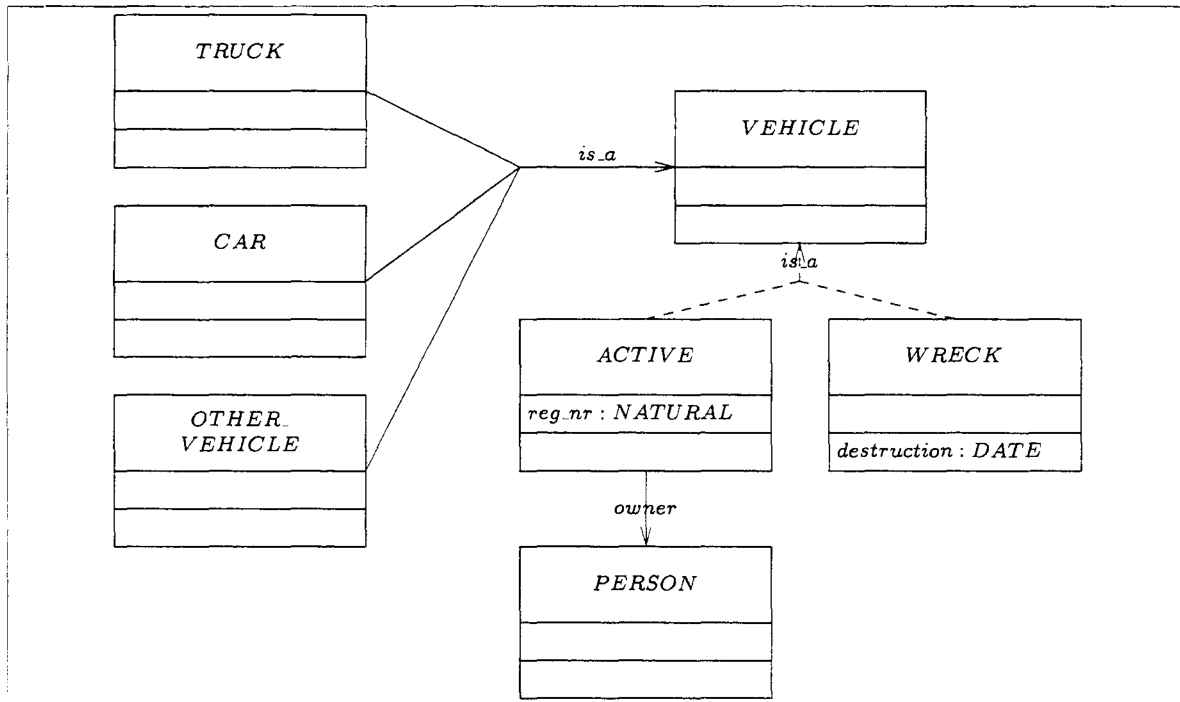



Figure 6.19: Dynamic partitioning of a class.

Each of these is a dynamic subclass that is the intersection between a static and a dynamic subclass.

The frequently occurring phenomenon of an object that "migrates to a subclass" must be modeled in MCM as an object that changes subclasses in a dynamic partition of a superclass. Figure 6.20 shows how we can model the situation that a person "moves to the student subclass". *NON_STUDENT*s have the same properties as *PERSON*s in general, except that *NON_STUDENT*s have the additional property that they can move to *STUDENT*. (This property is not shared with *STUDENT*s.) *STUDENT*s have a number of additional properties, plus the property that they can move to *NON_STUDENT*. (This property is not shared with *NON_STUDENT*s.)

Dynamic subclasses are called roles in version 1 of LCM [130] and in Troll [53]. They are similar to the roles of the Fibonacci language [3] and to the roles introduced by Pernici in office information systems [89], to the Aspects defined by Richardson and Schwartz [94] and the Predicate Classes defined by Chambers [14]. An early discussion of roles can be found in Bachman and Dayal [5], who introduced them in the network data model. In the object-oriented analysis method of Martin and odell [78], all classes are dynamic and the concept of object state is defined as the collection of classes that the object currently has. A survey of modeling approaches to roles is given by Reimer [92]. Although version 1 of LCM has a facility for defining dynamic subclasses, this was dropped in version 2 because of a switch to dynamic logic as specification formalism for event effects. LCM version 3 still has no facility for defining dynamic subclasses, but in a future version they will be included again [136].

### 6.3.3 Role-playing and delegation

There are cases where we want to give an identifier to an object that "moves to a subclass". For example, it is possible that a person becomes an employee at two different companies, or even is two different employees at the same company at the same time (with different jobs, employee numbers

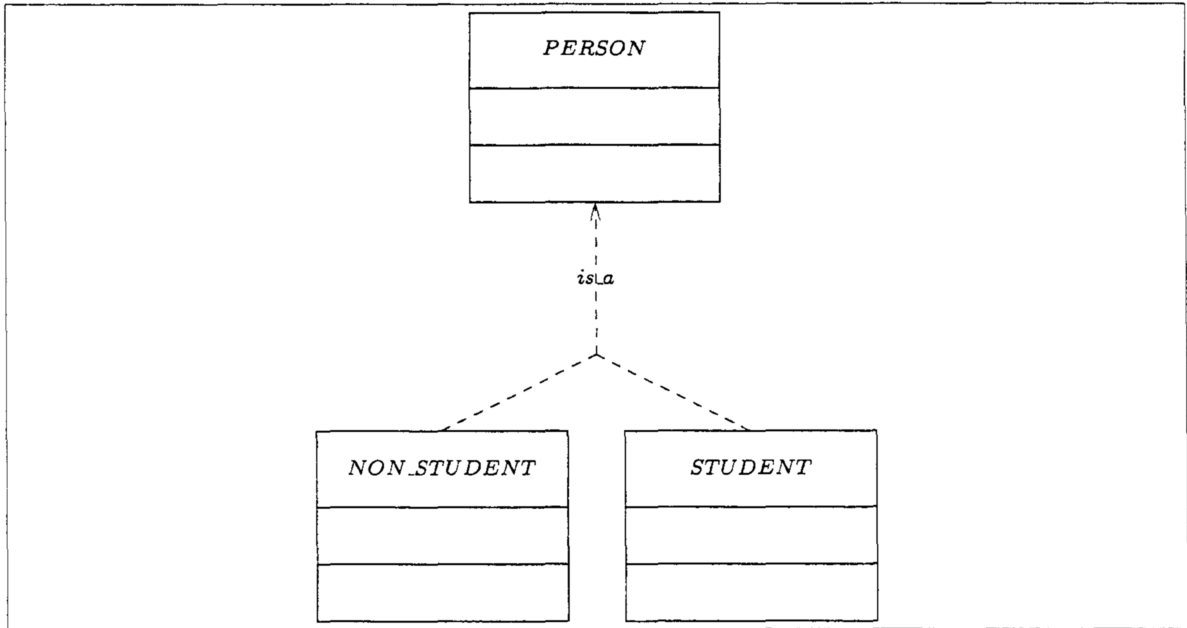



Figure 6.20: A $PERSON$ can move from one subclass to another in this dynamic partition.

and salaries). In these cases we do not need dynamic subclasses but roles. A **role** of an object is just another object, except that it has a special relationship with the object that plays the role. A role can be played by an object or by another role.

More formally, there is a function *player* such that if $R$ is a role class, then there is an object- or role class such that in each state $\sigma$ of the world we have

$$player : ext_\sigma(R) \rightarrow ext_\sigma(P),$$

where $P$ stands for a class of player objects. This implies the following:

1. There is exactly one **player** of $r$.
2. $r$ is **existence-dependent** upon its player, i.e. $r$ cannot exist if its player does not exist.
3. There may be any number of roles played by a player, even if these roles are instances of the same role class.

It is possible to define **delegation** from roles to players. For example, suppose we model an employee $e$ as a role of a person $p$, and *age* is an attribute of persons but not of employees. Then $age(e)$ would be a type error. We can recover from this error by *delegating* the evaluation of *age* to $played_by(e)$ [64]. This amounts to replacing $age(e)$ by $age(played_by(e))$. Delegation can also be defined for events.

Role-playing is represented by an arrow labeled *player*, in a way similar to static subclasses (figure 6.21). The semantics of role specialization in a class diagram is as follows:

- Each class (including a role class) can be specialized into one or more **role groups**. Each role group represents a set of mutually exclusive roles, i.e. the player cannot play roles from more than one role class in a role group.
- For each role class, a cardinality constraint must be given, that says how many instances of the role class the player can play simultaneously. Absence of a visible constraint means unrestricted cardinality. This is different from the cardinality of an *is_a* partition, which is always 0, 1.

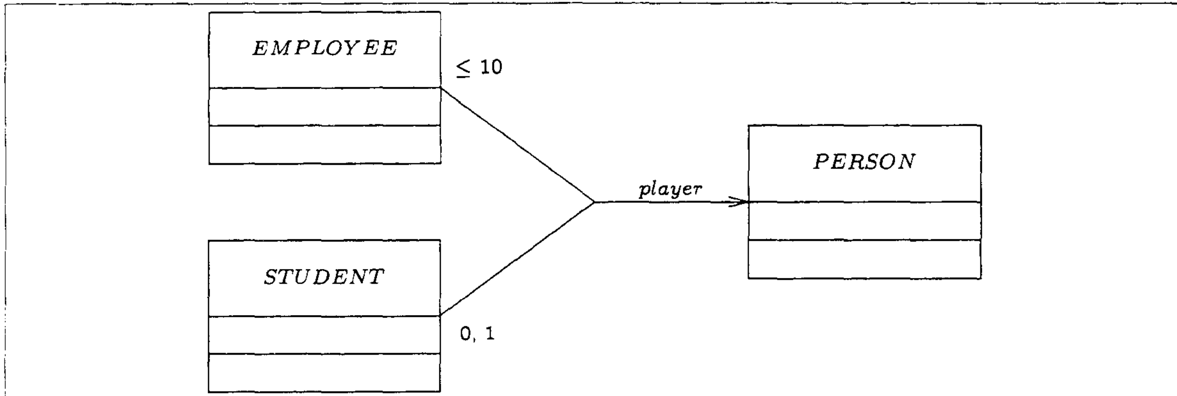



Figure 6.21: A $PERSON$ can play at most 10 $EMPLOYEE$ roles and at most 1 $STUDENT$ role. The two roles are exclusive, but it is possible that a $PERSON$ plays neither role.

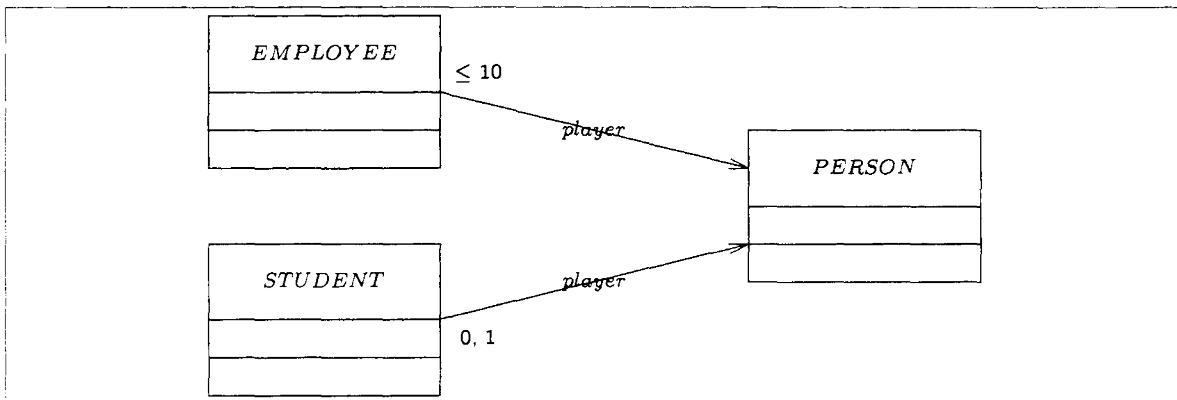



Figure 6.22: A $PERSON$ can play at most 10 $EMPLOYEE$ roles and at most 1 $STUDENT$ role. The two roles are *not* exclusive. It is still possible that a $PERSON$ plays neither role.

- Unlike *is_a* partitions, a role group is not "exhaustive". There may be instances of the player class that do not play any role in a role group.
- Role groups may consist of only one role class, which is also a difference with *is_a* partitions.

To illustrate the last point, suppose we want to allow the possibility that a $PERSON$ plays the roles of $EMPLOYEE$ and $STUDENT$ at the same time. Then we should put these in different role groups, as shown in figure 6.22 (the role groups each have only one element in this figure).

Consider the difference betwen modeling $EMPLOYEE$ as a role class and as a dynamic subclass (figure 6.23). In both cases, a $PERSON$ instance who is not an $EMPLOYEE$ may become an employee. However, in the first case, an instance $e$ of $EMPLOYEE$ *is identical to* a $PERSON$ in a certain state. In the second case, an instance $e$ of $EMPLOYEE$ is *different* from any $PERSON$ instance, but it is related to exactly one existing $PERSON$ instance $p$ by the *player* relationship class. As a role, $e$ is a state of a person, but as an instance of a dynamic subclas, it is a person in a certain state. This difference is behind the different default cardinalities: 0,1 in the dynamic subclass example, and $\geq 0$ in the role-playing example.

The need for separately identified roles is argued by Wieringa and De Jonge [135]. Roles are very

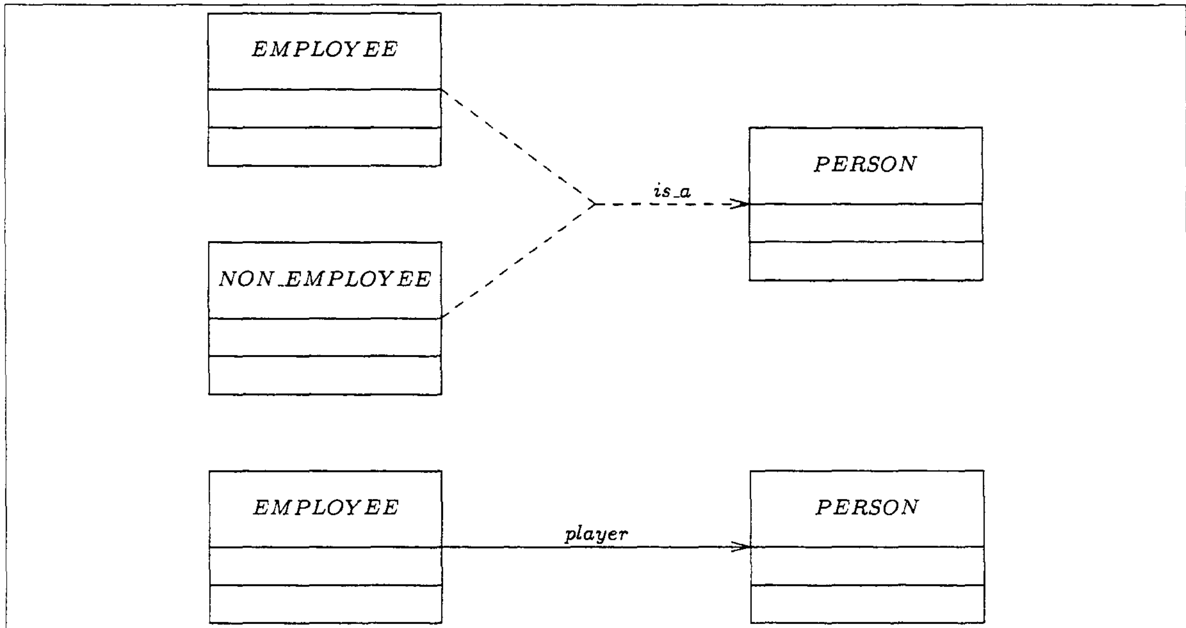



Figure 6.23: In the upper diagram, $EMPLOYEE$ is a dynamic subclass of $PERSON$ and any person is at any moment either an $EMPLOYEE$ or a $NON_EMPLOYEE$. In the lower diagram, $EMPLOYEE$ is a role class of $PERSON$ and any $PERSON$ can at any moment be 0, 1 or more $EMPLOYEE$s.

- **Classification principle.** For each *is_a* partition of $C$ there should be a classification principle that governs the division in subtypes. This principle should be clear, unambiguous, singular, and uniform for all subtypes.
- **Comparable rank.** All members of an *is_a* partition should be of comparable rank.
- **Informativeness.** By locating an object in one of the classes in an *is_a* partition, we should learn more about it than the classification itself tells us.

Figure 6.24: Classification checks for *is_a* partitions.

similar to the object specialization mechanism discussed by Sciore [103]. LCM version 3 has no facility for defining roles; this is planned as a future extension.

### 6.3.4 Quality checks: Classification principles

The logic of classification and taxonomic structure was studied extensively in Aristotelian logic. This section is based upon summaries of Aristotelian logic given by Joseph [51] and Rescher [93], as well as on a study of roles and dynamic subclasses by Wieringa and de Jonge [135].

Figure 6.24 gives a number of quality criteria for *is_a* partitions. First fo all, we should not choose just any grouping into *is_a* partitions, but for each group there should be a classification principle that is clear, unambiguous, singular, and uniform. What this means can best be explained by giving counterexamples to these principles.

- *Vague* classification principle: A division of people into talented and untalented people. It is unclear what the criterion applied here is.

- **Existence set containment.** $C_1$ $is_a$ $C_2$ if and only if $ext_\sigma(C_1) \subseteq ext_\sigma(C_2)$ for all possible states $\sigma$ of the world.
- **Equal extensions.** If $C_1 \xrightarrow{is_a} C_2$, then $ext(C_1) \subset ext(C_2)$, but if $C_1 \overset{is_a}{\dashrightarrow} C_2$, then $ext(C_1) = ext(C_2)$.
- **Migration.** $C_1 \overset{is_a}{\dashrightarrow} C_2$ iff the existence set of $C_1$ can change without a change in the existence set of $C_2$.

Figure 6.25: Quality checks to distinguish static classifications from dynamic classifications.

- *Ambiguous* classification principle: A division of documents into those about statistics, those about economics and others. This is ambiguous because some documents may be classified as having statistic as well as an economic subject matter.
- *Multiple* classification principles: A division of documents into books (non-borrowable), books (borrowable), periodicals (non-borrowable) and periodicals (borrowable). There are really two *is_a* partitions here, not one.
- *Non-uniform* classification principle: A division of animals into domestic animals, poisonous snakes and others. One subclass is defined along the dimension domestic-wild animals and another along the dimension poisonous-nonpoisonous.

The principle of *comparable rank* rules out an *is_a* partition of people into those living in Amsterdam, Americans, and others. There is a clear, unambiguous, single and uniform classification principle, viz. location of the place of birth, but the classes of entities grouped together according to this principle are not of comparable size.

Finally, the principle of *informativeness* rules out a classification of documents into

- those that can be borrowed for three weeks,
- those that can be borrowed for one week, and
- those that can be borrowed for one day.

Unless, say, extra attributes were applicable to instances of some of these subtypes, locating a document into one of those classes does not give any extra information above the information what the borrowing period is.

Incidentally, this is one reason why an intersection clas like $CAR * DIESEL$ should not be shown on a class diagram if the properties of instances of this class are just the sum of the properties of the intersected classes, $CAR$ and $DIESEL$.

### 6.3.5 Quality checks: Static versus dynamic specializations

Figure 6.25 gives quality checks that can be used to distinguish static from dynamic specializations. To illustrate, if $STUDENT \overset{is_a}{\dashrightarrow} PERSON$, then in all possible states of the world, each existing $STUDENT$ is identical to an existing $PERSON$, but the set of *all possible* $PERSON$s is the set of *all possible* $STUDENT$s. For every $PERSON$, there is a state of the world (which may not be reachable anymore), in which the $PERSON$ would be a $STUDENT$. Finally, if an existing $PERSON$ becomes a $STUDENT$ (i.e. migrates from $NON_STUDENT$ to $STUDENT$), then the existence set of $STUDENT$ increases but the existence set of $PERSON$ remains the same. Contrast this with $STUDENT \xrightarrow{is_a} PERSON$: if we create a $STUDENT$ in that case, we also create a

- **Counting check.**
  - If $C_1$ *is_a* $C_2$, then the total number of $C_1$'s in a given set is always less than or equal to the total number existing $C_2$'s in the same set.
  - If $C_1 \xrightarrow{player} C_2$, then the total number of $C_1$'s in a given set may be less than, equal to or greater than the total number of $C_2$'s in the same set.
- **Cardinality check.**
  - If $C_1$ *is_a* $C_2$, then an existing instance of $C_2$ is related to at most one existing instance of $C_1$.
  - If $C_1 \xrightarrow{player} C_2$, then an existing instance of $C_2$ can be related to any number of existing instances of $C_1$.
- **Identity check.**
  - If $C_1$ *is_a* $C_2$, then an existing instance of $C_1$ is identical to an existing instance of $C_2$.
  - If $C_1 \xrightarrow{player} C_2$, then an existing instance of $C_1$ cannot be identical to an existing instances of $C_2$.

Figure 6.26: Quality checks to distinguish (static and dynamic) *is_a* classifications from role-playing. $C_1$ *is_a* $C_2$ stands for $C_1 \xrightarrow{is_a} C_1$ and $C_1 \overset{is_a}{\dashrightarrow} C_2$.

*PERSON*. In a static subclassification, it is impossible to create an element of a subclass without at the same time creating an element of the immediate superclass.

### 6.3.6 Quality checks: Role playing versus specialization

Figure 6.26 gives a number of quality criteria to check whether roles have been modeled properly. We represent the role-playing relationship in the obvious way by $C_1 \xrightarrow{player} C_2$. To illustrate the checks, take the classes *EMPLOYEE* and *PERSON* and consider the question whether we should model this as $EMPLOYEE \overset{is_a}{\dashrightarrow} PERSON$ or $EMPLOYEE \xrightarrow{player} PERSON$. First of all this is a choice to be made by the modeler, not a metaphysical question to be solved by conceptual analysis nor an empirical question to be solved by experiment alone. The choice is evaluated by looking at the use that is made of modeling constructs (the quality checks discussed below), by comparing the choice with domain knowledge (validation checks, discussed in chapter 9), and by checking whether all relevant queries can be answered (utility checks, chapter 9).

- *Counting check.* If $EMPLOYEE \overset{is_a}{\dashrightarrow} PERSON$, then suppose we have a set of five employees (identified by, say, their employee numbers). Then this is also a set of five persons. By contrast, if $EMPLOYEE \xrightarrow{player} PERSON$, then it is possible that one person is two or more employees (has two employee numbers), for example because he or she has two or more jobs simultaneously. In that case, the set is not a set of persons at all, but of employee roles of persons. If pressed, we could say that the set "is" a set of $n$ persons, where $n \leq 5$.
- *Cardinality check.* If $EMPLOYEE \overset{is_a}{\dashrightarrow} PERSON$, then any person either is or is not in the state of being an employee. Any person is therefore at most one employee. Put differently, it is related to at most one employee by an identity relation. By contrast, if $EMPLOYEE \xrightarrow{player} PERSON$, then any person can play the role of $n$ employees, for $n \geq 0$.
- *Identity check.* If $EMPLOYEE \overset{is_a}{\dashrightarrow} PERSON$, then each employee is identical to a person but if $EMPLOYEE \xrightarrow{player} PERSON$, the each existing employee cannot be identical to an

- **Instance.** An object $o_1$ is an instance of an object $o_2$ if we can say that $o_1$ is a *case, illustration, example, sample, specimen* of $o_2$.
- **Counting.** An entity $e$ is a class if we can ask hown many $e$'s there are currently.

Figure 6.27: Instantiation checks.

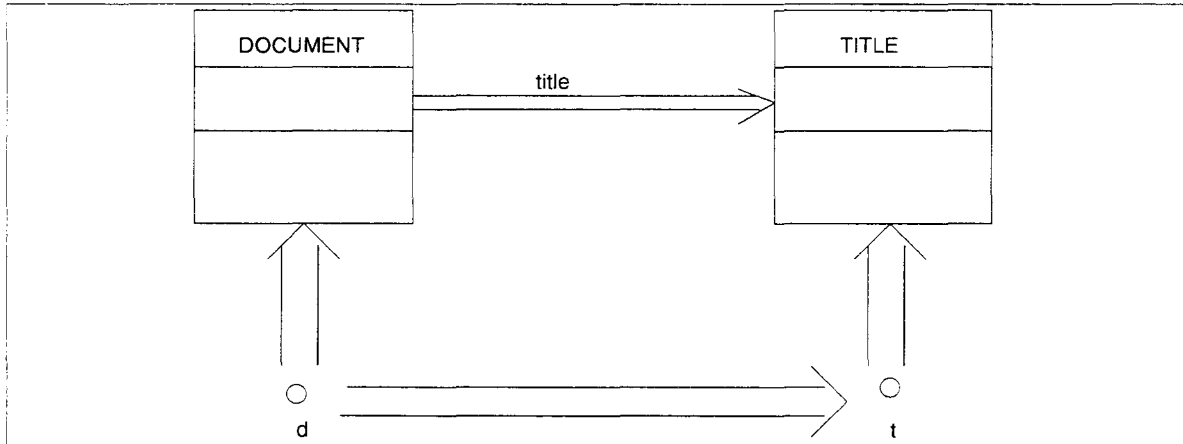



Figure 6.28: Metaclasses. The double arrow represents the instantiation relationship.

existing person.

These checks are just variations on each other.

## 6.4 The instantiation relationship

Subclassification and even object decomposition are often confused with instantiation. Figure 6.27 gives two criteria by which to decide whether an entity $o_1$ is an instance of entity $o_2$. They are quality checks that we can use to find out whether we confused a subclass with an instantiation.

To illustrate, $DOCUMENT$ and $TITLE$ are both classes, for we can ask how many $DOCUMENT$s and how many $TITLE$s there are in a library (the counting check). Now, let $t$ be a $TITLE$; then we can ask how many instances of $t$ there are in a library. For example, we can ask how many exemplars, copies, instances, there are of the title *An Introduction to Database Systems.* This means that $t$ is a class of $DOCUMENT$s and that $TITLE$ is a **metaclass** of copies. Figure 6.28 illustrates this. The double arrow represents the instantiation relationship.

The situation may be clarified if we draw Venn diagrams of the extensions of the classes (figure 6.29). $t$ is an element of the extension of $TITLE$, but since it is itself a class, it has an extension, $ext(t)$. This extension consists of all the documents if title $t$. The Venn diagram shows that $t$ is actually a specialization of $DOCUMENT$.

To see how the quality checks for the instantiation relationship can help to distinguish subclassing from instantiation, take for example the following "is_a" statement:

- A Ford is a car.

The phrase "a Ford" singles out an arbitrary entity of a certain type, Ford. We can ask how many Fords there are, so in this sentence, "Ford" denotes a class, say $FORD$. By the same argument,

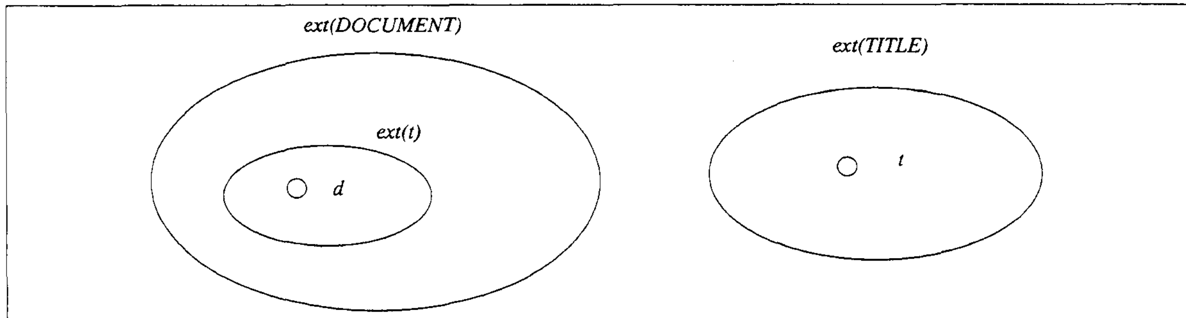



Figure 6.29: Venn diagram of $DOCUMENT$ and $TITLE$.

"Car" also denotes a class (we can ask how many cars there are), which we can call $CAR$. What is the relation between these two classes? We cannot say that $FORD$ is an instance of $CAR$. Rather, a $FORD$ instance is (also) an instance of $CAR$, so the statement expresses an identity between a $FORD$ with a $CAR$ instance. In other words, it expresses a subclass relationship $FORD \xrightarrow{is\text{-}a} CAR$.

Now take the following statement:

- Ford is a car model.

Again, "Ford" denotes a class (we can ask how many Fords there are), which we call $FORD$, and "car model" denotes a class (we can ask how many car models there are), which we call $CAR_MODEL$. But in this sentence, $FORD$ is an instance of $CAR_MODEL$. This is because $FORD$ is an instance of $CAR_MODEL$. But this means that $CAR_MODEL$ is a metaclass of Fords. The statement expresses an instantiation relationship. Note the difference with the subclass relationship above. An instance of $FORD$ is *identical* to an instance of $CAR$, but an instance of $FORD$ is not identical to an instance of $CAR_MODEL$. You may want to draw a Venn diagram of this example and the previous one.

The instantiation relationship can be expressed by a double arrow in an object diagram of MCM. This has no further semantics than that shown in figure 6.27. Often, it is clearer to just represent an instantiation relationship with an ordinary labeled arrow. LCM contains no facility for defining metaclasses.

## 6.5 Events

At any moment, the *state* of the UoD is represented by the set of objects, relationships and roles that exist, and by the state of the existing instances. The state of an existing instance consists of two parts, the attribute or predicate values of the instance and its position that the object has in its life cycle (this is explained later). In the previous sections, we have only considered attributes, predicates and relationships between objects, so we have until now only looked at the representation of the state space of the UoD. In this section, we start looking at the representation of the *dynamics* of the UoD. There are three important components to the dynamics of any object: the set of atomic events that it can perform, the way these events are composed into the life cycle of the object, and the way the object communicates with other object through these events. Events are discussed in this section, life cycles and communications are discussed in the next two subsections.

### 6.5.1 Requirements for events

An **event** in the life of an object is a relation on the possible states of the object. If the event is is a function on possible states, then it is called **deterministic**, otherwise it is called **nondeterministic**. If $(\sigma_1, \sigma_2)$ is an element of (the relation associated with) an event $e$, then we say that $(\sigma_1, \sigma_2)$ is an **occurrence** of $e$. because of this extensional view of ocurrences, a pair $(\sigma_1, \sigma_2)$ can be an occurrence of more than one event.

Events must satisfy the following criteria.

1. Events must occur in the UoD, not merely in the DBS.
2. Events must be *atomic* in the sense that all event occurrences have no intermediary states.
3. Events must be *discrete* in the sense that their occurrences take exactly one moment of (discrete) time.
4. Events are allocated to classes, which means that each event occurrence must be *local* to some instance of the class.

These requirements are derived from the requirements on actions in JSD [49] but they are not identical to them. The first requirement simply means that events are part of a model of the UoD, not merely of a model of the DBS. We should be able to define an event without referring to the DBS at all. For example, unless our UoD is a computerized DBS, *print_report* and *move_cursor* are events that cannot be defined independently from a (computerized) DBS of the UoD. By contrast, *borrow*, *return* and *lose* are events that can be defined independently from a DBS of the library. Note that these events will end up as registration transactions of a DBS. Hence the phrase "not *merely* in the DBS".

The second requirement means that the sequence

$$request_document \cdot receive_document$$

is not an event, because there is an intermediary state. (We use the dot to denote sequence.) Atomicity of events is required if we want DBS transactions to be atomic (see section 6.7). This is in turn required if we want to guarantee integrity constraints (see section 6.8). For example, if we want to guarantee the integrity constraint that a document is either borrowed or not borrowed by a library member, then the *borrow* transaction should not go through an intermediary state in which the book is borrowed but no member has borrowed it. The update program that implements *borrow* may perform an update to the *DOCUMENT* and to the *MEMBER* file in this sequence, but for an observer of the DBS this should appear as one atomic transaction, that either occurs or does not occur.

The atomicity of events is of course built into the definition of events as binary relations. If we would for example want to allow one intermediary state, we should add the possibility that events are triples of states. In that case, events would be confused with *processes.* Atomicity serves to distinguish events from processes.

The third requirement means that an event takes only one moment of time, i.e. only one tick of the clock. This serves to distinguish events from states. For example, *walking* is not an event but a state that exists for a period of time. By contrast, *start_walking* and *stop_walking* are events. Note that dynamic subclasses and roles represent states of objects, with which certain properties are associated, such as extra attributes, extra events, extra constraints, etc.

The fourth and last requirement on events means that an event occurrence can always be allocated to one object. For example, the event *change_address* can be allocated to the *MEMBER* class, which means that each occurrence of this event can be allocated to some instance of *MEMBER*. Different occurrences of *change_address* may of course be allocated to different instances of *MEMBER.* If an occurrence of $e$ is allocated to object $o$, this means that $e$ occurs in the life of $o$. This means two things:

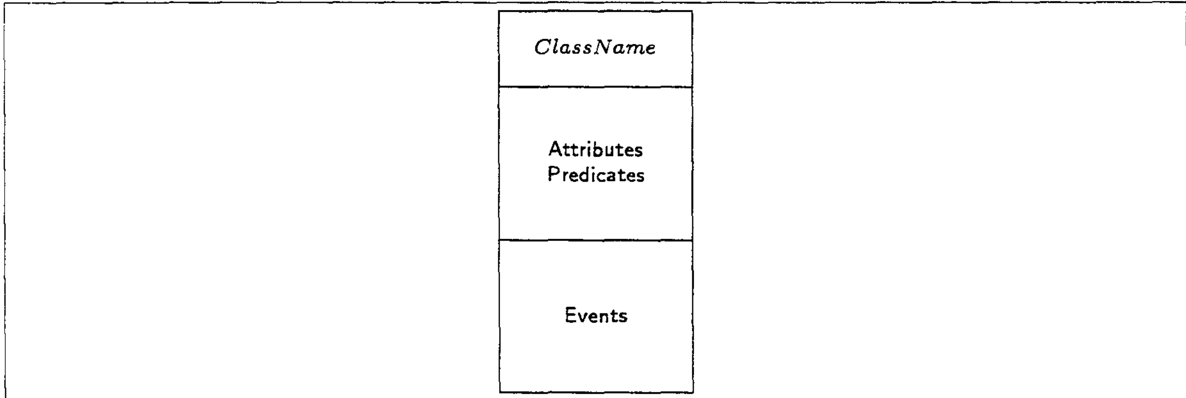



Figure 6.30: Graphical representation of event allocation in class diagrams.

- An observer of the event occurrence is observing $o$ when he or she observes the event occurrence.
- The event occurrence may change the state of $o$ (and not of any other object).

Locality of events is also called **encapsulation** of events in object-oriented methods. We avoid this term, because it has a different (although related) meaning in process algebra. This is explained when we look at life cycles below.

Events are allocated to classes just as attributes are allocated to classes, and we show this allocation diagramatically as shown in figure 6.1, repeated in figure 6.30.

### 6.5.2 Inheritance and overloading

Event names may be *overloaded*. For example, figure 6.9 shows three different classes with three different events called *create*. This is allowed because these three events can be distinguished by the type of their arguments. Formally, if an event $e$ is allocated to class $C$, then it is a function

$$e : C \times s_1 \times \cdots s_n \rightarrow EVENT.$$

The names $s_1, \ldots, s_n$ represent the types ("sorts") of arguments of $e$. The first argument is of type $C$ and must be instantiated by the object in whose life the event occurs. The codomain $EVENT$ is a distinguished sort that acts as sort of all events. As long as the string of argument types of two events is different, the events may be given the same name.

All events are **inherited** by all subclasses of the class to which the event is allocated. This means that if $C_1$ *is_a* $C_2$ (static or dynamic subclassing) and

$$e : C_2 \times s_1 \times \cdots s_n \rightarrow EVENT,$$

then we have an implicit declaration

$$e : C_1 \times s_1 \times \cdots s_n \rightarrow EVENT.$$

This is a case of overloading. It is also a case of inheritance, for all axioms for $e$ when it is applied to instances of $C_2$ are applicable to $e$ when it is applied to instances of $C_1$.

The rule that dynamic partitions must always be exhaustive prevents a problem that we would have if we would allow non-exhaustive dynamic subdivisions of a class. Consider what would happen if $STUDENT \overset{is_a}{- - -} PERSON$ and there would be no dynamic subclass $NON_STUDENT$. The

1. **UoD-orientation**. Events must occur in the UoD, not merely in the DBS.
2. **Atomicity**. Events must be *atomic* in the sense that all event occurrences have no intermediary states.
3. **Discreteness**. Events must be *discrete* in the sense that they take exactly one moment of (discrete) time.
4. **Locality**. Events are allocated to classes, which means that each event occurrence must be *local* to some instance of the class.
5. **Life cycle**. Each event allocated to a class must occur in the life cycle specified for the class.

Figure 6.31: Quality checks for events.

event *become_student* would then have to be allocated to $PERSON$ instead of to $STUDENT$. But *become_student* would then be inherited by $STUDENT$, because that is a subclass of $PERSON$, and that would be a modeling error.

### 6.5.3 Creation and deletion events

For each taxonomy, we define a **species** as a (possibly intersection) static subclass that has no static subclasses. It can be easily proven that in MCM models, the set of all possible instances of all classes is partitioned by species. Thus, each taxonomic structure defines a set of species such that

- each possible instance is an instance of exactly one species and
- instances never migrate from one species to another.

A **creation event** event is an event that moves an object into an existence set of a species and a **deletion event** is an event that moves an object out of the existence set of a species. We only allow creation and deletion events to be declared for species. One reason for this is that we do no want to allow partial creation of an object. Another reason is that the arguments of a creation event are usually the initial attribute values of the object to be created. Moving down in the taxonomic structure, the number of attributes may increase, so that the number of arguments of the creation event would also increase. This makes it impossible to define a general inheritance mechanism for creation events as it can be defined for all other events.

Creation and deletion events are marked in the a class diagram by an asterisk (*) and a plus sign (+), respectively.

### 6.5.4 Quality checks for events

The quality checks for events are just the requirements given earlier for events. They are listed in figure 6.31. Examples of the application of these criteria have already been given when the event requirements were discussed in subsection 6.5.1. The life cycle check is very useful to check whether all events are relevant. The inverse of this check is to see if the life cycle for instances of a class only consists of local events. This is a useful check to see if all relevant events have been found.

## 6.6 Life cycles

### 6.6.1 Process operators

Events can be composed into **processes** by means of the following operators:

| | |
|---|---|
| $+$ | Choice |
| $\cdot$ | Sequence |
| $\|\|$ | Parallel composition |
| $\|\lfloor$ | Left merge |
| & | Synchronous communication |

These operators are defined formally in the algebra of communicating processes (ACP) developed by Bergstra and Klop [9, 10, 11]. A convenient introduction and overview of ACP is given by Baeten and Weijland [6]. Their intuitive meaning can be explained as follows, where $x$ and $y$ stand for arbitrary processes.

- $x + y$ is the process that behaves as $x$ or as $y$, but it is not stated which.
- $x \cdot y$ is the process that first behaves as $x$ and if that terminates, then behaves as $y$.
- $x \parallel y$ is the process that behaves as $x$ and $y$ in parallel, i.e. at any moment we can observe the occurrence of a next possible event from $x$ or from $y$.
- $x \| \lfloor y$ is the process that behaves as $x \parallel y$, except that its first event is a first event from $x$.
- $x \& y$ is the process that performs the first event of $x$ synchronously with the first event of $y$, and then executes a parallel composition of the rest of $x$ and $y$.

We assume a set $EVENT$ of **deterministic events** that contains all possible deterministic local events of all possible objects. Definition of the set $EVENT$ in a particular application is an important task of conceptual modeling. For example, *borrow* and *lose* may be identified as atomic events in a library application, so they will be element of $EVENT$. The definition of the function decomposition tree of a DBS is an important step in the definition of the $EVENT$ set.

$EVENT$ is a subsort of $PROCESS$,

$$EVENT \leq PROCESS.$$

The operators listed above all have arity $PROCESS \times PROCESS \rightarrow PROCESS$. This means that that $borrow + lose$ is an element of $PROCESS$ and not of $EVENT$. However, it does satisfy the event criteria of subsection 6.5.1:

- it occurs in the UoD,
- it is atomic (has no intermediary states),
- it is discrete (takes one tick of the clock),
- and it is local (to a $MEMBER$ for example).

Although $borrow + lose$ is an event, it is **nondeterministic**. $EVENT$ only contains deterministic events and nondeterministic events are represented by terms of the sort $PROCESS$.

Processes are represented in MCM by **recursive process graphs**, as shown in figure 6.32. Recursive process graphs have been defined formally by Spruit [115]. A recursive process graph is a set of directed graphs, each with a root and with labeled edges. The root of each graph is pointed at by a small arrow labeled with the name of the graph. We write this name always in capital letters and call it a **process variable**. The edge labels may be names of atomic events or multisets of graph names. The meaning of an edge $\xrightarrow{e}$ is that traversing this edge amounts to executing event $e$. The meaning of an edge labeled by a multiset like $\xrightarrow{Y,Z,Z}$, is that traversing this edge amounts to executing process $Y$ and two copies of $Z$ in parallel.

There is a simple relationship between a recursive process graph and a set of process equations. For the example in figure 6.32, the set of process equations is

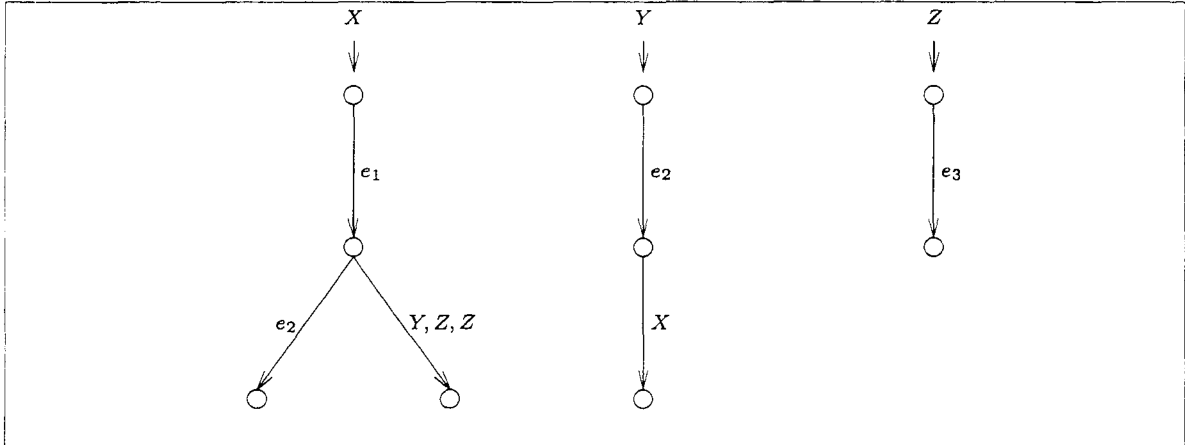



Figure 6.32: A recursive process graph.

$X = e_1 \cdot (e_2 + Y \parallel Z \parallel Z)$
$Y = e_2 \cdot X$
$Z = e_3.$

Recursive process graphs are specified in LCM by means of **recursive process specifications**. This is based on the theory of process algebra. A formal definition of how this syntax is embedded in LCM can be found in the syntax-definition report of LCM [31]. An outline of the formal semantics is given in a number of papers [132, 138].

### 6.6.2 Nondeterministic state and and bisimulation

The material in this subsection is based on research in process algebra by Bergstra and Klop [9, 10, 11], Baeten and Weijland [6], Milner [80, 81], van Glabbeek [39, 38] and others.

Choice allows us to build nondeterministic events. For example, we saw that the event *borrow+lose* is nondeterministic, because the two component events have different effects and we don't know which of the two occurs. Even though *borrow* and *lose* themselves have determinate effects (i.e. they are functions), their alternative composition *borrow + lose* is nondeterministic.

More generally, compare the two process graphs in figure 6.33. The event $a$ on the left-hand side is non-deterministic, because it has two possible next states. One state is characterized by the fact that $b$ can occur and one is characterized by the fact that $c$ can occur. Both states have an identical history but a different future.

By contrast, the event $d$ on the right-hand side is deterministic, because it leads to one state, which is characterized by the fact that $b$ and $c$ can occur. The difference between the two cases is that the specified process has a choice after doing the deterministic $d$, but does not have a choice after doing the non-deterministic $a$. Performance of $a$ does not *lead* to a choice, it *involves* a choice. However, this choice is not made by any system modeled by the process graph, but it is made randomly. Performance of $a$ it leads randomly to one out of a set of two possible states.

In deterministic systems, it is relatively straightforward to define what the state of the system is. Intuitively, the **state** of a deterministic system at any moment is the (possibly incomplete) memory that the system has of its past inputs. Thus, the state space of a deterministic system can be represented by the set of all finite **traces**, where a trace is a sequence of events. In terms of a process graph, a trace is a path through the graph starting from the root. In a deterministic system, each trace leads to exactly one state, although there may be several different traces that lead to the same state.

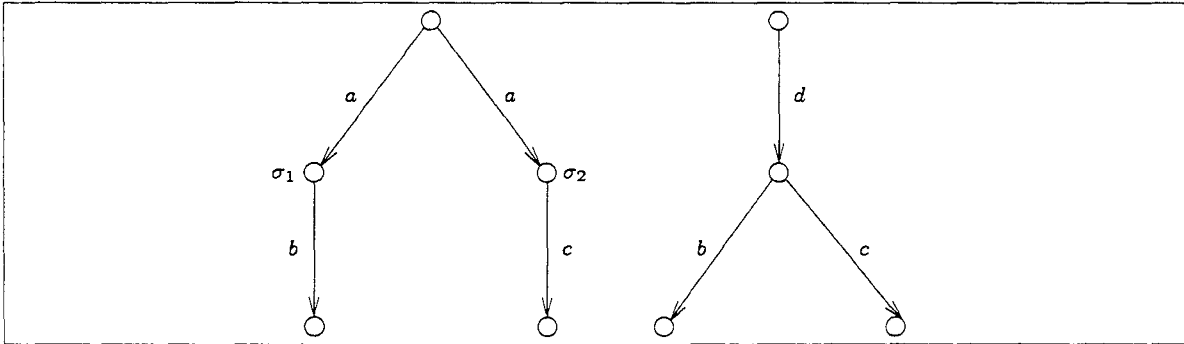



Figure 6.33: Nondeterministic and deterministic behavior. In the nondeterministic process graph on the left, there is an event, $a$, whose occurrence leads to one out of a set of possible states, and it is not specified which. In the deterministic process graph on the right, any event occurrence leads to exactly one next state.

Thus, each memory state is in general an equivalence class of traces, where two traces are equivalent if they lead to the same state. This is because a state of system usually does not remember everything from its past. For example, in the system with one button and one light-bulb we considered earlier, there are only four states, two of which are *on* and *off*. This system cannot distinguish the traces

*press* . *switch_on*
*press* . *switch_on* . *press* . *switch_off* . *press* . *switch_on*
etc.

Neither can the system distinguish the traces

*press* . *switch_on* . *press* . *switch_off*
*press* . *switch_on* . *press* . *switch_off* . *press* . *switch_on* . *press* . *switch_off*
etc.

However, it can remember whether a trace in one of these two sets occurred. In fact, the two trace sets are an extensional representation of the two states *on* and *off*.

In general, the set of all possible traces of a deterministic system is partitioned into sets such that each set is equivalent to exactly one state and different sets correspond to different states. A finite-state system can only distinguish traces if they are in different sets; two traces that are in the same trace set cannot be distinguished by the system, because they lead to the same system state.

If we go to nondeterministic systems, this view of system state does not hold anymore. It is now possible that one history (i.e. one trace) leads to one state in one execution but to another state in another execution. This very statement would be meaningless if we would use the concept of deterministic state defined above. We must use instead a deeper concept of system state, according to which a system **state** is *the set of all possible future events that the system can engage in.* If we restrict this to observable events, then an (observable) system state is the set of all possible future observations we can make of a system. For deterministic systems, we can always find a set of traces that lead to the same system state. For nondeterministic systems, there are at least two different states that have exactly the same trace set. The nodes labeled $\sigma_1$ and $\sigma_2$ in figure 6.33 represent two different states, because they have different futures. They represent different *nondeterministic* states, because both states have the same trace set, viz. $\{a\}$.

There are many different formalizations of this concept of state, which all use the concept of **bisimulation**. Bisimulation is a relation on process graphs which says that the two graphs represent

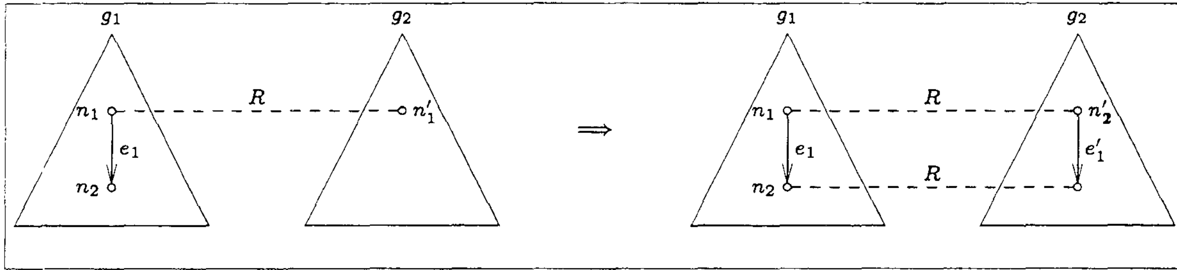



Figure 6.34: Bisimulation between process graphs.

the same process; or, put differently, they represent processes that cannot be distinguished by observations. Process graphs $g_1$ and $g_2$ bisimulate if there is a relation $R$ between the nodes of the graphs such that the following holds (see figure 6.34):

1. The roots of the two graphs are related by $R$.
2. If node $n_1 \in g_1$ and node $n_2 \in g_2$ are related by $R$ and there is an event $e_1$ that leads to a node $n'_1$ in $g_1$, then there is an event $e'_1$ that leads to a node $n'_2$ in $g_2$ such that $n'_1$ and $n'_2$ are related by $R$.
3. The same as 2, with the roles of $g_1$ and $g_2$ interchanged.

Figure 6.35 shows a number of pairs of bisimulating process graphs. The concept of bisimulation is studied extensively in process algebra research. It has been extended to recursive process graphs by Spruit [115].

We can now say that two nodes that bisimulate represent the same state. The two nodes $\sigma_1$ and $\sigma_2$ in figure 6.33 do not bisimulate, so they represent different states. If we would replace the event $c$ by $b$ in the graph, then the two nodes would bisimulate and they would represent the same state.

More on bisimulation relations can be found in Baeten and Weijland [6], Milner [80, 81], and van Glabbeek [38] and others.

### 6.6.3 Dialogs and life cycles

Recursive process graphs can be used to represent various kinds of processes.

- The **behavior** of a system is the process that it goes through in its life.
- A **thread of control** is a process that involves events local to a number of different systems. For example, the process leading from a request to withdraw money from an Automated Teller Machine (ATM) to the withdrawal of the requested money, is a thread of control that involves the ATM, a central communications computer shared by several banks, and one or more computers local to the bank owning the account from which money is withdrawn.
- A **dialog** between a system $S$ and its environment is a process involving $S$ and some systems in its environment that has a logical completion. A dialog is generally only a part of total system behavior.
- A **life cycle** is the dialog that an object in the UoD has with its surrogate in the DBS. Life cycles are called *entity structures* in JSD [49].

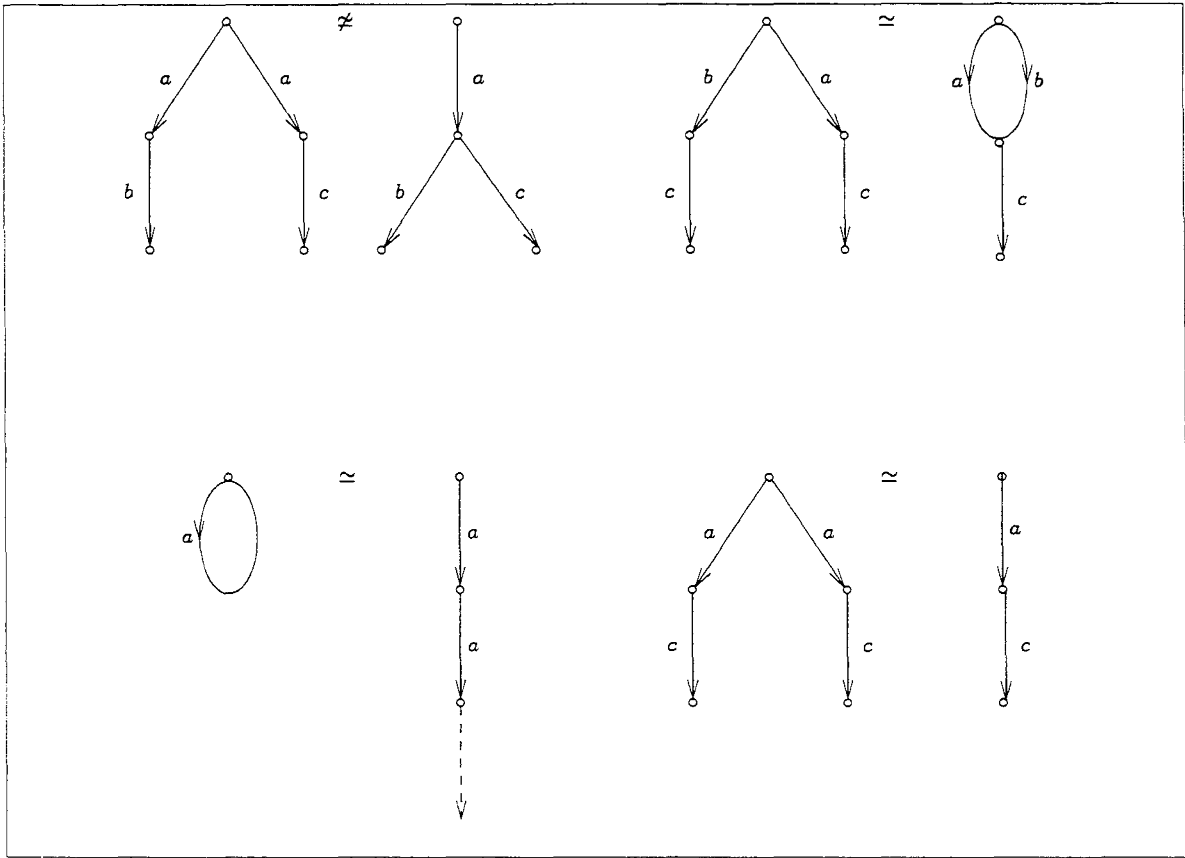



Figure 6.35: Examples and nonexamples of bisimulation.

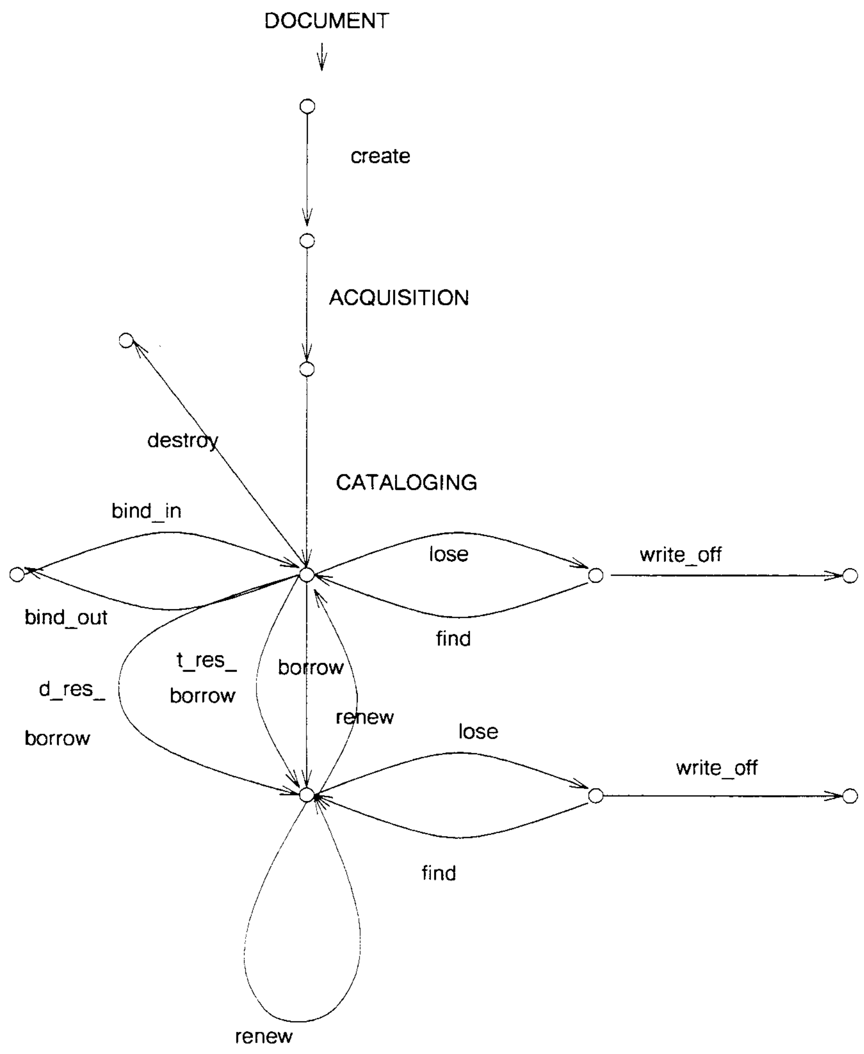



Figure 6.36: Life cycle of *DOCUMENT* instances.

- **Life cycle.** It should be impossible for an object to behave in a way other than as as described by its life cycle.
- **Local events.** A life cycle must consist only of local events.

Figure 6.37: Quality check for object life cycles.

These definitions are informal and will be refined in the future. Note that the life cycle of an object is the behavior of the object as seen by the DBS. Figure 6.36 illustrates the concept of a life cycle. This is only a part of recursive process graph, for the graphs of *ACQUISITION* and *CATALOGING* subprocesses are missing. Each event in the document cycle is a registration transaction.

For each class in the model, there must be a life cycle. Usually, we give this life cycle the name of the class, but this is not obligatory. The life cycle defined for the class describes the life of instances of the class and must consist of all and only the events allocated to the class. During its existence, each instance of the class goes through the life cycle defined for instances of the class. If no life cycle is defined for a class explicitly, we assume that after creation and before deletion, instances of the class can perform any event allocated to the class at any moment.

### 6.6.4 Life cycle inheritance

Life cycle inheritance is still a research topic. If $C_1$ *is_a* $C_2$ (static or dynamic inheritance), then each instance of $C_1$ is an instance of $C_2$ and therefore has the life cycle defined for $C_2$. What if we additionally define a life cycle for $C_1$? There must surely be some consistency requirement between the two life cycles, such as that the life cycle of $C_1$ is performed in parallel to the life cycle of $C_2$, i.e. the total life cycle for $C_1$ instances is $C_1 \parallel C_2$. A more general requirement would be to demand that the life cycle of instances of $C_2$ is a *projection* of the life cycle of instances of $C_1$. A more stringent version of this requirement would be that the life cycle of instances of $C_2$ is an *abstraction* of the life cycle of instances of $C_1$ in the sense of process algebra. This is the condition I suggested elsewhere [130]. Oblog and Troll use a projection condition on traces, which is different [22, 53].

The issue becomes even more complicated if we take dynamic specialization into account. The life cycle of a dynamic subclass is something like a subprocess of the life cycle of its immediate generalization. The projection condition mentioned above should cover this situation as well. Currently, research about this topic is still in a tentative state.

### 6.6.5 Quality checks for life cycles

We borrow the life cycle check shown in figure 6.37 from JSD. A life cycle should cover all contingencies and, more importantly, should not exclude anything that is even remotely possible in the life of an object. As Jackson puts it, if we ask "Could the object behave otherwise?" the answer should be "It cannot be". The behavior of objects in social systems may be of such an open-ended nature that no finite representation may be found of all possible exceptions other than an unstructured iteration over all possible events between creation and deletion of an object.

The local event check is useful to check if all relevant events have been found.

Borrowing

| | borrow | return | renew | overdue | remind |
|---|---|---|---|---|---|
| LOAN | create | return | renew | overdue | remind |
| DOCUMENT | borrow | return | renew | | |
| MEMBER | borrow | return | renew | | |

Figure 6.38: Transaction decomposition table for the Borrowing service.

## 6.7 Communication

### 6.7.1 Transaction decomposition

The representation of communication between objects is the most important part of a conceptual model produced by MCM. Arguably, it is the most important part of any object-oriented model, for this is where the intelligence of the model resides. Class specifications are usually easy to understand in isolation from each other, but these isolated specifications do not give the whole story. Much of the meaning of local events is hidden in the communications in which they participate. It is through communications that the locality of events is transcended and that anything gets going in the world at all. Nothing would ever happen if objects don't communicate.

There are three basic rules for object communication in MCM:

1. Each DBS transaction is either a local event occurring in isolation, or it is a synchronous execution of more than one local events in the life of different objects. In the second case it is a communication. For example, *change_address* might be a local event in the life of a library member, that can occur in isolation, but *borrow* may be a communication between the member and a document, that cannot occur in isolation but must involve a member and a document.

2. Different local events in a transaction are always performed or suffered by different objects.

3. Any relevant event that occurs in the UoD is a transaction in the UoD model of the DBS. For example, the *change_address* event will be modeled as a DBS transaction; it could not occur in the observable life of the DBS if it were not modeled as a DBS transaction. As another example, it means that the *borrow* communication mentioned above will end up as DBS transaction, because it could not occur if it were not modeled as a transaction. This means that objects communicate only via transactions.

These principles hold regardless of the kind of transaction we are dealing with, i.e. registration, directive control, declarative control or temporal events. For example, a DBS may send out a signal to polarize an elevator motor and simultaneously switch on an arrow-shaped light indicating the direction of movement of the elevator. This will be modeled as a directive control transaction that consists of two synchronous local events.

The communication structure of transactions is represented by a **transaction decomposition table**, a representation technique derived from entity/function matrices used in Information Engineering (IE) [75, 76, 77]. An example is given in figure 6.38. For each class in the model there is a row in the table and for each transaction in the model there is a column in the table. Transactions

are grouped into **services**, which are sets of transactions that intuitively belong to each other. There is no formal meaning to this grouping into services; it is merely done to partition the complete transaction decomposition table in manageable chunks. However, in MCM we require that each service in a transaction decomposition table is a service in the function decomposition tree and vice versa.

Since each model in MCM has a finite number of classes and a finite number of transactions, a single transaction decomposition table for all classes and all transactions may become too large to fit on a single page. We can split transaction decomposition tables by making a separate table for each service, and showing only the classes relevant for these transactions.

In each entry of a transaction decomposition table we put the name of the local event with which the class participates in the transaction. Optionally, we can add a C or a D according to whether the event is a creation or deletion event in the class to which it is allocated.

Suppose transaction $t$ is decomposed into local events $e_1$ and $e_2$. Then $e_1$ and $e_2$ are said to be **encapsulated** in the process algebra meaning of this word. This means, literally, that they are hidden from view. They cannot occur as part of the visible interface of the DBS. However, $t$ itself is not encapsulated and therefore *can* occur. There is no inconsistency in the statement that $t$ can occur while its components cannot occur, because $t$ differs from its components. (It is no more inconsistent than the statement "3 and 7 are odd numbers, but 3+7 is an even number".)

If $t$ consists of one local event only, say $e$, then $e$ is not encapsulated. There is an intriguing possibility of defining a local event $e$ that can occur independently as transaction $t$ but can also occur as part of another transaction $t'$. We then say that $e$ is **semi-encapsulated**.

If $C_1$ $is_a$ $C_2$, we only write a participating local event in the entry for $C_2$, not for $C_1$. For example, suppose in table 6.38 we would have $STUDENT_MEMBER \xrightarrow{is_a} MEMBER$, then we would not repeat the local events *borrow*, *return* and *renew* for this subclass of $MEMBER$.

Note that transactions may be overloaded, i.e. we may use the same name for several different transactions. This is allowed as long as the arguments of the transactions are different. The arguments of a transaction are the arguments of the components of the transaction and are declared in the formal transaction specification. It is for example allowed to define another *borrow* transaction, in addition to the *borrow* transaction in table 6.38, as long as the two *borrow* transactions can be distinguished on the basis of the type of their arguments. This is completely parallel to the possibility of overloading local event names.

Note that the components of a transaction may have the same name as the transaction, but this is not obligatory. Each transaction must consist of at least one local event, and each local event must be part of at least one transaction. It is possible to have local events participating in more than one transaction. Each occurrence of such a local event will occur as part of exactly one transaction, but different occurrences may occur as part of different transactions.

Part of the information in a transaction decomposition table can also be represented by a **communication diagram**, as shown in figure 6.39. If a local event participates in a communication, it protrudes from the class box. If it always occurs in isolation, it is drawn inside the class box, as *overdue* in figure 6.39. Each transaction name is represented in the diagram. If the transaction is a communication, the participating local events are connected to a black dot that represents the transaction, and name of the transaction is written next to the dot. If the transaction is a single local event and the transaction has a name different from the local event name, the name of the transaction is written next to the event box. The *overdue* transaction in the *Borrowing* service has the same name as its only component, so the transaction name has been omitted from the communication diagram in figure 6.39.

Experience shows that communication diagrams are harder to read than transaction decomposition tables. A simplified communication diagram, in which only the communications and not the local events are shown, is often useful to get an impression of the communication structure of the model. This is illustrated in figure 6.40.

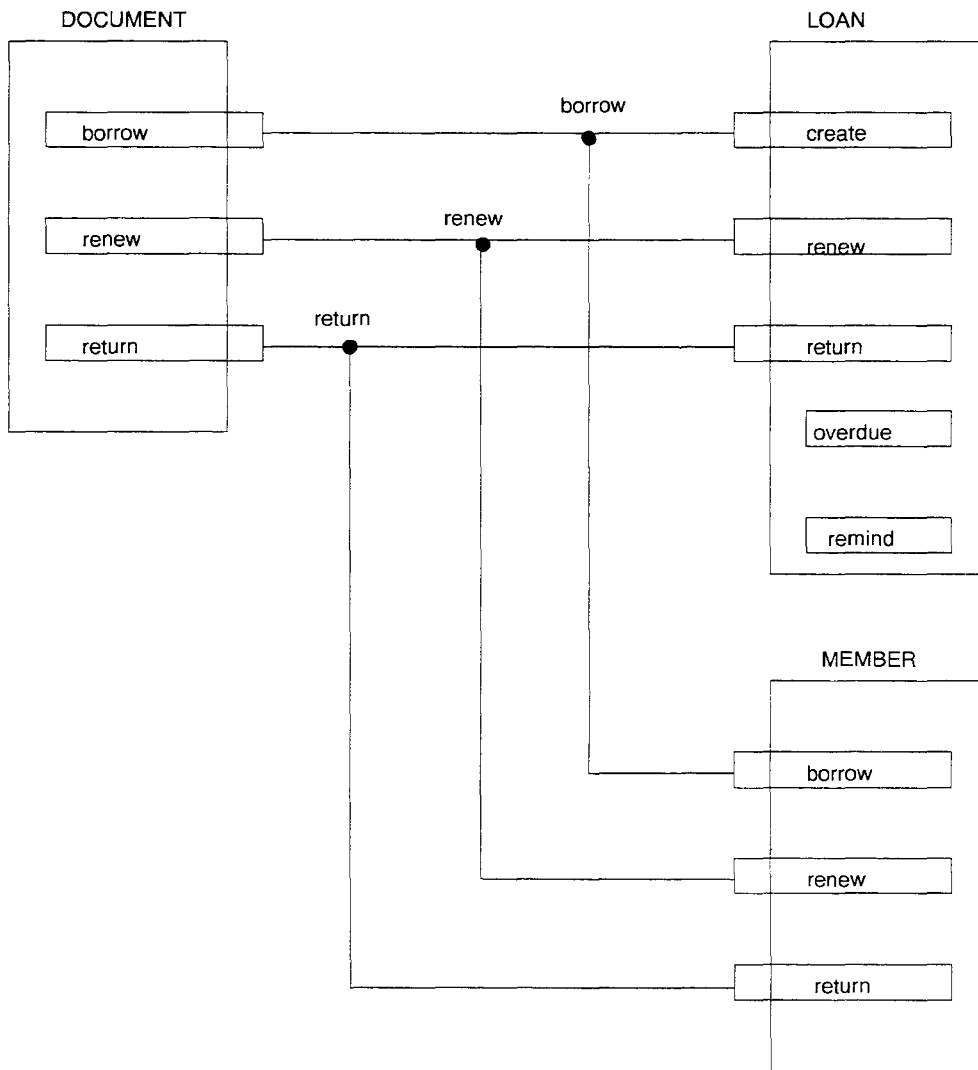



Figure 6.39: Communication diagram of the Borrowing service.

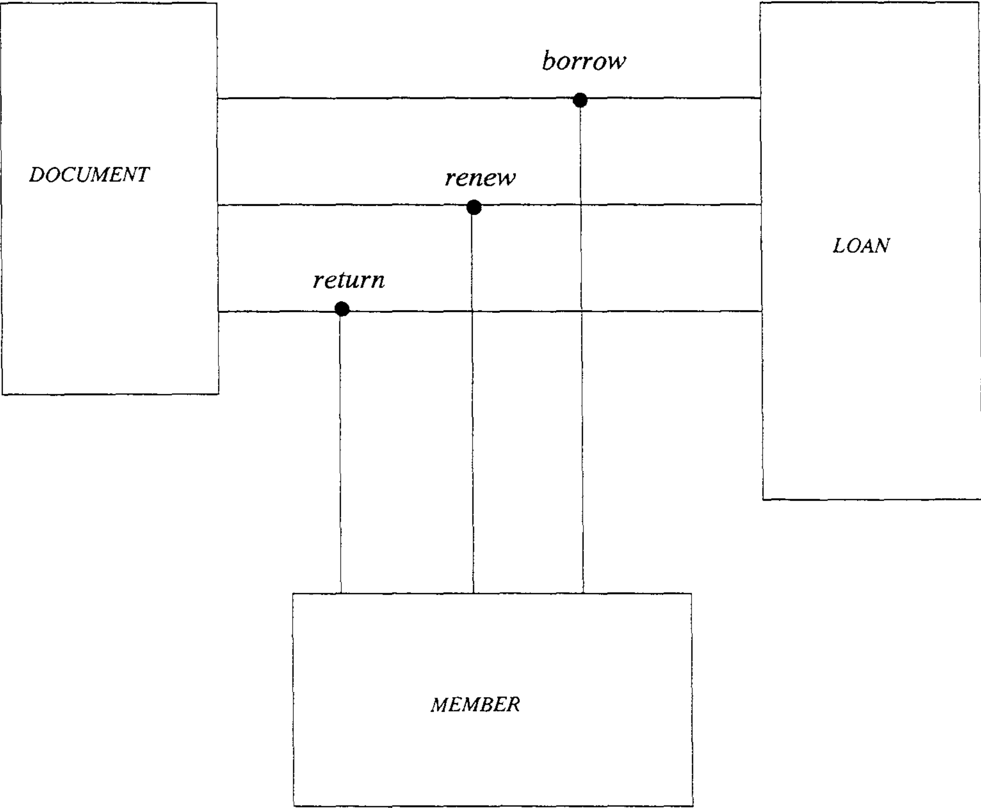



Figure 6.40: Simplified communication diagram of the Borrowing service.

- **State update.** An event must be allocated to an object if it can change the state of an object.
- **Querying.** We allocate an event to an object if we must remember how often the event occurred in the life of the entity.
- **Locality.** We do *not* allocate an event to an object if we cannot ask meaningfully how often the event occurred in the life of the object.

Figure 6.41: Quality checks for the allocation of events to objects.

- **Creation/deletion.** The presence or absence of creation or deletion events for a class should be explained.

Figure 6.42: Quality check for creation and deletion events.

### 6.7.2 Quality checks for communications

Events are allocated to objects according to the allocation criteria shown in figure 6.41. For example, since we cannot ask how often a *TITLE* is borrowed (f we do, what we mean to ask is how often documents of the title are borrowed), we do not allocate *borrow* to *TITLE*. Documents are borrowed, because we can ask meaningfully how often a document is borrowed. However, whether we actually do assign *borrow* to *DOCUMENT* depends upon whether *borrow* needs to change the state of the borrowed *DOCUMENT* and whether we need to answer queries about the *borrow* event that could not be answered without allocating *borrow* to *DOCUMENT*. Not that the query check is really part of the utility evaluation, treated more generally in chapter 9.

A useful quality check taken from Information Engineering is to check whether for each class, there is at least one creation and one deletion event (figure 6.42). Absence of creation or deletion events for a class is not wrong but should always be explained. For example, in some systems, there may be a fixed number of instances of a class, so that no C's and D's need to appear in the row corresponding to this class.

## 6.8 Integrity constraints

So far, the UoD model consists of a representation of the object–, relationship– and role classes, and of the attributes, predicates, events and life cycles of these class instances. There is more structure in the UoD than that, and this is represented by means of *integrity constraints*. We encountered two kinds of integrity constraints so far, cardinality constraints in the object model and an existence constraint on relationships, which says that relationships can only exist if their components exist. In this section, I give a classification of types of integrity constraints.

### 6.8.1 DBS constraints and UoD constraints

A constraint for a system $S$ is a **norm** for a system $S$, i.e. a statement that partitions the states and behavior of $S$ into desirable and undesirable states and behavior. One way to classify constraints is to look at what is being constrained. There are two kinds of systems that we are interested in here, database systems and the UoD. This gives us two kinds of constraints:

- A norm for a DBS implementation is called a **DBS constraint**.

- A norm for the UoD of a DBS is called a **UoD constraint**.

In practice, both kinds of constraints are called *integrity constraints*. Here, I will avoid that term and use the terms "DBS constraint" and "UoD constraint" instead. Examples of UoD constraints are:

- A borrowed document must be returned within three weeks.
- The balance of a bank account must not be negative.
- Stock level must not fall below a certain level.
- A salary should not decrease.

These norms are applicable to the UoD, not to the DBS. A characteristic of UoD norms is that they can be violated by the UoD but not by the DBS.

The following norms are applicable to the DBS:

- The age of a person must not be negative.
- A borrowed document must exist.
- A document cannot be represented as being borrowed by more than one person at the same time.
- An employee cannot be registered as fired before he or she is registered as hired.

These norms are all applicable to the DBS, not to the UoD. A characteristic of DBS norms is that they can be violated by the DBS but not by the UoD.

### 6.8.2 Necessary truths

If the possibility of a violation is absent from a statement then, if the statement is true, we call it a necessary truth. A **necessary truth** about a system $S$ is a statement that is true of all possible states and/or all possible behavior of $S$. Again, we may take for $S$ the UoD or a DBS. First, take necessary truths of the UoD. The most notable thing then is that *all examples of DBS constraints listed above are necessary truths about the UoD*. This is not a coincidence, for if a DBS constraint would not correspond to a necessary truth about the UoD, then we could not say, when DBS behavior implementation violates the constraint, that the DBS is wrong.

Looking next at necessary truths about the DBS, we may regard the specification of all system software as necessary truths about the DBS. During system development, we simply assume that this software is correctly implemented and that the computer system behaves as specified. The task of DBS development is to find a specification of how the DBS must behave, and then build an implementation of this behavior. The DBS specification is really a *norm* on the DBS implementation. Violations of this norm are relevant for the DBS developer, for he or she must take care that they will not occur and, if they nevertheless do occur, the developer must repair these violations. The DBS constraints listed above are example of such norms that are part of the specification. The specification of system software, on the other hand, is assumed by the DBS developer to be correct, i.e. it is treated as a necessary truth in all circumstances that system software behaves as specified.

**Analytical and empirical truths**

There are two kinds of necessary truths, analytical and empirical. A necessary truth is **analytically true** if its truth follows from the meaning of the words used in it. A necessary truth about the UoD is **empirically true** if its truth does not follow from the meaning of the words used in it, but follows from the way the UoD actually behaves.

Some of the DBS constraints above are analytically true when construed as statements about the UoD:

- The age of a person is non-negative.
- A borrowed document exists.
- A document cannot be borrowed by more than one person at the same time.
- No person can be fired before he or she is hired.

The truth of each of these statements can be inferred from the meaning of the words used in it, without any investigation of the state or behavior of the UoD about which the statement is made.

The truth-value of the following statement can only be determined by empirical investigation of the UoD about which it says something:

- The age of a person is under 150.

Empirical statements like these may be true or false, and the only way to find out is to do look at how the UoD behaves, i.e. to do experiments. This is the characterizing feature of empirical statements: An empirical statement can be falsified by the actual behavior of a system. It is always *logically* possible that an empirical statement is false, and we discover that it is false if we observe system behavior that contradicts the statement. By contrast, if an analytical statement is true, then it is logically impossible that it is false and no possible observation can falsify it.

The statement that the age of a person is under 150 is not very likely to be falsified in the next 30 years by any person that we will meet. From a pragmatical point of view, we can treat it as a necessary truth about persons, i.e. a truth that will never be violated by any existing person. We can therefore use it as a DBS constraint and treat any surrogate that represents a real person as having an age over 150 as incorrect.

The following examples are empirical statements that may very well be true but, if they are true, we are not absolutely sure of this, or we may not be absolutely sure that they will remain true during the useful life of a DBS (e.g. the next 30 years).

- People are under 110 years of age.
- Most people have less than 5 bank accounts.
- The balance of a current account does not exceed three times the monthly salary of its owner.

Because we are not absolutely sure that falsifications of these statements will not occur, we cannot treat these as necessary truths, and hence we cannot use them as DBS constraints.

### 6.8.3 Treating UoD norms as DBS constraints

It is possible to treat a UoD norm as if it were a DBS constraint. The effect of this is that the DBS cannot register a violation of the UoD norm. For example, we may treat the UoD norm that a withdrawal of a bank account must not exceed a certain amount $m$ as a DBS norm. This is usually coupled with user procedures which say that any withdrawal of money from an account must

be registered. If the DBS refuses to register some withdrawals, the joint effect of this with the user procedures is to refuse the withdrawal itself. Thus, the user procedure turns the *withdraw* transaction of the DBS into a *declarative control* transaction; a withdrawal occurs only if the DBS declares that it is a withdrawal.

This treatment of UoD norms as if they were DBS norms is not an invalidation of the distinction between UoD norms and DBS norms. On the contrary, we must make this distinction if we are to understand what happens in such cases as the *withdrawal* example above. I give a detailed analysis of this use of declarative control elsewhere [129, 133].

### 6.8.4 Dynamic and static constraints

Orthogonal to the distinction between DBS constraints and UoD constraints, we can make a distinction between static and dynamic constraints. A norm for a system $S$ is **static** if it can be formulated as a constraint on the set of possible states of $S$, without referring to state transitions. A norm that is not static is called **dynamic**. The following statements are examples of static constraints:

- At any moment, a document is borrowed by at most one person (DBS constraint).
- At any moment, a library member has borrowed at most 20 documents (UoD constraint).
- The age field of a person must be non-negative (DBS constraint).

Static constraints imply constraints on state transitions. For example, the first constraint above implies that a *borrow* event cannot occur in the life of a document that is currently borrowed.

The following statements are examples of dynamic constraints:

- The salary of an employee should not decrease. (UoD constraint)
- Withdrawals must not exceed Dfl 20000 (UoD constraint).
- An employee cannot be fired before he or she is hired (DBS constraint).

A dynamic constraint refers to at least two different states of a system, and cannot be translated into a statement that refers to only one state. A dynamic constraint can imply a static constraint. For example, suppose that salaries all start at Dfl 3000. Then the first constraint above implies that there should be no state in which the salary of an employee is below Dfl 3000.

### 6.8.5 Existence constraints

**Existence constraints** are constraint that involve the existence set. We encountered two examples so far.

- The component-existence dependence of relationships: a relationship can only exist if its components exist.
- Cardinality constraints: Each existing instance of $C_1$ must be related to $n$ existing instances of $C_2$.

The component existence constraint is applicable to all arrows $C_1 \longrightarrow C_2$. In each possible state of the world, such an arrow represents a function, which says that each existing instance of $C_1$ is related to exactly one existing instance of $C_2$.

We can turn the component existence constraint around and define the **aggregate existence constraint** as the constraint that in $C_1 \longrightarrow C_2$, each existing $C_2$ must be related to at least one existing $C_1$. This is a particular cardinality constraint. If $C_1$ is a relationship class, so that we have a

projection function $R \to C$, then the aggregate existence constraint says that a $C$ cannot exist without participating in at least one relationship $R$. In the ER modeling approach, this has been called *total participation* of $C$ in $R$ (as opposed to partial participation) and also *mandatory participation* of $C$ in $R$ (as opposed to optional participation).

All existence constraints so far are static constraints. Examples of dynamic existence constraints are the preconditions of creation and deletion events. A creation event has the precondition that the created object does not exist prior to the event occurrence and a deletion event has the precondition that the deleted object does exist prior to the deletion occurrence.

# Chapter 7

# The Structure of Specifications of UoD models

A specification of UoD models consists of two parts, an informal part and a formal part. The description of the formal part below assumes that it is written in LCM, whereas the description of the informal part is language-independent. The intention is that the formal part of the specification can be written in Oblog, Troll, FOOPS or another specification language without changing the informal part. An elaborate example of a UoD model, containing an informal and a formal specification as described in this chapter, is given in a companion report [137].

## 7.1 Informal specification

The informal specification is needed to be able to understand what the formal specification says. It acts as documentation for readers of the formal specification. In addition, it is also needed as a stepping-stone to the formal specification. Experience with using MCM so far suggests that, once the diagrams of the informal specification of a UoD model are made, the formal specification is easy. Iterations between writing the formal and the informal specification are expected, of course, but if the writer of the specification starts with the informal part, the rest of the process is considerably simplified compared to the case where he or she would start with the formal specification.

The informal specification consists of the components shown in figure 7.1. The function decomposition tree represents the boundary of the DBS and relates this to the overall function of the DBS for its environment. The class model shows which classes of objects, relationships and roles exist in the represented UoD and shows how their local state is structured (attributes and predicates) and how it may change (events). The life cycle model shows how the represented objects, relationships and roles change through their life. The communication model shows how each DBS transaction consists of one or more local events in the life of objects, relationships, or roles. The class dictionary, finally, documents each class in a way accessible for domain specialists.

We have seen examples of all components of the informal specification in the previous two chapters, except of the class dictionary. Figures 7.2 and 7.3 give an example of a class dictionary entry for the library example.

The different components of a class dictionary entry contain the following information:

- The class name.
- An informal definition of the class.

| Informal specification component | Graphical representation | Represented structures |
|---|---|---|
| DBS boundary | Function decomposition tree, context diagrams | The function decomposition tree lists all possible DBS transactions and groups them into services according to the contribution to the function of the DBS for its environment.<br>For each service there is a context diagram that shows for each transaction with which individual systems the DBS interacts. |
| Class model | Class diagram | Names of object–, relationship, role– classes.<br>Relationship composition.<br>Attributes of class instances.<br>External identifiers.<br>Predicates applicable to class instances.<br>Events applicable to class instances.<br>Creation and deletion events.<br>Static and dynamic *is_a* relationships.<br>Role-playing relationships.<br>Cardinality constraints. |
| Life cycle model | Recursive process graphs | Life of class instances from birth to death. |
| Communication model | Transaction decomposition tables, communication diagrams | Decomposition of DBS transactions into local events |
| Class dictionary | Formatted text | Definition of class intension,<br>Identity criterion of class instances.<br>Definition of the meaning of attributes.<br>Definition of the meaning of predicates.<br>Definition of the meaning of events.<br>Business rules encoded in the specification. |

Figure 7.1: The five components of the informal specification in MCM.

**Class name:** *DOCUMENT*

**Class definition:**

A *DOCUMENT* is a carrier of information that is of use to library members. The library exists to make *DOCUMENT* instances available to library members. Each *DOCUMENT* is a document of a *VOLUME*, which itself is a kind of *TITLE*. See these two classes for further descriptions.

**Identity:**

A *DOCUMENT* instance is identified by a barcode.

| **Attributes:** | |
|---|---|
| *available_at* | The department where the document can be borrowed. |
| *barcode* | Unique identification of the document issued by the library. Natural number consisting of 14 digits, built up as follows:<br>• First digit: 3 (indicating that a document is identified)<br>• Digits 2–5: identification of the department that owns the document<br>• Digits 6–13: unique identification of the document within this department<br>• Digit 14: checksum |
| *location_code* | The location in the store room where the document is stored when it is not borrowed or out to the binder. |
| *owner* | The department out of whose budget the document was paid for by the library. |
| *max_borrowing_period* | The maximal number of days that the document can be borrowed. This may be<br>• 0 (the document can only be read in the reading room)<br>• 1 (the document must be returned the day it is borrowed)<br>• 3 weeks<br>• unrestricted |
| *penalty* | The price to be paid by a member who loses this document. |
| *price* | The price paid by the library to acquire this document. |
| *volume* | The volume of which this document is an instance. |
| *technology* | The reading, storage or writing technology of the document. This may be printed paper, hand-written manuscript, microfilm, microfiche, movie, video, photo, fine print, or globe. |

Figure 7.2: Example class dictionary entry of *DOCUMENT*.

| Predicates: | |
|---|---|
| *Available* | The document is available for borrowing. |

| Events: | |
|---|---|
| *bind_in* | A document returns from the binder. |
| *bind_out* | A document goes to the binder. |
| *borrow* | A document is borrowed by a library member. |
| *create* | A document is ordered or is received as a gift. |
| *destroy* | A document that is not lent, missing or lost is destroyed. This deletes the document instance. |
| *d_res_borrow* | A document is borrowed by a library member who reserved it. |
| *find* | A missing or lost document is found. |
| *lose* | It is discovered that the actual whereabouts of a document is unknown. |
| *renew* | The borrowing period of a borrowed document is extended. |
| *return* | A document is returned by a library member. |
| *t_res_borrow* | A document is borrowed by a library member who reserved the document title. |
| *write_off* | A lost or missing document is written off. This deletes the document instance. |

**Business rules:**

- A document that has an allowable borrowing period of 0 days is not borrowable (axioms 8, 9, 10).
- A document can only engage in a *borrow* event only if there are no title- or document reservations applicable to the document (axioms 11, 12).
- A document can only engage in a *t_res_borrow* event only if there are no document reservations applicable to the document (axiom 13).
- A document can only be borrowed by a *t_res_borrow* event if it is an instance of a volume for which a reservation is outstanding (axiom 14). The member borrowing the document is then the member who placed the reservation (see the decomposition of *t_res_borrow* in the *Reservation* service).
- A borrowing period for a document can only be renewed when there are no title- or document reservations applicable to the document (axiom 15, 16).

Figure 7.3: DOCUMENT dictionary entry — continued.

- The identity criterion for class instances. This tells us how the class is instantiated, i.e. how we count instances of the class. For example, for an *EMPLOYEE* class it should answer such questions as whether a person hired twice by a company is the same employee, for a *PASSENGER* class it should answer the question whether a person entering the bus every day is one passenger or $n$ passengers, where $n$ is the number of times the bus was entered, etc.
- An informal definition of the meaning of the attributes.
- An informal definition of the meaning of the predicates.
- An informal definition of the meaning of the local events.
- The **business rules** encoded in the specification. A business rule is any rule that the domain specialists regard as a business rule. They may be encoded as part of the life cycle, as axioms, as part of the taxonomic structure, or as part of the attribute–, predicate– or event declarations. One business rule may end up in several parts of the specification and one part of the specification may encode several business rules.

All informal descriptions should use unambiguous language that is understandable for the domain specialist.

The class dictionary entries are best written after the first version of the formal specification is written. As explained in detail in chapter 8, the recommended order of specification is: first the diagrammatic parts of the informal specification, then the formal specification, and finally the class dictionary. Within this recommended sequence, some forward and backward jumping is likely to occur.

## 7.2 Formal specification in LCM

The formal part of the specification of a UoD model can be done in for example LCM. A LCM specification consists of the components shown in figure 7.4.

The **effect axioms** are formulas of the form $\phi \rightarrow [e]\psi$, where $e$ is an event applicable to the class $C$ in whose specification the axiom occurs, and $\phi$ and $\psi$ contain no modalities. They must satisfy the **locality requirement**, which says that $\phi$ and $\psi$ only contain attribute applications to one and the same argument, which must be a variable of sort $C$. In order to be an effect definition, $\phi$ must be equivalent to a conjunction of variable bindings of the form $a(x) = y$ or $P(x)$, with $a$ an attribute, $P$ a predicate, $x$ a variable of sort $C$ and $y$ a variable of an ADT sort, and all variables occurring in $\psi$ must occur in $\phi$ or $e$.

The **preconditions** are formulas of the form

$\phi \rightarrow [e]false$ ($\phi$ is a sufficient precondition for the failure of $e$),
$\phi \rightarrow \langle e\rangle true$ ($\phi$ is a sufficient precondition for the success of $e$),
$[e]false \rightarrow \phi$ ($\phi$ is a necessary precondition for the failure of $e$) or
$\langle e\rangle true \rightarrow \phi$ ($\phi$ is a necessary precondition for the success of $e$).

In some complex cases, $\phi$ may contain modalities.

The formal syntax of LCM is given in a syntax definition report [31]. An elaborate example containing a formal LCM specification is given in another companion report [137]. The following class and service specifications are taken from that report:

```
begin object class DOCUMENT
  attributes
    available_at: DEPARTMENT;
    barcode: NAT                                        fixed;
    location_code: STRING                               fixed;
    owner: DEPARTMENT;
    max_borrowing_period: NAT;
    penalty: MONEY;
    price:  MONEY;
    volume: VOLUME;
    technology: STRING;
  keys
    location_code;
  identifiers
    barcode;
  predicates
    Available                                           initially true;
  events
    bind_in(DOCUMENT);
    bind_out(DOCUMENT);
    borrow(DOCUMENT);
    create                                               creation;
    destroy(DOCUMENT)                                    deletion;
    d_res_borrow(DOCUMENT);
    find(DOCUMENT);
    lose(DOCUMENT);
    renew(DOCUMENT);
    return(DOCUMENT);
```

| Formal specification component | Meaning of the component |
|---|---|
| **Class specification** | Definition of properties shared by all instances of the class. |
| • Class name | |
| • Taxonomic structure | Declaration of one or more partitions per class. Currently only static *is_a* structure. |
| • Attribute declarations | Declaration of codomain of each attribute. The domain of each attribute is the class in whose specification the attribute declaration occurs. Declaration of external identifiers and keys, updatability of attributes, inverse attributes, injective, surjective and bijective attributes. Cardinality constraints. |
| • Event declarations | Declaration of event parameters. The class in whose specification the event declaration occurs is extra parameter called *subject* parameter. Declaration of creation and deletion events for species. |
| • Axioms | Presentation of the theory of all class instances, i.e. of properties shared by all possible instances, in dynamic logic. The *intension* of the class is the set of theorems derivable from the axioms by the rules of dynamic logic. |
| Static integrity constraints | Axioms without modal operator. These constrain the possible states of the model. |
| Effect definitions | Axioms of the form $\phi \rightarrow [e]\psi$, which say that in state $\phi$, the effect of $e$ is to bring about state $\psi$. |
| Event preconditions | Axioms which say that a precondition is necessary/sufficient for success/failure of an event performance (see text for the form of these axioms). |
| • Life cycle definitions | Definition of life cycle by means of a set of recursive process specifications from process algebra (ACP). |
| **Service specification** | Definition of all possible DBS transactions, partitioned into services according to the function decomposition tree. |
| • Transaction declarations | Declaration of name and arity of every possible transaction. |
| • Transaction decompositions | Equation of the form $e = e_1 \& \cdots \& e_n$ with $n \geq 1$, which says the components of a transaction $e$ are the local events $e_1, \ldots, e_n$. |

Figure 7.4: Components of a formal specification of a UoD model in LCM.

```
    t_res_borrow(DOCUMENT, VOLUME);
    write_off(DOCUMENT)                                   deletion;
  life cycle
-- ACQUIRE and CATALOGING are two subprocesses not defined here.
     DOCUMENT = create . ACQUISITION . CATALOGING. AVAILABLE;
     AVAILABLE = bind_out . bind_in . AVAILABLE +
                 lose .(find. AVAILABLE + write_off) +
                 (borrow + t_res_borrow + d_res_borrow) . LOANED +
                 destroy;
     LOANED = return . AVAILABLE +
              renew . LOANED +
              lose .(find . LOANED + write_off);

  axioms
-- 1, 2, 3, 4. A document is not available after being borrowed or lost,
-- missed, or after going to the binder.
-- Effect axioms.
   forall d: DOCUMENT:: [borrow(d) ] not Available(d);
   forall d: DOCUMENT, v: VOLUME:: [t_res_borrow(d, v) ] not Available(d);
   forall d: DOCUMENT:: [d_res_borrow(d) ] not Available(d);
   forall d: DOCUMENT:: [lose(d) ] not Available(d);
   forall d: DOCUMENT:: [bind_out(d) ] not Available(d);

-- 5, 6, 7. A document is available after being returned, found, or
-- brought back from the binder.
-- Effect axioms.
   forall d: DOCUMENT:: [return(d)] Available(d);
   forall d: DOCUMENT:: [find(d)] Available(d);
   forall d: DOCUMENT:: [bind_in(d)] Available(d);

-- 8, 9, 10. A document can only be borrowed if it is borrowable.
-- Local precondition necessary for success.
   forall d: DOCUMENT, ::
     <borrow(d)> true ->  max_borrowing_period(d) > 0;
   forall d: DOCUMENT, v: VOLUME::
     <t_res_borrow(d, v)> true -> max_borrowing_period(d) > 0;
   forall d: DOCUMENT, ::
     <d_res_borrow(d)> true -> max_borrowing_period(d) > 0;

-- 11, 12. A normal borrow event is only possible if there are no
-- title reservations for the volume of which an instance is borrowed and
-- if there are no document reservations for the document.
-- Global precondition necessary for success.
   forall d: DOCUMENT, r: T_RESERVATIONS::
     <borrow(d)> true -> not volume(r) = volume(d);
   forall d: DOCUMENT, d1, d2: DATE, r: D_RESERVATIONS::
     <borrow(d)> true -> not document(r) = d;

-- 13. t_res_borrow  only occurs when there is no document reservation
-- for the borrowed document. This gives precedence to document
```

```
-- reservations.
   forall d: DOCUMENT, v : VOLUME, r: D_RESERVATIONS::
     <t_res_borrow(d, v)> true -> not document(r) = d;

-- 14. A document is only borrowed by t_res_borrow if it is an instance
-- of the reserved volume. (Ensuring that the borrower has the right
-- identity is taken care of by the transaction.)
-- Local precondition necessary for success.
   forall d: DOCUMENT,  r: T_RESERVATIONS::
     <t_res_borrow(d, v)> true -> volume(d) = v;

-- 15, 16. Renewal is only possible if there are no title reservations for
-- the volume of which a document borrowed and if there are no document
-- reservations for the document.
-- Global precondition necessary for success.
  forall  d: DOCUMENT, t: DATE, r: T_RESERVATIONS::
    <renew(d, t)> true -> not volume(r) = volume(d);
  forall  d: DOCUMENT, t: DATE, r: D_RESERVATIONS::
    <renew(d, t)> true -> not document(r) = d;
end object class DOCUMENT

The dictionary entry of $DOCUMENT$ has been shown in
figures~\ref{fig.SPEC.doc1} and~\ref{fig.SPEC.doc2}.

Here is an example of a service specification:
\begin{verbatim}
begin service Borrowing
 transactions
   borrow(loan: LOAN, borrower: MEMBER,
          borrowed_document: DOCUMENT, today: DATE,
          return_before: DATE);
   overdue(loan: LOAN);
   return(loan: LOAN);
   remind(loan: LOAN);
   renew(loan: LOAN, today: DATE, return_before: DATE);
 decomposition
-- LOAN.create creates a LOAN instance b and initializes it so
-- that its components are c and m, and the return date is set to
-- d2. The DOCUMENT.borrow(d, d1) event sets the Available flag
-- of c to false. MEMBER.borrow increases the number of documents
-- currently borrowed by the member by one and also checks whether the
-- member is not excluded.
   forall  l: LOAN, d: DOCUMENT, m: MEMBER,  d1, d2: DATE::
     borrow(loan: l, borrower: m: borrowed_document :d,
            today: d1, return_before: d2) =
       LOAN.create(id: l, document: d, member: m,
                   date_borrowed: d1, return_before: d2) &
       DOCUMENT.borrow(d) &
       MEMBER.borrow(m);
```

```
-- LOAN.return kills the LOAN instance, DOCUMENT.return sets
-- the Available flag of the returned document to true and
-- MEMBER.return decreases the number of documents currently borrowed
-- by m by one.
   forall d: DOCUMENT, m: MEMBER::
     return(<document: d, member: m>)=
     LOAN.return(<document: d, member: m>) &
     DOCUMENT.return(d) &
     MEMBER.return(m)

-- LOAN.renew sets the return date for l = <m, d> to d2 and if the
-- difference between today and d2 is in the allowed borrowing
-- periods for the member.
-- DOCUMENT.renew merely adds the renewal event to the history of the
-- borrowed document, where it is then available for future queries.
-- MEMBER.renew checks whether the member is not excluded.
   forall d: DOCUMENT, m: MEMBER, d1, d2:DATE::
     renew(document: d, borrower: m,  today: d1, return_before: d2)=
     LOAN.renew(<document: d, member: m>,
                today: d1, return_before: d2) &
     DOCUMENT.renew(d) &
     MEMBER.renew(m);

   forall m : MEMBER, d : DOCUMENT::
     overdue(<document: d, member: m>) = MEMBER.overdue(m);

   forall m : MEMBER, d : DOCUMENT::
     remind(<document: d, member: m>) = MEMBER.remind(m);
end service Borrowing
```

The transaction decomposition table and communication diagram corresponding to the Borrowing service specification are given in figures 6.38 and 6.39.

- Each event in a class box of the class diagram corresponds to an event in the life cycle defined for instances of that class and vice versa.
- Each event in the class box of a class diagram appears at least once as a component of a transaction in a transaction decomposition table.
- Each transaction has at least one local event as component.
- The row corresponding to class $C$ (holding the local events with which instances of $C$ participate in transactions) contains exactly the set of events listed in the class box of $C$.
- Each service node in the function decomposition tree corresponds with exactly one transaction decomposition table and vice versa.
- The transactions decomposed in a transaction decomposition table are exactly the transactions listed under the corresponding service node in the function decomposition tree.

Figure 7.5: Relationships that must exist between the different parts of the informal specification.

- Each event in a life cycle definition must have the class to which the life cycle belongs as subject parameter.
- All events declared for a class must be used somewhere in the life cycle definition for the class.
- Each transaction declaration in a service specification must correspond to a transaction decomposition in the same service specification and vice versa.
- All events declared somewhere in a class specification appear at least once as component of a transaction in a transaction decomposition equation.
- Only events declared in a class specification can appear in a transaction decomposition equation.

Figure 7.6: Relationships that must exist between the different parts of the formal specification.

## 7.3 Relations between different parts of the specification

The relations that must exist between the different parts of the informal specification are given in figure 7.5 and the relationships that must exist between the different parts of a formal specification in LCM are given in figure 7.6. The relationships that must exist between a formal LCM specification and its corresponding informal specification are given in figures 7.7 and 7.8.

| Informal specification | Formal specification in LCM |
|---|---|
| Function decomposition tree | • For each service node in the tree there is one service specification and vice versa.<br>• For each transaction node in the tree there is exactly one transaction declaration in the service specifications.<br>• The transactions that are children of node $S$ are declared in the service specification corresponding to $S$; no other transactions are declared in that service specification. |
| Class diagram | • Each class in the class diagram corresponds to one class specification and vice versa.<br>• Each partitioning of a class in subclasses in the class diagram corresponds to a partitioning in the formal specification and vice versa.<br>• The cardinality constraints in the class diagram correspond to constraints declared in the corresponding class specification.<br>• All attributes in the class box of a class diagram are declared in the corresponding class specification and vice versa.<br>• All events in the class box of a class diagram are declared in the corresponding class specification and vice versa. |
| Recursive process graphs | • The recursive process graph of the life cycle of instances of class $C$ represents the unique solution of the life cycle definition for instances of $C$ in the recursive process graph model of process algebra, and vice versa. |
| Transaction decomposition tables | • Each transaction decomposition table corresponds with one service specification and vice versa.<br>• Each transaction in a transaction decomposition table corresponds with one transaction decomposition equation in the corresponding service specification and vice versa. |

Figure 7.7: Relationship between informal specification and formal specification in LCM.

| Informal specification | Formal specification in LCM |
| --- | --- |
| Class dictionary entry | • For each class specification there is exactly one class dictionary entry with the same name and vice versa.<br>• The set of attributes in the dictionary entry for $C$ equals the set of attributes declared in the specification of $C$.<br>• The set of events in the dictionary entry for $C$ equals the set of events declared in the specification of $C$.<br>• The business rules listed in the dictionary entry for $C$ may refer to axioms listed anywhere in the formal specification. |

Figure 7.8: Relationship between informal specification and formal specification in LCM — continued.

# Chapter 8

# An Induction Method for Finding UoD Models

## 8.1 Outline of an induction method

The regulatory cycle of information system development presented in chapter 2 consists of three tasks: requirements determination, conceptual modeling and implementation. Requirements determination produces a requirements specification, which consists of a statement of purpose, a functional requirements specification and a nonfunctional requirements specification.

Conceptual modeling then takes the functional model and starts with the identification of subsystems in this model. For each subsystem a conceptual model of its required behavior must be specified. The subsystems may be nodes in an EDI network, machines in a computer-integrated manufacturing (CIM) system, computers and people in a human-machine system, etc. These different situations require different methods and I do not claim that `MCM` is fit for all possible situations. Currently, `MCM` is intended for use administrative applications that run on a single machine as well as for embedded systems that interact with other machines. Extensions are planned for real-time and control applications that run in a distributed or heterogeneous environment, such as integrated administrative and -control applications and EDI networks.

Once subsystems are identified, we must find a conceptual model of each single DBS that is identified. This is done by filling in the skeleton of the empirical cycle presented in chapter 3 with tasks required to find a specification of an object-oriented conceptual model such as described in the previous two chapters. We can simplify this task by dividing it into two subtasks, that of finding a UoD model and that of finding a query model. `MCM` is concerned primarily with finding a UoD model, but contains tasks for finding system boundaries and for making a query model. An outline of the induction method of `MCM` is given in figure 8.1. This induction method is part of a completely rational modeling process, in which it is preceded by the *observations* task and is followed by the *evaluations* task. In practice, these tasks will occur interleaved. Like any rational method, it will have to be combined with management methods to deal with finite resources and a disturbing environment. There is no strict sequence in the induction method, except that one must start with determining system boundaries. After that, the other tasks may occur interleaved with each other and, if project procedures allow this, one may return to improving the definition of system boundaries many times.

In this chapter, we discuss making a function decomposition tree, various ways of doing a transaction analysis and performing scenario analysis. There is no special advice on how to perform the other tasks in the method. `MCM` allows the use of other methods and techniques to find the UoD model, such as record- and form analysis and normalization. These methods are not discussed here, but references

1. Identify **system boundaries** by making a *function decomposition tree* whose leaves consist *registration transactions*, *directive* and *declarative control transactions*, *temporal events* and *queries* (this includes an arbitrary ad hoc query).
2. **UoD modeling**.
   (a) Do a **transaction analysis** of registrations and triggered updates to produce a *class model* and a *communication model*.
      - Object analysis of transactions.
      - Elementary sentence analysis.
      - Transaction decomposition.

   (b) Do a **scenario analysis** to produce *life cycle models* for the object classes.
      - Narrative scenario analysis.
      - Event trace analysis

   (c) **Document** the object–, communication– and life cycle models in an *informal specification*.
   (d) **Formally specify** the model.
      - Produce *signatures* for the attributes and events.
      - Specify event *effects*.
      - Specify event *preconditions*.

   (e) Identify **business rules** and add the to the informal and formal specification.
3. **Query modeling**. Specify the *queries* in the decomposition tree and the *reports* produced by those queries.

Figure 8.1: A method for finding object-oriented database system models.

to descriptions of them are given in chapter 3 and in the appendix.

## 8.2 Making a function decomposition tree

The requirements specification document may already contain a function decomposition tree that represents the transactions to be performed by the required DBS. More likely is the situation where there is a function decomposition tree that is only partly worked out and that must be elaborated until the set of all transactions of the required DBS is reached. In other situations, there will not be a function decomposition tree for the DBS at all, and one must be built from scratch.

To make a function decomposition tree of a DBS, the developer must first make a function decomposition tree for the organization in which the DBS will be used. It is a simple matter to build a function decomposition tree for a DBS from one of the organization in which it will be used. An example of this is given in chapter 5 (Representing System Boundaries). In the rest of this section, I discuss making a function decomposition tree for organizations.

As indicated in chapter 3, we can do this using data collected by means of interviews, questionnaires, direct observations, etc. A useful source of inspiration for building a function decomposition tree is is a list of *generic functions* that can be found in most organizations, and which one can use to check which of these are present in the organization under study. The following list is taken literally from an introduction to the topic by Barker and Longman [7, pages 358–359].

- **Planning**.
  - Set objectives and critical success factors.

  - Analyze the industry in which the business operates: competitors, markets, competitive products.
  - Formulate a business strategy in financial, product and other terms.
  - Decide tactics to be employed.

- **Acquire resources.**
  - Raise capital.
  - Recruit, train, motivate staff.
  - Purchase (or otherwise acquire) building, machinery, furniture, vehicles, etc.
  - Takeover or merge with other businesses.
  - Ensure availability of resources provided by third parties.

- **Marketing and sales.**
  - Educate marketplace(s) and make aware of product.
  - Identify and qualify prospects.
  - Match product to prospect needs and complete sales.
  - Manage accounts.

- **Development, production and delivery.**
  - Research new product opportunities.
  - Design and develop product.
  - Produce product.
  - Deliver product to customer.

- **Support and maintenance.**
  - Resolve faults, complaints and product enhancement requests.
  - Identify and diagnose root causes of faults in product.
  - Expedite correction through product development or delivery, as applicable.

- **Monitoring and control.**
  - Assess performance against plan.
  - Identify and take action to correct dangerous anomalies and trends, and so on.
  - Instigate replanning if deviation is large enough.

Barker and Longman give other frequently occurring functions in government agencies and personnel departments. Other useful examples of frequently occurring functions are given by Martin [73, pages 23–24]. Starreveld et al. [116] give a very useful classification of organizations into trading companies, industrial companies, agriculture businesses, service businesses, banks, insurance companies, government agencies and non-profit organizations, and for each class they give a list of typical activities.

Turning from the input of the induction process to the induction process itself, the most important thing to remark is that building a function decomposition tree is a *negotiation process* between developers and sponsors.

To find a function decomposition tree for an organization, the developer typically interviews managers in the organization in order to find out what the organization does. The results of these interviews are presents in the form of a function decomposition tree, which is then discussed with the managers. In the resulting discussion about what is the decomposition tree that best represents the organization, the managers have the last word. For one thing, they are the domain specialists who know the most about their organization. Even more importantly, the tree will be used to represent not the current situation, but the situation as it ought to be. It is the managers who have the authority to determine the direction in which the company is going, not the developers.

A consequence of this procedure is that there is no unique function decomposition tree of an organization. Even if the current situation is modeled, and even if the current situation is unambiguous, then different developers would come up with different trees. This is simply because the set of all transactions in which the company engages can be ordered in many different ways and there is no one best way to do this. For example, functions may be grouped according to the material manipulated, according to the output produced, according to the kind of task performed, etc. Different classification criteria will lead to different trees that represent the same organization. This is no problem, because a function decomposition tree is independent of organization structure. What is important is that developers and sponsors agree about what the tree means and about the correctness of this representation of what goes on, or ought to go on, in the organization.

Function decomposition trees go through many incarnations, as you discover more and more functions that should be added to the tree. A very useful technique to build the first version of the tree, recommended by Barker and Longman [7], is to use small sticky notelets and write one function on each notelet. The notelets stick easily to a glass wall or to a table surface and can be moved around quickly as the ideas about the best organization of the tree change. When the tree is stable, and only then, a graphical editor could be used to draw the tree and give it a nice layout.

## 8.3 Transaction analysis

Once we have a list of relevant DBS transactions, we can proceed to build a UoD model on the basis of these transactions.

### 8.3.1 Object analysis of transactions

The method presented in this section is based on a transaction analysis method given by Flavin [34] and adopted by Ward and Mellor, who call it the passive event modeling approach [123, pages 37–38]. In an **object analysis of transactions**, we find relevant objects by considering each required DBS transaction in turn and asking which objects are *used* by that transaction. To use an object means to Create the object, to change the state of the object (Update), to look at the state of the object (i.e. Read it) or to Delete the object. These types of access are abbreviated by the well-known acronym CRUD. The result of this analysis is represented in a table called **transaction/use table**. Figure 8.2 gives an outline of the method.

Figure 8.3 gives an example transaction/use table for the circulation desk of the library. The function decomposition tree used as input for this analysis was already shown in figure 5.5. Figure 8.3 shows that the relationships $T_RES$, $D_RES$ and $LOAN$ are created by certain transactions, remembered for a while and then deleted. The title reservation transaction (creating a $T_RES$ instance) needs data about members and documents to check if a member is allowed to place a reservation and whether no document of that title is available for borrowing. The document reservation transaction (creating a $D_RES$ instance) checks whether the reserved document is available and marks the document as reserved. Borrowing a document may cause deletion of a reservation relationship. Renewing a document loan can only be done if there are no reservations for the title of that document, or for the

1. Decompose functions of the environment of the DBS down to the level of transactions.
2. List registration transactions, directive and declarative control transactions, temporal events, and queries.
3. List, for each transaction, the objects and/or relationships that are created, read, updated, or deleted by the transaction.
4. Put the object classes and relationship classes in an object diagram.
5. Add the attributes that are used or changed by the transactions.
6. Add cardinality constraints.

Figure 8.2: Object analysis of transactions.

| | Create | Read | Update | Delete |
|---|---|---|---|---|
| Reserve title | *T_RES* | *MEMBER*, *DOCUMENT* | *TITLE* | |
| Reserve document | *D_RES* | *MEMBER*, *DOCUMENT* | *DOCUMENT* | |
| Borrow document | *LOAN* | *T_RES*, *D_RES*, *MEMBER* | | *T_RES*, *D_RES* |
| Renew document loan | | *T_RES*, *D_RES*, *LOAN*, *MEMBER* | *LOAN* | |
| Return document | | *LOAN* | | *LOAN* |
| Lose document | *LOST*, *FINE* | *LOAN* | *MEMBER*, *DOCUMENT* | *LOAN* |
| Cancel reservation | | | | *T_RES*, *D_RES* |
| Send reminder | | *LOAN*, *MEMBER* | *LOAN* | |

Figure 8.3: A transaction/use table for the circulation desk database system.

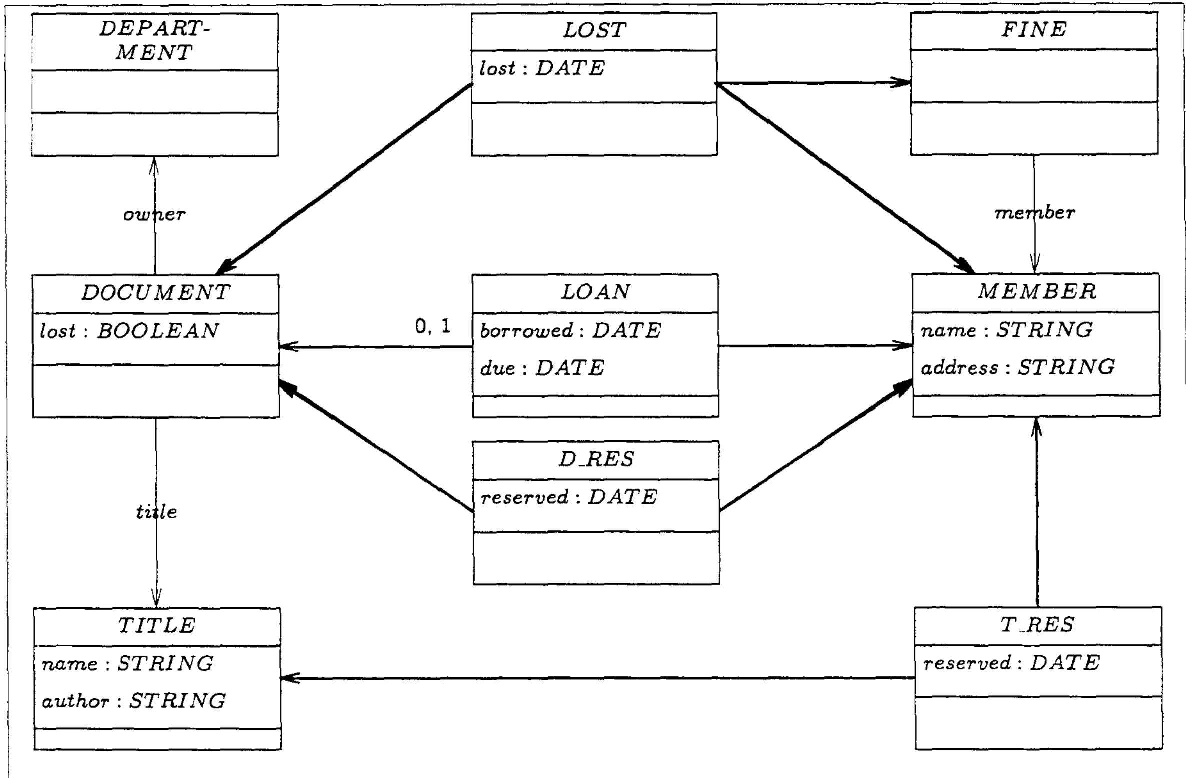



Figure 8.4: Object diagram produced from the transaction/use table.

document itself, and if the member is allowed to renew the loan. If these conditions are satisfied, the *LOAN* relationship is updated by setting a new return date. When a document is lost by a member, a *LOST* record is created that relates the lost document and the member, and a *FINE* is created that records the fine to be paid for this loss. The other entries of the table are self-explanatory.

To understand the table correctly, it is important to realize that *surrogates* for real-world entities are created, read, updated and deleted. For example, when we would include in the left-most column of the table the transaction order of the Finance department, then we would put down the *DOCUMENT* class under the Create column. This does not mean that ordering a document creates a document; it means that ordering a document creates a document *surrogate* in the system.

It is also important to realize that several instances of a class may be created, read, updated or deleted by a transaction. An entry in the table merely represents whether at least one instance of a class is accessed (i.e. created, read, updated or deleted) by a transaction.

A transaction/use table provides sufficient data to build the first version of an object diagram. Figure 8.4 shows an object diagram built from figure 8.3 The object class *DEPARTMENT* is not warranted by the transaction/use table but is the result of an analysis of the queries that the system must be able to answer. (Remember that these also appear in the function decomposition tree. As shown by the outline of the method in figure 8.2, object analysis of transactions can also be applied to queries.) Some of the attributes and cardinality constraints have already been shown in the diagram. Attributes are usually found gradually, during various tasks related to object modeling. For example, one may find that certain data are needed to specify event preconditions and that attributes to hold this data must be added. Another way to find the relevant attributes for objects is by checking

1. Invert the transaction/use table, marking the entries of the inverted table with a C, R, U or D.
2. Reduce the number of object accesses by applying transaction allocation criteria.
3. Split transactions whose decomposition is not conjunctive.
4. Choose names for local events.
5. Extend the object model with local event names.

Figure 8.5: Outline of a transaction decomposition procedure.

whether all relevant queries can be answered. This is discussed in chapter 9 on evaluation methods. The diagram also contains a cardinality constraint on *LOAN*. Like attributes, constraints are found gradually.

### 8.3.2 Elementary sentence analysis of transactions

Elementary sentence analysis has been criticized because it leads to a large set of mostly irrelevant elementary sentences, which produce a large set of mostly irrelevant candidate classes. A useful way to reduce the number of irrelevant sentences and classes found by elementary sentence analysis is to describe each transaction of the function decomposition tree by means of an elementary sentence. This is a sentence which have the simplest possible form of

Noun phrase – Verb phrase – Noun phrase

and that use a restricted subset of English. Each noun in an elementary sentence is a candidate object– or role class and each verb in an elementary sentence is a candidate for a relationship class. Often, the verb represents the transaction as a whole and then is likely to represent a communication between objects. More on elementary sentence analysis can be found in Rock-Evans [95, pages 44–53], Batini, Ceri and Navathe [8, pages 86–90] and Nijssen and Halpin [86].

### 8.3.3 Transaction decomposition

The list of required DBS transactions can be used to find the local events in the life of objects by **transaction decomposition**. The assumptions of transaction decomposition are:

1. Each transaction is a finite non-empty set of local events. If it consists of more than one event, then it is a synchronous execution of its component events, each of which is local to an object.
2. Different local events in a transaction occur in the life of different objects.
3. Any relevant UoD event is modeled by a DBS transaction. This means that objects communicate only through transactions.

As a first approximation, we can assume that the local events in a transaction all have the same name as the transaction itself. These names can be replaced by more intuitively reasonable names later on. The problem then is to *allocate* the transaction to one or more objects in whose life it occurs. Figure 8.5 gives an outline of a procedure by which to do this.

Transaction decomposition is equivalent to the allocation of events to classes. The simplest way to allocate transactions to objects is to take the *transaction/use table* and reorder it, so that the columns correspond to transactions and the rows to object classes. This reorganization is called **inversion** of the transaction/use table. An entry in the table contains a C,R,U or D according to the kind

| | DOCUMENT | D_RES | FINE | LOAN | LOST | MEMBER | TITLE | T_RES |
|---|---|---|---|---|---|---|---|---|
| *reserve_title* | R | | | | | R | U | C |
| *reserve_doc* | RU | C | | | | R | | |
| *borrow* | | RD | | C | | R | | RD |
| *renew* | | R | | RU | | R | | R |
| *return* | | | | RD | | | | |
| *lose* | U | | C | RD | C | U | | |
| *cancel* | | D | | | | | | D |
| *remind* | | | | RU | | R | | |

Figure 8.6: First version of a transaction decomposition table, obtained by reorganizing the transaction/use table.

| | DOCUMENT | D_RES | FINE | LOAN | LOST | MEMBER | TITLE | T_RES |
|---|---|---|---|---|---|---|---|---|
| *reserve_title* | | | | | | | | C |
| *reserve_doc* | | C | | | | | | |
| *borrow* | | D | | C | | | | D |
| *renew* | | | | U | | | | |
| *return* | | | | D | | | | |
| *lose* | | | C | D | C | | | |
| *cancel* | | D | | | | | | D |
| *remind* | | | | U | | | | |

Figure 8.7: Reduced version of a transaction decomposition table, obtained by applying transaction allocation criteria.

- **Updating**. An event $e$ *must* be allocated to object class $C$ when the $e$ must *change the state* of an instance of $C$.
- **Querying**. An event $e$ *must* be allocated to object class $C$ when the DBS must *remember* the fact that $e$ occurred in the life of an instance of $C$, for example because queries can be asked about it.
- **Locality**. An event $e$ *can* be allocated to object class $C$ if we can say that an instance of $C$ performed or suffered the event one or more times. This criterion says when it is *meaningful* to allocate an event to an object, not when it is *necessary* to do so.

Figure 8.8: Transaction allocation criteria. These are at the same time the quality checks for event allocation.

of access that the transaction has to the object class. Figure 8.6 shows the result of inverting the transaction/use table of figure 8.3.

Inversion usually gives a very dense communication structure, in which every transaction is a communication between a large number of objects. This makes the model unnecessarily complex and hard to understand. As was already stated earlier, each class specification is easy to understand in isolation, but that we need the context in which the event occurs in order to understand its meaning fully. This requires an understanding of the communication structure of the model. The more communications there are, and the more objects participate in a communication, the harder it is to understand what the meaning of an event is.

Luckily, the number of communications and the number of events per communication can usually be reduced without affecting the quality,. truth-value or utility of the model. To carry out this reduction, we use the **transaction allocation criteria** of figure 8.8. These are at the same time the quality checks for event allocation, already given in section 6.7.2. Figure 8.7 shows the result of applying these criteria to figure 8.6. The following considerations lead to the reduction in this example.

- *reserve_title* needs read access to *DOCUMENT* and *MEMBER* because it must be checked whether the member is allowed to reserve documents (e.g. is not suspended) and whether all documents of this title are unavailable (e.g. borrowed, lost, at the binder). These read accesses are performed in evaluating the precondition of *reserve_title* and are not modeled as transactions in themselves. We do not need a local event in the life of a member and a local event in the life of all documents of the reserved title to realize this read access. The same considerations lead to removing all R's from the table.
- A *TITLE* must be updated by a *reserve_title* transaction because the DBS must be able to answer the question whether a title is reserved, and by whom. However, *reserve_title* creates a *T_RES* instance and this is a relationship containing the identifier of the reserved title as component. This allows us to retrieve the necessary information without updating *TITLE*. We therefore drop the U from the *TITLE* entry of *reserve_title*. The same argument applies to *reserve_doc* (no update of borrowed *DOCUMENT* needed), and to *lose* (no update of *DOCUMENT* and *MEMBER* needed)

We now check whether each decomposition is **conjunctive**. By this is meant that each transaction must have a unique decomposition into a set of events. Given a transaction $t$ applied to arguments,

$$t(c_1, \ldots, c_n),$$

there should be exactly one column in the transaction decomposition table which says how to decompose this transaction. This is so even in the presence of overloading transaction names, for once we supply the transaction arguments, the overloading is resolved.

It is not allowed to have a choice of decompositions for a single transaction, because a transaction would then be nondeterministic. To illustrate, in figure 8.7, *borrow* and *cancel* events really have *disjoint* effects: If a document is borrowed that is not reserved, only a *LOAN* instance is created; if a reserved document is borrowed, then in addition, either a *T_RES* instance or a *D_RES* instance is deleted. This gives us *three* distinct borrow actions, which should really be modeled as such: borrowing after a title reservation, borrowing after a document reservation, and borrowing without a reservation. For *cancel*, we find two distinct actions, depending upon whether a title- or document reservation is canceled. The result of splitting these actions is shown in figure 8.9. For consistency of the documentation, the function decomposition tree and the transaction/use table must be updated to reflect this discovery of new transactions; this is not done here. Note that splitting the disjunctive transactions resulted in an even simpler communication structure than we had.

Finally, we choose names for the local events. The result is shown in figure 8.10. We could also show a C, U or D in the entries, but this would clutter up the table a bit.

We can represent the communications also in a *simplified communication diagram*, shown in figure 8.11. Figure 8.12 shows the object diagram for the circulation DBS, extended with the event names.

## 8.4 Scenario analysis

The method in this section is based on methods described by Jackson [49] for JSD, by Jacobson et al. [50] for Objectory and by Rumbaugh et al. [99] for OMT. Remember that the *behavior* of a system *S* consists of its interactions with all systems in its environment, whereas a *dialog* is that part of system behavior that consists of interactions with a few selected systems in its environment. In particular, a *life cycle* is the dialog that a surrogate in the DBS has with its counterpart in the UoD.

**Scenario analysis** is a method to find a representation of system behavior, including dialogs and life cycles. A **scenario** is a history of interactions between a system and one or more other systems. Scenarios are called *use cases* by Jacobson [50]. A scenario can be used to find a representation of a dialog, and if we study several scenarios, we may be able to discover a representation of the total behavior of a system. If we study scenarios involving only the dialog between a surrogate and its real-world counterpart, then we may be able to discover object life cycles.

Scenario analysis may not always yield results, simply because objects have no life cycle other than *birth . do_anything* . death*. In other cases there may be a lot more structure, but it it may be hard to give a guarantee that absolutely all possible behavior has been covered if we use a process graph that represents any behavior more complex than a simple iteration over all possible events between birth and death.

### 8.4.1 Narrative scenario analysis

A scenario can be written down as a narrative. For example, figure 8.13 shows a narrative describing a scenario in which a document is borrowed. This simple scenario can be represented by many different process graphs, of which figure 8.14 shows one. This process graph embodies a number of *hypotheses* about borrowing behavior, including the following:

- After *create*, a *LOAN* is in a state in which only two events can occur, *return* and *overdue*.
- After *renew*, a *LOAN* is in the same state as it is after *create*.

These hypotheses will be falsified if we investigate a scenario in which after *create*, the borrowed document is lost. This falsification will lead to a refinement of the *LOAN* life cycle, in which a *lose* event wil be added after *create*, etc. This refined life cycle must be tested against yet other scenarios, etc.

| | DOCUMENT | D_RES | FINE | LOAN | LOST | MEMBER | TITLE | T_RES |
|---|---|---|---|---|---|---|---|---|
| *reserve_title* | | | | | | | | C |
| *reserve_doc* | | C | | | | | | |
| *t_res_borrow* | | | | C | | | | D |
| *d_res_borrow* | | D | | C | | | | |
| *borrow* | | | | C | | | | |
| *renew* | | | | U | | | | |
| *return* | | | | D | | | | |
| *lose* | | | C | D | C | | | |
| *t_cancel* | | | | | | | | D |
| *d_cancel* | | D | | | | | | |
| *remind* | | | | U | | | | |

Figure 8.9: The result of splitting transactions whose decomposition is not conjunctive.

| | DOCUMENT | D_RES | FINE | LOAN | LOST | MEMBER | TITLE | T_RES |
|---|---|---|---|---|---|---|---|---|
| *reserve_title* | | | | | | | | *reserve_title* |
| *reserve_doc* | | *reserve_doc* | | | | | | |
| *t_res_borrow* | | | | *create* | | | | *t_res_borrow* |
| *d_res_borrow* | | *d_res_borrow* | | *create* | | | | |
| *borrow* | | | | *create* | | | | |
| *renew* | | | | *renew* | | | | |
| *return* | | | | *return* | | | | |
| *lose* | | | *lose* | *lose* | *lose* | | | |
| *t_cancel* | | | | | | | | *t_cancel* |
| *d_cancel* | | *d_cancel* | | | | | | |
| *remind* | | | | *remind* | | | | |

Figure 8.10: Transaction decomposition table for the circulation DBS.

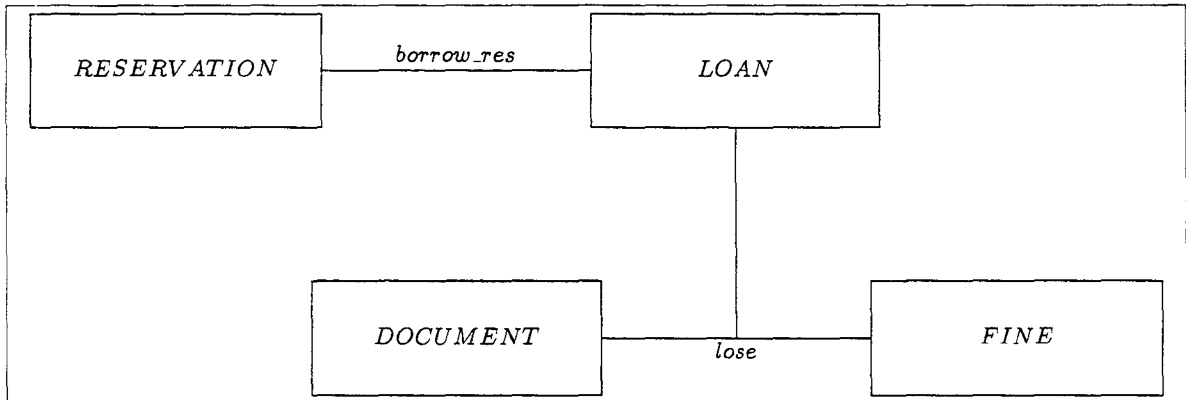



Figure 8.11: Simplified communication diagram of the circulation desk DBS.

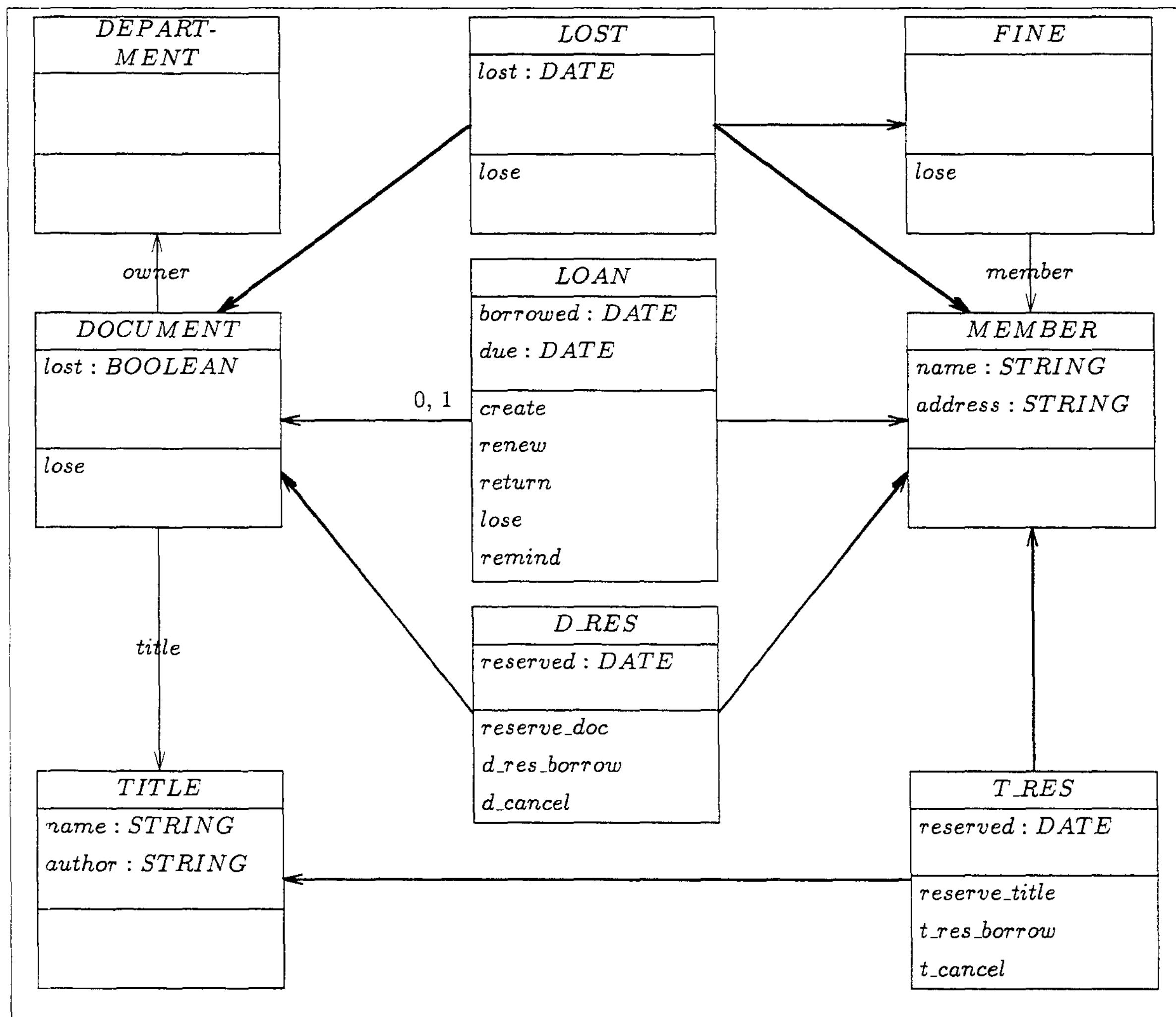



Figure 8.12: Object diagram extended with some event names.

Fred borrows *An Introduction to Database Systems*, Volume 1 and keeps it for 4 weeks. He receives a reminder from the library and then extends the loan period with another three weeks. After two weeks, he returns the book.

Figure 8.13: A narrative description of a document borrowing scenario.

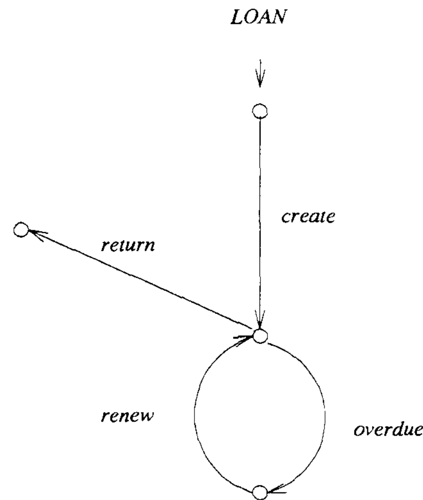



Figure 8.14: Process graph compatible with the scenario in figure 8.13.

In general, a life cycle must be corroborated against a large number of scenarios before we have reasonable certainty that is represents all possible courses of events. Total certainty connot be reached, because there is an *inductive jump* from a finite number of scenarios to a hypothesis about the life cycle. The life cycle covers infinitely many scenarios and can therefore not be proved correct on the basis of a finite number of examples.

Note that a scenario is written on a certain level of abstraction. It does not mention the dialog conducted when a document is borrowed, not does it mention the events that lead to Fred receiving a reminder. Either we already decided upon a level of transaction atomicity before writing the scenario, or this decision was made when writing the scenario. However, it is important that the decision about which events are relevant is not made implicitly. It can be recommended to perform scenario analysis only after we are reasonably sure about which transactions are relevant. Scenario analysis can then act as a check whether these transactions are indeed relevant — for each transaction there should be a scenario in which it occurs — and whether all relevant transactions have been found.

Another important decision to be made before or during scenario analysis, besides the identification of the relevant transactions, is the allocation of transactions to systems. If we use scenario analysis to find life cycles, then this question comes down to the question to which class the life cycle must be allocated, which reduces to the question which local events we can use to build the life cycle.

If the processes we build using scenario analysis consist of transactions shared by several systems, such as a thread of control or a dialog, then we cannot allocate the process to a single system. In this case, it may be better to use event traces instead of narratives, as shown next.

### 8.4.2 Event trace analysis

A scenario can always be represented by an **event trace diagram**. Figure 8.15 shows an event trace diagram of an elevator control system. The horizontal axis shows a number of individual systems that

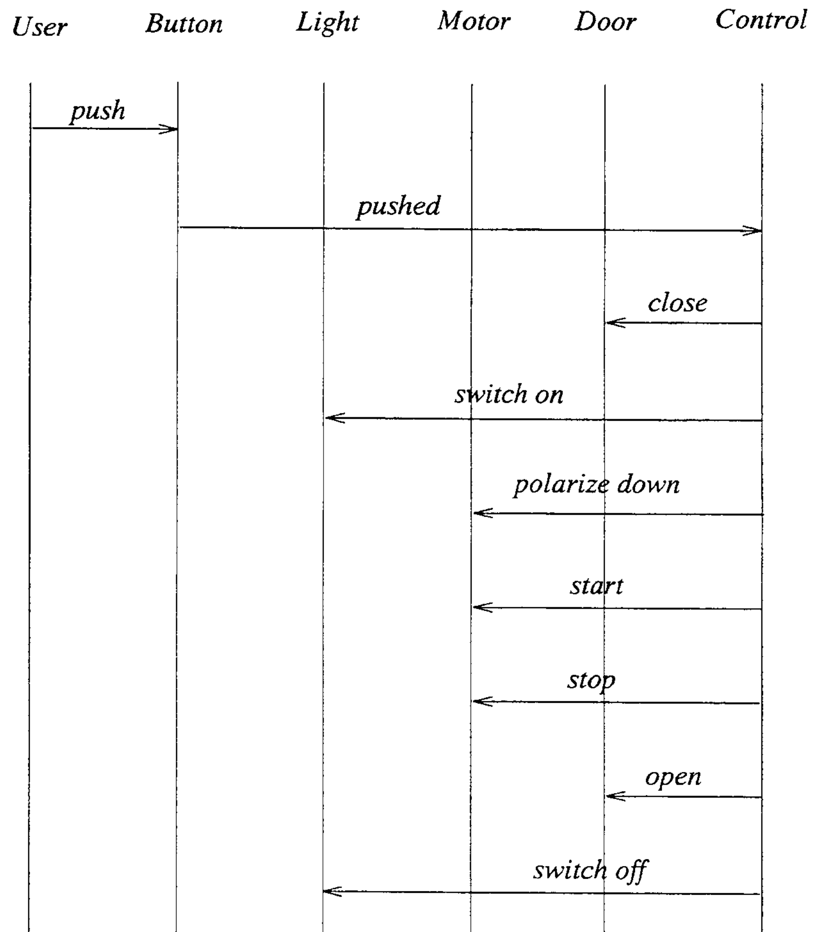



Figure 8.15: Event trace diagram of an elevator control system.

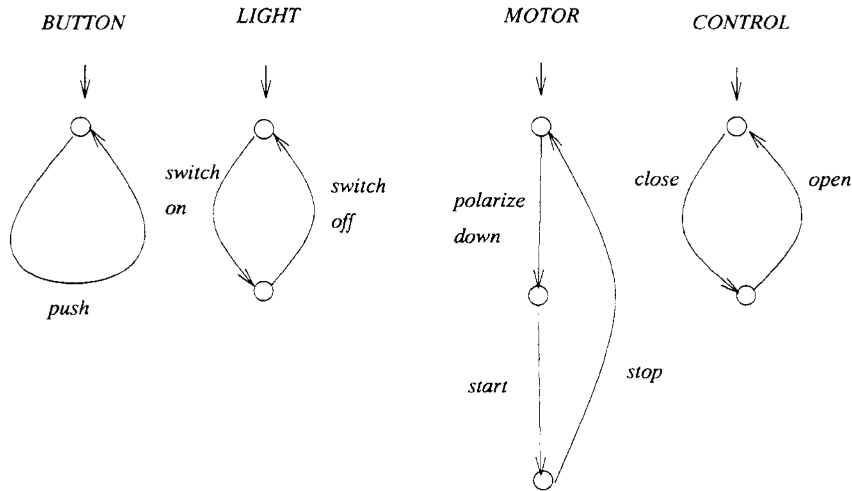



Figure 8.16: Life cycles built from an event trace diagram.

are engaged in transactions and the vertical axis shows the passage of time. Each occurrence of a transaction in the scenario is shown by an arrow from the sender of the transaction to the receiver.

An event trace diagram contains the information of a context diagram (at the level of transactions) plus information about the sequence of transactions in a particular scenario. We can therefore derive a context diagram from an event trace diagram that represents the interaction between the systems involved in this scenario.

We can use an event trace diagram to form a hypothesis about the life cycles of all systems depicted in it, but or about the dialogs between systems, or about a thread of control that moves through a number of systems. For example, from the event trace diagram in figure 8.15, we can hypothesize the life cycles shown in figure 8.16. The elevator control process is not shown, because from a single event trace diagram we can only infer a simple sequential process that goes through all events in the diagram, from top to bottom.

Event trace analysis assumes that we already have identified relevant systems and relevant transactions between those systems. This again illustrates that we should first identify those by other means than scenario analysis.

# Chapter 9

# Evaluation Methods for DBS Models

We now return to DBS models by including queries in our view. There are three kinds of evaluation methods for DBS models.

1. **Quality checks** that can be used to find out whether we made good use of the available modeling constructs. If we made appropriate use of the constructs, the model tends to be simpler, more pleasing, or even more "beautiful" than it would have been otherwise. Quality checks are applied interleaved with whatever induction method we are using, and they have already been discussed together with the modeling constructs to which they apply.

2. **Validation checks** are used to check if the model is a true representation of the requirements. This must be sharply distinguished from the question whether the requirements themselves are good, i.e. whether they will lead to an improved state of affairs. This last question is a development question, whereas the validation question is a modeling question. Validation checks are discussed in section 9.1.

3. **Utility checks** are used to find out whether the specified DBS would, indeed, be useful. This is really a development question, but it is a question that can only be asked and answered when we have a specification of a DBS, and an answer to the question will make us change the specification in order to make it more useful. Utility checks are discussed in section 9.2.

These checks are oriented to respectively appropriateness, truth and utility or, to give them an even more elevated status, to the classical triad Beauty, Truth and Goodness.

## 9.1 Validation methods

An important feature of conceptual modeling in the context of information system development is that the modeled DBS *does not yet exist.* This poses a problem for the validation of the truth-value of the conceptual model. This problem can be tackled by using a number of validation techniques.

- If we restrict ourselves to the class model, we can use the **elementary sentence check**. In this check we describe the class model in simple, general sentences and verify the resulting text with domain specialists.

- Still restricting us to the class model, we can use the **population check** to check whether the possible states of our model are plausible. In this check, we present a possible state of the DBS as specified by the class diagram in a readable format and ask the domain specialist whether this represents a possible state of the UoD. This is a useful way to check cardinality constraints. We can also try to produce states required to occur in the DBS or observed to occur in the UoD. The population check is important in NIAM [86].
- A **walkthrough** takes a domain specialist and/or others through the behavior described by the model to verify whether this is behavior is really possible and meaningful. This is a manual animation of the model. Walkthroughs are used in Structured Analysis.
- A graphical model of behavior, like behavior diagrams, can be animated by a software system with a graphical interface to make clear what has been specified. The animation system can attempt to produce behavior observed to occur in the UoD or required to occur in the DBS. Such an animation is proposed by Ward and Mellor for control flow diagrams [122] and it is used routinely (as the "token game") in Petri net applications.

All these checks come down to some form of **model animation**. In general animation consists of showing interesting properties of a model in a user-friendly, understandable way. Usually, this involves some graphical user interface that shows part of the model and shows some kind of movement in order to illustrate the the possible states and possible behaviors of the specified system. Animation is a current research topic.

## 9.2 Utility checks

The model should not only be true, it should also represent a useful system. One utility check that is frequently used in ER modeling is the **query check**. The query check is explained by Batini et al. in the context of ER modeling [8, pages 238-242]. This check can be applied to the class model of `MCM`. What we do is simply define navigation paths for all relevant queries to the DBS to see if they can be answered. These paths can be shown on the class diagram to make plausible that we will be able to implement the queries later on. Using the query check, we can verify whether all information needs identified during information needs analysis during the requirements determination task can be met.

In addition to answering queries, a DBS may have other functions, notably directive and declarative control functions. To check whether these functions are provided by the modeled system, we can build a **prototype** of the system represented by the model. This prototype should not contain all required system functionality — no error-handling, a rudimentary interface and poor performance for example — but it should contain the functions that we want to check can be implemented in the system. These functions can include query answering and control functions. Such prototypes can be easily generated using current fourth-generation system generation tools.

It is important that the prototype realizes only some of the required functionality, for otherwise we would not be building a prototype but the final system. The prototype should be experimental, easy to build and easy to change.

# Chapter 10

# Summary and conclusions

If we want to achieve progress in the field of methodology, we must not start from scratch in building a method but reuse as much as possible from older methods. MCM has been designed after studying a number of existing methods and is built by reusing good ideas from other methods. Sources for the ideas presented in this report have been noted wherever the ideas are used.

However, MCM does contain some new ideas in comparison with other methods, and I hope that overall, MCM will turn out to be an improvement with respect to other methods. Features of MCM include the following:

- Building a method from existing components where possible.
- Using the regulatory and empirical cycles as frameworks for development and modeling, respectively.
- Combining object-oriented modeling with formal specification.
- A flexible theory of taxonomic structures that includes static and dynamic subclasses as well as multiple role-playing.
- Explicit listings of quality checks that can be applied to the use that has been made of the available modeling constructs.
- The restriction of object communication to DBS transactions.
- The close link between formal and informal specification.

MCM presupposes a method engineering framework, that has been explained in the first two chapters of the report. This framework calls for a composition of an information system development method from four kinds of sources: requirements determination methods, conceptual modeling methods, implementation methods and project management methods. MCM can be used as the conceptual modeling component of such an information system development method.

Current and future research includes a number of important extensions to MCM and LCM. An important element present in MCM but as yet still absent from LCM is the ability to specify dynamic subclasses and roles. This will be added to LCM. In addition, there are many possible and useful joint extensions to MCM and LCM, which are listed in the following sections.

## 10.1 Method engineering and contingency

More research needs to be done on the kinds of methods that can be combined in requirements determination. This involves at least the following three questions:

- Which methods and techniques for requirements determination are logically compatible, and how can they be combined?
- Which guidelines are there for combining particular requirements determination methods in particular circumstances?
- Which requirements determination methods are compatible with MCM? In other words, what does the transition from requirements determination to MCM look like?

Answering these questions requires empirical, and possibly experimental, research.

## 10.2 Network modeling

The current trend in current data processing systems is to connect DBSs in networks and to let systems at the nodes of a network play a more active role than is played by traditional commercial database systems. For example, DBSs can place order periodically over an EDI network, or DBSs are connected with production control systems and computer-integrated manufacturing systems. These advanced applications call for a method by which one can model the behavior of data processing systems in a network. This would involve modeling dialogs between systems and modeling threads of control that are realized by a set of communicating systems in a network. This extension of MCM is subject of current research.

## 10.3 Real time and deontic logic

Network systems call for a battery of advanced techniques for specification languages, including real time and deontic logic. For example, if one system sends a directive control message to another, it puts the other system under an obligation to do something, without having the power to ensure that the other system does indeed perform the obligation. This means that systems will have to be able to represent violations occurring in their environment and take appropriate action in case violations occur. To deal with this adequately in the specification language, we need deontic logic. In addition, we need real time, because obligations are usually obligations to perform an action before a real-time deadline. Another important issue in this respect is the synchronization of clocks in a distributed system. Some of these issues have already been studied in the past by others, and all we need to do is gather the results of that research and incorporate it in LCM and MCM. Other issues need more research, and some of this research is planned for the near future.

## 10.4 Historical DBS modeling

Independently from whether a DBS functions in a network, there is a need to represent and manipulate historical data. This has always been the case in administrative database systems and it will also be the case in networked systems. This calls for an extension of the specification language with a past tense temporal logic and this involves corresponding extensions to the modeling method. These extensions are planned for LCM and MCM in the future.

## 10.5 Animation

Even without all planned additions, validating a LCM specification of a MCM method is a daunting task that needs computational support. Current research includes defining a subset of LCM that is useful for conceptual modeling and at the same time executable on a computer. A prototype animation for this subset will be implemented, that allows the modeler to get answers to questions about the model in such a way that his or her understanding of the model is increased. The system will involve theorem-proving techniques.

## 10.6 Metamodeling

A workbench for MCM used should not only include an animation system, it should also include editors and syntax checkers to help writing a specification. In addition, it should support the user of the workbench in the application of heuristics and quality checks during the modeling process. This calls for an extension of the concept of a repository with a model of the development process. The informal description of the relations between the different parts of a model in chapter 7 is a first step in the direction of a specification of a repository. The field of software process modeling is still in its infancy, and it is not yet clear whether it is at all possible to give automated support beyond making the modeling chapters of this report interactively available, say through a hypertext system. I belief it is impossible to automate the whole software process, or even to describe it as if it were an automated process. Modeling is a process that moves from what is informal to what is formal. The possibility to bridge the gap between the formal and informal worlds is, in my opinion, one of the great mysteries of the human mind and cannot be implemented in a machine [128]. Nevertheless, it is interesting to try how far we can get in providing automated support for the modeling process that goes beyond current CASE tools and yet does not get in the way of what the modeler tries to do.

## 10.7 Query modeling

Query modeling has been ignored in this report. However, the benefits of formal specification would increase dramatically if we can use the specification language as a query language for the specified database system. In the case of LCM, this would involve a kind of equational query language, in which the query is a set of equations to be solved against the database. Addition of modal operators for transitions, past time, obligations, permissions, prohibitions etc. would increase the utility of such a query language by an order of magnitude. Although the computational complexity may be beyond anything we can hope to cope with, this remains a tantalizing research topic. No research is currently being planned in this direction, but it remains on the shopping list of things to do in the (unspecified) future.

## 10.8 User interface modeling

In this report we ignored all aspects connected to user interfaces. A natural way to add user interface modeling to MCM is to associate one screen form to every transaction, but other alternatives are possible. Current fourth-generation languages go a long way in allowing the modeler to prototype user interfaces and test them with users, but it is likely that with the advent of high-level formal specification languages like LCM and advanced modeling methods like MCM, new problems as well as new possibilities arise. No research is currently planned for user interface modeling, but the topic is important enough to deserve mentioning here.

## 10.9 Modeling user procedures

Another important aspect of conceptual modeling that is ignored in this report is the modeling of user procedures, or, more generally, modeling the work users have to do. This calls for an entirely different set of modeling methods and techniques, that are studied in organizational psychology. The current and future generation of system users are more critical than ever, and they will demand due care to be taken that the quality of working life is enhanced by the introduction or redevelopment of data processing systems. The method engineering framework of chapter 2 contains a slot in which methods for the design of user procedures can be put, but this is clearly not sufficient. The method should not merely contain a slot for designing user procedures, it should contain hooks by which it can be linked to methods for designing high-quality workplaces. More research is needed into the relations between the formal model of mechanical work and the informal model of human work, and into the quality that one can build into mechanical as well as human work. No research in this topic is currently planned, but it remains high on the list of topics to be studied later.

# Bibliography


[1] R.J. Abbott. Program design by informal English descriptions. *Communications of the ACM*, 26:882–894, 1983.

[2] W.W. Agresti, editor. *New Paradigms for Software Development.* Computer Society Press, 1986.

[3] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In R. Agrawal, S. Baker, and D. Bell, editors, *Proceedings of the 18th International Conference on Very Large Databases*, pages 39–51, Dublin, Ireland, August 24–27 1993.

[4] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10:230–260, 1985.

[5] C.W. Bachman and M. Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Databases*, pages 464–476, 1977.

[6] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[7] R. Barker and C. Longman. *Case*Method: Function and Process Modelling.* Addison-Wesley, 1992.

[8] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[9] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[10] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[11] J.A. Bergstra and J.W. Klop. Algebra of communicating processes. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science (CWI Monographs 1)*, pages 89–138. North-Holland, 1986.

[12] J. Cameron, editor. *JSP & JSD - The Jackson Approach to Software Development.* IEEE Computer Science Press, second edition, 1989.

[13] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Datatypes*, pages 51–67. Springer, 1984. Lecture Notes in Computer Science 173.

[14] G. Chambers. Predicate classes. In *European Conference on Object-Oriented Programming (ECOOP93)*, pages 268–296. Springer, 1993.

[15] P. Checkland and J. Scholes. *Soft Systems Methodology in Action.* Wiley, 1990.

[16] P.B. Checkland. *Systems Thinking, Systems Practice.* Wiley, 1981.

[17] P.P.-S. Chen. The entity-relationship model – Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[18] P.P.-S. Chen. English sentence structure and Entity-Relationship diagrams. *Information Sciences*, 29:127–149, 1983.

[19] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press/Prentice-Hall, 1990.

[20] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4:397–434, 1979.

[21] S. Conrad, M. Gogolla, and R. Herzig. TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[22] J.F. Costa, A. Sernadas, and C. Sernadas. *OBL-89 User's Manual, version 2.3.* Instituto Superior Técnico, Lisbon, May 1989.

[23] L.W. Crumm. *Value Engineering: The Organized Search for Value.* Longman, 1971.

[24] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 5th edition, 1990.

[25] G.B. Davis. Strategies for information requirements determination. *IBM Systems Journal*, 21:4–30, 1982.

[26] F. Dehne. Formal specification and transformational implementation of registration systems in CMSL. Master's thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[27] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press/Prentice-Hall, 1978.

[28] E. Downs, P. Clare, and I. Coe. *Structured Systems Analysis and Design Method: Application and Context.* Prentice-Hall, second edition, 1992.

[29] L. Drenth and I.L. Paulides. Object modeling technique and conceptual model specification language: A comparison. Master's thesis, Faculty of Mathematics and Computer Science, Free University, De Boelelaan 1081a, 1081HV Amsterdam, 1993.

[30] J.M. Edwards and B. Henderson-Sellers. A graphical notation for object-oriented analysis and design. *Journal of Object-Oriented Programming*, pages 53–74, February 1993.

[31] R.B. Feenstra and R.J. Wieringa. LCM 3.0: a language for describing conceptuel models. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[32] P. Feyerabend. *Against Method: Outline of an Anarchistic Theory of Knowledge.* Verso, 1975.

[33] J. FitzGerald and A. FitzGerald. *Fundamentals of Systems Analysis.* John Wiley, 1987. Third edition.

[34] M. Flavin. *Fundamental Concepts of Information Modeling.* Yourdon Press, 1981.

[35] D. Gabbay and J.M. Moravcsik. Sameness and individuation. *The Journal of Philosophy*, 70:513–526, 1973.

[36] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Prentice-Hall, 1979.

[37] A. Giddens. *New Rules of Sociological Method: A Positive Critique of Interpretative Sociologies.* Basic Books, 1976.

[38] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Vrije Universiteit/Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[39] R.J. van Glabbeek. The linear time-branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90. Theories of Concurrency: Unification and Extension.* Springer, 1990. Springer Lecture Notes in Computer Science 458.

[40] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[41] A. Goldberg and D. Robson. *Smalltalk-80: The language.* Addison-Wesley, 1989.

[42] J.J. van Griethuysen (ed.). Concepts and terminology for the conceptual schema and the information base. Technical Report TC97/SC5/WG3, International Organization of Standards, 1982.

[43] A.D. Hall. *A Methodology for Systems Engineering.* Van Nostrand, 1962.

[44] P. Hall, J. Owlett, and S. Todd. Relations and entities. In G.M. Nijssen, editor, *Modelling in Database Management Systems*, pages 201–220. North-Holland, 1976.

[45] T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kusch. TROLL–2 report. Informatik-bericht, TU Braunschweig, 1993. *In preparation.*

[46] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, 1987.

[47] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19:201–260, 187.

[48] B. Ives and G.P. Learmont. The information system as a competitive weapon. *Communications of the ACM*, 27(12):1193–1201, December 1984.

[49] M. Jackson. *System Development.* Prentice-Hall, 1983.

[50] I. Jacobson, M. Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Prentice-Hall, 1992.

[51] H.W.B. Joseph. *An Introduction to Logic.* Clarendon Press, 1916.

[52] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[53] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-oriented specification of information systems: The TROLL language. Technical report, Abt. Datenbanken, Tech. Universität Braunschweig, P.B. 3329, Braunschweig, Germany, 1991.

[54] J.G. Kemeny. *A Philosopher Looks at Science.* Van Nostrand, 1959.

[55] K.E. Kendall and J.E. Kendall. *Systems Analysis and Design.* Prentice-Hall, 1992. Second edition.

[56] W. Kent. A rigorous model of object reference, identity, and existence. *Journal of Object-Oriented Programming*, 4(3):28–36, June 1991.

[57] S.N. Khoshafian and G.P. Copeland. Object identity. In *Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, 1986. SIGPLAN Notices 22 (12).

[58] S.O. Kimbrough. On representation schemes for promising electronically. *Decision Support Systems*, 6:99–121, 1990.

[59] S.O. Kimbrough, R.M. Lee, and D. Ness. Performative, informative and emotive systems: The first piece of the PIE. In L. Maggi, J.L. King, and K.L. Kraenens, editors, *Proceedings of the Fifth Conference on Information Systems*, pages 141–148, 1984.

[60] W.R. King. Strategic planning for information systems. *MIS Quarterly*, 2(1):27–37, 1978.

[61] J.A. Kowal. *Analyzing Systems.* Prentice-Hall, 1988.

[62] T. Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, second, enlarged edition edition, 1970.

[63] L. Laudan. *Progress and Its Problems. Towards a Theory of Scientific Growth.* University of California Press, 1977.

[64] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, October 1986.

[65] M. Lundeberg. The ISAC approach to specification of information systems and its application to the organization of an IFIP working conference. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*, pages 173–234. North-Holland, 1982.

[66] M. Lundeberg, G. Goldkuhl, and A. Nilsson. A systematic approach to information systems development - I. Introduction. *Information Systems*, 4:1–12, 1979.

[67] M. Lundeberg, G. Goldkuhl, and A. Nilsson. A systematic approach to information systems development - II. Problem and data oriented methodology. *Information Systems*, 4:93–118, 1979.

[68] M. Lundeberg, G. Goldkuhl, and A. Nilsson. *Information Systems Development — A Systematic Approach.* Prentice-Hall, 1981.

[69] B.J. MacLennan. Values and objects in programming languages. *Sigplan Notices*, 17(12):70–79, December 1982.

[70] J.-L. Malouin and M. Landry. The mirage of universal methods in systems design. *Journal of Applied Systems Analysis*, 10:47–62, 1983.

[71] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent System Specification.* Springer, 1992.

[72] D.A. Marca and C.L. Gowan. *SADT: Structured Analysis and Design Technique.* McGraw-Hill, 1988.

[73] J. Martin. *Strategic Data-Planning Methodologies.* Prentice-Hall, 1982.

[74] J. Martin. *Information Engineering.* Prentice-Hall, 1989. Three volumes.

[75] J. Martin. *Information Engineering, Book I: Introduction.* Prentice-Hall, 1989.

[76] J. Martin. *Information Engineering, Book II: Planning and analysis.* Prentice-Hall, 1989.

[77] J. Martin. *Information Engineering, Book III: Design and construction.* Prentice-Hall, 1989.

[78] J. Martin and J.J. Odell. *Object-Oriented Analysis and Design.* Prentice-Hall, 1992.

[79] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis.* Yourdon Press/Prentice Hall, 1984.

[80] R. Milner. *A Calculus of Communicating Systems.* Springer, 1980. Lecture Notes in Computer Science 92.

[81] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[82] H. Mintzberg, D. Raisinghani, and A. Théorêt. The structure of "unstructured" decision processes. *Administrative Science Quarterly*, 21, June 1976.

[83] E. Mumford and M. Weir. *Computer Systems in Work Design – The ETHICS Method.* Associated Business Press, 1979.

[84] E. Nagel. *The Structure of Science.* Routledge and Kegan Paul, 1961.

[85] J.D. Naumann, G.B. Davis, and J.D. McKeen. Determining information requirements: a contingency method for selection of a requirements assurance strategy. *Journal of System and Software Sciences*, 1:273–281, 1980.

[86] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design.* Prentice-Hall, 1989.

[87] C. Pava. *Managing New Office Technology: An Organizational Strategy.* The Free Press, 1983.

[88] J. Peckham and F. Maryanski. Semantic data models. *ACM Computing Surveys*, 20:153–189, 1988.

[89] B. Pernici. Objects with roles. In *IEEE/ACM Conference on Office Information Systems*, Cambridge, Mass., 1990.

[90] M.E. Porter. *Competitive Strategy: Techniques for Analyzing Industries and Competitors.* The Free Press, 1980.

[91] M.E. Porter. *Competitive Advantage: Creating and Sustaining Superior Performance.* The Free Press, 1985.

[92] U. Reimer. A representation construct for roles. *Data and Knowledge Engineering*, 1:253–251, 1985.

[93] N. Rescher. *Introduction to Logic.* St. Martin's Press, 1964.

[94] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple, independent roles. In *ACM-SIGMOD International Conference on Management of Data*, pages 298–307, Denver, Colorado, May 1991. ACM. Sigmod Record, Vol. 20.

[95] R. Rock-Evans. *A Simple Introduction to Data and Activity Analysis.* Computer Weekly Publications, 1989.

[96] J.F. Rockart. Chief executives define their own data needs. *Harvard Business Review*, 57(2):81–93, April/May 1979.

[97] P. Rook. Project planning and control. In J.A. McDermid, editor, *Software Engineer's Reference Book*, page Chapter 27. Butterworth/Heinemann, 1992.

[98] D.T. Ross. Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, SE-3(1):16–34, January 1977.

[99] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[100] M. Saeki, H. Horai, and H. Enomoto. Software development process from natural language specification. In *11th International Conference on Software Engineering*, pages 64–73. IEEE Computer Society press, May 15–18 1989.

[101] E.H. Schein. *Process Consultation: Its Role in Organization Development.* Addison-Wesley, 1969.

[102] A. Schuetz. Common-sense and scientific interpretation of action. *Philosophy and Phenomenological Research*, 14:1–37, 1953.

[103] E. Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, 1989.

[104] J. Searle and D. Vanderveken. *Foundations of Ilocutionary Logic.* Cambridge University Press, 1985.

[105] J.R. Searle. *Speech Acts. An Essay in the Philosophy of Language.* Cambridge University Press, 1969.

[106] J.R. Searle. A taxonomy of illocutionary acts. In *Expression and Meaning*, pages 1–29. Cambridge University Press, 1979.

[107] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Prentice-Hall, 1988.

[108] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice-Hall, 1992.

[109] J. Shomenta, G. Kamp, B. Hanson, and B. Simpson. The application approach worksheet: an evaluative tool for matching new development methods with appropriate applications. *MIS Quarterly*, 7(4):1–10, 1983.

[110] H.A. Simon. A behavioral model of rational choice. *Quarterly Journal of Economics*, 69:99–118, 1955.

[111] H.A. Simon. *The New Science of Management Decision.* Harper and Row, 1960.

[112] H.A. Simon. *The Sciences of the Artificial.* MIT Press, 1969.

[113] J. M. Smith and D.C.P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2:105–133, 1977.

[114] K. Southwell. Managing software engineering teams. In J.A. McDermid, editor, *Software Engineer's Reference Book*, page Chapter 32. Butterworth/Heinemann, 1992.

[115] P.A. Spruit and R.J. Wieringa. Some finite-graph models for process algebra. In J.C.M. Baeten and J.F. Groote, editors, *2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 495–509, 1991.

[116] R.W. Starreveld, H.B. de Mare, and E.J. Joëls. *Bestuurlijke Informatieverzorging, Deel II.* Samsom, 1985.

[117] P.J. van Strien. *Praktijk als Wetenschap. Methodologie van het Sociaal-Wetenschappelijk Handelen.* Van Gorcum, 1986.

[118] A. Sutcliffe. *Jackson System Development.* Prentice-Hall, 1988.

[119] R.H. Thayer, editor. *Software Engineering Project Management.* IEEE Computer Science Press, 1988.

[120] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1989.

[121] G.M.A. Verheijen and J. Bekkum. NIAM: An information analysis method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*, pages 537–589. North-Holland, 1982.

[122] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Volume 1: Introduction and Tools.

[123] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Volume 2: Essential Modeling Techniques.

[124] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Prentice-Hall/Yourdon Press, 1985. Volume 3: Implementation Modeling Techniques.

[125] V. Weinberg. *Structured Analysis.* Yourdon Press, 1978.

[126] R.J. Welke, K. Kumar, and H. van Dissel. Methodology Engineering: Een voorstel om te komen tot een situationeel specifieke methode-ontwikkeling. *Informatie*, 33:322–328, May 1991. In Dutch.

[127] R.J. Wieringa. *Machine Intelligence and Explication.* Eburon, 1987. Filosofische Reeks, Universiteit van Amsterdam.

[128] R.J. Wieringa. *Machine Intelligence and Explication.* Eburon, 1987.

[129] R.J. Wieringa. Three roles of conceptual models in information system design and use. In E.D. Falkenberg P. Lindgreen, editor, *Information System Concepts: An In-Depth Analysis*, pages 31–51. North-Holland, 1989.

[130] R.J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.

[131] R.J. Wieringa. A conceptual model specification language (CMSL Version 2). Technical Report IR-248, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1991.

[132] R.J. Wieringa. A formalization of objects using equational dynamic logic. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *2nd International Conference on Deductive and Object-Oriented Databases*, pages 431–452. Springer, 1991. Lecture Notes in Computer Science 566.

[133] R.J. Wieringa. *Information System Development and Conceptual Modeling: A Comparative Study.* Department of Mathematics and Computer Science, Vrije Universiteit, 1993.

[134] R.J. Wieringa and R.B. Feenstra. The university library document circulation system specified in LCM 3.0. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[135] R.J. Wieringa and W. de Jonge. The identification of objects and roles. Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1991.

[136] R.J. Wieringa and W. de Jonge. Roles and dynamic subclasses: a logic and methodology. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, In preparation.

[137] R.J. Wieringa, R. Jungclaus, P. Hartel, and G. Saake. Omtroll — object modeling in troll. Proceedings of the International Worskshop on Information Systems — Correctness and Reusability (IS-CORE'93), Udo W. Lipeck and G. Koschorrek (eds), pages 267–283. Institut für Informatik, Universität Hannover, Postfach 6009, D-30060, Hannover., September 1993.

[138] R.J. Wieringa and J.-J.Ch. Meyer. Actors, actions, and initiative in normative system specification. *Annals of Mathematics and Artificial Intelligence*, 7:289–346, 1993.

[139] C. Wiseman. *Strategy and Computers. Information Systems as Competitive Weapon.* Dow Jones-Irvin, 1985.

[140] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, 1989.

# Appendix A

# Summary of the Method

MCM presupposes a certain context, which is that it is part of the following information system development method:

1. **Requirements determination**

   1.1 *Identify the reasons for development*

   - *Problem-driven development*: Identify problems and problem owners. Example methods to use: ISAC change analysis, with activity diagrams can replaced by diagram techniques that are more familiar to analysts, such as data flow diagrams [66, 67, 68, 65]; the problem analysis part of ETHICS, again with an opportunistic choice of diagram techniques [83]; Analysis of Strengths, Weaknesses, Opportunities and Threats. For ill-defined problems, Pava's sociotechnical analysis [87] or Checkland's Soft Systems Methodology [16, 15] may be helpful.
   - *Norm-driven development*: Identify business goals and objectives. Example methods to use: Critical Success Factor Analysis [96], Value Chain Analysis [90, 91]; Strategic option generation [139]; Value analysis [23]; and other general methods to develop business strategies. These methods are independent from information system development and logically precede it.

   1.2 *Analyze* problems and/or goals and objectives.

   - *Determine development goals and objectives*: change needs, information needs and constraints from the environment of the system to be developed. Example methods to use: the above business analysis methods can be used with a focus on information problems and information strategies. In addition, the following methods can be used: strategy set transformation [60]; customer resource life cycle analysis [48]; the Information Strategy Planning stage of Information Engineering [73, 75]; sociotechnical systems analysis [87].

   1.3 *Generate alternative solutions*: observable behavior of developed system, performance level, required resources. Example methods to use: Brainstorming techniques, representation and specification techniques from the analysis methods listed above, or from the conceptual modeling methods listed below.

   1.4 *Evaluate alternative solutions* on the probable effectiveness and efficiency in realizing development objectives. Example methods to use: Economic cost/benefit analysis, sociotechnical analysis [83, 87], risk analysis.

   1.5 *Choose solution* (done by sponsor).

2. **Conceptual modeling**

   2.1 Gather **observational** data of the required situation. Example observation methods: Direct observation, participant observation, interviewing, questionnares, gathering company documents [55, 33]. See also section 3.2 for a summary.

2.2 Identify **system boundaries** by making a *function decomposition tree* whose leaves represent *registration transactions*, *directive* and *declarative control transactions*, *temporal events* and *queries*. Example methods to use: Making a function decomposition tree [7, 73, 75, 76, 77], making a context diagram [79, 123]. See also chapter 5 and section 8.2.

2.3 Specify user procedures. Example methods to use: Sociotechnical system design [83, 87].

2.4 Specify observable behavior of each required DBS. Example methods to use: ER modeling [17, 8]; NIAM [86, 121]; Structured Analysis [36, 61, 27, 79, 125, 140]; SADT [72, 98]; the Information Analysis task of ISAC [66, 67, 68, 65]; the UoD modeling and system network specification tasks of Jackson System Development [12, 49, 118]; OMT [99]; The Ward/Mellor [122, 123, 124] or Hatley/Pirbhai [46] methods of Structured Analysis for real-time systems; The Shlaer/Mellor method for object-oriented analysis [107, 108]; The Martin/Odell method for obejct-oriented analysis [78]; MCM (chapters 6 to 9), etc. In addition, methods for user interface design must be used.

3. **Implementation**

3.1 Design and

3.2 build specified systems. Example methods to use: build yourself; let someone else build it; buy something already built. For each of these: program it using structured or object-oriented methods; generate it using fourth-generation software; adapt parametrized software; reuse existing software; transformational implementation from a formal specification; evolutionary implementation; incremental implementation.

3.3 Reorganize and convert to the new situation.

The above framework must be supplemented with methods for project management, that allow one to deal with finite resources and a potentially disturbing environment. Examples are methods for planning, budgeting, staffing, presentation, communication etc.

MCM is a method for finding and evaluating formal and informal specifications of object-oriented conceptual models of data processing systems (task 2.4 above). It consists of the following tasks.

1. **UoD modeling**. See chapter 6 for the structure of UoD models, chapter 7 for the specification of UoD models, chapter 8 for induction methods and chapter 9 for evaluation methods.

1.1 Do a **transaction analysis** of registrations and triggered updates to produce a *class model* and a *communication model*.

- Object analysis of transactions. Subsection 8.3.1.
- Elementary sentence analysis. Subsection 8.3.2.
- Transaction decomposition. Subsection 8.3.3.
- A class model of the UoD of an *existing* DBS can also be produced with form analysis, record analysis or normalization. See Rock-Evans [95, pages 37–41], and Batini, Ceri and Navathe [8, pages 92–100] for form analysis; Rock-Evans [95, pages 42–43] and Batini, Ceri and Navathe [8, pages 101–109] for record analysis; Rock-Evans [95, pages 60–72] or almost any textbook on database management systems, e.g. Date [24] for normalization.
- Essential systems analysis is a method to find a model of the transactions of an existing DBS, that may be adapted to an object-oriented modeling approach. McMenamin and Palmer give a detailed essential systems analysis method that derives data flow models [79]. There is currently no essential systems analysis method that leads to object-oriented models.

1.2 Do a **scenario analysis** to produce *life cycle models* for the object classes.

- Scenario analysis. Subsection 8.4.1.
- Event trace analysis. Subsection 8.4.2.

1.3 **Document** the object–, communication– and life cycle models in an *informal specification.* Section 7.1.

1.4 **Formally specify** the model. Sections 7.2 and 7.3.

- Produce *signatures* for the attributes and events.
- Specify event *effects.*
- Specify event *preconditions.*

1.5 Identify **business rules** and add the to the informal and formal specification.

2. **Query modeling**. Specify the *queries* in the decomposition tree and the *reports* produced by those queries. Not treated.

3. **Evaluation** of the model. See chapter 9.

- The *quality checks* that can be performed depend upon the kind of model structures. Section 5.2.3 gives quality checks for function decomposition trees, chapter 6 contains all quality checks for the modeling structures of MCM.
- The *validation* checks consist of some form of animation of the model. See section 9.1.
- The *utility checks* consist of some form of prototyping the required system functions. See section 9.2.

# Index


*is_a*, *see* Subclass

Aboutness, 22
ACP, *see* Algebra of Communicating Processes
Administrative transaction, 24
Aggregate, 48
Aggregate existence constraint, 78
Algebra of Communicating Processes, 64
Analytical truth, 77
Animation, 109
Atomicity
  and interface technology, 20
  of events, 19, 61, 63
  of transactions, 20, 105
Attribute, 22, 39
  codomain, 39
  domain, 39
  finding attributes
    by preconditions, 98
    by queries, 98
  value, 39

Behavior, 5, 67, 102
Bisimulation, 66
Boundary, 26
Business rule, 84

Cardinality constraint, *see* Constraint
Causation, 19
CIM, *see* Computer-integrated manufacturing
Class, 38
  definition, 80
  existence set, 38
  extension, 38
  instance, 38
  intension, 38
    in LCM, 86
  species, 63
Class diagram, 39
Class model, 94, 121
Classification principle, 56
Clock, 22
Codomain, 39
Communication, 71
Communication diagram, 72
Communication model, 94, 121
Component, 44
Component existence constraint, 44, 78
Computer-integrated manufacturing, 93
Conceptual model, 10
  file-oriented, 14
  object-orientation, 14
Conceptual modeling, 6, 11
Conceptual modeling method, 8
Constraint, 44
  aggregate existence constraint, 78
  cardinality constraint, 46
    as an existence constraint, 78
  component existence constraint, 44, 78
  DBS constraint, **75**
  dynamic constraint, 78
  existence constraint, 78
  static constraint, 78
  UoD constraint, **76**
Context diagram, 26
Contingency approach, 2
Control
  declarative, 23
  directive, 23
Creation event, 63
Critical success factor analysis, 8
Customer resource life cycle analysis, 8, 120

Data dictionary, 24
Data directory, 24
Data dredging, 16
Data type, 38
Database system, 11
  constraint, 75
DBS, *see* Database system
DBS model, 25
Declarative control, 33
Declarative control transaction, 23
Declarative specification, 25
Declared event, 23
Delegation, 54
Deletion event, 63
Deontic logic, 111
Dependent identity check, 49

Determinism, 61
Deterministic state, 65
Development, 5
  iterative, 9
  linear, 9
  prototyping method, 9
  spiral, 9
  system development, 5
Dialog, 67, 102
Directive control transaction, 23
Discreteness of events, 19, 63
Discreteness of time, 19
Domain, 39
Dynamic constraint, 78
Dynamic partition, 52
Dynamic subclass, *see* subclass, dynamic

Effect axiom (LCM), 85
Elementary sentence analysis, 16, 99
Elementary sentence check, 108
Embedded database system, 11
Empirical cycle, 11
Empirically true, 77
Encapsulation
  in object-orientation, 62
  in process algebra, 72
Entity, 21
Entity structure (JSD), 67
Entity-Relationship modeling, 8, 12, 16, 17, 79, 109, 121
ER, *see* Entity-relationship modeling
ER modeling, 1
Essential model, 10, 16, 30
Essential system analysis, 16
Essential system model, 20
ETHICS, 6
Evaluation method, 17
  quality check, 17
    classifications, 56
    creation/deletion, 75
    dependent identity check, 49
    event, 63
    event allocation, 75
    function decomposition tree, 33
    instantiation checks, 59
    is-a vs. role-playing, 58
    life cycle, 70
    object vs. relationship, 49
    object vs. value, 43
    static vs. dynamic classification, 57
  utility check, 17
    prototyping, 109
    query check, 109
  validation, 17
    animation, 109
    elementary sentence check, 108
    population check, 109
    walkthrough, 109
Event, 19, 22, 61
  allocation, 75
  atomicity, 19, 61, 63
  creation, 63
  deletion, 63
  deterministic, 61, 64
  discreteness, 19
  encapsulation
    in object-orientation, 62
    in process algebra, 72
  inheritance, 62
  locality, 61, 63
  nondeterministic, 61, 64, 65
  occurrence, 61
  spatial, 21
  temporal, 20, 23
  versus process, 61
  versus state, 61
Event trace diagram, 105
Evolutionary prototyping, 9
Existence, 37, 44
  in the DBS, 37
  in the UoD, 37
Existence constraint, 44, 78
Existence set, 38
Existence-dependence, 54
$ext(C)$, 38
$ext_\sigma(C)$, 38
Extension
  of a class, 38
  of a type, 38
External identifier, *see* Identifier, external

File-orientation, 14
Form analysis, 16
Formal specification, 85
Function, 5, 22, 33
  as mathematical function, 18
  as service, 18
Function decomposition tree, 27, 72
  making, 94
  used to find events, 64
Functional decomposition strategy, 30
Functional requirements specification, 9
Functional system, 18

Identifier
  external, 42
  internal, 42
Identity, 37, 41

rigidity, 37
uniqueness, 37
IE, *see* Information Engineering
Imperative specification, 25
Implementation, 5, 6
Implementation method, 8
Induction method, 14
elementary sentence analysis, 16, 99
essential system analysis, 16
event trace analysis, 105
form analysis, 16
narrative scenario analysis, 102
normalization, 16
object analysis of transactions, 96
record analysis, 16
scenario analysis, 17
transaction analysis, 17
transaction decomposition, 99
Inductive jump, 14, 105
Informal specification, 80
Information Engineering, 1, 6, 27, 71, 75
Information needs analysis, 8
Information system, 11
Inheritance, 52
of events, 62
Instance, 38
Instantiation check, 59
*int*(*C*), 38
Integrity constraint, 75
Intension, 38
in LCM, 86
of a type, 38
Interactive function (JSD), 21
Interface technology, 20
Internal identifier, *see* Identifier, internal
Inversion of transaction/use table, 99
ISAC, 6, 8
Iterative development, 9

Jackson System Development, 1, 8, 12, 17, 21, 25, 36, 61, 67, 70, 102
JSD, *see* Jackson System Development

Key, 43

Life cycle, 67, 102
Linear development, 9
Locality of events, *see* Event, locality
Locality requirement, 85

Management method, 8
Mandatory participation (ER), 79
Many-one relationship, *see* Relationship, many-one
meta-DBS, 24
Method engineering, 2, 8
Mission, 27
Monotonic designation, **41**

Namespace, **40**
Necessary truth, 76
analytic, 77
empirical, 77
Negotiation, 95
NIAM, 8, 12, 17, 109, 121
Non-functional requirements specification, 9
Nondeterminism, 61, 64, 65
of transactions, 102
Nondeterministic state, 66
Norm, **75**
Normalization, 16

Object, 36
as system, 37
existence, 37
in the DBS, 37
in the UoD, 37
identity, 37, 41
observability, 36
versus value, 38
Object analysis of transactions, 96
Object Modeling Technique, 8, 17, 102
Object-orientation, 14, 24
Objectory, 8, 102
Objectspace, **40**
Observability, 36
Observation method, 13
Occurrence, 61
oid, *see* Object identifier
OMT, *see* Object Modeling Technique
Optional participation (ER), 79
Overloading
of attribute names, 39
of event names, 62
of predicate names, 39
of transaction names, 72, 101

Partial participation (ER), 79
Partition
dynamic, 52
static, 49
Planned response system, 18
Player, 54
Precondition (LCM), 85
Predicate, 39
versus Boolean attribute, 40
Process, 61, 63
Process graph, 64
Process variable, 64
Prototype, 109

Prototyping, 9
evolutionary, 9
throw-away, 9

Quality check, *see* Evaluation method
Query, 24
Query check, 109
Query modeling, 94, 122

Rational choice process, 5
Rationality assumption, 6
Reactive system, 18, 22
Real time, 111
Record analysis, 16
Recursive process specification, 65
Recursive relationship, *see* Relationship, recursive
Registration transaction, 22
Regulatory cycle, 6
Relationship, 44
existence, 44
many-one, 47
recursive, 44
Relationship attribute, 44
Relationship class, 44
Relationship predicate, 44
Report, 24
Requirements determination, 6
Requirements determination method, 8
Requirements specification, 9, 93
functional, 9
non-functional, 9
statement of purpose, 9
Reverse engineering, 16
Rigid naming, **41**
Rigid reference, **41**
Role, 54
as state of object, 61
Role group, 54

SA, *see* Structured Analysis
SA/SD, *see* Structured Analysis/Structured Design
SADT, *see* Structured Analysis and Design Technique
Scenario, 102
Scenario analysis, 17, 102
Semantic modeling, 1
Semi-encapsulation, 72
Service, 30, 72
Singular naming, **41**
Singular reference, **41**
Soft Systems Methodology, 8
Spatial event, 21
Species, 63
Specification
declarative, 25
formal, 85
imperative, 25
informal, 80
Spiral development, 9
SSADM, 6
SSM, *see* Soft Systems Methodology
State, 61
nondeterministic, 66
of deterministic system, 65
Statement of purpose, 9
Static constraint, 78
Static subclass, *see* Subclass, static
Strategy set transformation, 8, 120
Structured Analysis, 1, 8, 12, 16, 17, 20, 27, 33, 109, 121
Structured Analysis and Design Technique, 121
Structured Analysis/Structured Design, 6
Subclass
dynamic, 52
as state of object, 61
static, 49
Surrogate, 21, 98
and internal identifiers, 42
creation, 98
existence in the DBS, 37
for a relationship, 44
for an object, 42
System, 5
behavior, 67, 102
boundary, 26
functional system, 18
planned response system, 18
reactive system, 18, 22
System development, 5

Temporal event, 20, 23
representation in a context diagram, 27
Thread of control, 67, 105
Throw-away prototyping, 9
Time, 22, 27
and clocks, 22
discreteness, 19
not an object, 24
Total participation (ER), 79
Trace, 65
Transaction, 19, 71
administrative transaction, 24
atomicity, 20, 105
declarative control transaction, 23
directive control transaction, 23
input transaction, 21
nondeterministic, 102

output transaction, 21
overloading, 72
query transaction, 24
registration transaction, 22
report transaction, 24
temporal transaction, 23
Transaction allocation criteria, 101
Transaction analysis, 17, 94, 121
object analysis, 96
Transaction decomposition
assumptions, 71, 99
conjunctive, 101
Transaction decomposition table, 71
Transaction/use table, 96
inversion, 99
Trigger, 21
Type, 38
extension, 38
intension, 38

Universe of discourse, 11, 21
UoD, *see* Universe of discourse
UoD constraint, **76**
UoD model, 24, 35
object-oriented, 24
UoD modeling, 94, 121
Use case (Objectory), 102
User, 5
User directory, 24
User interface, 11, 121
Utility check, *see* Evaluation method

Validation, 12, 17
Validation check, *see* Evaluation method
Value, 37
versus object, 38
Value type, 38

Walkthrough, 109