# LOGICS FOR OO INFORMATION SYSTEMS

## a semantic study of object orientation from a categorial-substructural perspective

ILLC Dissertation Series 2001-03

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-525 6051
fax: +31-20-525 5206
e-mail: illc@wins.uva.nl
homepage: http://www.illc.uva.nl/

# LOGICS FOR OO INFORMATION SYSTEMS

## a semantic study of object orientation from a categorial-substructural perspective

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.dr. J.J.M. Franse
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op woensdag 9 mei 2001, te 12.00 uur

door

Erik de Haas

geboren te Neuss, Duitsland.

| | |
|---|---|
| Promotores: | Prof.dr. P.W. Adriaans |
| | Dr. P. van Emde Boas |
| Overige commissieleden: | Prof.dr. J.F.A.K. van Benthem |
| | Prof.dr. P. Klint |
| | Prof.dr. G. Renardel de Lavalette |

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam
Kruislaan 404
1098 SM Amsterdam



Cover design and photography by Mariska van de Cappelle

ISBN: 90–5776–066–5

*Schaatsenrijder*

*Over zijn strenge cirkels heengebogen*
*eigent hij zich de middelpunten toe.*
*Hun trots bezit staat in zijn harde ogen.*
*Hij wordt de mathematica niet moe,*

*waarmee elk nieuw uitvieren zich voltrekt*
*om elke nieuwe inkeer op te vangen.*
*Zie hem in rustige beslissingen hangen*
*boven het tijdloze, dat hij wekt*

*en kantelend in tegenkringen leidt*
*voor het een snelle, ronde dood zou vinden.*
*Hij heeft zich van de wereld al bevrijd;*
*enkel de smalle ijzers die hem binden*

*aan 't evenbeeld. Een laatste trouw misschien?*
*Wat kan hij in de spiegel nog verwachten?*
*Of houdt een vrouwenschim, die wij niet zien,*
*hem vast binnen dit eenzaam veld van krachten?*

*IJskoude liefde, die niet sterven wil,*
*omdat de dode lelies onder water*
*haar eenmaal droegen in hun gouden harten,*
*waarmee de vijver vol lag, zwaar en stil.*

Gerrit Achterberg, SPHINX

# Contents

**Acknowledgments** **xi**

**Preface** **xiii**

**I General Analysis of Object Oriented Technology** **1**

**Introduction** **3**

**1 The object oriented development practice** **5**
1.1 Labels, partial descriptions and non-wellfoundedness . . . . . 7
1.2 Object Oriented Information Systems . . . . . 8
1.2.1 An ontology of object oriented information systems . . . . 8
1.2.2 The Unified Modeling Language (UML) . . . . . 11
1.2.3 Object oriented modeling languages, database languages, and programming languages . . . . . 15
1.3 Object Oriented Software Development . . . . . 17
1.3.1 A very brief history of the OO software development process 17
1.3.2 An overview of the OO Software Development Process . . 18
1.4 Summary . . . . . 20

**2 Concepts in object orientation** **21**
2.1 Concepts . . . . . 21
2.1.1 Object and object identifier . . . . . 22
2.1.2 Complex value, type and class . . . . . 27
2.1.3 ISA hierarchy, subtyping and inheritance . . . . . 31
2.1.4 Methods and operations . . . . . 36
2.1.5 Encapsulation . . . . . 38

2.1.6 Declarativeness . . . 38
2.1.7 Rules and knowledge . . . 40
2.1.8 Graphical representation . . . 40
2.1.9 Partial specifications, Identity, and the Extendibility principle 44
2.2 Summary . . . 45

**II A Model for Object Oriented Technology 47**

**Introduction 49**

**3 A generalized language for object oriented information systems 51**
3.1 A Case for Object Oriented Information Systems . . . 52
3.2 The Syntactic Theory . . . 54
3.2.1 Edge Graphs . . . 54
3.2.2 Operations . . . 64
3.2.3 Types, Objects and Constraints . . . 66
3.2.4 Categorial Graphs . . . 71
3.2.5 Imploding and Exploding of categorial graphs . . . 74
3.3 Summary . . . 77

**4 A semantics for object oriented information systems 79**
4.1 Desiderata for meta language for categorial graphs . . . 80
4.2 The meta language of categorial graphs . . . 85
4.3 Calculus for the meta language of categorial graphs . . . 87
4.4 The semantic domain for object oriented information systems . . . 91
4.5 Summary . . . 101

**III Logical Aspects 103**

**Introduction 105**

**5 Methodology: semantics, logic and applications 107**
5.1 Formal Semantics . . . 108
5.1.1 Semantics in computer science . . . 108
5.1.2 Semantics for categorial graphs . . . 109
5.2 The roots of the logic of categories . . . 111
5.2.1 The categorial graph logic as a modal logic . . . 111
5.2.2 The categorial graph logic as a substructural logic . . . 114
5.2.3 Related logics of information systems . . . 116
5.3 Applications . . . 117
5.3.1 Applications in object oriented information system practice 117
5.3.2 New computational applications . . . 117

5.3.3 Logical and philosophical repercussions . . . . . 118
5.4 Summary . . . . . 119

**6 Logic of object oriented information 121**
6.1 Models for object-oriented information systems . . . . . 122
6.1.1 Intended models . . . . . 122
6.1.2 Abstract models . . . . . 124
6.1.3 Representation . . . . . 125
6.2 Modal languages . . . . . 126
6.2.1 Definition language and semantics . . . . . 127
6.2.2 Adjacency logics . . . . . 129
6.2.3 Extendibility logics . . . . . 134
6.2.4 Aggregate logic . . . . . 136
6.2.5 The combined system . . . . . 142
6.2.6 Conclusion . . . . . 146
6.3 Other logical formalizations . . . . . 146
6.3.1 Translation . . . . . 146
6.3.2 First Order approach . . . . . 148
6.3.3 Resource approach . . . . . 148
6.4 Axioms and completeness . . . . . 150
6.4.1 The first-order case . . . . . 151
6.4.2 The modal case . . . . . 152
6.4.3 The resource case . . . . . 154
6.4.4 What does this mean for our object models? . . . . . 156
6.5 Complexity . . . . . 157
6.5.1 Benchmark tasks . . . . . 157
6.5.2 Model checking . . . . . 157
6.5.3 Satisfiability . . . . . 159
6.5.4 Inference . . . . . 165
6.6 Extensions . . . . . 165
6.7 Further logical considerations . . . . . 166
6.7.1 'Object'/'type' duality . . . . . 166
6.7.2 Treating 'facts' as first-class citizens . . . . . 167
6.8 Summary . . . . . 171

**IV Philosophical Backgrounds 173**

**Introduction 175**

**7 Four philosophical issues 177**
7.1 Examples from 2500 years of modeling information systems . . . . 178
7.2 The philosophical issues in terms of categorial graphs . . . . . 183

7.3 Summary . . . . . 187

**V Conclusion** **189**

**8 Categories for Profit** **191**
8.1 The object oriented development practice . . . . . 193
8.2 Concepts of object orientation . . . . . 193
8.3 A generalized language for object oriented information systems . . 194
8.4 A semantics for object oriented information systems . . . . . 194
8.5 Methodology: semantics, logic and applications . . . . . 195
8.6 Logic of object oriented information . . . . . 195
8.7 Four philosophical issues . . . . . 195

**Bibliography** **197**

**Samenvatting** **207**

# Acknowledgments

First I would like to thank my promotores Pieter Adriaans and Peter van Emde Boas. I thank Peter for giving me the opportunity to do research in his group, and for directing me to an interesting field of science. The extensive package of themes he called my attention to, and his knowledge of the relevant literature have been very valuable. A special thanks I owe Pieter, who provided the insight to build solid ground under my raw intuitions and theories. I thank Pieter also for his creative ideas, that, next to contributing to this thesis, also let me rediscover the relevance and pleasure of scientific research.

Another special thanks I owe Johan van Benthem for giving a lot of support, and for his generous time and effort that helped me to complete the logical analysis.

Then I would like to thank all the other persons that provided me with numerous valuable suggestions. In this respect I would like to mention Paul Klint, Gerard Renardel de Lavalette and Anne Troelstra, who have given nice and constructive comments on the almost-final version of my thesis. Moreover I would like to thank Mark Ballantyne for taking the effort to thoroughly proofread my thesis and providing me with numerous suggestions to improve my English language. I also want to thank Mariska van de Cappelle for helping me with the graphics in this thesis, and for designing the wonderful cover.

While writing this thesis I have been working both in a scientific and in a business environment. In both environments people have given me a lot of support and provided valuable input through many discussions about the topic of this thesis. In particular I would like to thank my close colleagues at the university: Karen Kwast, Ernest Rotterdam, Sophie Fisher, Marten Trautwein, Erik Aarts, Wilco Quak, and Carlo de Boer. Also I want to thank my colleagues from the Syllogic R&D department, who provided an environment with a scientific quality that is seldom seen in business: Erik Darwinkel, Marc-Paul van der Hulst, Daniel van der Wallen, Otto Moerbeek, Chris Thieme, Michiel Greuter, Eelco den Heijer, Arno Knobbe, Arjo Duineveld, Derk Bijmolt, Sander van Geloven,

Bert Laverman, Ramesh Srinivasan, (again) Mart Trautwein, and (again) Ernest Rotterdam.

Finally, I would like to thank some people from my personal world. I owe very special thanks to Ingrid, who caringly supported me for many years. I also thank Mariska for motivating me to launch a final sprint for finishing my thesis, and for standing by me even when the sprint turned out to be a somewhat longer than expected. I also want to thank my parents, Gedi and Ko, and my brother Marc, because they always care warmly.

Amsterdam Erik de Haas
March, 2001.

# Preface

*There is no formal transformation from the informal to the formal*

*(Anonymous)*

Recent years have seen the convergence of many disciplines in information systems facilitated by the concepts of *Object Orientation*. Not least has been the convergence of the languages for Object Oriented analysis and design, manifested in the definition of the industrial standard UML (Unified Modeling Language [UML97], [UML99]) for such languages. Moreover the integration of information design languages into integral software development tools, enabling automatic database (persistent) model generation and code generation, indicate that these kinds of languages and concepts have grown to a mature state.

The theme of this thesis is a *semantical investigation in Object Oriented (OO) modeling and database languages*. The semantical investigation strives to give a thorough mathematical description of the concepts used in OO design and database languages. Such a mathematical description gives an insight into the constructs used, and can be used to develop and refine automatic development tools and query optimalization techniques for computing with OO information objects. Fact is, most object oriented design languages, and especially UML, have no clear mathematical foundation. Nevertheless a lot of 'formal' tasks like code generation and 'database modeling' are performed in these languages. The resulting systems therefore are suspect of ambiguities and inconsistencies, and hence sometimes valid UML expressions cannot be processed. Research in the mathematical foundations of OO concepts aims to aid the development of OO language processing, by taking away the non-clarities and providing a formal and consistent way of interpreting the language.

The semantical investigation in Object Oriented design languages is especially interesting because the concepts of object orientation originate from practice and were designed to help information analysts and designers to accurately describe information models that reflect aspects of the real world. In this respect this research touches on themes from philosophy, where it is an important goal to accurately describe aspects of the real world.

In this thesis we will study the semantics of object oriented design and database languages in detail. The thesis will provide a thorough description of the concepts that can be expressed in UML and like languages We will cover all the main concepts of object orientation such as identity, inheritance, encapsulation etc.. Moreover, we will study languages for specifying information systems from a more general perspective and then identify the really basic concepts of talking about information objects. In this exercise we will encounter serious philosophical controversies that are inherent in talking about objects, but often ignored in the information system practice. It turns out that in the practice of information analysis, the information modeler runs into hard philosophical problems in his attempt to accurately describe the aspects of the real world he wants to capture.

The major artifact we will present in this thesis is a language for modeling information systems. This language contains all the main concepts of object orientation. It is a generalization of the object modeling part of UML (a fragment of the language constructs of UML, present in several diagramming techniques of UML). The basic building block of the language is a so called *category* and contains graphical and textual components. We will do the necessary mathematics for this language in order to obtain a formal semantics for the object oriented concepts. We will develop a formal syntactic theory for the language and provide a rigid mathematical model in which we will interpret the language. In this setting we can give a clear semantics for the basic language constructs of both object oriented modeling and design languages and object oriented database and programming languages. For the semantic study we will use the arsenal of modal and substructural logic and categorial grammars. This branch of mathematics is used heavily in the study of natural language and computation theories and the study on the OO concepts contributes a nice application of the theory with promising extensions for intelligent information systems and data mining. Moreover, we can identify the potential philosophical controversies associated with describing aspects of the real world in the information analysis practice. Such an identification will enable the information modeler to choose a consistent interpretation of the models he writes down.

Several parts of this thesis have already been communicated to the scientific community in various papers. A first version of the language of categorial graphs

appeared in [Haas95] and [Haas94]. Extensions on this research in relation to natural language learning and data mining were published in [HaasAdriaans99], [AdriaansHaas99] and [AdriaansHaas00]. Preliminary research on object orientation and information systems theory, which provided the inspiration to explore this interesting subject more thoroughly, appeared in [HaasEmdeBoas93], [PomykalaHaas93], [PomykalaHaas94], [PomykalaHaas96].

This thesis is structured as follows:

- Part1: General analysis of Object Oriented technology. Part 1 contains a general analysis of the concepts and intricacies of object orientation in information systems. It is the conceptualization of the domain of our semantical investigations.
  - In Chapter 1 we will describe the information system analysis and design practice. We will focus specifically on the object oriented analysis and design practice and the related object oriented database models. We will discuss the use of languages for analysis and design and databases, and give an overview of the languages used in practice (especially the industry standard UML). We will see that this practice imposes requirements on the language and its interpretation in the research context.
  - In the second chapter, we present in detail the family of notions and concepts for which we will do the semantic research. Note that much debate is possible on the exact interpretation of information system notions that originate from actual use. We will discuss the notions for object oriented (new generation) information systems in a critical way, and provide a motivation for the interpretation we will use.
- Part 2: OO Modeling Proposal: Categorial Graphs. In this part we propose a model in which we can research the object oriented analysis and design practice.
  - Chapter 3 will introduce a language for talking about information systems. This is the syntactic domain in which we can denote (graphically and textually) the concepts discussed in part 1. This language is a generalization of the common OO information system design languages. We will especially show its expressiveness by comparing it to UML. In effect, the language presented will be a formal syntactic theory for a generalized fragment of UML. The language is built from a syntactic construct we call a *categorial graph* (borrowing the term 'category' from Aristotle); and the language therefore is called the *language of categorial graphs*.

  - Chapter 4 contains the semantics of the categorial graph language. We will present an interpretation of the language that talks about object oriented information systems. This interpretation will be a logic based on the theory of modal and substructural logics.

- Part 3: Logical aspects. The chapters in this part present logical aspects of the theory of object oriented information systems.

  - In chapter 5 we will explain the benefits of formal semantics and describe the approach and attitude to tackle the semantics for information systems taken in this thesis. We will explain the logical aspects of doing semantics, and also position this research in the research field of logic, as it touches some very interesting problems in current logic research.
  - In chapter 6 we will investigate the logic of categorial graphs. We will discuss logical aspects, especially soundness, completeness and the computational complexity of the logics for categorial graphs.

- Part 4: Philosophical backgrounds. In this part we discuss philosophical issues involved in information system modeling and object oriented concepts.

  - In Chapter 7 we take a little step back, and will formulate only the basic concepts we like to have in our language that talks about information systems. We will right away discover that this basic list of desiderata already confronts us with hard problems that are (still) very actual in philosophy.

- Part 4: Conclusion This part contains a wrap up of the themes we discussed in this thesis.

  - In chapter 8 we summarize what we have done and evaluate what we have achieved.

# Part I

# General Analysis of Object Oriented Technology

# Introduction

The first part of this thesis contains a general analysis of the concepts and intricacies of object orientation in information systems. It contains a description of the practical use of object technology and an analysis of the concepts of object orientation. The description of the practical context serves two purposes:

- To establish the context for which we will do semantical research. It will explain for which languages and technologies this thesis aims to contribute scientific analysis. These are in short the field of object oriented development, and in particular UML (unified modeling language).
- To show that the use of the object oriented design and coding languages imposes requirements on these languages and its interpretation. To be more specific, the incremental nature of the development practice demands that the languages cope with labels, partially or even non-wellfounded defined objects, and partial descriptions of objects.

Moreover we discuss the concepts that are most emblematic for object technology. More specifically we analyze:

- object identity and object identifier
- complex structure, abstract type and class
- ISA hierarchy, subtyping and inheritance
- operations and methods
- encapsulation
- declarativeness
- rules and knowledge
- graphical representation

# Chapter 1

# The object oriented development practice

> When it comes down to it, the real point of software development is cutting code. Diagrams are, after all, just pretty pictures. No user is going to thank you for pretty pictures; what a user wants is software that executes.
>
> Martin Fowler, UML Distilled; A brief guide to the Standard Object Modeling Language ([FowlerScott00])

*"Here at our company we are doing business in Wuzzels. Wuzzels have Wazzels and this distinguishes them from the Buzzels."*. When, in practice, an information modeler comes into a company to commence his task, he will need to capture information on objects he possibly does not know anything about. Nothing about the structure, the behavior, nor the interrelations between the objects. He will more or less start with a growing collection of labels, that gradually gets more structure and meaning. Moreover he will discover kinds of objects, relations between certain kinds of objects and constraints on the structure and relations of objects. Nevertheless the information modeler will need to immediately start writing down preliminary versions of the model of the world he is trying to capture. The preliminary model he writes down will be used to communicate with the experts and users that play an important role in the piece of the world he is modeling.

Languages that bear concepts from the object oriented methodology are used in the information capturing process and in analysis and design. Such practice imposes some requirements on the language in which information system models are written down. For example the language must be able to elegantly denote objects that have a partial nature or that have unknown structure and behavior,

and still we need to be able to interpret the language such that the expressions really denote some part of the information model. Moreover because of the additive way of working the language must be such that we can easily extend existing descriptions of the world, and that the interpretation of such an extended model is an 'elaboration' of the former model[1].

Object oriented languages have already been used much longer in programming and database practice. The most referred reason to use these kind of languages in the semi-formal world of computer coding was that the object oriented languages contain concepts that enable one to talk 'naturally' about the information that needs to be coded. This means that the idea is that the notions used in object orientation are founded on a natural intuition to talk about information. This also explains the popularity of object oriented languages in analysis and design.

In this chapter we will introduce the reader into the problem domain covered and researched in this thesis. We will discuss three important concepts that, in the general process of information capturing with object oriented languages, are important items in our analysis of the object oriented development practice. These are *labels, partial descriptions of objects* and *partially or even non-wellfounded defined objects*.

Moreover we will give a brief overview of the origins of the concepts of object orientation in object oriented information systems. We will present in more detail the Unified Modeling Language (UML), which is a standard in the object oriented design and analysis practice. This language is the most influential reference for the concepts of object oriented information systems. Consequently we will explain the connection between a design language, like UML, and the conceptual 'lower level' coding languages for databases and programming constructs.

In this chapter we will also briefly describe the information development practice. This practice covers the whole trajectory from analysis and design to implementation in programming and database coding languages. We do this for two completely different reasons. First of all we want to identify a number of requirements of the object oriented languages that are related to the *use* of these languages (especially for analysis and design languages). Secondly we want to make the reader acquainted with the field of application for which this research is done: *Object Oriented modeling, design and development*.

The main focus of this chapter will be on conceptual aspects of object orientation. These notions are most apparent in the analysis and design practice, but

[1]More precisely, it should not be the case that expressions that hold in the former model are not satisfied by the elaborated model, unless this was explicitly stated in the additive modeling step.

are also relevant to the coding practice, because the conceptual strength of the OO languages motivates its use in coding practice.

## 1.1 Labels, partial descriptions and non-wellfoundedness

Notions that are key in this thesis are labels, partial descriptions and non-wellfoundedness. In this section we will explain these notions in more detail. Let us take a look again at the example from the beginning of this chapter:

> "Here at our company we are doing business in Wuzzels. Wuzzels have Wazzels and this distinguishes them from the Buzzels."

**Label.** An information analyst who has never been introduced to the subject of wuzzels, wazzels and buzzels will need to start his model, based on the information from the above sentence, with a collection of labels. These labels here clearly denote types of objects. But even though the information modeler does not have a clue of what its interpretation should be, he can start to interpret that a particular wuzzel is some kind of object. This object does not have anything but a label.

**Partial descriptions.** An important piece of information in the above sentence is that 'the fact of holding a wazzel' is a discerning fact. This means the following: To hold a wazzel can be a property of an object. And moreover, holding a wazzel is a characterization of certain types of objects, among which are wazzels. Also to hold a certain wazzel is a partial description of a wuzzel. To make the example more concrete consider the following: To have a 'beard' is a property of 'ancient philosophers'. Having a 'beard' is characteristic for 'ancient philosophers'. And 'he having that white beard' is a partial description of Socrates.

**Non-wellfoundedness.** The information modeler is quite certain that he will learn a lot more about the wuzzels, wazzels and buzzels. He will get to know intrinsic properties and accidental properties of certain wuzzels, wazzels and buzzels. But he will never know that he grasped all that can be said about the wuzzels; even more drastically at certain stages in the iterative process he will be certain that he did not grasp everything he needs of the wuzzels, wazzels, and buzzels. This is part of the way he works. Every object can be extended by discovering more and more properties (descriptions), and the properties in turn (seen as objects themselves) can again be extended. The process of extending can possibly never end (either by cycles or by infinite chains). But that means that these objects will be not well founded. Moreover one never knows whether the objects are totally described in terms of it properties.

The above notions are only briefly introduced in this section. They will be elaborated in the semantic study later on in this thesis, and there the importance will become evident.

## 1.2 Object Oriented Information Systems

The numerous developments in information systems from the last two decades, both in practice and in theory, have contributed to the obtaining of proposals for so called *Next Generation Information Systems* ([Comm.ACM91n10]). One of the most influential developments is object orientation. The concepts of object orientation are the subject of this chapter. Concepts from other conceptual worlds, however, have also contributed significantly to the arsenal of technologies used in current information systems. We will also briefly give attention to the *relational paradigm* and the *logical paradigm*. The relational paradigm is well known from the Entity Relationship design languages (ER) and the language for relational databases (SQL). The logical paradigm is used in knowledge based rule systems, deductive databases and logic programming.

### 1.2.1 An ontology of object oriented information systems

Among the recent developments in information systems, the most influential developments are probably those following the principles of the *object-oriented (OO) programming paradigm*. The OO programming paradigm has its origin in the SIMULA programming language ([Pooley87]), which was proposed in the late 1960s. The concepts underlying this paradigm became especially popular in the 1980s with the introduction of the programming languages SMALLTALK ([Smith95]), EIFFEL ([RistTerwilliger95]), and later C++ ([Stroustrup91]) and JAVA ([ArnoldEtAlii00]).

In the 1980s the paradigm of object orientation entered into the world of databases. Together with other developments in databases, in particular complex values and notions from semantic databases , this constituted the *object oriented database (OODB) paradigm* ([AtkinsonEtAlii89])

The concepts of the OODB paradigm entered the world of databases in several different disguises. Firstly, in OO programming there was a need for a *persistent* store for the objects that were created by the executions of the (OO) programs. In this context persistence means that the lifetime of the objects that occur in a program is longer than the lifetime of the program run that created the objects; this is in order to give other programs the opportunity to use these objects. In this disguise, some primitive database notions were incorporated into the world of OO programming languages. Another disguise of the concepts of the OODB paradigm was created the other way around. The new concepts of the OODB

paradigm were incorporated into existing database models. This gave rise to models such as the 'object relational database model' and the 'deductive object oriented database model' ([Abiteboul90]).

It has to be noted that the OODB paradigm rose from implementation efforts, and is not based on a precise formal model. Since its appearance, several models of various degrees of formality have been developed. None of these models developed so far encompasses *all* the features that are associated with the OODB paradigm, and also, a universally agreed upon model has not yet emerged. There have been proposals for a standard model of OODB systems, of which the ODMG-93 proposal ([Cattell94]) and its successors ODMG 2.0 ([Cattell97]) and ODMG 3.0 ([CattellEtAlii00]), from the ODMG group is the proposed standard. This is the case, because a large number of influential providers of OODB systems committed their efforts to the proposal. Unfortunately, the proposal suffers from many conflicting compromises, and is very sloppy. It also completely lacks formality and rigor in the semantics[2], and is much criticized. Another very influential proposal for standardizing the OODB concept is the SQL-3 standard ([SQL3]). Although this standard is historically based on the relational database model, it has incorporated many of the popular concepts of object oriented information systems, and in particular the OODB concepts. Much work is done comparing and synthesizing the two above mentioned proposals. The efforts to clarify the different views on the concepts of the OODB paradigm have initiated three manifests ([AtkinsonEtAlii89], [StonebrakerEtAlii90] and [DarwenDate95]). These manifests list in an informal manner the required features of OODB systems.

Whereas the first manifest concentrated solely on the concepts of the OODB paradigm, the second and third manifest stressed the importance of incorporating the fruitful notions of more traditional database systems, especially of the relational database systems. Contrary to OODB systems, relational systems ([Codd70], [Ullman88]) evolved from a precise formal model, equipped with a high level declarative language. The advantages of the theoretical clearness, the ad hoc query mechanism and the declarativeness of the language, inspired the implementation efforts of the relational database model for use in practice. The relational model, enriched with a lot of features and provided with a standard[3] query and data definition language (SQL), is considered one of the most successful theoretically impaired languages in information systems.

Another mathematical model that found its way into information systems is that of logic programming. The resulting paradigm of deductive databases is commended for is declarativity in combination with its computational power. Furthermore it enables one to incorporate rules of knowledge into the database, i.e. it

[2]syntax of the languages in the ODMG proposal are formally defined in BNF, although the BNF syntax conflicts sometimes with the informal presentation of the syntax

[3]Although not everybody is happy with SQL, it *is* the unchallenged standard.

gives the possibility to bring in concepts of the world of *knowledge bases*. Combining notions from deductive databases and OO databases has become an important matter of research and controversy. The main problem here concerns the question whether the notions of the OODB paradigm and the deductive database paradigm are compatible; especially the matter of combining declarativeness and the notion of an 'object' from the OODB paradigm ([Ullman91]). Although influential database experts refuted the combination of the deductive database model (declarativeness) with the OODB model, many others proved this combination possible, with respect to their own interpretation of the concepts in question Especially worth mentioning in the context of combining database paradigms is the language OORL [Rotterdam96] which combines in a declarative setting the relational model, notions from OODB systems and logical rules similar to the rules in deductive databases.

In the same era as OODB, from a different but maybe even more influential field of practice and research, notions relating to object technology originating from *analysis and design* made their advent in the conceptual world of information systems. The languages of analysis and design are tailored to describe at a high level of abstraction (conceptual level) the information structure of a system and its surroundings. For the languages supporting the object oriented notions in analysis and design in 1997 a standard language called UML has been accepted in industry. Although not formal, the models and languages of analysis and design contained appealing and well developed notions for information systems. Moreover the notions from analysis and design fit in and enrich nicely the concepts of the OO paradigms.

A very appealing feature of the languages of analysis and design is the ability to use sophisticated graphical schema techniques (UML, BOOCH, FUSION, OOSE, RDD, SYNTROPY, OMT, NIAM, EER etc.). Many of these schema techniques were gratefully adopted by implementation languages which included some practical database languages[4] and (with some limitations) and programming environments[5]. Graphical syntax has also been introduced in scientific, and theoretically wellfounded database languages[6]. It is probably is impossible to imagine next generation information systems without means of graphical representation.

[4]e.g. 'Gemstone', and 'O2' (pronounced 'O-deux').

[5]strictly the graphics are not part of the programming languages, but are abbreviations of programming language expressions in the integrated development environments (IDE's). Examples of such IDE's are Visual Age (IBM), Visual Studio (Microsoft), Forte (SUN).

[6]e.g. IFO ([AbiteboulHull87]), GOOD and HQL ([AndriesEngels94]).

The most important ingredients of the Object Oriented paradigm include[7] the following concepts:

- object and object identifier,
- complex value, type and class,
- ISA hierarchy, subtyping and inheritance,
- operations and methods
- encapsulation.

Related concepts that are usually not mentioned under the object technology label are:

- Declarativeness
- Graphical syntax

Concepts that are important for the use of the languages with object orientation are:

- partial specifications, identity and the extendibility principle

### 1.2.2 The Unified Modeling Language (UML)

The need for a uniform and consistent visual language in which to express the results of rather numerous object oriented (design and analysis) methodologies extent in the early 1990s became very evident. During that period the authors of three influential object oriented methodologies began an effort to unify their methods, when they were 'recruited' around 1995 by the Rational Software Company, a company that had developed a number of software development tools and practices. These authors were Grady Booch (author of the Booch method [Booch94]), Ivar Jacobson (initiator of the use case driven approach [JacobsonEtAlii92]), and James Rumbaugh (principle developer of the Object Modeling Technique OMT [RumbaughEtAlii91]). They released a first version (version 0.9) of the Unified Modeling Language UML in 1996. The effort was expanded to include other methodologists and a variety of companies including IBM, HP, and Microsoft, each of which contributed to the evolving standard. The standardization process resulted in the release of UML version 1.1 under the authority of the Object Management Group (OMG) standard organization in November 1997. UML has

[7] We note here that many concepts of the object oriented database paradigm already existed in earlier paradigms. We do not imply that the notions mentioned here originated in this OODB way of looking at databases, but merely that they are present in and characteristic for the OODB paradigm

now grown into the de facto visual language for writing down information system models for most (if not all) methodologies and tools that use the concepts of object technology (object orientation). Currently UML has evolved to version 1.3 (June 1999) with only minor revisions.

The Unified Modeling Language (UML) is a standard modeling language for software. It is a language for visualizing, specifying, constructing and documenting the artifacts of a software intensive system. Basically UML enables a information system modeler to visualize its work in standardized diagrams. For example the characteristic icon to write down an object is a layered rectangle with an underlined name in the upper layer. Such an icon is just a graphical notation. It is syntax. The icons of UML also have an intended meaning, a semantics. Below we will list an overview of the syntax of UML and a brief description of the informally defined meaning of graphical UML terms. For a thorough treatment of the UML language and its semantics we refer to [FowlerScott00] and [WarmerKleppe99]. There also exists the public documentation set of UML that was delivered when UML was released as a standard by the Object Management Group [UML99], but it is suited as a reference only. Note that the semantics of UML as defined in the books and the standard consist of brief English (natural language) sentences. It is not a formal semantics. In the evolving process of this standard the definition has become more consistent, but still no formal semantics is planned for future releases (UML 2.0 plans to contain a number of major enhancements, again driven by practical use). The formal semantics of the core concepts of UML is the subject of this thesis.

UML provides developers with a vocabulary that includes three meta categories: *things*, *relationships*, and *diagrams*. There are four kinds of *things*:

- *structural things*: these are building blocks that can specify structure of the world. These are: *class*, *active class*, *use case*, *interface*, *component*, *collaboration* and *node*, *object*[8], *attribute*[9] *in vocabulary* and *operation*[10].
- *Behavioral things*: these are building blocks that specify behavior of the world. These are: *Interaction* and *state machine*.
- Grouping things: These are containers that organize the world. These are: *package*, *model*, *subsystem*, *framework*.
- *Annotational thing*: This is a construct for adding arbitrary information in natural language. This is: *note*.

[8]not classified as such by the three UML founders [JacobsonEtAlii99]. They somehow only mention 'object' under the diagram meta category

[9]also not listed in [JacobsonEtAlii99]

[10]also not listed in [JacobsonEtAlii99] in vocabulary

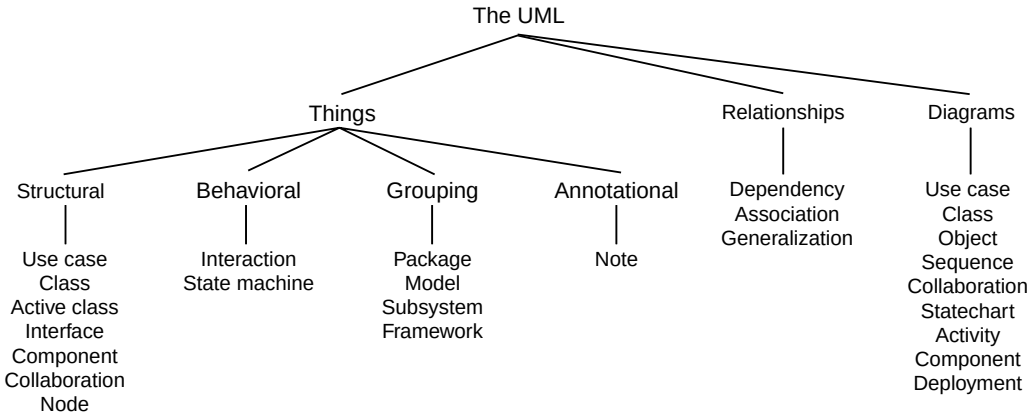



Figure 1.1: The vocabulary of UML in a tree form.

Within the second meta category, *relationships* we find three building blocks:

- a *dependency* denotes a dependency between things
- a *association* denotes an association (or relation) between things
- a *generalization* denotes an inheritance or isa relation between two things

In the last meta category *diagrams* we find 9 types of graphical containers:

- *use case*
- *class*
- *object*
- *sequence*
- *collaboration*
- *state chart*
- *activity*
- *component*
- *deployment*

The remaining figures in this section show an overview of the graphical notation of UML

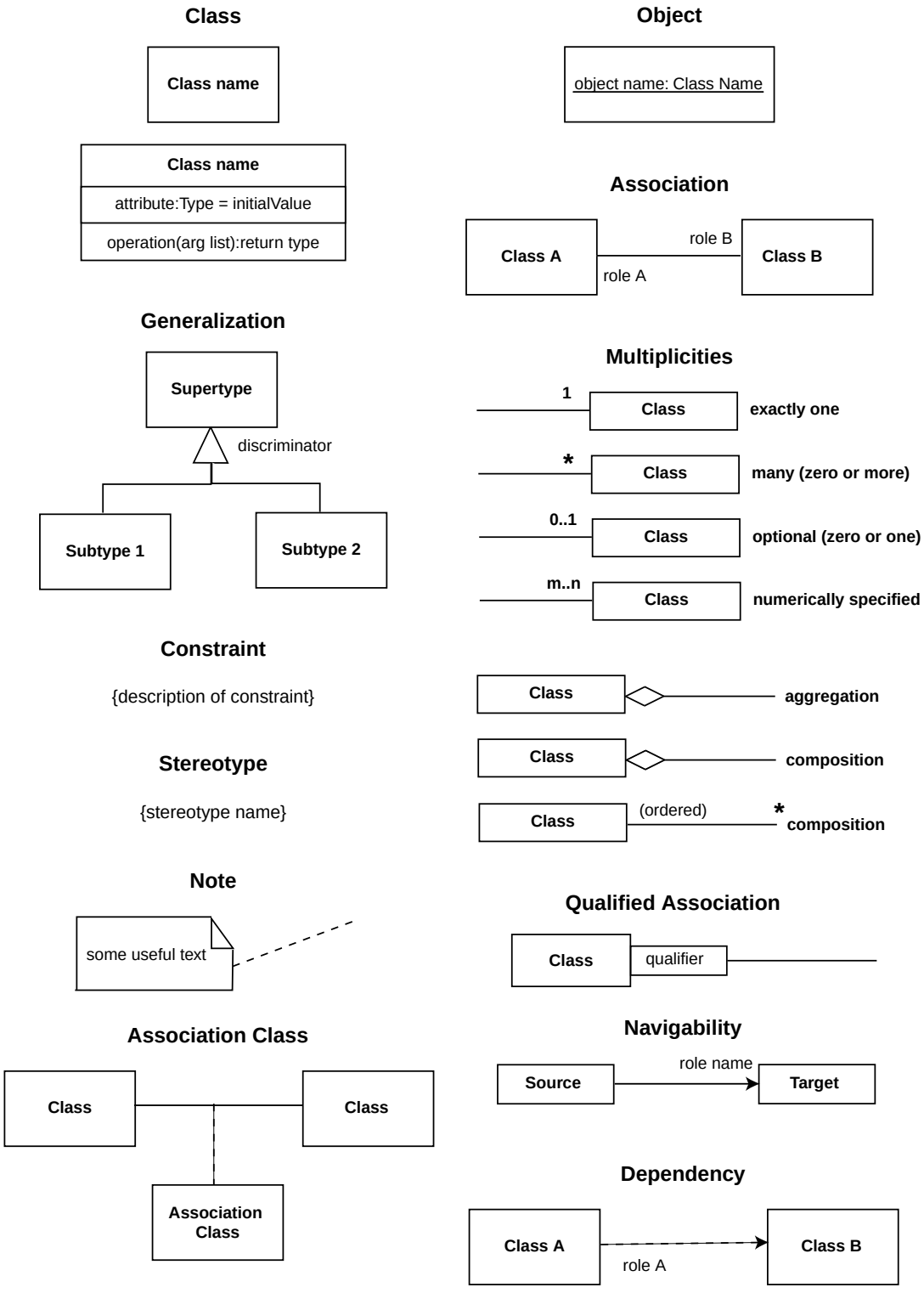



Figure 1.2: The relevant graphical constructs of UML

### 1.2.3 Object oriented modeling languages, database languages, and programming languages

The Unified Modeling language (UML) has originally been defined for object oriented modeling and design, and as such is the official standard language in that field. A development process is more than analysis and design. In the end we will also need programs and databases. Programs and databases still require other languages. The important glue between these languages, the glue that makes a straightforward transformation from specification from analysis and design to implementation possible, are *common concepts of object orientation*. This makes it possible to transform a design model in UML to database and programming languages, writing down all the information in the detail level as specified in the design model. In the implementation activity then implementation specific coding needs to be added (more detail) in order to obtain an operational system. In current development tools the uniformity of concepts is often good enough to do some part of the transformation automatically. Moreover automatic transformation for implementation back to design is also possible to some extent in order to make the design model consistent with the implementation model in the event that in an implementation language something has been changed that is also prevalent on the design level. *And this all without a rigorous semantics.*

We will spend some time on two kinds of implementation languages: object oriented *programming* languages and object oriented *database* languages. In practice the most commonly used object oriented programming languages are Smalltalk, C++ and Java. Especially Java, chronologically the latest developed from the three, uses concepts very similarly used as UML (probably because these languages were defined at the same time). This has an advantage that the transitions are relatively smooth. Therefore there are quite some tools that do automated code generation from UML to Java, and vice versa re-engineering code to UML diagrams[11]. For the object oriented programming languages it holds that they are well accepted in practice.

Object oriented database languages also exist in many flavors, but in this field standardization attempts are also being made. The leading standard is the already mentioned ODMG standard ([CattellEtAlii00]). Although the value of the use of object oriented database languages[12] is recognized, they are hardly used. Even in the context of object oriented software development, the main stream database systems are based on a different, namely *relational* information

[11] An example is the roundtrip tool for the Rational Rose UML modeling and design suite and IBM Visual Age for Java. A note to make here is that the translation is performed based on an intermediate 'meta language' for storing object design information called XMI. This language is based on the XML standard and is a proposed standard for interchanging object meta data.

[12] i.e. database systems that support object oriented database languages, and thus provide the possibility to handle their content as objects in the sense of object technology

model. Even though the relational model is a very elegant one, the transition from concept to implementation we mentioned above needs to leap over a completely different way of looking at information. This leap is normally bridged by an *object to relational mapping*. Even though such a mapping works out well in many cases, it is not based on a formal semantics of the object world (while the relational world has a more or less formal semantics). Such a formal model could be a good vehicle to define such a mapping.

There are some things to be said about some object oriented concepts that play an important role in the transformation. The object oriented concepts on static models has matured to an extent that the structure is common to most object oriented languages. Things get complicated when *constraints* start to play a role. Nevertheless constraints are getting more important in recent trends in software development with the focus on so-called 'business objects'. From the technological perspective business objects are persistent objects that can be shared by all the information systems in an enterprise or organization. Business objects model core information entities for the business of an enterprise or organization. Such business objects can have complicated structure and behavior, but particularly they have complex constraints in the form of *business rules* defined on them. These rules should be expressed by the modeling and design languages and should be forced upon the objects by the realizing database and programming languages.

In the modeling language UML there is a constraint language, called OCL ([WarmerKleppe99]), to express constraints, which enables a declarative way to write down constraints on objects. In the various object oriented databases there are constraint languages but there is not a constraint language in the proposed standard for object database languages by the ODMG. In the standard, constraints are to be forced by the operations on the objects, exactly like it is done in a programming language. This means that there is not a general mechanism that enforces the constraints, but every operation on the objects should make sure, in the end, that the constraints are satisfied. This practice imposes quite a gap between the design, in which we can declare the constraints, and the implementation, in which we need to enforce them. Most of the time custom mechanisms need to be defined to enforce the constraints. Alternatively, some constraints may be forced by the relational database system through an object to relational mapping. We see an important role in this area for a formal model for object orientation. The large complexity of the matter needs a thorough formalism for (better understanding of) the object oriented concepts in order to capture the constraints and perform the transformation. A lot can be gained if one could provide a general mechanism for solving this problem.

## 1.3 Object Oriented Software Development

A software development *process* defines *who* is doing *what*, *when*, and *how* to reach the goal of building or enhancing a software product. A very important tool in such a process is the language that formulates the accomplishments during the development process. This language is used in the descriptions of requirements, documents containing analysis of the universe of discourse for which one builds IT support, design models of the system to be built, and even the actual source code of the programs that denote in detail the working of parts of the system.

### 1.3.1 A very brief history of the OO software development process

In the late 1960s when software products became more and more complex, the need to present software architecture by means similar to engineering blueprints became apparent, in order to be able to communicate the content of the software and to guide the development of the software product to a successful end ([Jacobson85]). A significant milestone in the streamlining of software development processes was the issuance in 1976 by CCITT, the international body for standardization in the telecommunications field, of the *Specification and Description Language (SDL)* for the functional behavior of telecommunication systems. SDL was the first specialized object modeling language. Periodically updated it is still in use by a large number of developers. In the same context many other (non-specifically object technology based) languages with their companion methodologies were developed, of which the most influential is the language SA (Structured Analysis) with its methodology SADT (Structured Analysis and Design Technology) ([Ross77], [Ross85]).

Where the SA technology kept evolving in a steady way, the object technology inspired methodologies and accompanying languages became a real hype when object orientation became very popular (and more mature) in the late 1980s. Many object oriented development methodologies and design languages were introduced. Well known examples are the Booch method ([Booch94]), OOA/OOD (Object Oriented Analysis/Object Oriented Design) by Coad and Yourdan [CoadYourdan91a] [CoadYourdan91b], OMT (Object Modeling Technique) by Rumbaugh et alii ([RumbaughEtAlii91]), OL (Object Lifecycles) by Shlaer and Mellor ([ShlaerMellor88]), OOAD (Object Oriented Analysis and Design) by Martin and Odell ([MartinOdell92]), FUSION by Coleman et al. ([ColemanEtAlii94]), OOSE (Object Oriented Software Engineering) by Jacobson et al. ([JacobsonEtAlii92]), OOSD (Object Oriented System Development) by de Champeaux et al. ([deChampaeuxEtAlii93]), and MOSES by Henderson-Sellers and Edwards ([HendersonEdwards90]). The enormous amount of design languages provided a problem in communication of designs and automation of the development process. This fact initiated a considerable reduction in object

oriented design languages by the standardization effort of the leading forces in object oriented software development, resulting in a standard language for denoting information using concepts from object orientation. This language is the already mentioned *the unified modeling language*.

Now the industry has a standard object design language for use in object oriented software development, the trend to get to a unified process is also ongoing. This process is (of course) called *the unified software development process*, often abbreviated with 'UP' ([JacobsonEtAlii99]). We will use UP as a reference to give an overview on the object oriented software development process in the following section.

### 1.3.2 An overview of the OO Software Development Process

The aim of software development is to build a software system. A software development process is the set of activities needed to transform a user's need or requirement into a software system. The need or requirement that is to be transformed into a software system can vary from a simple processing demand of well understood entities to a request for sophisticated computations on various kinds of complex information to serve unintelligible processes. The sprouted software system, in the end, will have some purpose in the (more or less abstract) world in which the requirements make sense. In order to achieve this result an understanding of the world is needed, as well as an accurate description of the information that is processed. In order to get to this understanding and description (and eventually program code) a development process defines workflows and steps to gradually build the understanding (a model of the world). In this process languages are needed to write down the gained knowledge of the world. Here we get to the object technology in the object oriented development process. We will use languages that bear concepts of object orientation to write down the achievements in several stages of the process. These notions are important in the process because they are developed based on an intuition to talk about the worlds for which we build the software systems. The concepts are described in detail in the next chapter. The process itself also imposes some requirements on the languages as we shall see.

The general activities in software management are usually categorized by the following terms: *requirements*, *analysis*, *design*, *implementation* and *test*. Each of these activities (called *core workflows* in UP) are comprised of several tasks and have several deliverables. We give a short description the core workflows here:

- **Requirements**. The here goal is to find out what the purpose or the need of the users for the system is. A result of this activity is a list of requirements.

- **Analysis**. Here we gain a conceptual understanding of the world in which the system shall live, and what its function shall be. We make a conceptual model of the system

- **Design**. Here the conceptual model of the analysis phase is transformed to a technical description in terms that relate closer to what can be implemented with the current information technology. We expect from this activity a technical model and a systems architecture description

- **Implementation**. Here we build the system with actual information technology. We code programs to be compiled and executed on operating systems and middleware platforms, and code database schemas and procedures for storage of information on database platforms. The result should be a fully operating information system.

- **Test**. Here the operational system is tested. We verify that its performance satisfies the requirements set, assess that it serves its purpose in its world and make sure that it does not malfunction technically. This activity will result in a number of defect descriptions and additional requirements for the software system.

In a software development process these activities are organized in phases and steps. Traditional software development processes like the waterfall process for software development typically organize these activities in a strict sequence, where at the end of each activity one aims to have a completely finished deliverable for the whole system. Deficiencies in either of the deliverables force one to step back to the activity in which this deliverable was constructed, aiming again to fully complete the deliverable This process has some drawbacks[13], because in order to fully deliver the requirements or the conceptual model, one already needs full understanding of the world. In practice this is hardly ever the case.

The notions of object orientation enable an object oriented software development process to organize the activities differently. In an object oriented development processes the basic language constructs to build the deliverables are *objects*. These objects have a 'generality' that enables one to talk about the objects on an arbitrary abstract level. Objects can be referred to as meaningless labels, or as complex structures with sophisticated behavior. This feature enables one to let the objects that make up the deliverables evolve from an abstract indefinite version to a version that carries enough meat to realize the software system. Taking advantage of this the software development process organizes the activities in *iterations*. In an iteration artifacts of requirements, analysis, design, implementation and test evolve in parallel. In the early iterations most of the emphasis will

[13]Because we want to emphasize the notion introduced by OO software development we only mention this drawback, and do not go further into other fruits or drawbacks of the waterfall process.

be on requirements and analysis activities and only little on design, implementation and testing. In later iterations the most of the effort will go into design, implementation and testing, and less in requirements and analysis. In order to steer the software development process these iterations are organized in phases. For UP (but similar for the other OO development processes) these phases are:

- **Inception**. The primary goal of this phase is to establish the business case. After this phase one needs to be able to judge feasibility of the software system and validate its purpose.
- **Elaboration**. This phase focuses on do-ability. Here we need to establish the main part of the conceptual model and a basis for the architecture.
- **Construction**. Here we refine the conceptual and technical artifacts and do most of the building. This phase should deliver an initial system that operates and has all the main functions.
- **Transition**. Here the system and the artifacts are finalized and we validate its integral correctness.

## 1.4 Summary

In this chapter we described the practical context which is subject to scientific analysis and formalization in the coming chapters. The results presented in this thesis have their applications in precisely this context, and strive to contribute to the scientific fundaments of the domains of 'information processing' and 'software development'.

# Chapter 2

# Concepts in object orientation

*"Object-oriented programming is a wonderful example of how fruitful things don't happen very precisely"*

*(Robin Milner)*

One conjecture of this thesis states that the concepts which are actually used in the practice of object oriented information systems are *not* similar to the concepts that are common in contemporary mathematics. In contrast, the concepts of information systems evolved from the need and "way of looking at things" in practice. This entails that these concepts do not have a rigorous mathematical definition. There exists, however (and fortunately) quite some level of consensus on the meaning of those concepts (although hardly mathematically defined).

In this chapter we analyze and exemplify the concepts and notions for which we will construct a thorough mathematical theory in the succeeding chapters. We believe that the concepts we captured mathematically are amongst the most pronounced and widely used (and interesting) concepts that have become important in the field of object oriented information systems. Our attempt is to capture at least the *common* part of the conceptual world of object technology.

In this chapter we start to give an account on the sources from which we deduced the concepts mentioned. Thereafter we will present *our* interpretation of the concepts we strongly believe to be the most basic and interesting.

## 2.1 Concepts

The notions of object oriented information systems are defined in many different ways. Not only do the definitions differ in the level of formality, the definitions also differ in the level of *conceptuality*; i.e. in some presentations the notions

are explained using low level concepts of implementation (directions on how to implement the notions), and at other places the same notions are explained at a high conceptual level in terms of their desired behavior (abstracting totally on implementation). For example the use of an OID (object identifier) is more an implementation 'trick' to obtain the ability to distinguish two objects that have the same values, rather than a philosophically justifiable property of an object. Nevertheless it is a property that is used many times to describe the properties of an object. We do recognize the importance to have both the high (conceptual) level and the low (implementation oriented) level descriptions, because both kinds are actually used. Below we strive to strictly separate the low level and high level descriptions. We believe this results in a more clear picture of the concepts we will describe. In this section we informally describe and analyze the concepts of interest in this thesis.

### 2.1.1 Object and object identifier

In informal terms, an *object* in an information system represents some 'actual' entity, whether this entity is, for example, a person, or a scalar value. The most important property of an object is that it has an *identity*. The notion of identity is, although very common in use, quite difficult to understand ([Leeuwen93]). Many philosophers quarreled with this seemingly unproblematic notion, and, fortunately solved many paradoxes involving the use of identity (e.g. [Frege1892]). In information systems where object identity is important, there is also some quarrelling ([EmdeBoas96]). This quarrelling, though, hardly takes into account the research (in philosophy) on the notion of identity that has already been done.

To avoid many paradoxes when talking about identity, one can make (as in philosophy) a distinction between 'language' and 'meta language'. The meta language contains expressions that reflect directly things in the real world. In the 'language' one can talk about the real world, and expressions in this language are interpreted via this meta language.

In the object oriented information system practice both 'language' and 'meta language' is developed when building a model of the real world. For example rows in a database form pieces of the 'meta language vocabulary', while in some input screen one can type in 'language' expressions to specify a query. However in the object oriented practice no distinction is made between 'language' and 'meta-language' when talking about the real world. Although this is not really harmful, it requires a lot of accurate administration to avoid running into non-clarities and paradoxes. Especially because the concept 'object' and 'identity' is such an important ingredient of the paradigm. Below we sketch out some of the problems.

In (more) formal definitions of objects in information systems, an *object identifier* (OID) is usually associated with an object. Frequently an OID is some

unrelated value, which is used to distinguish objects for which we may have the same data in our information system, but which are known to be different[1]; and also for identifying an object of which some of its data is changed in the course of time. Furthermore an OID often serves as a handle to refer to an object, or in other words it acts as a *name or reference* for the object of which it is the identifier.

**2.1.1.** EXAMPLE. Imagine we have a bag with three marbles, one white marble and two black marbles. We can model this information as follows:

| **marbles** | OID | COLOUR |
|---|---|---|
| | x123 | black |
| | x456 | white |
| | x789 | black |

| **bags** | OID | CONTAINS |
|---|---|---|
| | b789 | x123, x456, x789 |

Note that we used the OIDs of the marbles to denote which marbles are in the bag.

> **Question:** Suppose we pick blindfolded a marble from the bag and this marble happens to be black. Which marble did we pick, x123 or x789? If the marbles 'look' exactly the same, does it matter which marble we picked? And what after someone told you that one of the black marbles is cursed?

▲

The 'trick' of using OIDs to solve the matters of identity usually works out fine. Sometimes, though, it does not. For example, if we want to be able to reason with an infinite set of objects, like the natural numbers, we have to assign an OID to all the natural numbers. As this is generally not possible we get the unnatural situation that some natural numbers -the ones we have stored somewhere in the information system- have an OID, and most others, -the ones we have not used yet- do not. To solve this situation most of the systems apply another trick: among the objects they distinguish so called *literals*, which are 'objects without an OID that are identified by their value' (e.g. [Cattell94]). Although this solves the problem of infinite sets of OIDs, the resulting non-uniformity of the collection of all objects is very inelegant and on a conceptual level, even incorrect[2]. In

[1] Using this trick, an attempt is made to satisfy Leibniz principle that states that no two individuals can be the same in all properties, without actually being the same. However the property that is introduced to accomplish this (the OID) is without any independent meaning.

[2] 'all objects have identity, but some objects have more identity than other objects'

our opinion using OIDs is a way to solve the matter of *implementing* the matter of object identity, but it should not be confused with concepts that handle the notion of identity semantically.

The notion of OID becomes really problematic if we consider incomplete information on the identity of objects or information that possibly contains wrong identifications. In some cases we may not know whether two objects are the same or not. The requirement of associating an OID to an object will usually assign different OIDs to these two objects. But if we discover in due time (by additional information) that these two objects are actually the same, we will have to identify the two objects. How to do this with OIDs is unclear: do we keep both OIDs, or only one of them, do we assign a new OID. Again we argue that the OIDs may implement the notion of identity correctly, but as long as we cannot actually identify two different persons (in reality) this notion is vague, sloppy, and incorrect from a philosophical point of view[3]. There is an alternative way to do it: come up with a philosophically sound 'trick' to handle identity, which is strictly separated from the tricks used to implement this notion. This gives a clear view on the matter of identity of objects, and enables one to reason about identity of objects without encountering disturbing paradoxes or inconsistencies.

We propose the following approach to handle objects and their identity[4]. In the languages we use to talk about the objects in an information system, we have *names* for the objects. A name is simply a word in the language used, and is interpreted to refer to the actual object it denotes[5]. The most important difference with an OID is that a denotation is *not* a property of an object. We must take care that different objects have different names. In contrast with the OID approach we do allow one object to have several different names. In order to keep track of the identities of the objects we denote, we maintain an *identity relation* ($\approx$) between the names and/or a *difference relation* ($\not\approx$). Thus if we know that two names $a$ and $b$ denote the same actual object, we will put $a \approx b$. If we know that they denote different objects we put $a \not\approx b$.

**2.1.2.** Example. Consider again the bag of marbles of example 2.1.1. We will use names, instead of OIDs to refer to an object.

| **bags** | | |
|---|---|---|
| $b$ | denotes | the bag containing $m_1, m_2$ and $m_3$ |

[3]Assuming that the identity is an immutable property of an object. Note however this assumption is not totally unquestioned (even outside 'science fiction').

[4]We also do not want to reinvent the fruits of 2000 years of philosophy, we simply take a Frege style approach ([Frege1892]) to handle the matter of identity. It is not the switch of philosopher (from Leibnitz to Frege) that bears fruit here, but the fact that we apply the principle of Frege in a semantically correct manner, instead of forcing the Leibniz principle with philosophically doubtful tricks.

[5]It is, roughly, the way mathematicians deal in a language with the identity of the mathematical objects.

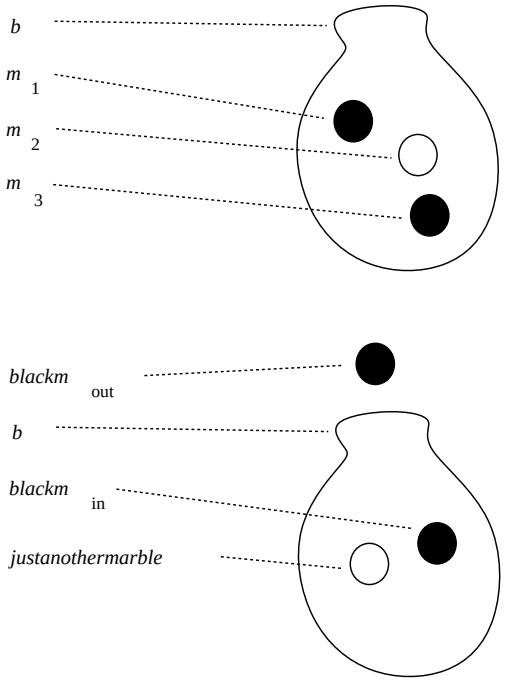



Figure 2.1: Denoting marbles

| **marbles** | | |
|---|---|---|
| $\text{name}_1$ | denotes | black marble $m_1$ |
| $\text{name}_2$ | denotes | white marble $m_2$ |
| $\text{name}_3$ | denotes | black marble $m_3$ |

| **inequality** | | (Take symmetric closure of the below) |
|---|---|---|
| $\text{name}_1$ | $\not\approx$ | $\text{name}_2$ |
| $\text{name}_1$ | $\not\approx$ | $\text{name}_3$ |
| $\text{name}_2$ | $\not\approx$ | $\text{name}_3$ |

Note that for simply modeling the structure this approach hardly differs from the one taken in example 2.1.1. The difference in approach becomes apparent if we want to determine identities[6].

Let us now pick a black marble from the bag. From its appearance we cannot determine whether we took out the marble we denoted with $name_1$ or the marble we denoted by $name_3$. We can only say that we now have a black marble out of

[6]"*merely to know that a name has as its referent an object with which we are confronted, or which is presented to us in some way, at a particular time, is not yet to know what object the name stands for: we do not know this until we know, in Frege's terminology, 'how to recognize the object as the same again', that is, how to determine, when we are later confronted with an object or one is presented to us, whether or not it should be taken to be the same object*" [Dummett73]

the back (in the picture denoted by $blackm_{out}$), and a black marble in the bag (denoted by $blackm_{in}$), and that they are different (i.e. $blackm_{in} \not\approx blackm_{out}$). Using fixed identifiers would cause problems here. ▲

This approach solves the problems with OIDs we mentioned above. If we want to talk about literals -objects that are nothing more than an immutable value- for example natural numbers, we simply take the token of that value as the name of the object. For example, for the immutable object 'the natural number 1', we take the name '1' or 'one', and for 'the natural number 2' we can take the name '2' or 'two' or 'b10'. This avoids the administration of superfluous OIDs. Also it solves the asymmetry of having object with or without OIDs, because all objects have their names, both non-literals and literals. When an object is changing some of its properties dynamically, its name still remains a valid reference to the object. When in course of time we discover that two names, $a$ and $b$ actually denote the same object, we only have one unambiguous action to take: add the equality $a \approx b$ to our equality relation.

When we analyze this phenomenon, especially the fact that we introduce an explicit (non) identity relation to handle the identity matter, we tend to conclude that this matter reveals a very strong intuition about objects that is neglected when making things precise. Drawing the three marbles from our example (or writing them down) as is done in the object oriented way of looking at things, we implicitly mean to give a lot *more* information than just labeling objects and giving them a color. Not only do we say that we refer to three objects wich are marbles, and of which one is white and two are black; i.e. in logical notation:

$$\exists x, y, z(\mathsf{marble}(x) \wedge \mathsf{marble}(y) \wedge \mathsf{marble}(z) \wedge \mathsf{black}(x) \wedge \mathsf{white}(y) \wedge \mathsf{black}(z))$$

but implicitly we also mean that these objects are different and that we really have exactly three objects in this bag and also that being white disqualifies being black; i.e.

$$\begin{aligned}&\exists x, y, z(\mathsf{marble}(x) \wedge \mathsf{marble}(y) \wedge \mathsf{marble}(z) \wedge \mathsf{black}(x) \wedge \mathsf{white}(y) \wedge \mathsf{black}(z)\\ &\quad \wedge x \not\approx y \wedge x \not\approx z \wedge y \not\approx z\\ &\quad \wedge \forall u[\mathsf{marble}(u) \rightarrow u \approx x \vee u \approx y \vee u \approx z]\\ &\quad \wedge \forall u[\mathsf{black}(u) \rightarrow \neg\mathsf{white}(u)]\\ &\quad \wedge \forall u[\mathsf{white}(u) \rightarrow \neg\mathsf{black}(u)])\end{aligned}$$

To strengthen our case, we remark that when we pick a black marble, we really cannot infer from the logical rules which one we picked, while if we had picked the white one, we could really infer from the logical rules that we picked the $y$ marble (in the scope of that quantor) and from that we could indeed determine its name; i.e.

$$\forall x, y \mathsf{marble}(x) \wedge \mathsf{marble}(y) \wedge \mathsf{white}(x) \wedge \mathrm{white}(y) \rightarrow x \approx y$$

A modern practical language for analysis and design like UML, which operates on a conceptual level, has circumvented this problem quite wisely by only requiring the existence of a notion of identity that is strong enough to distinguish different objects, leaving it to the implementer of the system how to realize it (in the extensions of the information specification). It also circumvented the mathematical obligation to 'realize' the identity of the objects on a high level in a model, because this language has no formal semantics. This omission, however, can from a practical point of view be justified, because the realizations of UML will not be formal either and moreover will probably use tricks like the OID (depending on the programming/database language used), so the axiom of identity normally suffices to be clear enough about this matter. The challenge is more that the realizations must be such that the problems that occur by using the tricks like the OID are to be solved. A formal model that realized this matter elegantly may provide a good example of how to realize it properly[7].

### 2.1.2 Complex value, type and class

Objects can be classified according to their *type*. Informally a type can be seen as a collection of all objects that have a certain property[8] in common; e.g. the property of being a person or the property of being able to jump, which includes a horse, a flea, and even a person. A common way to classify objects in information systems is to distinguish objects according to their *signature*. A signature describes what type of *basic building blocks* an object of this signature should at least[9] posses and which *abilities* it should at least have and to which other objects it is *associated*. Given a set of so called *basic types*, one can build *complex types* or signatures using several type construtors; e.g.

**2.1.3.** EXAMPLE. The signature of the type Book

```
type       Book
signature
           attr  title       :  string
           attr  author(s)   :  SET OF(person)
           attr  ISBN        :  ℕ × ℕ × ℕ × ℕ
           attr  publisher   :  company OR institute
           attr  year        :  year > 1450
```

▲

In the above example we used a couple of common type constructors. Firstly the title part of a book (hence forward called an *attribute* of book) should be of

[7] for example using a relation administering identity like the Frege style of dealing with identity [Leeuwen93]

[8] or collection of properties

[9] We will consider type requirements to be existential

type string. This type is considered a common 'basic' and 'predefined' type in most information systems. Other basic types are integer, bool, float and blob[10]. Most object oriented languages offer, next to the common predefined types, the ability to introduce new basic types, by defining the extension of that basic type. For example,

**2.1.4.** Example.

$$\begin{array}{lcl} \textsf{primary-colors} & := & \{\text{red, yellow, blue}\} \\ \mathbb{N} & := & 0|S(\mathbb{N}) \end{array}$$

▲

Another common type constructor is the `SET OF` type constructor. In the above example a book has a set of authors, which models the event that a book can have an arbitrary number (including zero) of authors.

A well known type construction in mathematics is taking the Cartesian product ($\times$) of existing types. In the example, an ISBN number consists of a row of 4 natural numbers; e.g. 90-351-1372-1[11]. We remark that the constructions of taking objects together like the Cartesian product and the 'is attribute of' constructor are not unproblematic from a 'design' point of view. For example we could have chosen to model the object of type Book to be the Cartesian product[12] of a title, a set of persons, an ISBN, a publisher and a year object. On the other hand we could have chosen to model an ISBN number to be a complex object with 4 attributes of type $\mathbb{N}$. Although for the Book object and for the ISBN object these are quite clearly not the intuitive ways to model them, for other object types this may be less clear a matter. The distinction that is generally made is that an aggregation operator for taking things together (like the Cartesian product) represents an 'IS A' relation between the object and the whole of the aggregation expression; or in other words a member of the aggregation IS part of the whole object. The attribute construction on the other hand models a 'HAS A' relation between the object and each of its attributes. In the example this means the a Book 'HAS A' title and HAS An ISBN number etc. etc.; while the ISBN number IS An (ordered) aggregation of four natural numbers.

To illustrate that this is a realistic problem we point to the practical programming language C++. Most C++ books (e.g. [Stroustrup91]) explain the attribute constructor (member in C++) as a modeling a 'HAS A' relation to distinguish it from an inheritance relation between classes. For example, if an ISBN number IS An aggregation of four numbers, it should be a subclass of the type $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, inheriting its properties. A Book on the other hand HAS

[10] blob is an abbreviation of 'binary large object', a type that is much used in multi-media information systems.

[11] We realize that the ISBN number encodes some information. For the sake of the example we only consider it as a sequence of numbers.

[12] we will call such a construction an aggregation.

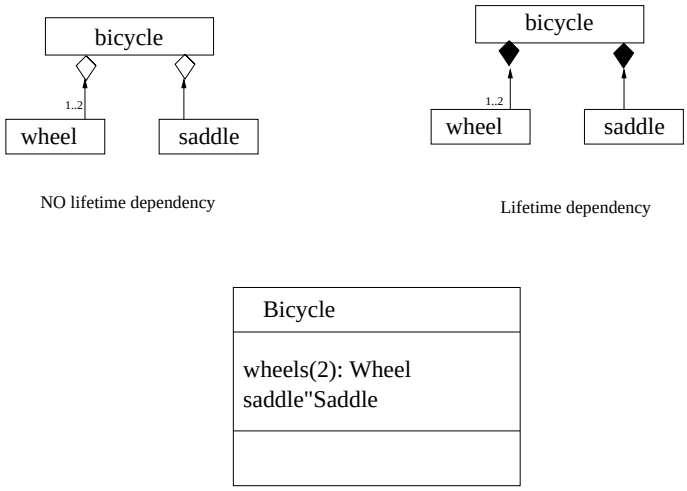



Figure 2.2: example of a HAS A and IS A aggregation in UML

A title, meaning that title should be a member of a Book. The new generation information system language UML also acknowledges complexity of the modeling choices that have to be made in this case and proposes next to attribute (HAS A) construction an aggregation operation in two flavors (to provide a 'middle course'). UML has two types of aggregations, relating these types of aggregation to a notion of 'life time dependency'. Between two objects with a lifetime dependency there exists an 'IS PART OF' relation (i.e. between the 'aggregation of all these parts' and the 'object' there exists an IS A relation), while between two objects that have no lifetime dependency there is a HAS A relation. A typical example of this phenomenon is the following: IS a bicycle a frame together with a saddle and wheels and a steer, or HAS a bicycle a saddle, a frame wheels and a steer (see figure 2.2). In the first view the bicycle is not the same anymore from an identity perspective when you replace its front wheel, because the parts that make up the definition of the bicycle changed. In the second view you can replace all its parts (all the parts that are modeled) and still have the same bicycle from an identity perspective.

The 'OR' type constructor in the above Book example is yet another type constructor which is generally known as the *union type* constructor. In the example, a book can be published by a company (e.g. North Holland publ. company) or an institute (e.g. the Institute of Logic Language and Computation (ILLC)).

It can be the case that for some reason one wants to constrain the set of objects of an existing type to some subset of this set. For example one wants to consider only printed books, and book printing was invented in 1450, so one considers books that were published after 1450. The ability to constrain a type is also a type construction. Although in type theory there is not much difference in defining a type by giving its signature or by formulating constraints, in information systems

these two activities are often separated, and often totally different languages are used to perform these two ways of defining types. For example the design language UML has many kinds of graphical schema techniques to write down the signature with limited ability to add some constraints. To write down constraints with full expressive capacity, UML has defined a (textual) constraint language OCL ([WarmerKleppe99]).

The concept of a *class* coincides largely with the concept of type. A class is also used to classify objects according to several properties. Next to properties and constraints, a class also epmhesizes *abilities* (which for simplicity we also consider to be properties of an object) by separately listing the operations that are associated with the type of objects in the class. Additionally, a class also contains the implementations of the functions (i.e. we can talk about methods) and the manner in which the objects are actually represented in the implementation. In other words a class consists of:

1. type
2. operations
3. body of implementation containing:
    - actual representation of the type of objects
    - implementations of the operations

**2.1.5.** Example. In this example we see the specification (coding) of a banking account type and its behavior.

```
class Account {
    // the signature of Account
    int      account;
    string   owner;
    float    balance;
public:
    // the methods of Account

    Account(int accountnr, string owner, float balance);

    float    get_balance();
    void     incr_balance(float amount);
    void     decr_balance(float amount);
private:
    // some things needed for implementation
    float     my_percent;             // the interest percentage
    date      prev_mutation_date;     // date of previous mutation
```

```
    float     build_interest;         // to keep track of the interest
};

// The body with the implementation of the constructor of the class
// and the implementations of the methods.

Account::Account(int accountnr, string owner, float balance)
{
    my_percent              := NORMAL_PERCENT;
    prev_mutation_date      := current_date;
    build_interest          := 0.0;
};

float Account::get_balance()
{
    return balance;
};

void Account::incr_balance(float amount)
{
    float add_interest;
    float add_percent;
    add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
    add_interest := balance * add_percent;
    balance := balance + amount;
    build_interest := build_interest + add_interest;
    prev_mutation_date := CURR_DATE;
};

void Account::decr_balance(float amount)
{
    float add_interest;
    float add_percent;
    add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
    add_interest := balance * add_percent;
    balance := balance - amount;
    build_interest := build_interest + add_interest;
    prev_mutation_date := CURR_DATE;
};
```

▲

### 2.1.3 ISA hierarchy, subtyping and inheritance

Types can be ordered in a type hierarchy or *ISA hierarchy*. For example consider the two types student and person and suppose one would like to assert that a

student is a (ISA) person, or in other words, one would like to assert that student is a subtype of person. This means that the set of all objects of type student should be contained in the set of all objects of type person. Usually a subtype relation like the one above is defined by adding phrases to the information system that formulate the subtype relation; e.g.

student ISA person

From a collection of these phrases, other unstated subtype relationships can be inferred. For example, if we also add the phrase graduate-student ISA student, we should be able to infer graduate student ISA person. Usually there is a formal system with inference rules that take care of these things. An influential paper on this subject is from Cardelli ([Cardelli84]). A system for subtyping usually contains, next to rules for reflexivity and transitivity of the ISA relation, the following rule (*):

> If type $A$ contains at least all the attributes that type $B$ has, then $A$ is a subtype of $B$[13]

For example:

**2.1.6.** Example. Look at the following three type definitions.

$$\begin{array}{llcl} \texttt{type} & \mathsf{vehicle} & = & (\text{age} : \mathsf{integer}, \text{speed} : \mathsf{integer}) \\ \texttt{type} & \mathsf{machine} & = & (\text{age} : \mathsf{integer}, \text{fuel} : \mathsf{string}) \\ \texttt{type} & \mathsf{car} & = & (\text{age} : \mathsf{integer}, \text{speed} : \mathsf{integer}, \text{fuel} : \mathsf{string}) \end{array}$$

According to the rule (*) we can derive the following relations:

$$\begin{array}{lll} \mathsf{car} & \text{ISA} & \mathsf{vehicle} \\ \mathsf{car} & \text{ISA} & \mathsf{machine} \end{array}$$

▲

In some cases though this inference rule can result in undesired subtype relationships; for example:

**2.1.7.** Example. Consider the following type definition:

$$\texttt{type} \quad \mathsf{fuelcontainer} \quad = \quad (\text{age} : \mathsf{integer}, \text{fuel} : \mathsf{string}, \text{size} : \mathsf{measure\text{-}of\text{-}volume})$$

[13] The rule I am aiming at is even more general then that stated above. The above rule, though, already suffices for the argument. Usually the rule looks as follows:

> If $A = [c_1 : C_1, \ldots, c_m : C_m, \ldots, c_n : C_n]$ and $B = [d_1 : D_1, \ldots, d_m : D_m]$ and also $C_1$ ISA $D_1 \cdots C_m$ ISA $D_m$ and $c_i = d_i (1 \leq i \leq m)$ then $A$ ISA $B$

Using the rule (*) we can unfortunately derive fuelcontainer ISA machine.

Even more treacherous is the application of rule (*) in the following example. Consider the following two type definitions for a polygon and a polyline respectively,

```
type       polygon
 signature
            attr    pointlist  :  LIST OF point
```

and

```
type       polyline
 signature
            attr    pointlist  :  LIST OF point
```

It is usual to represent both a polyline and a polygon by a sequence of points. But neither is a polyline a polygon nor vice versa. i.e. polyline IS NOT A polygon and polygon IS NOT A polyline ▲

In our model, which we present in the chapters to come, we have chosen to omit the mentioned subtyping inference rule (*). This will give the typing system more freedom. We will discuss this matter when addressing the notion of 'knowledge rules'. It is feasible to regain the possibility to accomplish the subtyping inference from example 2.1.7 when it is desired, by putting things just a little differently; e.g.

**2.1.8.** Example. Consider the type definitions of example 2.1.6. We can define the types for machine, car, and fuel-container with the following phrases:

> If an object is a machine then it has an *age* attribute and a *fuel* attribute; i.e.
>
> $$\mathsf{machine} \rightarrow \mathtt{attr}(\text{age}) \wedge \mathtt{attr}(\text{fuel})$$
>
> *If an object is a* car *then it has an* age *attribute and a* speed *attribute and a* fuel attribute*; i.e.*
>
> $$\mathsf{car} \rightarrow \mathtt{attr}(\text{age}) \wedge \mathtt{attr}(\text{speed}) \wedge \mathtt{attr}(\text{fuel})$$
>
> *If we want to be able to derive that a* car *ISA* machine *we will have to add the phrase*
>
> > If an object is a car then it is a machine; i.e.
> >
> > $$\mathsf{car} \rightarrow \mathsf{machine}$$

If we want the effect of the (*) rule we should formulate the properties of a type a little different; namely:

*Every object that has a* speed *attribute and an* age *attribute is a* vehicle*; i.e.*

$$\texttt{attr}(\text{age}) \wedge \texttt{attr}(\text{speed}) \rightarrow \text{vehicle}$$

Now we can derive that a car ISA vehicle without explicitly stating the ISA rule (i.e. we do not derive it from simple rules of reflexivity and transitivity of the ISA relation). ▲

Classes can be ordered in a class hierarchy, similar to the way types are ordered in a type hierarchy. The ordering among classes is given by a so called *inheritance* relation. In many cases the notion of inheritance largely coincides (in effect) with the notion of subtyping. But again, with inheritance there are matters of implementation that significantly determine the ordering of the classes, where this is not the case with types and subtyping. If we state that a class $B$ *inherits* the properties of a class $A$, we will say that $B$ is a *subclass* of $A$. Also, here the objects of class $B$ have all the important properties such that we can view them as objects of its *superclass* $A$. Some of the properties that are inherited (i.e. properties that determine the inheritance relation) are related to the implementation of the classes, which include the methods and their code and the attributes with their representations. A precise definition of inheritance is hard to give. An elaborate taxonomy article on inheritance results 7 different ways inheritance is used in the literature (see [Tailvalsaari96]).

In most Object Oriented programming languages there is a subtle difference between the use of subclasses versus subtypes. In C++, an instance of a subclass is not seen as an instance of the superclass, while an instance of a subtype is always an instance of the supertype. For example, in C++ if a class circle is a subclass of a class shape, then an instance of circle is *not* also an instance of shape. In this case it is said that an instance of class circle can play the *role* of an instance of class shape. The reason for this is related to the interpretation of properties of the whole class like the number of instances of a class.

Furthermore, some implementation matters play a role, like problems of choosing which implementations of methods have to be executed for an instance when one does not know how much further down in the class hierarchy the object may be specified. Object oriented design languages (e.g. UML) explicitly leave room for both the subtype and the subclass interpretations. The user then may choose the interpretation based on the language he will use to realize his designed system.

Even though in effect the concept subclass coincides largely with the concept subtype the main drive for subclassing seems to be based on *code sharing*. The code of class $A$ is used to implement a large part of its subclass $B$. This is, of course, a typical scenario when $B$ is a subtype of $A$; i.e. when a $B$-object is also an $A$-object. Not all the code of $A$ is always inherited though. In practice it became clear that the most strict notion of inheritance, which proscribes that all

the code of a superclass should be inherited, is not flexible enough to obtain both a high amount of code sharing and nice looking hierarchies. For that reason, in a subclass it is allowed to re-implement a method that is inherited from a superclass (i.e. not inherit the code of that method). This notion is known under the name *overriding*. The ability to override code from an inherited method necessitates another, purely implementation oriented phenomenon, which is that of *dynamic binding*. Traditionally, in the compilation of a program, a function (method) name is uniquely bound to a piece of code, which has to be executed when the function is called by its name. With methods in a class hierarchy this is not possible, because the implementation of a method can be overwritten at some point in the class hierarchy (while the typing of the method stays the same!). E.g.:

**2.1.9.** EXAMPLE. Consider the class definition of example 2.1.5. A subclass of the class Account should be a Savings-account, which is an account that gives more interest relative to the amount that is on the account but which forbids a negative balance and has a limit of how much money you may draw from your account in one withdrawal[14].

```
class Sav_account: superclass Account {
float my_decr_limit;
public:
Sav_account(int accountnr, string owner, float balance, float my_decr_limit);
};

// The body with the implementation of the constructor of the class
// and the implementations of the methods. This class inherits the methods
// show_balance and incr_balance, but has a constructor of itself and overrides
// the decr_balance method of its superclass.

Sav_account::Sav_account(int accountnr, string owner, float balance)
{
my_percent := HIGHER_PERCENT;
prev_mutation_date := current_date;
build_interest := 0.0;
};

void Sav_account::decr_balance(float amount)
{
float add_interest;
float add_percent;
if (amount <= MAX_WITHDRAW && balance - amount >= 0.0)
{
```

[14]Note again that, for clarity and non-C++ speakers, we do not use pure C++ syntax but a pidgeon OO programming language to declare some class to be a subclass

```
   add_percent := my_percent * (CURR_DATE-prev_mutation_date)/ONE_YEAR;
   add_interest := balance * add_percent;
   balance := balance - amount;
   build_interest := build_interest + add_interest;
   prev_mutation_date := CURR_DATE;
}; //fi
};
```

▲

For this reason, the code of a method can only be bound to its name when the method is actually called by an object. Only then, from the (sub)class membership of the object that calls the method, one can deduce which code to execute. This event is called dynamic binding.

### 2.1.4 Methods and operations

An important ingredient in current information systems is the ability to add *dynamics* to the objects in the system. This is done by adding *operations* to the information system. These operations alter the information in the information system and/or produce side effects that exhibit the desired behavior of the information system. In traditional OO systems these operations are exclusively associated with a type or class. For example:

**2.1.10.** Example. For specifying a bank account consider the following type:

```
type  Account
      signature  account-number
                 owner
                 balance
```

With the type Account we can typically associate the following operations:

```
operations
            get-balance
            increase-balance(amount)
            decrease-balance(amount)
```

▲

For operations that take more then one argument the strict connection between a type and a operation often amounts to problems of symmetry, also known as the *problem of the cow and the milk-can*. Suppose you have an operation $M$ that models the event of milking a cow resulting in a filled milk-can[15]. The

[15] i.e. The value of the milk attribute of the cow decreases with the minimum of the amount of milk that the cow passes and the amount of milk that fits in the milk-can, while the milk attribute of the milk-can increases by the same amount

problem now arises in determining which type we have to associate the milking operation $M$, the cow, or the milk-can, or something else. In other words, do we say to the cow: "Put your milk in the milk-can"; or do we say to the milk-can: "Extract the milk from the cow"; or alternatively, do we create a farmer and say to him: "Extract the milk from the cow and put it in the milk-can". The first two possibilities simply point to the asymmetry of the operation call, which arises when an operation with more then one argument is associated to one type only[16]. The last possibility of introducing a farmer solves the asymmetry but introduces dummy objects of dummy classes, i.e. objects that only exists to ensure the symmetry of the operation calls, but do not carry any necessary information[17]. In mathematical type theory the above case is not considered a problem. The milking operation $M$ is simply associated to the Cartesian product of the types for the cow and the milk-can, i.e. to type $\mathsf{cow} \times \mathsf{milk\text{-}can}$. Unfortunately in most database systems that carry the object notion combined with dynamics, this type constructor is not readily available for combining complex types. The constructor is very common in the traditional relational database model, but here we have no object notion and no dynamics. We will consider the ability to combine arbitrary types as an important and basic constructor for types. This way we can simply associate operations with the types of the objects they process.

In summary, an operation in an information system will be associated with a type, and is assumed to (possibly) change the content of the information system performing some combination of the following:

1. Altering the attributes of existing[18] objects,
2. deleting existing objects,
3. creating new objects,
4. perform side effects that do not alter the content of the information system.

In OO programming languages and database languages operations are, next to a name and a type, also associated with a specific *implementation*. Taking the name and the type together with the implementation we obtain what is usually called a *method* (although in a fairly new language like UML both items are separated; a signature there is called an *operation* while its implementation is called a *method*).

**2.1.11.** Example. Consider the type and methods of example 2.1.10. Note that the 'get-balance' method is actually a one-argument function, taking an account and returning a balance. The methods 'increase-balance' and 'decrease-balance'

[16] This also is the case if both the cow and the milk-can have their own milking operations
[17] they only have identity and nothing else
[18] i.e. present in the considered information system

are actually two-argument functions, taking an Account and an amount, and returning an Account. i.e.

$$\begin{array}{lcll} \text{get-balance} & : & \mathsf{Account} & \mapsto \text{balance} \\ \text{increase-balance} & : & \mathsf{Account} * \text{amount} & \mapsto \mathsf{Account} \\ \text{decrease-balance} & : & \mathsf{Account} * \text{amount} & \mapsto \mathsf{Account} \end{array}$$

The main restriction in the last two functions (and in all functions that are methods that update the object they are a method from) is that the identity of the Account in the domain and the co-domain is the same. So, in a general setting, the first method can be better viewed as a function of the following signature:

$$\text{get-balance} : \mathsf{Account} \mapsto \mathsf{Account} * \text{amount}$$

Instances of these methods can be viewed as 'Curry-ed' versions ([Barendrecht84]) of the general function taking an Account from the domain into the function. ▲

### 2.1.5 Encapsulation

Consider an information system for which there exists a nice categorization of the objects in classes. It is necessary to force a programmer that makes an application with this information system to really use this categorization. So instead of giving this programmer access to the representation of the objects, we only allow (her/him) the use of the methods of a class to operate on the objects of this class. From the point of view of the mentioned programmer, the representation of the objects is hidden, and only the method names are visible. This is called *encapsulation*. Encapsulation is a mechanism that enables the concept of a 'class' to be a real categorization, it makes sure that the category specified by the class is really used as such an informal category or type. In a sense the mechanism of encapsulation enables one to force a behavior of the 'classes' to be abstract (very similar to the behavior of 'types'), because it makes certain that a class is more than a collection of code that can be inherited.

### 2.1.6 Declarativeness

One of the main advantages of relational databases and deductive databases is their *declarativeness*. Declarative languages have some clear benefits over procedural languages, which are common in most OO databases. In order to query a database with a declarative query language one has to specify *what* information one wants to obtain from the database, and not *how* to obtain it.

Not only for a query language, but also for an object data definition language, declarativeness is a desirable, because then one only has to state what the type of data (signature) should be and which constraints the data should satisfy, instead of actually specifying how the data should be represented and how the constraints should be enforced.

**2.1.12.** EXAMPLE. In this example, we specify an electrical circuit with resistances. Instead of telling how the different objects are stored in our information system and telling how the different physical quantities can be obtained from the representation of the objects, we list all the attributes and the physical laws that relate the physical attribute to each other. This means we have the following types (expressed in a semi-logical language):

$$\begin{array}{lcl} \mathsf{circuit} & \rightarrow & \mathtt{attr}(v) \wedge \mathtt{attr}(i) \wedge \mathtt{attr}(r) \\ \mathsf{serial} & \rightarrow & \mathtt{attr}_1(\mathsf{circuit}) \times \mathtt{attr}_2(\mathsf{circuit}) \\ \mathsf{parallel} & \rightarrow & \mathtt{attr}_1(\mathsf{circuit}) \times \mathtt{attr}_2(\mathsf{circuit}) \end{array}$$

The first type definition says that a circuit has three attributes: a voltage ($v$), a current ($i$) and a resistance ($r$). The second type signature definition says that a serial circuit has two (serially connected) circuits, and similarly the third definition says that a parallel circuit has two (connected in parallel) circuits. The following constraint says that a circuit is either a resistance or a pair of parallel connected circuits or a pair of serial connected circuits. Note that this specification covers all possible simple circuits of parallel and serial resistances.

$$\mathsf{circuit} \rightarrow \mathsf{serial} \vee \mathsf{parallel} \vee \mathsf{resistance}$$

We now declaratively specify the physical rules for these simple electrical circuits, i.e. Ohm's law ($v = i * r$) and the rule for current in parallel circuits[19] and for resistance in a serial circuit[20]:

$$\begin{array}{lcl} \mathsf{circuit} & \rightarrow & \mathrm{equal}(\mathtt{attr}(v), \mathtt{attr}(i) \cdot \mathtt{attr}(r)) \\ \mathsf{circuit} \wedge \mathsf{parallel} & \rightarrow & \mathrm{equal}(\mathtt{attr}(i), \mathrm{sum}(\mathtt{attr}_1(\mathtt{attr}(i)), \mathtt{atttr}_2(\mathtt{attr}(i)))) \\ \mathsf{circuit} \wedge \mathsf{serial} & \rightarrow & \mathrm{equal}(\mathtt{attr}(r), \mathrm{sum}(\mathtt{attr}_1(\mathtt{attr}(r)), \mathtt{attr}_2(\mathtt{attr}(r))), \end{array}$$

Now it is possible to store (or have for insert or update) the information on a specific circuit in many different ways that is not complete, while (by inference) one can compute all defined quantities of the circuit. Given a proper constraint solver one does not have to specify how the values have to be computed for every case. For example a constraint solver like RL [Denneheuvel90] could infer from the above rules that for parallel circuits, 1 divided by the resistance of the circuit equals the sum of 1 divided by the resistance of its parts; i.e.

$$\frac{1}{r_{tot}} = \frac{1}{r_1} + \cdots + \frac{1}{r_n}$$

▲

[19]The current in a parallel circuit equals the sum of the currents in its parts.

[20]The resistance in a serial circuit equals the sum of the resistances of its parts.

Declarativeness is in our opinion an important key to nice and user friendly information systems. So next generation information systems should support declarative languages for defining and querying data.

Most declarative languages have nicely and mathematically defined relations as the basic structure of the database they talk about. Being a little ahead of events, we remark that in the coming chapters we will build a mathematical structure for complex objects for which we provide a declarative language for both defining and querying an object database[21]. For defining databases, this language will enable us to state many things that are common in non-declarative OODB languages. The only difference is that those statements are logical (they state what has to be the case) and are not directions towards the implementation or representation.

### 2.1.7 Rules and knowledge

In the preceding chapters we have seen some rules that define constraints on the types of an information system. We mentioned subtyping statements as described in section 2.1.3, like: 'A car is a vehicle'. We have also seen a phrase stating the following constraint: 'Every printed book should have appeared after the year 1450'. We mentioned constraints in section 2.1.2 where we discussed signatures of types. But using a general language, many more complex statements are possible. We could introduce negation and disjunction etc.. For example: 'If a vehicle is older then 100 years, or if it has a propeller, then it is not a car'. Or more complex: all laws of quantum mechanics. What we are aiming at is that in using a language which talks about information systems, it is desirable that this language is expressive enough to state phrases as the one above. This enables one to utilize the information system as a *knowledge base*. We will be able to check or enforce or even prove complex statements that bear the knowledge of experts in the field in which we use the information system. All of this, of course, depends on the expressiveness of the language in which one can express the constraints and rules.

### 2.1.8 Graphical representation

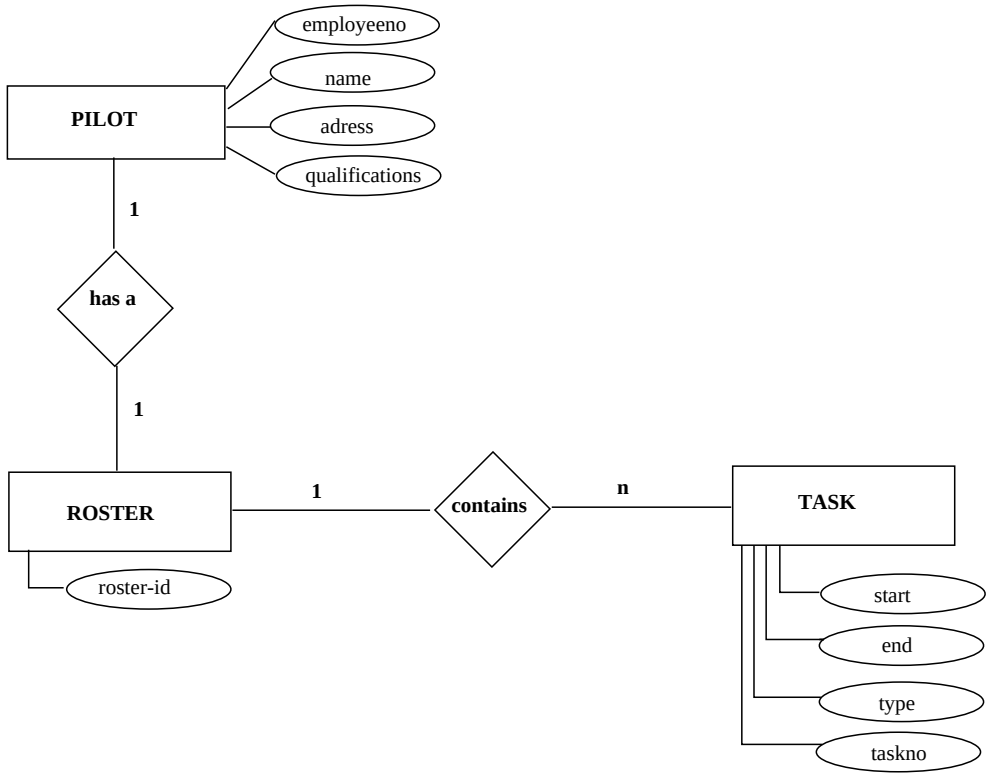
In the field of analysis and design there exist many graphical techniques to model information systems. The models, written using these graphical techniques, are usually translated to database and programming languages that in turn implement the modeled information system. Some of the graphical techniques from analysis connect closely to certain programming database languages. The most apparent example is the close connection between the (E)ER[22] model and the

[21] Remember that this is what this thesis will deliver

[22] (Extended) Entity Relationship model

languages of the relational model[23]

**2.1.13.** Example. An example of the tight connection between an ER model and a specification of the ER model in the relational database schema.




The ER schema above relates to the following relational database schema[24]:

| PILOT | |
|---|---|
| employeeno | integer |
| name | string |
| address | string |
| qualifications | list-of-airplane-types |
| roster-id | integer |
| key: | employeeno |

[23] e.g. Relational Algebra, QUEL, SQL, QBF, etc. etc..

[24] There are a number of ways to turn an ER schema into a relational schema. This one is a common transformation taking into account the cardinality of the associations. For a discussion about such transformations one can look in standard relational database textbooks (for example in [Ullman88]).

| TASK | |
|---|---|
| taskno | integer |
| start | time*date |
| end | time*date |
| type | string |
| key: | taskno |

| ROSTER-TASKS | |
|---|---|
| roster-id | integer |
| taskno | integer |
| foreign key: | rosterid, taskno |

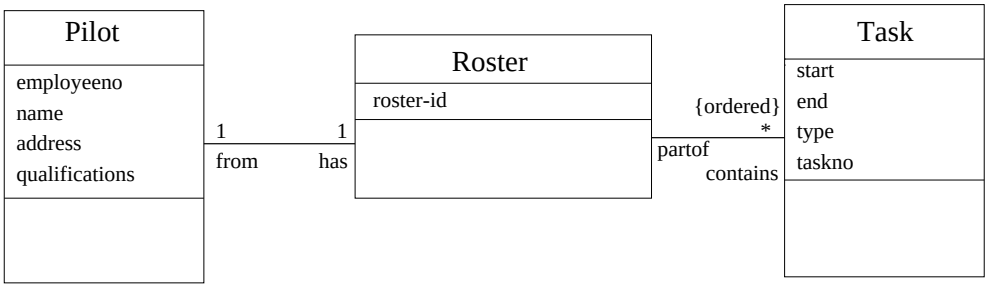
▲

The tight connection between the (E)ER syntax and the language of the relation databases initiated actually incorporating the (E)ER graphics in a database definition and query language for relational databases (e.g. look at the language HQL [AndriesEngels94]). The reason for the existence of a seamless transition from the (E)ER language to a language for relational databases lies in the fact that both languages talk about the same basic and mathematically defined structure: the relation. In other words, both the (E)ER model and the relational database model have a clear and rigorously defined mathematical semantics, based on the same mathematical structure called relation.

With the connection between graphical object-oriented or object-based analysis and design languages, and object-oriented or object-based database models the situation is completely different. Not only do these models lack a rigorous mathematical semantics, which alone makes the task of establishing the connection difficult, but there also exists a difference in the level of conceptuality between the notions of analysis and design versus the notions of object oriented or object based databases. Although the notions in both domains are similar, the definitions of these notions in analysis and design are more abstract and oriented towards reasoning (debating) about the information system. The similar notions in object-oriented databases are often oriented towards implementing the structures which formulate the notions.

On the other hand, the languages of object-oriented analysis and design use the notions of new generation information systems and are much more liberal in mixing graphical syntax with textual syntax, thereby improving the usability.

**2.1.14.** Example. The following UML class diagram with OCL (object constraint language) phrases denotes a slightly more informative model for the pilot roster example in the (E)ER diagram of 2.1.13.




Roster
self.contains→collect(type) INCLUDED IN self.from.qualifications→asSet

A note on the notation: The underlined header denotes the context of the OCL rule; with the dots (.) one can walk along the associations; and the arrow (→) denotes the reference to a method. In the above rule the expression *self.contains* in the context of *Roster* denotes the ordered list of task objects that can be reached from a Roster object. The expression *self.contains→collect(type)* in turn, denotes the invocation of the *collect* method of the ordered list objects where the recipient of the method is the ordered list of tasks objects that is reachable from a particular Roster object and the argument is *type* (an attribute name; i.e. a string object). This method returns the set of types as an ordered list of tasks. The *self.from.qualifications→asSet* expression denotes the invocation of the *asSet* method on the qualification object that is reachable from a roster object via the pilot object. The rule says that that for all the tasks (in a roster) the pilot (of that roster) should have the necessary qualifications. ▲

In this thesis we aim to provide a mathematical foundation of the basic concepts of the object oriented and object based databases and analysis and design methodologies. The foundation will play the same role as the basic concept of relation in the (E)ER model and the relational database model. We will use this foundation both for interpreting the graphical syntax[25] like that which exists in analysis and design, and the non-graphical syntax which is common in the languages of databases. As a matter of fact, in some cases we even prefer the graphical syntax over the non-graphical because it shows more clearly the structure of the entity it denotes. We will also incorporate the graphics into the database languages.

[25]We need to be more formal about the term 'graphical syntax', because for a real 'syntax' one needs a formal 'syntactic theory'. We will explain our view on graphical syntax in chapter 4 when we do the theory, because this matter really is theory.

### 2.1.9 Partial specifications, Identity, and the Extendibility principle

The concepts of class and type hierarchy of the object oriented information systems both give, in effect, a way of manipulating entities for which one only needs a partial specification. For example, one can specify rules or actions for a machine without knowing whether this is a car or a generator or whatever machine. In other words, we are able to talk about objects taking into account only a part of its information. In order to talk about a (structural) part of an object we need to distinguish *partial descriptions* of the specification of an object. In most OO languages one has the ability to write down (talk about) the individual connections between an object and its attribute. These connections are the building blocks for describing structure of an object. For example, if we want to talk about a car or a generator as if it is a machine, we look only at the connections between the object we consider and the objects that describe the machine part of that object[26]. These connections (later on we will name them *links*) make up part of the specification of the car or generator we generalized to a machine.

Analysing this feature that enables one to look at only a part of the specification, one can generalize its intent and assume that objects *have* (potentially) only a partial specification. This enables one to consider objects in an information system, for which we, at a certain point in time, or at some level of abstraction, have only partial knowledge, and for which this knowledge can be augmented in course of time or when we de-generalize. The fact that in our models the objects have an identity, and therefore are not identified by their structural and behavioral specification, enables one quite naturally to handle the objects like they have only a partial specification. Furthermore it is very relevant in practice, where often at some point in time there is no complete knowledge about all objects in the information system.

We argue that this incompleteness of the specification is tightly connected to the concept of identity in information systems. The fact alone that we can distinguish an object by its identity without knowing its complete specification gives a lot more strength to the concept of identity. In fact it is the concept that makes the Object Oriented paradigm work. The ability to make powerful generalizations (i.e. consider superclasses) makes the languages that support object orientation very expressive. And furthermore, being able to dynamically discover more specific information about an object, and to classify an object at a finer granularity (by considering the subclass), makes the system very flexible. If we would need a complete specification, like for example in the relational database model, the identity would be nothing more then a label to distinguish it from other objects with the same specification. The concept of identity in a system that allows partial specifications and generalizations enables one to talk about object

[26] i.e. the part of the structure of the object that makes this object a machine.

at all levels of granularity without coming into problems with generalizations and de-generalizations.

The assumed partialness of the specification of an object has an important consequence for the models of information systems, though: in such a model, the composition of all the parts of the object does not result in the object itself, because one never knows whether the specification is complete. We will, in the system we present in the next chapters, introduce the concept of *extendibility*, and say that a partial specification can extend to the whole object it is part of.

## 2.2 Summary

In this chapter we analyzed the most advertised concepts of object orientation. In the subsequent chapters we will present a mathematical formalization of a large part of these concepts. We will see a mathematical model for object oriented information that could play a role in clarifying and enhancing object oriented information processing, similar as the relational model does for relational information.

# Part II

# A Model for Object Oriented Technology

# Introduction

In this part we propose a model in which we can do theoretical research in object oriented technology. We will introduce two artifacts for this model:

- a *language* for expressing information in the object oriented way
- an *interpretation* of this language in a mathematical model; i.e. *semantics*

The language we present has a formal syntactic theory that has both textual and graphical components. The graphical components are called *edge graphs*. From these constructs we build so called *categorial graphs* to denote types and information models, and *object graphs* to denote actual information[27]. The languages of categorial graphs and object graphs are generalizations of the language constructs in object oriented languages. The constructs connect closely to the high level languages for object oriented analysis and design, like UML.

The semantics consists of a mathematical model in which we have objects and partial descriptions. If an information modeler constructs a categorial description of an information system, all the instances of information content he envisages to be possible are actual models of the description.

So how do these models relate to the practical situation of an operational information system? A particular situation in an operational information system coincides with a particular semantic model that satisfies a description written down by the information system designer (written down in the language of categorial graphs). One can write down parts of actual or possible models using

[27]In other words we use the categorial graphs to write specifications of objects, and object graphs to denote the actual objects categorized by the categorial graphs, just like a data model denotes specifications of information objects in a relational database, and records the actual information objects in a database.

the object graphs, just like one can write down diagrams in UML that exemplify actual or possible situations in an information system.

And how do these models relate to the practical object oriented languages? In analysis, design and implementation languages, one writes down (possibly graphical) symbols. These symbols have a meaning, i.e. we interpret these symbols to be something they stand for. In order to give a *precise* meaning, one needs to give that meaning in the form of a mathematical construction in which information objects live and behave as one has specified in the language. Such a mathematical construction will be presented in this part of the thesis.

We will start this part with a presentation of the language for object oriented information systems; the syntactical theory in chapter 3. The semantic domain, the mathematics in which we interpret the language of object oriented information systems will be presented in chapter 4.

# Chapter 3

# A generalized language for object oriented information systems

*"Again we face most basic questions like what is the right logic and even what are the right structures"*

*Yuri Gurevich ([Gurevich88])*

The large popularity and numerous occurrences of modeling and database languages using graphical syntax and object notions suggest that *it underlies an important intuition on how to model parts of the (real) world*. We will present a language that bears these notions. In [Adriaans92] Pieter Adriaans proposed graphical structures called '*categorial graphs*' that in a very general way describe most of the structures used when modeling for analysis and design[1]. We will, in this thesis, take the notion of 'categorial graph' as a starting point for our theory on modeling with object oriented information systems. We will 'mathematize' (i.e. turn into mathematics) the notion of 'categorial graph' to obtain a model for object oriented information systems.

In this chapter we present a language, called *categorial graph language*, that combines both graphical and textual phrases and is tailored to define information systems. The categorial graph language is modeled after the real life practical languages of modeling and design, and bears the features of object orientation we described in the previous chapters. We also present a language of *object graphs* that enables us to write down particular models of object oriented information systems that satisfy the categorial description in the categorial graph language.

In the framework of categorial graphs we will make a distinction between the way in which we write down matters of signature, and matters of constraints. The matters of signature are denoted graphically, while for the constraints we assume

[1]There also exists a modeling tool for these graphs (see [Adriaans92]).

the presence of a constraint language in which we write down the restrictions. From a type-theoretic viewpoint this distinction may seem artificial, but in the eyes of an information system designer, it is the way it is traditionally done. Our aim is to stay as close as possible to the practitioners intuition as possible.

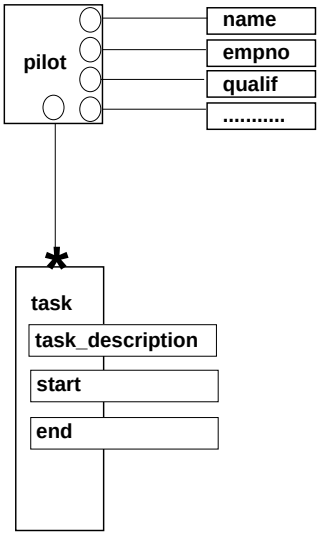
In the next section we will start with an example to illustrate the language and notions for object oriented information systems in an informal presentation of a case. This example will emphasize important features for object oriented information system languages. We will use this example in the whole chapter to solidify the abstract definitions of the syntactical theory.

In the subsequent section we will display the syntax of a class of languages, called the categorial graph languages. These languages contain a graphical and a textual part. The graphical part is built from so called *edge graphs*. The edge graphs form the core of the class of languages. The textual part is a language that is used to write down constraints. Next to presenting the syntactical building blocks of the language we will discuss syntactic matters like writing down schemas and (typed) instances for information systems. Furthermore we will tackle a common problem of writing down large schemas for information systems using a good way of imploding and exploding the syntactic structure (the edge graphs), without losing means to interpret the syntax in a proper way.

## 3.1 A Case for Object Oriented Information Systems

In the example below we will speak about objects with a complex signature: the have attributes and aggregations. We write down both the schema and an instance of an information system. Furthermore we extend the schema by adding constraints to the signature.

**3.1.1.** Example. Consider a working roster of an aviation pilot. As a simple example, his roster will look like a sequence of flights, time-offs, and possibly some obligatory training courses. His roster then could be modeled like an aggregation of objects that model flights, time-offs and courses. These flight, time-off and course objects then, would typically have attributes (adjacents) like start-date-and-time, end-date-and-time. A flight also would have as an attribute the departure airport, and the arrival airport. In a picture:



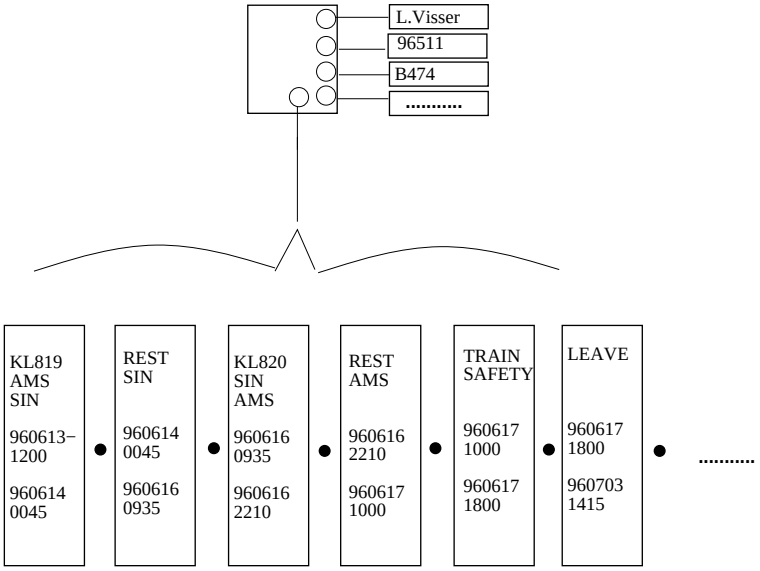

The picture also contains *cardinality* constraints. The '*' denotes the adjacency of *zero or more* tasks to the pilot. It would be convenient to have the ability to denote that we have an object of type roster that is an arbitrary long aggregation of edges of some other type (i.e. task). The roster is an aggregated object, and can be addressed (as a whole) just like any other object.

- A roster ISA aggregation of zero or more tasks

Furthermore we add a constraint that touches *inheritance*. For a clear division between object-structure and object-constraints we do not denote it graphically like in most OO design languages (this will become clear later on in this chapter).

- A pilot ISA person

An instance of the above schema could look as follows:



For the safety of the passengers, the rosters of pilots should respect certain rules. In this example we state two rules:

- A flying duty should be rostered to a pilot and not to some other type of person
- A flying duty should be preceded by a rest of at least 11 hours

▲

## 3.2 The Syntactic Theory

The language of categorial graphs combines both graphical and textual phrases. For a proper presentation of this language we need a nice syntactic theory like the theory of formal languages (in a sense we are leaving the good developed theory of words as described in [Davis58], [HopcroftUllman79], [LewisPapadimitriou81]). Note that it is not problematic to consider graphical structures in the syntax, as long as there exists a well defined mathematical formulation for these syntactic structures. One should provide a simple syntactic theory for the graphical structures in the same style as done for conventional textual structures. Observe that graphical structures like graphs are simple mathematical entities, just as strings, words and sentences are.

The language of categorial graphs is tailored to defining and writing down information models and database schemas. It consists of a graphical part, in which one can specify the signatorial matters, and a textual part in which one can state constraints on the specified signature. This is exactly as most information system languages, especially those in analysis and design, are structured. The most commonly known are UML, NIAM and (E)ER.

The syntax for the graphical part of the language of categorial graphs is built from edge graphs, which are presented below. The edge graphs define a signature for an information system like a theory of categorial grammars Using a sufficiently rich language, one can put (additional) constraints on these categories (types).

### 3.2.1 Edge Graphs

When we draw pictures of the objects we are modeling, we in a way 'talk' about these objects in terms of boxes and lines and other graphical constructs. Defining the graphical syntax more formally we could say we talk about the objects in terms of edges and vertexes. In our language we want to talk about objects as if they were structured entities. We could, of course, encode the structure of an object using vertexes and edges. However, when one starts modeling some part of the world, the structure and complexity of the objects are not known in their full extent. One will not know 'a priori' which possible attributes of an object

to take into the model or which relations to other objects exist. In order to talk about the objects one would then need a way to address the objects without being specific right away (or to a full extent) about the structure of an object. Moreover, it would be very natural if one did not need to change the structure of all the sentences when in the process of modeling the world one needs to add parts of an object to the model. This is only possible if one has a direct way of denoting complex objects and furthermore that the interpretation of these objects are *incomplete or even non-wellfounded* in the sense that parts of the object can be left 'unspecified' and filled in later.

A direct way to represent a structured object can be achieved by having it denoted by a structured graphical entity which is a basic building block of the graphical syntax. Moreover, we need a special 'structured object' that is empty, so we will have ways to extend the complex structure of the objects in our syntax without having to change all of the syntax.

The structured entity we will use as a basic building block is mathematically equivalent to a *hyper edge*. An edge has a structure, namely it has *adjacents*. We will talk about structured objects using these edges. This is a powerful generalization of using graphical entities for describing objects. We will be able to denote all objects, whether simple or complex, using a mathematical structure that is formally an edge. This way, all objects, both simple and complex, can be treated uniformly, because they are all denoted by an edge. We will also introduce an 'empty edge' that marks the end of the structure of an object[2]. This way additions in the structure of an object will only affect the denotation of the object itself, not of its context.

An edge graph is a simple generalization of an ordinary graph. Consider a collection of nodes. In between these nodes we can draw edges. Such an edge has as its source and as its target a node. Having drawn edges, we can imagine we can draw edges between edges; i.e. we can draw an edge which has an edge as its source and an edge as its target. The same way we can draw edges between a node and an edge, or between an edge and a node. Let us now assume that we can imagine a node to be an edge as well. We define node as being an edge with as its source and as its target some distinguished abstract edge, called the empty edge, denoted by **1**. The structure we then obtain contains edges only. We will call such a structure an '*edge graph*'.

**3.2.1.** Example. Look at figure 3.1. Here we have drawn two edges $a$ and $b$ that both have the empty edge as their source and as their target. These edges are in a sense basic edges (or nodes) because their source and target are 'empty'. Note that in the figure we 'copied' the empty edge **1** a couple of times for drawing purposes. Edge $c$ is a complex edge, having two other edges as its source and as

[2]e.g. An object will always have an empty adjacent

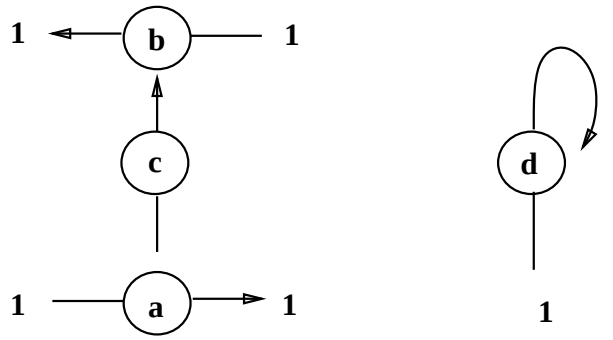



Figure 3.1: an example of a binary edge graph

its target. We can also make very weird edges like edge $d$, which has the empty edge as its source and itself as its target. ▲

We can generalize the above concept of edge graph to a concept of *hyper* edge graph, in the same manner as graphs are generalized to hyper graphs. Instead of letting an edge have exactly two adjacents, its source and its target, we can allow an edge to have an arbitrary number of adjacents. Such an edge is called a hyper edge. From hyper edges we can build hyper edge graphs.

When modeling part of the real world, there are several ways to talk about the structure of an object. One can be interested solely in the fact that an object has a certain attribute or adjacent, i.e. in graphical denotation we are interested whether there is or is not a line between one object and another. For example, if we are interested whether an object of type man has an arm adjacent. In this case the adjacency structure can be described by a *set*. An object has a set of adjacents, and membership to that set determines whether an object is adjacent to another object.

One could also be interested in *counting* the adjacents of an object. For example, one can be interested in how many adjacents of a certain type an object has. In this case the adjacency structure of an object is resource conscious, and can be described by a *bag* or *multiset*. In this case we may draw more then one line between two objects. E.g. one can then express that an object of type man has two arm adjacents.

One could even more specifically be interested in the fact that the first or second adjacent of an object is of a certain kind. As another example one could want to say that a certain adjacent of an object is the the adjacent labeled by 'child' and another adjacent is labeled by 'parent'. To be able to express these things the adjacency structure needs to be ordered. This can be expressed using an *ordered list* for describing the adjacency structure of an object. For example

one could say of an object of type man that it has an arm adjacent labeled by 'left' and an arm adjacent labeled by 'right'.

It is important to observe that if we want to have some specific abilities to talk about the objects one is modeling, this has direct consequences for the theory on it. Not only does the language need a way to denote the things one wants to say, moreover the semantic domain in which this language is interpreted needs to model the desired behavior. In other words if, we want to talk resource consciously about objects, we need a syntactic graphical entity that can denote a multiset structure, and moreover, one needs to be able to count the adjacents of an object in the semantic domain in which we interpret the syntactic constructions.

In order to be able to talk about objects in the different ways indicated above, the syntactic entity of a hyper edge comes in different flavors. Starting with the most expressive one, there are the so called '*directed*' hyper edges. For these hyper edges the order of the adjacents is important. For a directed hyper edge we have a first adjacent, a second adjacent, a third adjacent and so forth. The adjacents of a directed hyper edge form an *ordered list*. We will also consider '*undirected*' hyper edges. For these edges the order of their adjacents is irrelevant. The adjacents of an undirected hyper edge constitutes a *multiset*. The most abstract flavor of hyper edges we will consider are the so called '*set*' hyper edges. With a set hyper edge we abstract over the multiplicity of its adjacents. The adjacency structure of such an edge can be given by an ordinary set[3].

**3.2.2.** Definition. (universes of edges and the empty edge) We will consider three universes of edges, one for set hyper edges denoted by $\mathbf{Edge}_{\text{set}}$, one for undirected hyper edges denoted by $\mathbf{Edge}_{\text{undirected}}$and one for directed hyper edges denoted by $\mathbf{Edge}_{\text{directed}}$. All three universes contain a designated edge called the *empty edge* which is respectively denoted by $\mathbf{1}_{\text{set}}$, $\mathbf{1}_{\text{undirected}}$ and $\mathbf{1}_{\text{directed}}$.

On the edges of the three universes, so called *adjacency functions* will be defined that map these edges to their adjacents. For edges in $\mathbf{Edge}_{\text{set}}$ these adjacency functions map edges to a *set* of edges, for edges in $\mathbf{Edge}_{\text{undirected}}$ to a *multiset* of edges and for edges in $\mathbf{Edge}_{\text{directed}}$ to a *list* of edges.

We also introduce the empty edge for the hyper edge case. Even though it is not necessary to have the empty edge in the hyper edge case, because we have genuine empty sets, empty bags and empty lists in the definition of a hyper edge (this was not the case with the binary edges). We introduce it here because it enables us later on, in an algebraic setting, to conveniently talk about (being or having) an empty structure. To ensure that $\mathbf{1}_{\text{set}}$, $\mathbf{1}_{\text{undirected}}$ and $\mathbf{1}_{\text{directed}}$ really behave like the empty edge, we will require the following:

For all adjacency functions *Adj* we put

[3]Note that we do not demand the adjacency structure (list, multiset or set) to be finite, although in almost all practical cases it will be finite.

- $Adj(\mathbf{1}_{\text{set}}) = \emptyset$ (empty set)
- $Adj(\mathbf{1}_{\text{undirected}}) = \ddot{\emptyset}$ (empty multiset)
- $Adj(\mathbf{1}_{\text{directed}}) = []$ (empty list)

To enforce the neutral behavior of the empty edge ($\mathbf{1}$) we will identify (in the interpretation) the adjacency-sets $\{a, b, c\} \cup \{\mathbf{1}_{\text{set}}\}$ and $\{a, b, c\}$. In other words, $\{\mathbf{1}_{\text{set}}\}$ will behave like the empty set. Similarly $\ddot{\{}\mathbf{1}_{\text{undirected}}\ddot{\}}$ will behave like the empty multiset and $[\mathbf{1}_{\text{directed}}]$ will behave like the empty list[4]. △

Note that the presentation of an edge graph relates to the second most popular presentation[5] of conventional graphs, where a graph is given by a set of nodes $V$ and a set of edges $E$ together with two functions $source : E \to V$ and $target : E \to V$ mapping the edges respectively to their source and their target.

**3.2.3.** Definition. (set edge graph) A *set edge graph* is given by a pair $(G, Adj_{set})$ where

- $G \subseteq \mathbf{Edge}_{\text{set}}$ is a set of objects called *set hyper edges*,
- $Adj_{set} : G \to \mathcal{P}(G)$ is an adjacency function mapping the edges of G to their adjacents (which form a set).

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{set}$ or even plainly $Adj$, provided this does not lead to confusion. △

The definitions for undirected edge graph and directed edge graph are similar to the definition of the set edge graphs. For the sake of completeness we will give these definitions anyway.

**3.2.4.** Definition. (undirected edge graph) An *undirected edge graph* is given by a pair $(G, Adj_{undirected})$ where

- $G \subseteq \mathbf{Edge}_{\text{undirected}}$ is a set of objects called *undirected hyper edges*,
- $Adj_{undirected} : G \to \ddot{\mathcal{P}}(G)$ is an adjacency function mapping the edges of G to their adjacents (which form a multiset)[6].

[4] Admittedly these identifications look a little peculiar in this set-theoretic presentation. Note however that in algebraic formalisms we have no problems at all with these kind of neutral elements.

[5] The most popular presentation, of course, consists of two sets $V$ and $E$ where $E \subseteq V \times V$.

[6] The notation $\ddot{\mathcal{P}}(G)$ denotes the set of all possible multisets constructed with elements from $G$, similar to the powerset set construction for ordinary sets.

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{undirected}$ or even plainly $Adj$, provided this does not lead to confusion. △

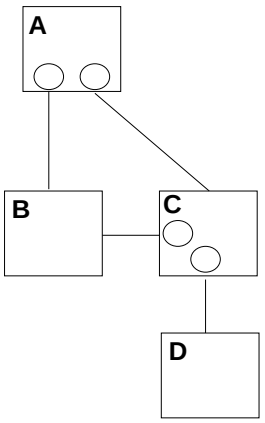
**3.2.5.** DEFINITION. (directed edge graph) A *set edge graph* is given by a pair $(G, Adj_{directed})$ where

- $G \subseteq \mathbf{Edge}_{\text{directed}}$ is a set of objects called *directed hyper edges*,
- $Adj_{directed} : G \to \mathsf{list}(G)$ is an adjacency function mapping the edges of G to their adjacents (which form a list)[7].

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{directed}$ or even plainly $Adj$, if this does not lead to confusion. △

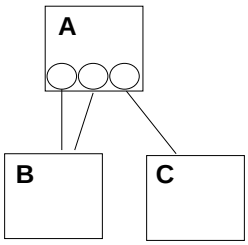
**3.2.6.** EXAMPLE. Below we have drawn 3 edge graphs, one of each flavor. The first one is a set edge graph, the second an undirected edge graph, and the third a directed edge graph. We have denoted the graphs graphically and have given their mathematical description.

[7]The notation $\mathsf{list}(G)$ denotes the set of all possible lists constructed with elements from $G$, similar to the powerset set construction for ordinary sets.



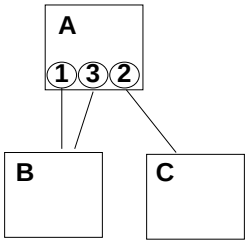

**SET edge graph**

**G={A,B,C,D}**
**Adj={ (A,{B,C})**
**(B,{})**
**(C,{B,D})**
**(D,{}) }**




**UNDIRECTED edge graph**

**G={A,B,C}**
**Adj={ (A,{B,B,C})**
**(B,{ } )**
**(C,{ } ) }**




**DIRECTED edge graph**

**G={A,B,C}**

**Adj={ (A,<BCB>)**
**(B,<>)**
**(C,<>) }**

▲

Note that in theory it is possible that an edge has itself as its source or its target. It is even possible that an edge has itself as the target of the source of its target. We will call this kind of edge '*cyclic edges*', and graphs containing such edges '*cyclic edge graphs*'.

We can also imagine that it is possible that a given edge has the property that one can infinitely many times 'descent' to its adjacents (source or target). Note that, among others, a cyclic edge has this property. If an edge has this property, we will call it an *unfounded* edge. On the other hand we will say that an edge is '*founded*' if there is no infinite chain in its adjacency structure. This means that at a certain point in descending along the adjacencies, one should encounter an edge that has an empty adjacency structure. For this purpose we

introduced the *empty edge*, denoted by **1**, which behaves similar to the empty word in standard formal language theory. The empty edge enables us to denote the fact that an edge has no source or target, by saying that the source or target is the empty edge. Of course the empty edge itself also has no source or target, meaning the same as saying that it has only itself as its source and target[8]. We will call an edge graph that contains, apart from the empty edge, only founded edges a '*founded edge graph*'.

**3.2.7.** Definition. (founded edge graphs) A set edge graph $G$ is called *founded* if

1. $\mathbf{1} \in G$ ($G$ contains the empty edge) [9],
2. all edges are *'founded on one'* (FOO) where
$$\mathrm{FOO}(a) \Leftrightarrow a = 1 \text{ or } (\mathrm{Adj}(a) = A \text{ and } \forall b \in A[\mathrm{FOO}(b)])$$
3. acyclism[10]
4. nothing else but implied by 1,2,3 and 4.

△

The definitions for founded undirected and founded directed edge graphs are very similar. We leave these definitions as an exercise for the reader[11].

As an aside, we note that we can *encode* an edge graph with a normal (conventional) directed graph with nodes *and* binary edges. In the representation using a conventional graph, an edge graph is simply a graph in which the nodes denote the names of the 'edges' of the edge graph and the directed edges of the conventional graph point to the nodes that denote the names of the adjacent 'edges' of the edge graph.

**3.2.8.** Example. Consider the following (set) edge graph G:

$$\begin{array}{lcl} G & = & \{a, b, c, 1\} \\ \mathrm{Adj} & = & \{(a, \{1\}), (b, \{1\}), (c, \{a, b\}), (1, 1)\} \end{array}$$

This graph is drawn in figure 3.2 in three ways, the upper two different edge graph notations for graph $G$, and below these two, a conventional graph $G'$ is drawn

[8] This infinity in the possible denotation of the empty edge motivated us to use the term *founded* instead of *wellfounded*, because the latter term in set theory surely forbids the empty edge

[9] Redundant.

[10] implied by 2

[11] Hint: Only one symbol in the second clause needs to be changed!

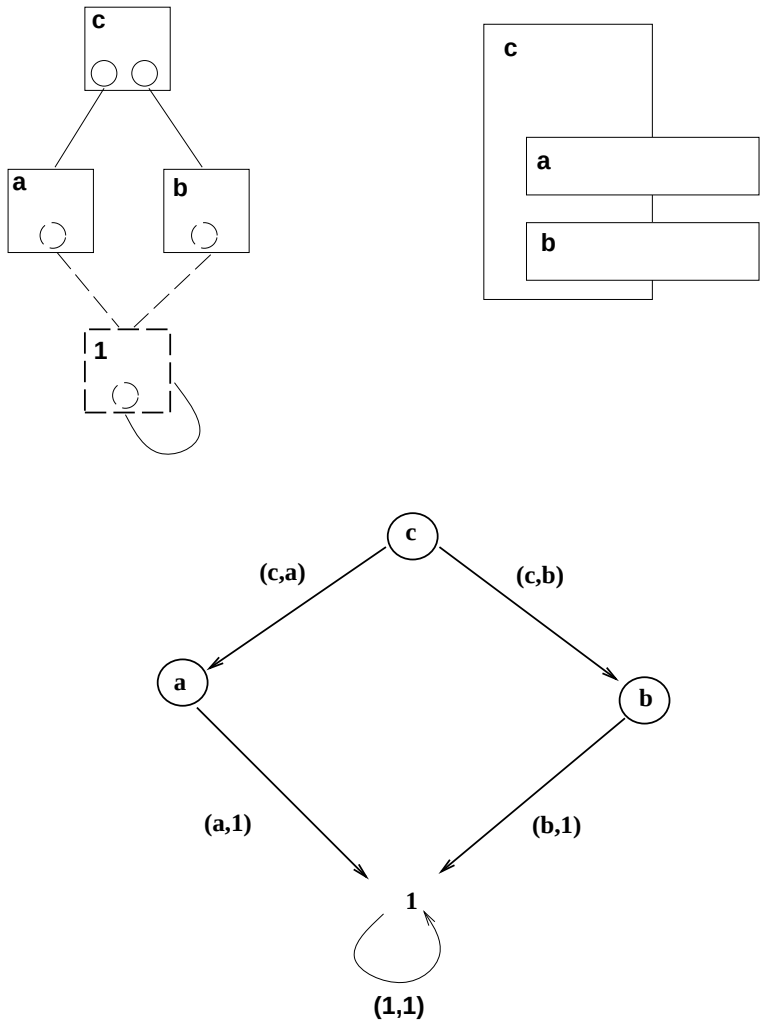



Figure 3.2: An edge graph represented by a conventional graph

encoding the edge graph $G$. The mathematical description of the conventional graph is as follows:

$$G' = (V, E) \text{ where}$$
$$V = \{a, b, c, 1\}$$
$$E = \{(a, 1), (b, 1), (c, a), (c, b), (1, 1)\}$$

▲

In such a representation, a founded edge graph is nothing more than a directed acyclic graph (with a sink[12] if we have the empty arrow having itself as its only adjacent). We note however that the nodes and edges in the conventional graph

[12]the empty arrow **1** is the sink, i.e. has a loop.

model will not correspond directly to a (semantic) concept in which we commonly reason about information systems; In particular they do not directly denote a type or object in a traditional object oriented diagram. This flaw will make it necessary to use concepts in the semantics, corresponding to these nodes and directed edges, which are not common in information systems. The nice thing about the edge graphs, is that the syntactic primitives here (complex edges) *do* have a natural semantic interpretation in terms of common concepts of information systems[13]. We will take advantage of this phenomenon in the following chapters.

As an aside we want to remark that for the syntax of the categorial graph language we have no problems with different graphical representations of the same graph, because we have taken the mathematical notion of edge graph as a syntactic entity. In fact we choose the mathematical structure of an edge graph as a syntactic entity with the intention that we then will not have to worry about trivial identity; i.e. when in many other computer languages one has language expressions that very clearly denote the same thing, in the edge graph language one has actually either the same syntactic expression, or else there are important structural differences in the expressions, and these differences are not trivially amounting to the same thing. By considering the mathematical structure of the edge

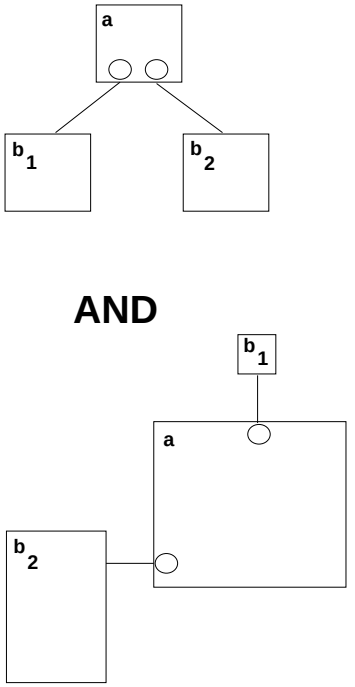
$$< a, \{(a, b_1), (a, b_2)\} >$$

we avoided the problem of identifying this syntactic structure with the structure

$$< a, \{(a, b_2), (a, b_1)\} >$$

where the order of the symbols differs, or even

[13]Using the terminology of information systems: The complex edges are *first class citizens* of the system.




which have different geometric properties. These denotations are all the same syntactic structure in our language, so we do not have to worry whether or not they are interpreted (semantics!) as the same object.

### 3.2.2 Operations

Above we defined edge graphs, the basic syntactic entities of the language(s) of categorial graphs. As in most syntactic theories there are some basic *operations* or *constructions* defined on the syntactic entities. For example the most basic and common of these operations in textual languages is the *concatenating* of two strings, forming a new string. In traditional programming languages even much more complex operations are common, for example in procedural programming languages there normally exists an 'if- then-else-operator' that takes a Boolean expression[14] and two statements[15] and returns a statement. Complexity of these formalisms can reach even to undecidable systems like the two level grammars of ALGOL 68 ([WijngaardenEtAlii76]). The point we want to make with the above examples is that a syntactic theory can be quite complex, and that the operations we will introduce below for the syntactic entities of the categorial graph language are quite modest with respect to their complexity.

We already saw one operation for edges: the forming of an edge graph (see definitions 3.2.3, 3.2.4 and 3.2.5). Given a token and a set (multiset) [list] of edge denotations, it produces an edge.

[14] A Boolean expression is a syntactic category.
[15] A statement is a syntactic category.

One important operation used in modeling the world is *taking objects together*. Such an operation is used to tie objects together and to look at the tied-up objects as one player in the model. Both in syntax and in semantics we will present an aggregation operation that can be used for this purpose. For example one can think of a 'Pilot' object, having 'a number of tasks' as its roster (see example 3.1.1).

**3.2.9.** DEFINITION. (Aggregation) Consider a universe of edges **Edge**. Given two edges $a$ and $b$ then $a \cdot b$ denote the aggregation of $a$ and $b$. Given two sets of edges $A$, $B$, then $A \cdot B$ denotes all the products $a \cdot b$ where $a \in A$ and $b \in B$.

Now let $\mathbf{Edge}^*$ denote the set of all products of edges and products of products of edges etc. etc. (strings over **Edge**); i.e.

$$\mathbf{Edge}^* = \mathbf{Edge} \cup \mathbf{Edge} \cdot \mathbf{Edge}^*$$

Now we will define all products in $\mathbf{Edge}^*$ being edges as follows:

| | |
|---|---|
| set edges: | $\mathrm{Adj}(a \cdot b) := \mathrm{Adj}(a) \cup \mathrm{Adj}(b)$ |
| undirected edges: | $\mathrm{Adj}(a \cdot b) := \mathrm{Adj}(a) \ddot{\cup} \mathrm{Adj}(b)$ |
| directed edges: | $\mathrm{Adj}(a \cdot b) := \mathsf{append}(\mathrm{Adj}(a), \mathrm{Adj}(b))$ |

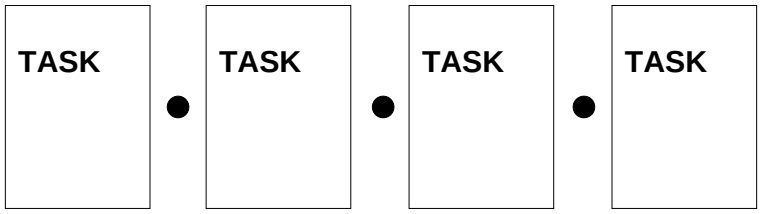
Note that $\cup$ is the set-union (for set edges), and $\ddot{\cup}$ is multiset-union (for undirected edges), while append is list-union (for directed edges).

For the edges in $\mathbf{Edge}^*$ we will have to make some non-trivial identifications:

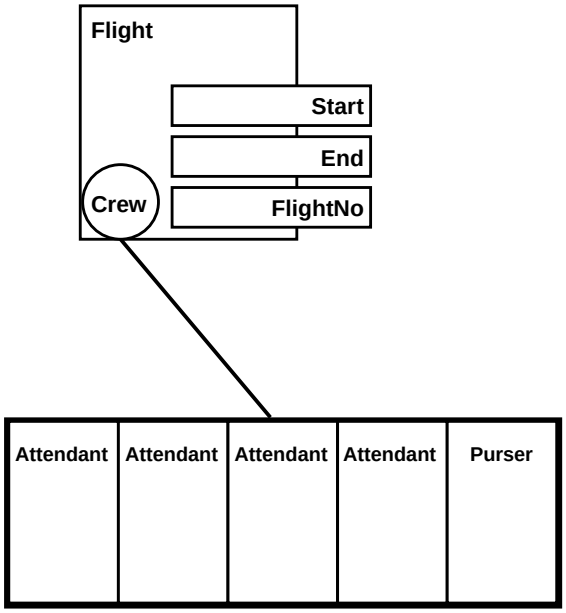
- concatenation for set edges must obey the axioms of set union;
- concatenation for undirected edges must obey the axioms of multiset union[15];
- concatenation for directed edges must obey the axioms of list union[15].

△

**3.2.10.** EXAMPLE. In the running example the roster of a pilot is modeled as an aggregation of tasks. The roster is an ordered sequence of tasks:




An other example of aggregation would be the model a finite set. Suppose you have an object 'cabin-crew' that is an attribute of a object 'flight' and models a set of stewards or stewardesses (let's say they are modeled by 'cabin-personnel' objects having attributes like name, age etc.). Then this object 'cabin crew' is an aggregation of a couple of 'cabin-personnel' objects. In a picture:




The attributes of the aggregated objects like 'roster' or 'cabin-crew' will be the attributes of the individual parts of the aggregation. For example a roster will have several arrival station attributes, each belonging to another flight. Also a cabin-crew would have several name-attributes, one for each cabin personnel in the aggregation, and similarly, several 'age' attributes. One could think of having operation on these attributes like SUM, AVERAGE and COUNT resulting in attributes of the aggregation object ('roster' or 'cabin-crew'). These well known operations are called aggregates in relational database languages like SQL. ▲

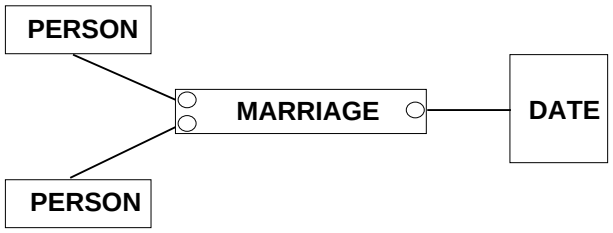
To avoid confusion we note that in this section we talk about *static* operations, in the sense that we talk about operations on the structures that we will interpret in a static semantic model. Objects may have both attributes *and* abilities. Although abilities have a dynamic content, dynamics itself is not covered in this thesis. The model represents an information system instance, without a past or a future. In the paper [Haas01] we embryonically discuss dynamics of information systems based on the language presented here. There we look at models that have, in a formal sense, a past and a future. The subject matter of dynamic operation is orthogonal on the matter of static operations as discussed above.

### 3.2.3 Types, Objects and Constraints

We will us edge graphs to denote a signature that specifies a *type*[16]. For example we can denote with an edge graph that an object of the type MARRIAGE involves

[16]We will give a precise definition of the notion of *type* when we present the semantics of our language. Here it will remain a 'vague', but well known semantic concept that will enable us to talk about the language without having to stay totally abstract

two objects of the type PERSON and one object of the type DATE; i.e.




In object oriented modeling and database languages we most often talk about types of objects; i.e. the most commonly used modeling documents are database class diagrams. However we also need to talk about instances. For example the modeling language UML has a syntax for describing instances which are used in several diagramming techniques such as sequence diagrams and collaboration diagrams ([FowlerScott00] note that these diagrams also cover operational aspects that are not covered by the syntax described here). The language to talk about instances is very strong related to the language that talks about objects. Even more strictly, one can not talk meaningfully about instances if one does not have a notion of the types of these instances, because we need to know what kind of objects we are talking about (described by the types) before we can interpret the denoted instance.

For example, an object or instance of the type MARRIAGE should typically be of the same 'structure' as the edge for the type MARRIAGE itself; i.e. in mathematical terms there should exist a homomorphism between the object and the edge denoting the MARRIAGE type. We can fairly write down the structure of such an object using, again, an edge graph[17]; i.e.


The homomorphism between the edge graph of objects (from now onwards called *object graph*) and the edge graph denoting types (from now onwards called *type graph*) will be called a *typing functor*; i.e. it assigns a type in the type graph to the edges in the object graph. In the above example the typing homomorphism assigns the upper PERSON edge to the AL edge, the other PERSON edge to the PEG edge, the MARRIAGE edge to an actual marriage act, and finally the DATE edge to August 21 1967. We will consider an edge graph with a typing functor

[17]Note that this is not uncommon, see for example IFO of Abiteboul [AbiteboulHull87]

as a basic building block of the information in an information system; i.e. basic entities of an instance of an information system are *typed complex objects*. We can also use the typing functor to validate a typed object graph. If the typing functor is not a homomorphism to a given type graph, then we will say that the typed object graph is *syntactically not valid* with respect to that type graph.

Note that the restriction we put on the typing functor (i.e. being a homomorphism, or in other words structure preserving) is a syntactical restriction on the denotation of an instance of an information system. The purpose of this restriction is that if we have a schema and a typed instance, we are certain that all the things we wrote down in our instance can be talked about using the definitions in our schema[18].

As an aside we note that this typing homomorphism is in a sense an extension to traditional syntactic systems, that do not have complex objects but only atomic ones and aggregates. For example suppose you have a type graph modeling an English sentence as follows:

**NOUN · VERB**

Then **"John · walks"** would be a syntactic proper instance of the graph modeling a sentence if the typing functor would map "John" to the type "NOUN" and "walks" to the type "VERB". On the other hand the sentence **"John · loves · Mary"** does not have a sensible typing function to the above type graph. In order to construct a homomorphic typing function we have several possibilities:

1. We can map **"John"** to **"NOUN"** and **"loves ·Mary"** to **"VERB"**,
2. or alternatively **"John · loves"** to **"NOUN"** and **"Mary"** to **"VERB"**
3. or even map **"John · loves · Mary"** to **"NOUN"** and the empty word **""** to **"VERB"**,

[18]In other words this means that if we type an object in some instance we should assign to it the initial (most specific) type, because only then can we speak about all the information of the object we wrote down in the instance. If we would have typed an object with a less specific type (i.e. a supertype of the specific type), we will have no means to talk about the object in the way we wrote it down (i.e. as being of a more specific subtype of the type assigned to it). The object then would have adjacents we cannot classify being of that object. If an object is typed properly, then when we interpret the object (semantics!) we can, of course, infer the object being of this less specific type. But this is on the semantic level. On the syntactic level we are only interested in writing things down properly such that when we interpret the things written down, we get a proper meaning for the things written down. In other words the syntactic rules should prevent us writing down things we cannot give a proper meaning to. Chomski said: Colorless green dreams sleep furiously.

4. or map the empty word **""** to **"NOUN"** and **"John · loves · Mary"** to **"VERB"**.

5. More peculiarly we can map **"John"** to **"NOUN"** and **"loves"** to **"VERB"** and **"Mary"** to the empty type "**1**"

None of these alternatives are sensible typing functions for the object graph **"John · loves · Mary"**. The first 4 examples clearly give undesirable types to the objects (it is especially undesirable that an empty word is a NOUN or a VERB). The last example, though, needs some more attention. The last example shows the strength of the empty edge as a type. In the last example the objects "John" and "loves" have a proper type and object "Mary" is postulated to be of the empty type. In a sense this means that we *ignore* the object "Mary" in our analysis, using the 'datamodel' $\mathbf{NOUN} \cdot \mathbf{VERB}$. The empty type enables one to write syntactically correct type graphs and object graphs without the necessity to be fully specific in our analysis. In other words we give the data analyst a chance to (syntactically correct) write down data models and instances during the process of his modeling task, when he does not know the full complexity of the object in the universe of discourse he is modeling. To continue with the above example, only when the analyst has gained the insight that an English sentence could alternatively be of type $\mathbf{NOUN} \cdot \mathbf{VERB} \cdot \mathbf{NOUN}$, he can fully specific type the instance **"John · loves · Mary"**.

Note that for graphs with complex objects, the empty object plays the same role as the empty or 'unspecified' type. A trivial example for this is a type graph consisting of one atomic edge $A$ and an object graph with an object $a$ and an adjacent $b$ that in turn has an adjacent $c$; this object graph can be typed as follows: $a : A, b : 1, c : 1$.

Let us consider two disjoint universes, $\mathcal{T}$ and $\mathcal{O}$; and let us reserve $\mathbf{Edge}(\mathcal{T})$ for denoting edge graphs for types and $\mathbf{Edge}(\mathcal{O})$ for denoting edge graphs for information objects. An edge graph $G$ over $\mathcal{T}$ (i.e. a type graph), determines a set containing all typed edge graphs over $\mathcal{O}$ (i.e. object graphs) $H$ for which the typing functor is a homomorphism from $H$ to $G$. These typed object graphs are members of this set because they have the signature defined by the type graph $G$. In information systems we often want to classify objects, not only by their signature, but also by some other properties concerning the whole graph. In information systems these properties are usually called *constraints*. Examples of constraints in information systems are constraints that are common in relational databases like functional dependencies, join dependencies, inclusion dependencies, primary keys, foreign keys, or constraints that are common in object oriented databases like subtyping constraints, etc.. We want in our language for information systems the ability to express a restriction on a set of object graphs that have the 'proper' signature, that is satisfied by those object graphs of this

signature that also satisfy some additional *constraints*[19]. The *language of categorial graphs* we present in the next section will be able to talk about objects by drawing a type edge graph and adding textual constraints. These categorial graphs will denote data or information models. Instances of these data models can be denoted by object edge graphs.

**3.2.11.** EXAMPLE. Let us again take the example of a roster of a pilot (see example 3.1.1). All the entries in the 'roster' object[20], i.e. the flights, the time-offs, and the courses, will certainly have things in common, because they are all entries in a roster. Suppose we want to model that by saying that these three kinds of objects are things we call 'tasks'. The attributes of a task are typically the attributes that flights, time-offs and courses have in common, for example start-date-and-time and end-date-and-time. Furthermore, we can say now that a roster is an aggregation of tasks, and that the types 'flight', 'time-off' and 'course' are subtypes of the type 'task'. Moreover, we can force that all tasks in a roster do not overlap in date-and-time. To assert that some type is a subtype of another type, e.g. 'flight' is a subtype of 'task', amounts to putting a constraint on the models that have objects of these types. It says that all objects of type 'flight' should also be objects of type 'task'[21]. And, obviously, saying that tasks in a roster should not overlap is evidently a constraint on the models that have objects of type 'roster'. ▲

Note that in many formalisms, among which UML is one, the subtype constraint is drawn graphically with an arrow. As subtyping is a matter of constraint and not a matter of signature (structure) we choose to keep it a textual constraint in this formal presentation of the language of categorial graphs, so it will not cause any confusion with the edges that denote matters of signature.

We can draw categorial graphs in many ways. In fact, because we take a very abstract point of view, there are many existing formalisms that might be seen as a drawing of categorial graphs. We want the categorial graphs to be general enough to capture most object oriented information system formalisms. Because the categorial graph language will be given a proper semantics, it provides a vehicle to give a proper semantics to these formalisms

[19]Note that we are talking about *static* constraints; i.e. properties that should be satisfied regardless of the past or future of the models (if you let a model be dynamic of course). We will consider *dynamic* constraints when we look at dynamic models of categorial graphs.

[20]i.e. an object of type 'roster'.

[21]Note that for the *syntax* it suffices for a typing function to map an edge that will be interpreted as an object to only *one* other edge that will be interpreted as a type. In the semantics however an object of type 'flight' should be both of type 'flight' and type 'task'.

### 3.2.4 Categorial Graphs

A categorial graph will be the syntactic vehicle to define a database schema. A categorial graph consists of en edge graph and a set of phrases. Each edge in categorial graph will represent a *type*[22]. The structure of the edge will determine the *signature* of the type it represents. The phrases will define *constraints* on the types.

**3.2.12.** EXAMPLE. Consider figure 3.3. It contains an edge graph $G$. The figure shows that $G$ consists of the basic edges[23] HUSBAND, WIFE, NAME, SEX and YEAR, four complex edges: one edge MARRIAGE with adjacents HUSBAND,WIFE and DATE, one complex edge BIRTH with adjacents MARRIAGE DATE and PERSON, one complex edge PERSON with adjacents NAME and SEX, and finally one complex edge DATE with adjacent YEAR. Note that we have also written down some constraints:

- a HUSBAND ISA PERSON
- a WIFE ISA PERSON
- the year in the DATE of a MARRIAGE should be after 1848
- the date of a MARRIAGE that is accounted in a BIRTH should be before the DATE of a BIRTH

It is easy to see that the typed object edge graph $H$ of figure 3.3 is an instance[24] of $G$. ▲

**3.2.13.** DEFINITION. (constraint phrases) Given a universe of edges $\mathbf{Edge}(\mathcal{T})$ and an edge graph $G$ over this universe; let $\mathsf{Prop}$ be the set of propositional variables ranging over $\mathbf{Edge}(\mathcal{T})$, and let $\mathsf{Prop}(G) \subseteq \mathsf{Prop}$ be the set of propositional variables ranging over the edges in $G$. Let $\mathsf{Con}$ be a set of constants. Furthermore let $\mathsf{Op}^i$ be a finite set of operators of arity $i$. Let also $\mathsf{Fun}$ be a countable set of function symbols. Then the language $L$ of constraint phrases is defined as follows:

$$L = \begin{array}{ll} P| & \text{(propositional variables)} \\ \mathsf{Con}| & \text{(constants)} \\ \mathsf{Op}^1(L)|L\mathsf{Op}^2L|\mathsf{Op}^3(L,L,L)|\ldots & \text{(operations)} \\ \mathsf{Fun}^0|\mathsf{Fun}^1(L)|\mathsf{Fun}^2(L,L,L)|\ldots & \text{(funstions)} \end{array}$$

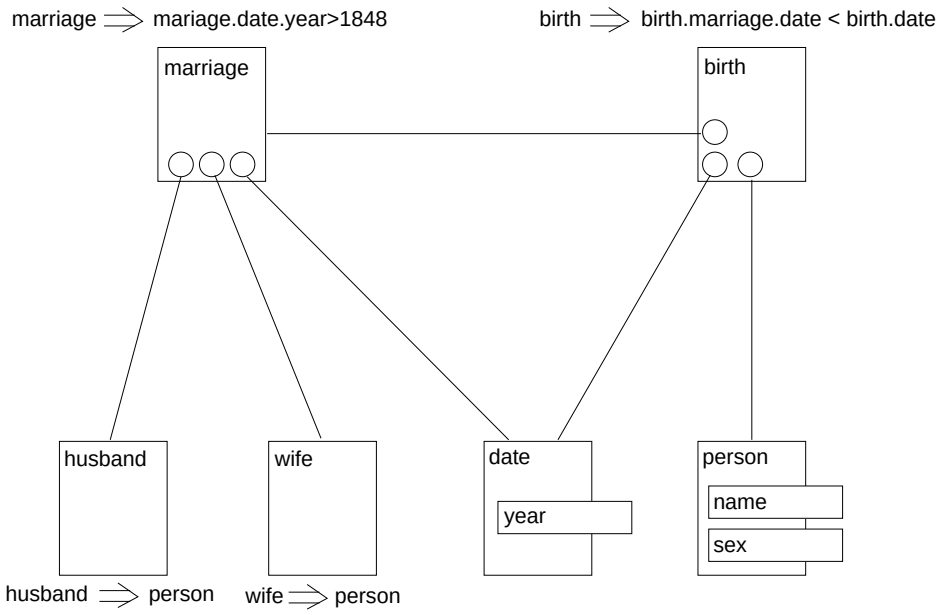
The set $phrases_L(G)$ will be defined as the set of all phrases of the form $L \Rightarrow L$ for which all propositional variables occur in $G$. The intended meaning for a phrase $A \Rightarrow B$ is "all objects satisfying $A$ should satisfy $B$". △

[22]Semantics will be given in the next section

[23]We consider an edge to be simple (opposed to complex), if it has only the empty arrow $\mathbf{1}$ as adjacent. We abbreviate its denotation by omitting its adjacents because $\{1\} = \emptyset$

[24]In the picture of $H$ we wrote the types after the denotation of the object, e.g. we wrote i:marriage for an object i that is typed to be a marriage

*A type graph with constraints*




*and an instance*

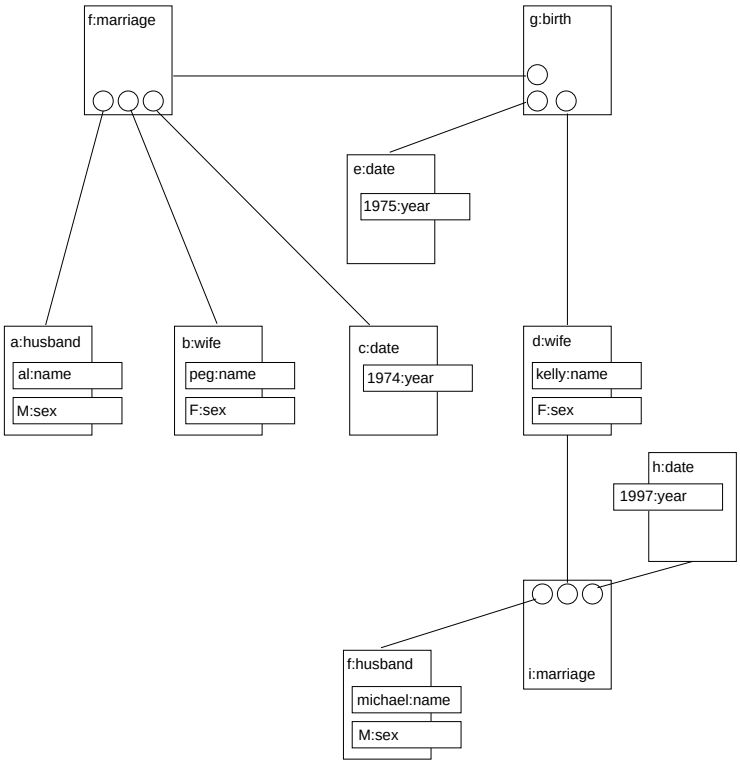



Figure 3.3: Example of a categorial graph: Married with children

We defined very generically the syntax of the phrases (i.e. the constraint language), because there are many possibilities, flavors, operators, constants, functions etc. to choose from. For each different feature one wants to be able to talk about, one needs a hook in the language that enables one to talk about it. Below we will introduce a number of basic constants and operators that are important for expressing constraints.

**3.2.14.** Definition. (constraint operators) Constants for talking about structure:

**1** (representing the empty edge)

Unary operators for talking about structure

$\Diamond$ (for talking about adjacency; $\Diamond A$ means "having an adjacent of type $A$")

Binary operators for talking about structure:

$*$ (for aggregation; $A * B$ means "being an aggregate of two objects, one of type $A$ and one of type $B$")

Constants for reasoning:

$\bot$ (representing falsehood)
$\top$ (representing truth)

Unary operators for reasoning:

$\neg$ (for negation: $\neg A$ means "being not of type $A$")

Binary operators for reasoning:

$\sqcap$ (for conjunction: $A \sqcap B$ means "being both of type $A$ and of type $B$")
$\sqcup$ (for disjunction: $A \sqcup B$ means "being either of type $A$ or of type $B$")

Constants to talk about identity

self (representing the object itself; i.e. to talk about the object in whose scope we write down the constraint)

$\triangle$

We also have function symbols in the constraint language. The function symbols enable one to constrain objects within the domain of the type of the object. The most common examples of such functions are arithmetic functions for objects that are natural numbers, or for objects that are strings one can have string operations like alphabetic-ordering or pattern-matching[25].

[25] comparable with the SQL operator 'like ¡pattern¿'

**3.2.15.** DEFINITION. (categorial graph) Given a universe of edges **Edge** and a language $L$ as above. A *categorial graph* is a pair $(G, S)$ where

- $G$ is an edge graph over **Edge**,
- $S \subset phrases_L(G)$ is a set of phrases over the tokens of $G$

△

**3.2.16.** EXAMPLE. Consider again the type graph with constraints of figure 3.3 above. The constraints ornamenting the graph can be formulated as follows in the constraint language:

$\mathsf{husband} \Rightarrow \mathsf{person}$
$\mathsf{wife} \Rightarrow \mathsf{person}$
$\mathsf{marriage} \Rightarrow (\Diamond \mathsf{date} \sqcap \Diamond f_{>1848}(\mathsf{year}))$
$\mathsf{birth} \Rightarrow f_{\text{before-in-time}}(f_{\text{get-date}}(f_{\text{get-marriage}}(\mathsf{self})) \ , \ f_{\text{get-date}}(\mathsf{self}))$

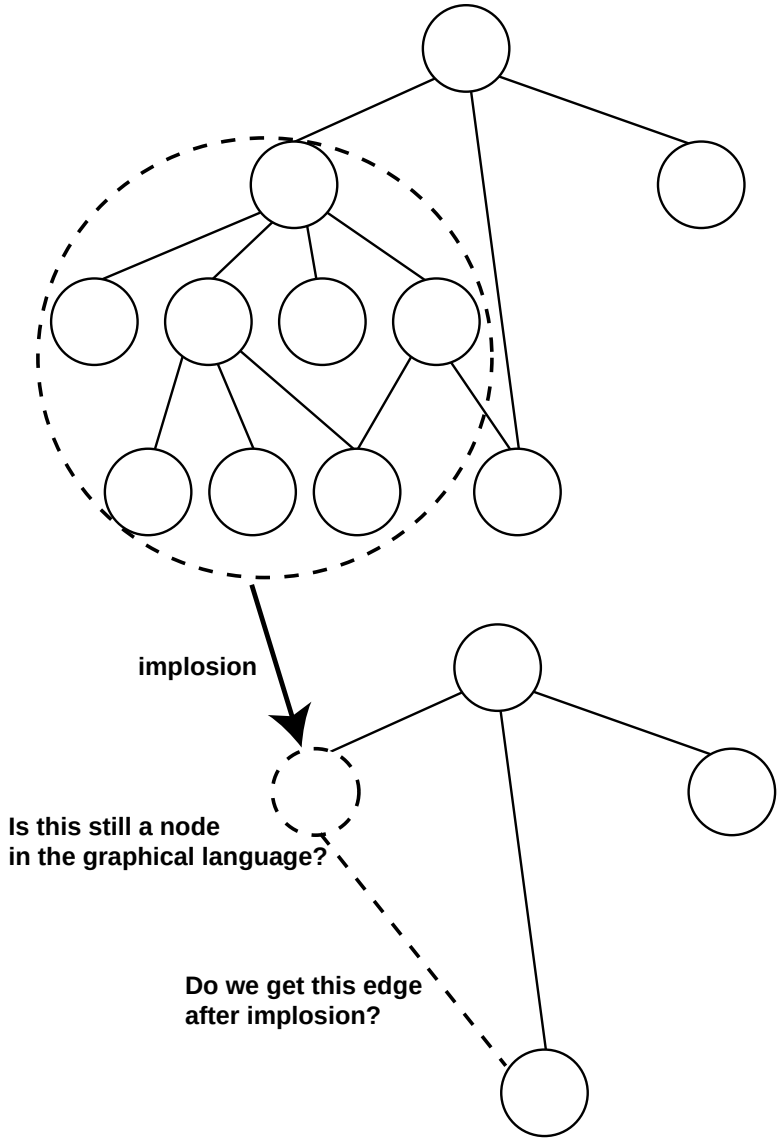
▲

## 3.2.5 Imploding and Exploding of categorial graphs

Categorial graphs are structures that have both a graphical and a textual part. The edge graphs are mathematical structures that can be classified as graphical, because they can be represented conveniently with 'pictures' (graphs). We mentioned this before. The advantages of graphical representation of information is stressed a lot of times in many fields of science, especially in the field of artificial intelligence (psychology and graphical knowledge representation) ([Dastani98]). To (bluntly) summarize these advantages it has often been stressed that a picture says more than a thousand words. There exists a limit, though, in using pictures: if the amount of graphical elements in the picture becomes very large, the meaning of the picture becomes incomprehensible[26]. This phenomenon is thoroughly observed and studied ([Adriaans90]). From a practical point of view it forces graphical syntaxes to provide some mechanism for reducing the number of graphical objects in a picture. In this area also there are some proposals. Most of these proposals, however, are very ad hoc, and imprecise in defining the meaning of the syntactic elements of a reduced picture. Evidently this is problematic. For example, suppose you have a graphical language in which the basic syntactic elements are nodes and edges[27]. Let us assume that reducing the number of syntactic elements in a picture (text) amounts to imploding a subgraph into a new node. The problem now arises to define a proper meaning for this new node of the imploded graph. This is not trivial, even though we already have a

[26] A picture with a thousand objects says totally nothing.
[27] i.e. conventional graphs.

proper meaning for normal nodes and edges. These new nodes that represent an (imploded) subgraph can hide very complex structure, which becomes apparent only if we explode them. For example the connectivity to the context is lost in the following implosion of a traditional graph:

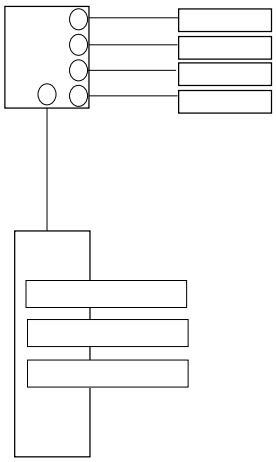



In edge graphs this matter is solved in a natural manner. Instead of coding the complex structure of an object in atomic graphical elements, each graphical element in an edge graph (i.e. each edge) can be of arbitrary complexity. This means that if we draw an edge like this:

□

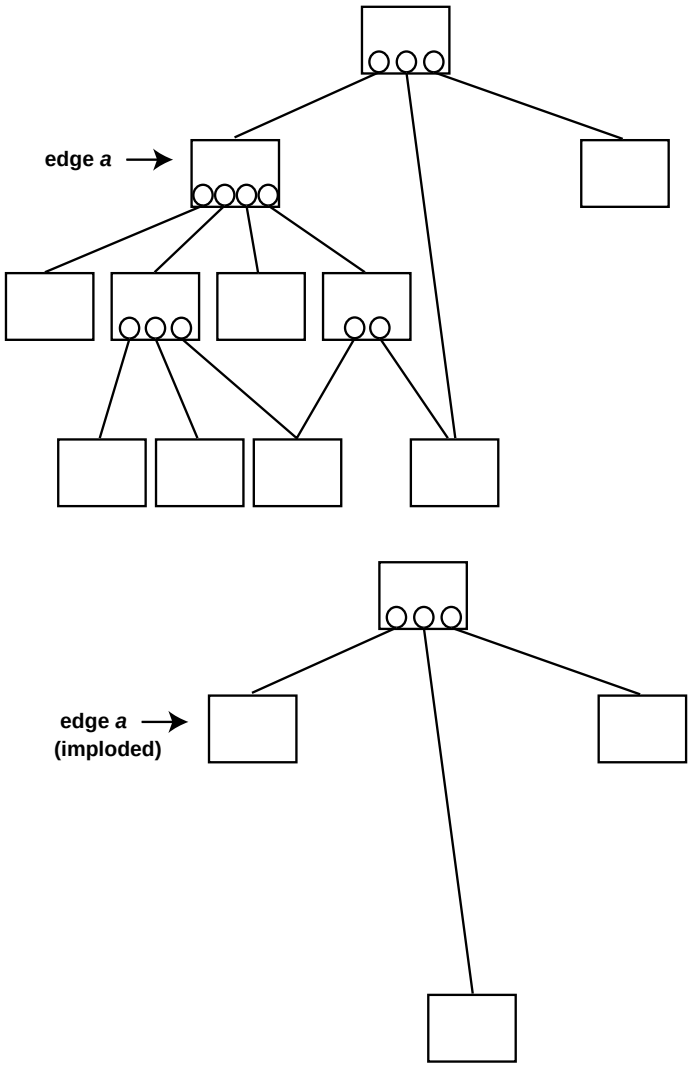
it can be the picture of a very simple edge (for example the empty edge **1**) or a very complex one. The denotation of an edge contains its *whole* complex structure.

We may draw it as one simple edge and still interpret it as the complex edge it is. The reason is that we do not encode a complex structure using atomic ones, but give a complex edge the same status as a simple one. They are both edges. We may also choose to reveal something of its complexity by drawing some of its structure, i.e.



All in all we can implode an edge as one syntactic element, and explode it again revealing all of its structure, without running into problems of interpretation. Actually the reason for this freedom is inherent in the difference between the mathematical nature of an edge graph (which we view as syntax) and a drawing of it in a two dimensional space (a denotation we see with our eyes). Just like you can draw a conventional graph in arbitrary many ways, you can vary the drawing of an edge graph. The holistic nature of an edge[28] then enables one to abbreviate the drawing of an edge. In the following picture the effect of the implosion is perfectly clear (objects just hide their structure); i.e.

[28] An edge contains its own structure.




A little more realistically, recall the pilot example of example 3.1.1. For instance, if one would display the full roster of a pilot, and take along all its details like the flight concerned with the task, the plane that will be flown in this flight, the cabin crew that will be on the particular flight together with their hobbies etc. (note that in reality a task in a roster carries a lot of complex objects as adjacents), the view of the roster would be seriously obscured. Note that in the 'real' world the information of complex objects like a roster is enormous. For example the roster information of the cabin crew of a real airline is typically stored in more then 40 different tables.

## 3.3 Summary

In this chapter we presented a language for object oriented information systems. This language has both graphical and textual ingredients and a formal syntactic

theory. In the next chapter we will give a proper mathematical semantics to this language. This semantics will reveal the meaning of the constructs in a conceptual object oriented world.

# Chapter 4

# A semantics for object oriented information systems

*Semantics is a strange kind of applied mathematics; it seeks profound definitions rather than difficult theorems. The mathematical concepts which are relevant are immediately relevant. Without any long chains of reasoning, the application of such concepts directly reveals regularity in linguistic behavior, and strengthens and objectifies our intuitions of simplicity and uniformity.*

*(J.C. Reynolds [Reynolds80])*

The categorial graph language is a language for object oriented information systems for which we claim that the concepts it expresses are those concepts used in practical languages for object oriented information systems. The theory for the language of categorial graphs aims to show that these concepts are materialisable (in mathematics), and are furthermore so in a direct manner (i.e. without encoding). In this chapter we will present the semantics of the family of categorial-graph languages. The semantics will be constructed in the following manner:

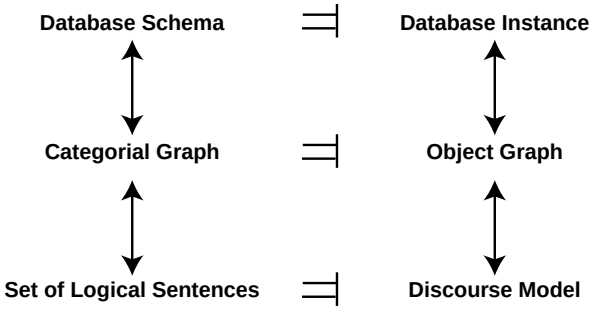
We will translate a language of categorial graphs into a logical meta language. This meta language will itself have (again) a semantics and an inference system to reason about the semantic domain. The semantic domain consists of instances of object oriented information systems. These instances contain complex objects and partial descriptions of objects. The three ingredients: the meta language, the inference system, and the semantics, form a *logic*.

To interpret a categorial graph, the graph and the constraint phrases will be translated into a collection of phrases in the logical meta language. These phrases will constitute a theory in the given logic. Valid models in this theory will be models of valid information system instances of the given categorial graph.

In order to make this scheme work properly, we provide a meta language that inhabits language constructs that are (we are using another vague term here) 'very close' to the language constructs we want to interpret. This means that we provide a meta language that contains high level constructs that reflect directly the primitive structures in the categorial graph language. This way, we have a semantics in which the concepts that are expressed with the categorial graph are interpreted using primitive constructs of the semantics, and not by encoding in low level mathematical concepts.

Note that the distinction between signature and constraints becomes more vague if we turn to the meta language. If we translate the graphical (signatorial) syntax and the textual (constraint language) syntax to the logical language, we have only *one* textual logical language. Still we pursue the proposed distinction between signature and constraints by designing the logical language in such a way that the graphical ingredients of the object language are *intrinsic* features of the meta-language. We achieve this by the 'direct' translation of the graphical constructs in its logical constructs.

Summarizing, our semantics for categorial graphs looks as follows: We will provide a meta language in the form of a logical language to interpret the categorial graphs. A categorial graph, then, will be interpreted by a set of logical sentences in the meta language. All models of the theory of these logical sentences are object oriented information systems that satisfy the description of the categorial graph. In a picture:




## 4.1 Desiderata for meta language for categorial graphs

The definition of the family categorial graphs languages presents us with a list of concepts that we need to be able to express in the meta language in order to be able to translate a categorial graph into the meta language. These concepts are coded in the graphical constructs of the categorial graph language, and we now need to make them explicit to mould them into a logical form.

In the view of the categorial graph language information is present in the form of *objects*. In principle everything you model is an object. An object has an identity, essential properties -i.e. an object *is* of some type- and an object has some aspects -i.e. an object *has* some properties-. Objects and aspects of an object can appear with structure. For example one can say 'an object **has** *two* aspects of a certain type'. Moreover one can take objects together. For example one can say 'this object **is** the aggregation of two objects of some type'. Finally one can express complex constraints on the structure or content of an object.

In the categorial graph language we talk about the objects with graphical entities -edges- that denote *types*. As we saw in the previous chapter, a type denotes an essential property of an object that is assigned to that type by the typing functor (or inherits from that type because it is assigned to a specialization of that type).

Recall that aspects of an object are denoted by the adjacents of a category. An adjacent of a category types a property of an object of that category. In the object model (the instance) the adjacent object is a property of the object itself. The adjacents form a structure. If one wants to talk solely of 'having a certain kind of adjacent', a set structure is suited to express the adjacents of a category. If one wants to talk about a certain number of adjacents of some kind, the adjacency structure needs to be able to count. A multiset structure can denote this. If one wants to talk about the first adjacent and the second adjacent, one needs a list adjacency structure to express such structure.

The language of categorial graphs also provides an operation to take categories together. Aggregation of two categories delivers a category that is uniquely determined by its components.

Complex constraints can be formulated in a constraint language. This constraint language could, for example, enable one to use Boolean constructs to logically combine properties, formulating a complex constraint.

Below we list the desirables for what we want to be able to say about complex objects.

1. talk about *essential properties* of objects
2. talk about *aspects* of objects, which are in the OO philosophy information objects in their own right
3. talk about complex constraints on the objects using Boolean connectives like conjunction ($\sqcap$), disjunction ($\sqcup$), negation ($\neg$) and implication ($\rightarrow$)

4. talk about *aggregations* of objects (structurally and resource consciously)

5. talk structurally and resource consciously about aspects of objects

We will use the elaborated arsenal of modern formal logic to express the concepts listed in the desiderata. The first desiderata means that we need to be able to state (at least) propositions on whole objects. The second states that we need to be able to assert propositions about the adjacents of an object. Propositions about aspects are expressed with modal propositions ($\Diamond P$ where $P$ is a proposition). The third desiderata -complex constraints- introduces the need for Boolean connectives in order to make complex assertions about objects and their aspects[1]. The 4th item -aggregation- introduces the need for a connective that is interpreted as taking together objects. This will be the $*$ ( we use the same symbol as the related resource conscious conjunction of linear logic). The last item in our list requires an aggregation operation in a modal context. Although the items by them selves seem to introduce clear ingredients to the language, the combination of all the desiderata will appear to be problematic when we want to build a logic for the language with all the desired ingredients.

Note that when we look at the discussion in the previous chapters, we have even more desiderata. These are things we want to be able to express, but are not bound to language constructs. The most important of these are : labels, non-wellfoundedness and incomplete specification of objects (i.e. not all aspects known and/or the aspect structure not known). These are not desiderata that influence the language constructs, but are inherent to the interpretation of the logical connectives we use. This will become clear when we present the interpretation of the meta language.

In talking about the adjacency structure, we add the following remark. The languages we will define will have an interpretation in a semantic domain populated by complex objects. When issues like structural properties become important (due to items 4 and 5), the complex objects in the semantic domain need to have structural properties as well. This means that when we can say things about objects taken together, such aggregation needs to be defined on the objects in the semantic domain. We note that there are several variants on the result of taking objects together, which can all be accounted for in the structural rules for the logic that talks about the variant in focus. This will be elaborated when we present the logical calculus and the interpretation of the logic for categorial graphs below.

[1]Note that we omitted the self operator here. Although very powerful, and important in the broad OO context, the self operator is not an intrinsic part of the core system. We will see that we can add it nicely to the core system in a logical context. We will spend some words on that when discussing the logical aspects of our system.

Recall the enumeration of desiderata (1-5). We summarize the possibilities[2] we will consider in the table below.

| | **1** | **2** | **3** | **4** | **5** | **comments** |
|---|---|---|---|---|---|---|
| (i) | X | | | | | atomic typing of essential properties only |
| (ii) | | X | | | | atomic typing of aspects only |
| (iii) | X | X | | | | atomic typing of essential properties and aspects |
| (iv) | X | | X | | | Propositional logic for essential properties |
| (v) | | X | X | | | Propositional logic for aspects |
| (vi) | X | | | X | | atomic typing of essential properties only and possibility to say something about aggregate whole objects (structure) |
| (vii) | | X | | X | | atomic typing of aspects only and possibility to say something about aggregated whole objects (structure) |
| (viii) | | X | | | X | atomic typing of aspects only and possibility to say something about the adjacency structure |
| (ix) | X | X | | X | | atomic typing of essential properties and aspects and possibility to say something about aggregated whole objects (structure) |
| (x) | X | X | | | X | atomic typing of essential properties and aspects and possibility to say something about the adjacency structure |
| (xi) | X | X | X | | | propositional modal logic for essential properties and aspects |
| (xii) | X | | X | X | | propositional logic with structural (aggregation ,product) connective |
| (xiii) | | X | X | | X | propositional logic for aspects with structural connective (aggregation ,product) |
| (xiv) | X | X | X | X | | Propositional modal logic for essential properties and aspects combined with a structural (aggregation, product) connective for taking together essential properties |
| (xv) | X | X | X | | X | Propositional modal logic for essential properties and aspects combined with a structural (aggregation, product) connective for specifying structure of the aspects of an object |
| (xvi) | X | X | X | X | X | Propositional modal logic for essential properties and aspects combined with a structural (aggregation, product) connective for taking together essential properties and a structural (aggregation, product) connective for specifying structure of the aspects of an object |

The languages without logical connectives (without 3) are actually simple type systems, for which there are hardly any rules (we only type data with it). Nev-

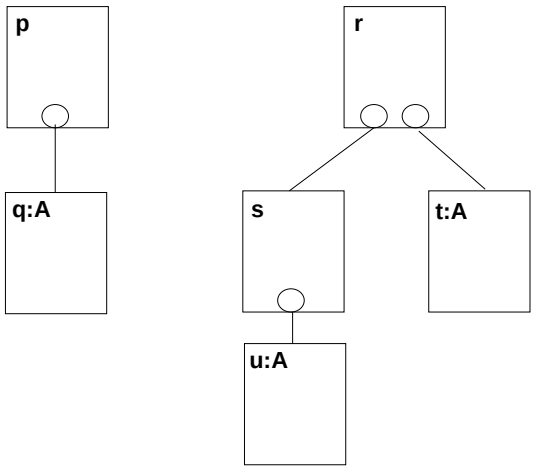
[2]Note that some combinations make no sense, like combining propositions on whole objects (essentials) with a product for adjacents (and nothing else).

ertheless they are widely used as subsets of the modeling languages. Many Data models are written with only the typing language. The calculus for these languages is nearly trivial, because we only need to have rules for the aggregation or product connectives, in order to let them behave as the aggregation does in the model. For example, if in the model we cannot count then the 'type' $A$ will be the same as $A * A$. We will go formally into these matters after we have defined the models.

For the languages with connectives things get complicated right away. Especially if we want to combine with these connectives the things we say about essential properties of objects with things we say about aspects of objects. Moreover we have the classical problem of using negation in an information model, where the description specified by $\neg A$ is satisfied by information we may not have in our information model.

In summary the meta language of categories has the following features:

- *Expresses propositions on the complex structure of an object using a modality for adjacency.* i.e. a proposition about the structure of an object that says that an object has some kind of adjacent is expressed with a modality. The modality will have an existential character, and will be denoted by the $\Diamond$. This means we can express that an object has some structural properties without knowing its structure totally. For example the complex objects[3] p, r and s will all be of type $\Diamond A$.

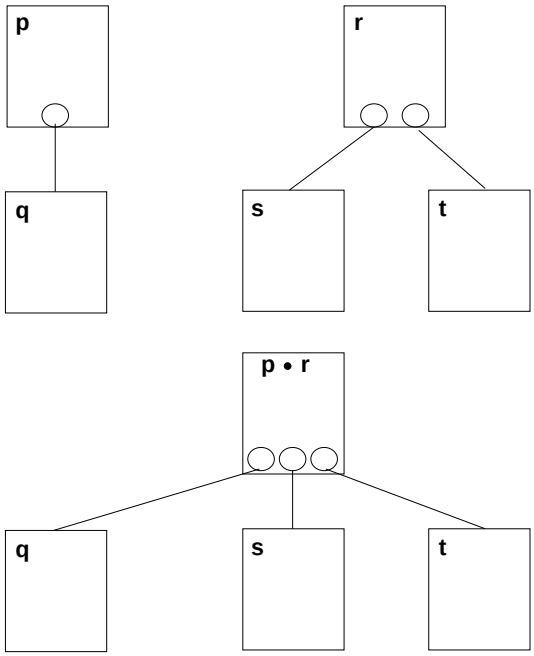



  If we want to talk about a directed adjacency structure, i.e. about the first adjacent, the second adjacent, etc. etc., we need to label the modal

[3] A note on the informal notation for an object: the structure of a complex object is denoted by a box using a circle as placeholder for an adjacent object and a line from the placeholder to the box that denotes the adjacent. This line is called a *link*. The label for the object will be a lower case letter $(a, b, c, \ldots)$, and the type of an object will be denoted by a capital letter $(A, B, C, \ldots)$. The label $a : A$ will mean 'object $a$ of type $A$'.

adjacency operator: $\Diamond^1, \Diamond^2, \Diamond^{\text{role}}$, etc.. Note that we may use symbolic names as labels, if we want.

- *Has a linear aggregation operation.* This means that we can express the 'taking together' of two objects resulting in a new complex object. For example if we have two objects $p$ and $r$, and $r$ has two adjacents $s$ and $t$, $p$ has an adjacent $q$, then their aggregation $p \cdot r$ will have three adjacents, $q$, $s$ and $t$. In a picture:




## 4.2 The meta language of categorial graphs

**4.2.1.** Definition. (The meta language of categorial graphs)
Let $S_{\mathsf{cat}}$ be a set of operators containing one unary modal operator $\Diamond$, together with the modal constant $\mathbf{1}$. We define the language $L_{\mathsf{cat}}$ to be the pair $(S_{\mathsf{cat}}, Q_{\mathsf{cat}})$, where $Q_{\mathsf{cat}}$ is a set of propositional variables. The set $\Phi(L_{\mathsf{cat}})$ of formulas in $L_{\mathsf{cat}}$ is defined as usual, using, next to the modal operators, the connectives $\_ * \_$, $\_ \sqcap \_$, $\_ \sqcup \_$, $\neg \_$ $\triangle$

The $\Diamond$ modality will talk about the adjacency relation $R$, and $\mathbf{1}$ models the empty (or unknown) objects. Furthermore, the * denotes aggregation, or in other words, the multiplicative or resource conscious conjunction. The connectives $\sqcap$ and $\sqcup$ are respectively additive (*non* resource conscious) conjunction and additive disjunction. Finally the $\neg$ denotes (*non* resource conscious) negation.

Let us give some informal interpretation of the connectives before getting formal. The $*$ (multiplicative conjunction) is the resource conscious 'and' connective. Informally, an object of type $A * B$ will be an object that is an aggregation of two objects, one of type $A$ and one of type $B$. The $\sqcap$ (additive conjunction) is the traditional 'and' ($\wedge$) connective. If an object is of type $A \sqcap B$ it informally means that the one object itself is both of type $A$ and of type $B$. The $\sqcup$ (additive disjunction) is similar to the disjunction ($\vee$) of classical logic. Informally, an object is of type $A \sqcup B$ if it is either of type $A$ or of type $B$. The $\neg$ (negation) is a non constructive and non-resource-conscious negation. An object is of type $\neg A$ if it is not of type $A$. The $\Diamond$ will model adjacency. An object of type $\Diamond A$ is an object that has an adjacent of type $A$. Finally $\mathbf{1}$ will be the type of the empty object. We will make sure the type $A * \mathbf{1}$ will have the same interpretation as the type $A$; in other words an object composed from an object of type $A$ and the empty object will be of type $A$, simply because composing (aggregating) with the empty object will be the identity operation.

We can extend the language by adding the following modalities that are induced by the adjacency relation:

1. the $\Diamond^+$ modality, which talks about the transitive closure of $R$.
2. the $\Diamond^{-1}$ modality, which talks about the inverse of $R$.

These modalities need axioms to be formally defined in the calculus.

**4.2.2.** Example. Recall the running example of chapter 3 (example 3.1.1). We can express the graph expressions in the meta language of categorial graphs as follows:

$\mathsf{pilot} \Rightarrow \Diamond\mathsf{name} * \Diamond\mathsf{empno} * \Diamond\mathsf{qualif}$
$\mathsf{pilot} \Rightarrow \Diamond\mathsf{roster}$
$\mathsf{roster} \Rightarrow \Diamond(1 \sqcup \mathsf{task})$ (for set flavoured this suffices)
$\mathsf{roster} \Rightarrow \Diamond(1 \sqcup \mathsf{task} \sqcup (\mathsf{task} * \mathsf{task}) \sqcup \ldots \sqcup (\mathsf{task} * \cdots * \mathsf{task}))$
($\mathsf{roster} \Rightarrow \Diamond(!\mathsf{task})$ when we introduce the bang (!))
$\mathsf{task} \Rightarrow \Diamond\mathsf{task\_desciption}$
$\mathsf{task} \Rightarrow \Diamond\mathsf{start}$
$\mathsf{task} \Rightarrow \Diamond\mathsf{end}$
$\mathsf{pilot} \Rightarrow \mathsf{person}$
$\mathsf{task} \sqcap \Diamond(\mathsf{task\_description} \sqcap \mathsf{flying\_duty}) \Rightarrow \Diamond^{-1}(\mathsf{person} \rightarrow \mathsf{pilot})$
when we introduce **self** and the *functions* we can tackle the other constraint:
$\mathsf{start} \sqcap \Diamond^{-1}((\mathsf{task} \sqcap \neg\Diamond\mathbf{self}) * (\mathsf{task} \sqcap \Diamond(\mathsf{task\_descrtipion} \sqcap \mathsf{flying\_duty}) \sqcap \Diamond\mathbf{self})) \Rightarrow$
$\Diamond^{-1}((\mathsf{task} \sqcap \Diamond f_{>11}(\mathsf{start}, \mathbf{self})) * (\mathsf{task} \sqcap \Diamond\mathsf{self}))$

▲

## 4.3 Calculus for the meta language of categorial graphs

In this section we present a variety of rules and axioms for the meta language of categorial graphs. This language, together with the calculus, constitutes a logic we call *the logic of categories*. Several rules and axioms directly correspond to the structural issues we presented in the previous sections. With this we mean that, in a multiset structure for example, we have rules that respect that we can count aspects.

The rules and axioms for our logic are based on the calculus for linear logic ([Girard87], [Troelstra92]). The 'resource consciousness' paradigm of Girard's linear logic ([Girard87]) triggered the use of such a logic for categorial graphs. Logics such as linear logic emerged from a broader landscape of logics, which is the framework of substructural logics. In a Gentzen-style sequent formulation, a substructural logic distinguishes itself by the *absence* of some *structural rules* that are common in the Gentzen-style formulation of the most common logics like classical or intuitionistic logic. Well known substructural logics are 'relevance logic' ([Dunn86]), categorial logic ([Lambek58], [Benthem91]) and 'BCK logic' ([OnoKomori85]). Linear logic differs from these substructural logics by allowing some limited or controlled use of the structural rules using logical (modal[4]) operators.

The axioms and rules for the logic of categories will be presented in Gentzen-style sequent calculus with the restriction that the sequents have, exactly, one formula on the right. A sequent thus will have the following format:

$$\Gamma \Rightarrow A$$

where $\Gamma$ is a sequence of formulas $A_1, \ldots A_n$. The sequence of formulas can intuitively be interpreted as a comma separated list of resources. The $\Rightarrow$ is interpreted as provability:

$$\text{if } \Gamma \Rightarrow B \text{ then from } \Gamma \text{ we can prove } B$$

For basic connectives $*$, $\sqcup$, $\sqcap$, $\mathbf{1}$, $\bot$, and $\top$, we present the usual axioms and rules. We also have modalities in the linear language that are different from the ones that are usually studied in the field of linear logic and modal logic (see e.g. [Bucalo94]). The fundamental difference lies in the fact the accessibility relations we consider are not only *set*-based, but are also *multiset*-based, or *list*-based.

[4]In order to avoid misunderstanding we note that for regulated use of the structural rules we, of course, use other modal operators than the modal operator for adjacency. In fact when we introduce a modal operator for regulated use of the structural rules, we get a *multi modal logic*

**4.3.1.** DEFINITION. (The basic rules for categories)
Rules and axioms for the non-modal part of the calculus:

$$(AX) \qquad A \Rightarrow A$$

$$(CUT) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma', A \Rightarrow B}{\Gamma, \Gamma' \Rightarrow B}$$

$$(L\sqcap) \qquad \frac{\Gamma, A \Rightarrow C}{\Gamma, A \sqcap B \Rightarrow C} \quad \frac{\Gamma, B \Rightarrow C}{\Gamma, A \sqcap B \Rightarrow C} \qquad (R\sqcap) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \sqcap B}$$

$$(L*) \qquad \frac{\Gamma, A, B \Rightarrow C}{\Gamma, A * B \Rightarrow C} \qquad (R*) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma' \Rightarrow B}{\Gamma, \Gamma' \Rightarrow A * B}$$

$$(L\sqcup) \qquad \frac{\Gamma, A \Rightarrow C \quad \Gamma, B \Rightarrow C}{\Gamma, A \sqcup B \Rightarrow C} \qquad (R\sqcup) \qquad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \sqcup B} \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \sqcup B}$$

$$(L\mathbf{1}) \qquad \frac{\Gamma \Rightarrow A}{\Gamma, \mathbf{1} \Rightarrow A} \qquad (R\mathbf{1}) \qquad \Rightarrow \mathbf{1}$$

$$(R\top) \qquad \Gamma \Rightarrow \top$$

$$(\text{no } L\top)$$

$$(L\bot) \qquad \Gamma, \bot \Rightarrow A \qquad (\text{no } R\bot)$$

$$(L\neg) \qquad \frac{\Gamma \Rightarrow A \sqcup \Delta}{\Gamma \sqcap \neg A \Rightarrow \Delta} \qquad (R\neg) \qquad \frac{\Gamma \sqcap A \Rightarrow \Delta}{\Gamma \Rightarrow \neg A \sqcup \Delta}$$

For the adjacency modality we add the following:

$$(\Diamond I) \qquad \frac{A \Rightarrow B}{\Diamond A \Rightarrow \Diamond B}$$

$$(\Diamond EXISTENTIAL) \qquad \frac{\Gamma \Rightarrow \Diamond A * \Diamond B}{\Gamma \Rightarrow \Diamond A}$$

△

The basic set of axioms and rules axiomatizes a language that can talk about essential properties of objects (i.e. whole objects) and about aspects of objects (i.e. partial description of objects). Expressions on aspects of objects can be done using the $\Diamond$ modality. The existential character of the $\Diamond$ modality is axiomatized by the '$\Diamond EXISTENTIAL$' rule. This rules says (informally) that when an object of type $\Gamma$ has an $A$ and a $B$ adjacent, then we may conclude that an object of type $\Gamma$ has an $A$ adjacent.

In the presentation of the models of discourse spaces we stated that the adjacency relation either consist of list, multiset, or set adjacency structures. Note that this basic set of rules can only be sound and complete for an interpretation that involves edges with a list adjacency relation because we cannot show with this basic collection of rules the behavior of a multiset adjacency relation,

$$\Diamond A * \Diamond B \Rightarrow \Diamond B * \Diamond A \ ,$$

nor behavior of a set adjacency relation,

$$\Diamond A * \Diamond A \Rightarrow \Diamond A \ .$$

For axiomatizing this kind of behavior we need structural rules. In the list below we will introduce rules that enable or disable certain expessivity of the language and behavior of the models.

- *rules for adjacency structure*

  1. Adjacency structure is undirected; i.e. there is no order in the adjacents of an object.

  $$(\Diamond EXCHANGE) \quad \frac{\Gamma, \Diamond A, \Diamond B, \Gamma' \Rightarrow C}{\Gamma, \Diamond B, \Diamond A, \Gamma' \Rightarrow C}$$

  2. Adjacency structure is non-resource-conscious; i.e. there is no notion of counting adjacents (only existence of a type of adjacent matters)

  $$(\Diamond CONTRACTION) \quad \frac{\Gamma, \Diamond A, \Diamond A \Rightarrow \Delta}{\Gamma, \Diamond A \Rightarrow \Delta}$$

  $$(\Diamond WEAKENING) \quad \frac{\Gamma, \Diamond A \Rightarrow \Delta}{\Gamma, \Diamond A, \Diamond A \Rightarrow \Delta}$$

- *Rules for whole object structures*

  1. Aggregate structure is undirected; i.e. there is no order in the aggregates of an object.

  $$(RESTRICTED\text{-}EXCHANGE) \quad \frac{\Gamma, A, B, \Gamma' \Rightarrow C}{\Gamma, B, A, \Gamma' \Rightarrow C} \quad \begin{array}{l}\textit{provided that } A \textit{ is} \\ \text{not of the form } \Diamond G\end{array}$$

  2. Aggregation of whole objects is non-resource-conscious; i.e. there is no notion of counting aggregates.

  $$(RESTRICTED\text{-}CONTRACTION) \quad \frac{\Gamma, A, A \Rightarrow \Delta}{\Gamma, A \Rightarrow \Delta} \quad \begin{array}{l}\textit{provided that } A \textit{ is} \\ \text{not of the form } \Diamond G\end{array}$$

  $$(RESTRICTED\text{-}WEAKENING) \quad \frac{\Gamma, A \Rightarrow \Delta}{\Gamma, A, A \Rightarrow \Delta} \quad \begin{array}{l}\textit{provided that } A \textit{ is} \\ \text{not of the form } \Diamond G\end{array}$$

- *General rules*

  1. All structure is undirected (i.e. there is no order in the aggregates and neither in the adjacents of an object)

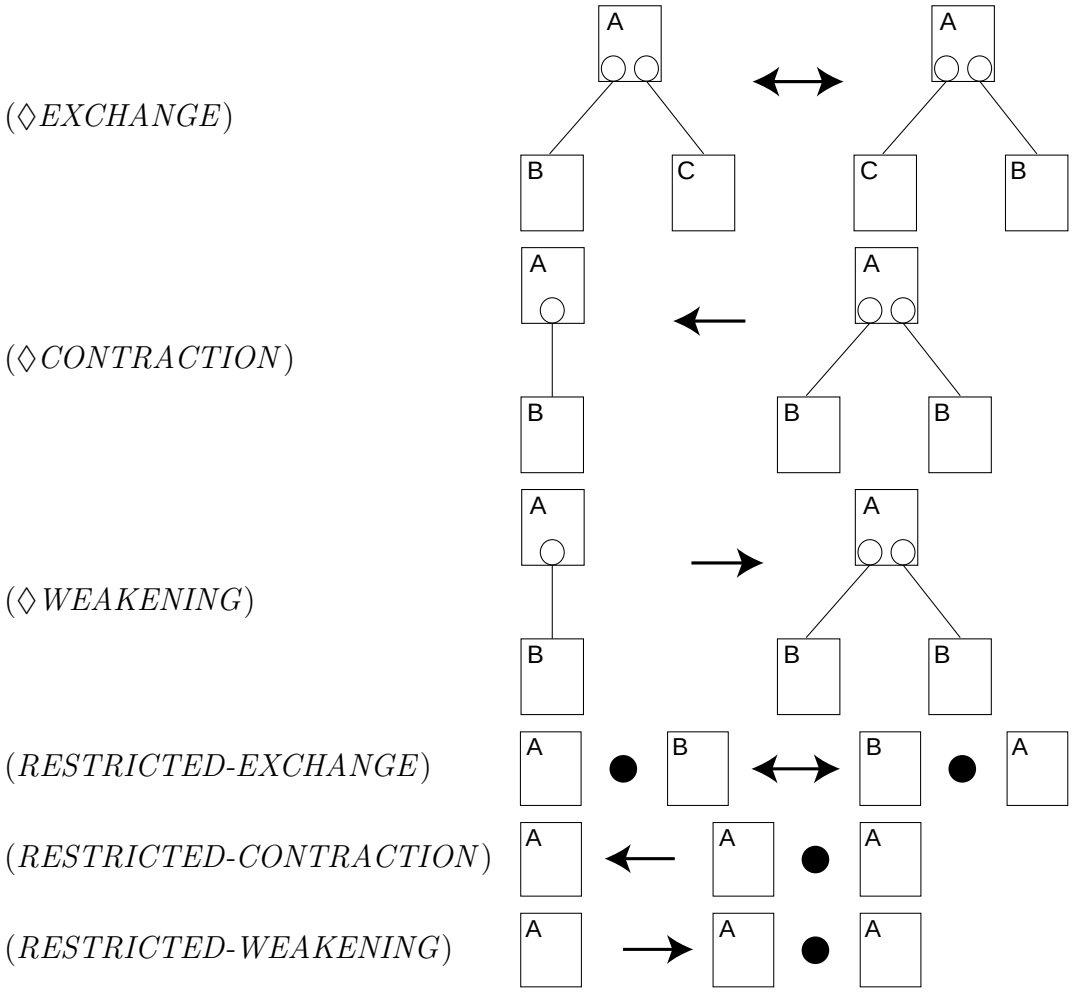
  $$(EXCHANGE) \quad \frac{\Gamma, A, B, \Gamma' \Rightarrow C}{\Gamma, B, A, \Gamma' \Rightarrow C}$$

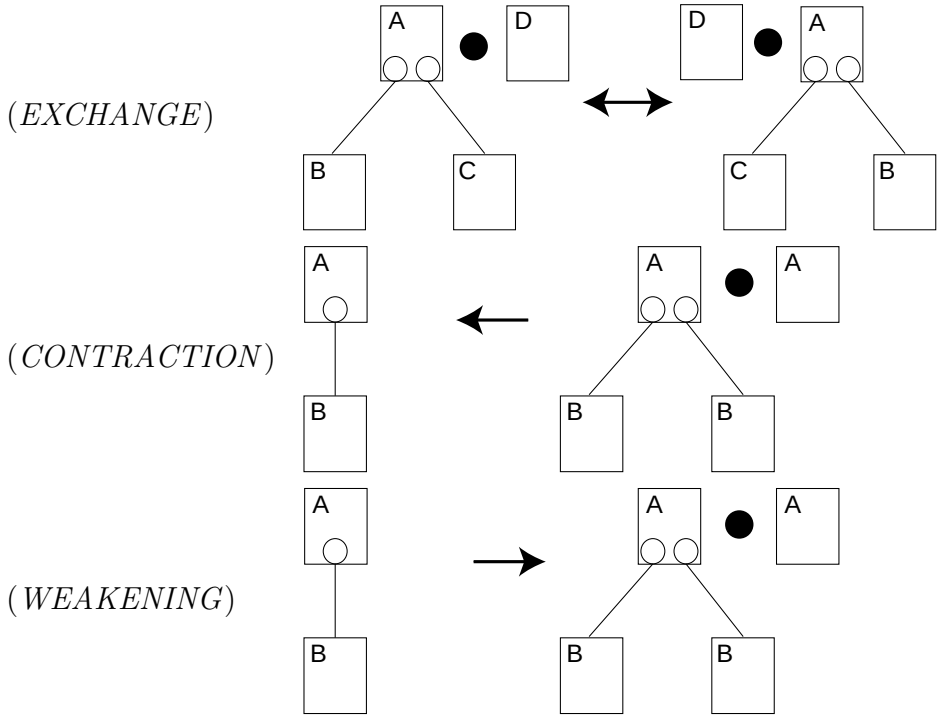
2. All structure is non-resource-conscious; i.e. there is no notion of counting whole objects, neither is there a notion of counting adjacents (only existence of a type of adjacent matters)

$$(CONTRACTION) \quad \frac{\Gamma, A, A \Rightarrow \Delta}{\Gamma, A \Rightarrow \Delta}$$

$$(WEAKENING) \quad \frac{\Gamma, A \Rightarrow \Delta}{\Gamma, A, A \Rightarrow \Delta}$$

In order to illustrate the effect, the rules from above enable the following identifications in the categorial graphs language (specifying types):






The rules from above have serious implications on the semantics of the language of categorial graphs. For example, by introducing $\Diamond EXCHANGE$ the theory can not distinguish between objects with different order of adjacents. This means that the semantics of the language can not contain any reference to the order of the adjacents. In other words, the two expressions $\Diamond A * \Diamond B$ and $\Diamond B * \Diamond A$ will have the same meaning, just like the two graphs above at the $\Diamond EXCHANGE$ label. We will typically interpret the language with the $\Diamond EXCHANGE$ in a semantic domain with objects with an adjacency structure which has no order.

## 4.4 The semantic domain for object oriented information systems

The meta language of categories has an interpretation in a semantic domain. We can construct a landscape of semantic domains for which we can present different sets of axioms and rules that are sound with respect to the proposed semantics by adding or omitting these particular rules[5]. The semantic domains will (all) have the following features:

- *Both 'whole objects' and 'partial description of objects' are members of the semantic domain.* This means that in our semantics we can point at an

[5]very much like the landscape of substructural logics is given its variety by adding and omitting structural rules

object $a$, but also at an aspect $b$ of an object $a$[6].

- *An adjacency structure models the complex structure of an object.* The adjacency structure of an object contains the adjacents of an object, and this structure is as rich as the rules of the calculus describe. In effect this means that the adjacency structure of an object is either a set, a multiset or a list. For example in the multiset case this means that if some object $a$ has three adjacents: $b$, $c$ and again $c$, then[7] $\ddot{\{}(a,b),(a,c),(a,c)\ddot{\}}\ddot{\in}R_{\mathrm{Adj}}$, where $R_{\mathrm{Adj}}$ denotes the adjacency multiset relation.

- *A monoid structure over the object domain interprets aggregation.* This means that there is an operation '$\cdot$' on complex objects that interprets aggregation, and that aggregated objects are members of the domain of objects.

- *Identity.* For handling the notion of identity properly we may not 'a priori' identify an object by its structure. In a world with an object notion there can be two *different* objects with the same adjacents (contrary to the relational world, which requires semantically meaningless attributes as key to force that two objects that in some knowledge state cannot be distinguished by there properties, but may turn out to be two different objects anyhow). This means that characterizations of objects by describing their structure are, in general, *partial.*

- *Links and partial descriptions.* In order to interpret formulas with an existential character, like "an $A$ object is an object that has at least two adjacents, one $B$-adjacent and one $C$-adjacent"[8], we will calculate in our models with partial descriptions of objects. These partial descriptions, called *aspects* or *links*, will be the witnesses of one particular object being the adjacent of another particular object. If we graphically write down an object, a link can be seen as the line between the object and its adjacent. In our semantics we denote an aspect or link by an ordered pair $(a,b)$ meaning the aspect witnessing that[9] $b$ is adjacent to $a$.

We interpret the connectives and operators of the meta language $L_{\mathsf{cat}}$ of categorial graphs in the semantic domain. The construction of this semantic domain is a variation on a Kripke style semantic domain. The domain will be generated by a set of atomic objects $E_{\mathrm{At}}$ and a function $f_R$ (e.g. the characteristic function

[6]For example both the whole object named *Socrates* and its partial description *the white color of Socrates* can be members of the semantic domain

[7]A remark on notation: We use two dots to distinguish the multiset symbols and operations from their normal set variants

[8]in our language that is denoted by $\Diamond A * \Diamond B$

[9]or more concrete: the aspect witnessing that Sorcrates is white

of a relation $R$) that describes the adjacency structure of the objects. The mathematical items $E_{\mathrm{At}}$ and $f_R$ together generate a hybrid structure which is called a *discourse frame* with two dimensions of so called 'structures' (or monoidals) of mathematical elements:

1. one dimension is a 'structure' with mathematical elements behaving like whole objects (and aggregations of whole objects) which can interpret the propositions on whole objects; this structure will be called '*space of wholes*'.

2. the other dimension contains for every (aggregation of) whole object(s) a lattice in the 'structure' of mathematical elements behaving like information-pieces (called 'aspects' or 'links' or 'infons') which can interpret the propositions on aspects or partial descriptions of whole objects; these structures will be called '*adjacency spaces*'.

This discourse frame, together with a valuation (interpretation), form the so called *discourse models* that interpret the language of categorial graphs.

**4.4.1.** Definition. (structure domain)
A *structure domain* is a triple $< C, \cdot, 1 >$ where

- $C$ is a collection of elements (structures)
- $\cdot : C \times C \to C$ is a product
- $1 \in C$ is the unit element for '$\cdot$'

In the course of this section we distinguish 4 types of structure domains based on two properties:

1. commutativity: $e_1 \cdot e_2 = e_2 \cdot e_1$
2. idem-consuming/cloning: $e \cdot e = e$

The 4 variants of the structures are now:

1. non-commutative and non-idem-consuming/cloning (i.e. lists)
2. commutative and non-idem-consuming/cloning (i.e. multisets)
3. non-commutative and idem-consuming/cloning (i.e. lists with no identicals right after another)
4. commutative and idem-consuming/cloning (sets)

Within a structure we have an ordering $\leq$ called a *substructure ordering* that satisfies the following condition:

$$a \leq b \text{ implies } a \cdot c \leq b \cdot c \text{ and } c \cdot a \leq c \cdot b$$

$$a \leq b \text{ if } \begin{cases} a = 1 \\ a = b \\ a = a_1 \cdot \ldots \cdot a_n \& b = b_1 \cdot \ldots \cdot b_n \& a_i \leq b_i (1 \leq i \leq n) \\ (\text{where} = \text{means equivalence in the structure domain}) \end{cases}$$

△

Both dimensions of the model, the *space of wholes* and all the *adjacency spaces* will be structures.

**4.4.2.** Definition. ('space of wholes')
Given a set $E^{\mathrm{At}}$ of objects called "atomic objects", a *space of wholes* $\mathcal{E}$ is a structure domain $< E, \cdot, 1 >$ freely generated from $E^{\mathrm{At}}$. (The ordering in the space of wholes is simply the substructure ordering, which we do not need in the definitions further on) △

The unit element 1 of the space of wholes will interpret the empty type **1** (i.e. the empty type is interpreted by the empty object). If the space contains objects $a_1, a_2$ and $a_3$, then it also contains aggregates $a_1 \cdot a_2$, and also $a_1 \cdot a_2 \cdot a_3$. But also $a_1 \cdot a_1 \cdot a_2 \cdot a_3 \cdot a_2$ etc. etc. etc. . The multiplication operation is defined on objects. We will also define the multiplication on sets of objects (denoted by the same token ('$\cdot$')) as usual: $X \cdot Y := \{x \cdot y \mid x \in X, y \in Y\}$.

Objects have adjacents. The adjacency structure of objects will be modeled by an adjacency function. This function maps a whole object to its (full) adjacency structure element, which is the structure element that models the complex structure of the object. Such a structure element is called an 'infon' and is an element of a structure with the 'information pieces' or 'links' as atomic elements. Such an atomic link is a witness to the fact that 'one object $b$ is in the adjacency structure of another object $a$', and will be denoted by an ordered pair $(a, b)$. The mathematical behavior of the infons (structure elements) will be determined by the rules that hold in the adjacency structure. For example if the adjacency structure is commutative, the behavior of the infons (structure elements) $(a, b) \cdot (a, c)$ and $(a, c) \cdot (a, b)$ containing both the two atomic links $(a, b)$ and $(a, c)$ will be identical. In effect an infon (structure element) representing the (whole or part of the) adjacency structure will be either a *set* (when both commutativity and idem-consumption hold), *multiset* (when only commutativity holds), or a *list* (when neither of the rules hold) of pairs. The function that maps an object to its adjacency structure element will interpret the $\Diamond$-modality. The proposition $\Diamond A$ will be interpreted as *'has an A-type adjacent'*. In other words the adjacency operator talks about *aspects* (links or infons) of an object; i.e. partial descriptions.

**4.4.3.** Definition. (links)
Let $\mathcal{E} =< E, \cdot, 1 >$ be a space of wholes and $e_1, e_2 \in E$. Then a *link* of object $e_1$ is a pair $(e_1, e_2)$. This pair witnesses the fact that $e_2$ is an adjacent of $e_1$. The pair $(e_1, e_2)$ is an information piece that partially describes object $e_1$. It is an *individual aspect* of object $e_1$. △

**4.4.4.** Definition. ('Adjacency spaces')
Let $\mathcal{E}$ be a space of wholes. Then for each element in $e \in E$ we define a structure domain $\mathcal{A}_e$ freely generated from all possible links of $e$. i.e. let $A_e^{At} = \{e\} \times E \subseteq A_e$ be the set of all links of $e$; then

$$\mathcal{A}_e =< A_e, \cdot_e, (e, 1), \leq_e >$$

is the structure domain freely generated by $A_e^{At}$. $\mathcal{A}_e$ is called the *adjacency space* of $e$, and the elements of $A_e$ are called *partial descriptions* or *infons* of $e$. △

**4.4.5.** Definition. (adjacency (structure) mapping) An adjacency structure mapping is a function $f_R : E \mapsto \bigcup_{e \in E} A_e$ that maps each atomic object to a structure element in its adjacency space. This structure element describes the full adjacency structure of the whole object. i.e.

$$f_R(e) \in A_e$$

The domain of $f_R$ can be extended to range over all objects including aggregates ($E$) by putting (*regularity*):

$$\begin{array}{rcl} f_R(a \cdot b) & = & (a \cdot b, c_1) \cdot_{a \cdot b} \ldots \cdot_{a \cdot b} (a \cdot b, c_n) \cdot_{a \cdot b} (a \cdot b, d_1) \cdot_{a \cdot b} \ldots \cdot_{a \cdot b} (a \cdot b, d_n) \\ & \text{iff} & \\ f_R(a) & = & (a, c_1) \cdot_a \ldots \cdot_a (a, c_n) \\ & \text{and} & \\ f_R(b) & = & (b, d_1) \cdot_b \ldots \cdot_b (b, d_n) \end{array}$$

△

**4.4.6.** Definition. (adjacency (structure) relation)
Given a space of wholes $\mathcal{E}$ as above, we define an adjacency (structure) relation $R : E \times \bigcup_{e \in E} A_e$ by the adjacency structure mapping as follows:

$$eRa \text{ iff } f_R(e) = a' \& a \leq_e a'$$

In other words, $R$ relates an object to a partial description (element from an adjacency space) iff the partial description is a substructure of the partial description that is the adjacency structure element of that object. i.e. the whole adjacency structure is related to its object, together with all the infons that are a substructure of that (total) adjacency structure element. △

To clarify the definition of adjacency structure relations we list the 'special cases' where the structure is respectively a set, a multiset, and a list.

> Case **set**: $f_R$ maps an object to a set of partial desciptions. The adjacency structure of an object is a set. $R$ is a set of pairs; pairs have only 'presence' as feature in $R$. The adjacency space will have as elements sets of pairs with set inclusion as ordering.
>
> Case **multiset**: $f_R$ maps an object to a multiset of partial descriptions. The adjacency structure of an object is a multiset. $R$ is a multiset of pairs; pairs will have 'presence' and 'arity' as feature in $R$ (that is asserting arity=0 means 'not present', we could say they only have 'arity' as feature in $R$). The adjacency lattice (defined below) will have as elements multisets of pairs with multiset inclusion as ordering.
>
> Case **list**: $f_R$ maps an object to a list of partial descriptions. The adjacency structure of an object is a list. $R$ consists of lists of pairs; pairs will have 'presence', 'arity', and 'positions' (one position for each occurrence) as feature in $R$. The adjacency lattice will have as elements lists of pairs with list inclusion as ordering.

In this informal formulation the generic structure is formulated as follows:

> Generic case **structure**: $f_R$ maps an object to a structure of partial descriptions. The adjacency structure of an object is a structure. $R$ consists of structures of pairs (a generalization of the relation concept, from which a traditional relation, a multiset relation and a relation consisting of lists are special cases). The structures of pairs will behave according to the rules of the structure. The adjacency lattice will have as elements structures of pairs with structure inclusion as ordering.

An adjacency lattice contains (unordered) sequences of pairs $(e_1, e_2)$, where $e_1$ is an object and $e_2$ is one of its adjacent objects. We will treat the elements of the adjacency lattices, i.e. the infons, as objects with a special property: they are '*extendible*' to all elements that are more informative than themselves, with in the limit the 'whole' that they are a partial description of[10]. For example the infon $(e_1, e_2)$ is interpreted to be extendable to the 'whole' $e_1$. As an other example the aggregation of the infons $(e_1 \cdot e_3, e_2) \cdot (e_1 \cdot e_3, e_2)$ will be extendible to $e_1 \cdot e_3$. Whole objects will be extendible to themselves only, because they describe exactly one individual, and there is no description more informative or precise than the individual itself.

[10]In a sense an infon has an existential character: *'There exists an edge object that I am a partial description of'.*

We will now combine the two worlds, the world of infons (partial descriptions) and the world of whole objects. The combined structure is a hybrid structure containing as elements both the infons and the whole objects. This hybrid space will be called *discourse space*. The structure has two dimensions, one dimension playing at the level of the individual adjacency lattices (i.e. within the description of one whole object), and the other dimension playing at the level of (aggregation of) whole objects. Furthermore the interaction between these dimensions needs to be controlled. This means the following:

- The aggregation of infons from one individual adjacency lattice for an object $e$ is the product in this lattice obeying the rules for taking together partial descriptions, and
- The aggregation of two whole objects is the product within the space of wholes, obeying the rules for taking together propositions about whole objects.
- the aggregation outside an individual adjacency lattice or involving both whole objects and infons is a product operation regulating the traffic between the two worlds.

The first two products are already defined. The product between elements of different worlds (i.e. between two infons of different adjacency lattices or between an infon and a whole object), called a *hybrid product*, needs some more discussion. In general the product operation is modeling the aggregation operation that is taking together two pieces of information. This taking together of two pieces of information can behave in different ways, it can for example be resource conscious or sensitive for the order in which the information is taken together. For a product on uniform behaving elements in our model, e.g. the product of two whole objects or the product of two infons of one object, this product can simply obey the rules in the specific subspace[11]. For the product of two differently behaving elements we need to take a little more care. The behavior of taking together elements in the different subspaces should be conservative w.r.t. the taking together of elements in the different subspaces, in the sense that it should not be possible to obtain equivalences in the separate subspaces by using the hybrid product which one can not obtain using soley the product in the different subspaces.

In effect this means that the product of two elements of different worlds will be a term that will have structural properties (like commutativity, associativity, idem-consumption, and cloning) only if the products in the other two worlds both have these properties.

**4.4.7.** Definition. (hybrid product)
Let $\mathcal{E}$ be a space of wholes and let $\mathcal{A}_e$(for all $e \in E$) be a collection of adjacency

[11]The space of wholes and the separate adjacency lattices are all spaces with a product.

spaces generated from a collection of objects $E^{\mathrm{At}}$ and an adjacency relation $R$. Let $o_1, o_2 \in E \cup \bigcup_{e\in E} A_e$, then

$$o_1 \cdot o_2 = \begin{cases} o_1 \cdot_{A_e} o_2 & \text{if } o_1, o_2, o_1 \cdot_{A_e} o_2 \in A_e \\ o_1 \cdot_E o_2 & \text{if } o_1, o_2 \in E \\ o_1 \cdot_H o_2 & \text{otherwise} \end{cases}$$

△

Now we formalize our notion of *extendibility* relating the separate worlds of infons to the world of whole objects. An infon will be extendable to the infons that are more informative than itself but still a partial description of the whole object, and to the (aggregation of) whole object(s) of which it is the witness of one or more aspects. Whole objects will be extendable to themselves only.

**4.4.8.** Definition. (extendibility)
Let $\mathcal{E}$ be a space of wholes and let $\mathcal{A}_e (e \in E)$ be a collection of adjacency spaces generated from a collection of objects $E^{\mathrm{At}}$ and an adjacency relation $R$. For $o \in E \cup \bigcup_{e\in E} A_e$, we define

$$\mathsf{Ext}(o) = \begin{cases} \{o\} & \text{if } e \in E \\ \{a |\ o \in A_e\ o \leq a \& a \leq f_R(o)\} \cup \{e |\ o \in A_e\} & \text{if } a \in \bigcup_{e\in E} A_e \\ \emptyset & \text{otherwise} \end{cases}$$

(Note that the $\mathsf{Ext}$ operations filters out all undefined products) △

**4.4.9.** Example. Look at the following objects $E^{At} = \{a, b_1, b_2, c, d\}$ with the adjacency multiset relation $R = \ddot{\{}(a, b_1), (a, b_2), (c, d)\ddot{\}}$. The following then are examples of adjacency lattices (in the case of a multiset adjacency structure, i.e. associative and commutative space for the adjacents):

$A_a = \{(a, b_1), (a, b_2), (a, b_1) \cdot (a, b_2\}$
where $(a, b_1) \leq (a, b_1) \cdot (a, b_2)$ and $(a, b_2) \leq (a, b_1) \cdot (a, b_2)$
$A_c = \{(c, d)\}$
$A_{a\cdot c} = \{(a \cdot c, b_1), (a \cdot c, b_2), (a \cdot c, d), (a \cdot c, b_1) \cdot (a \cdot c, b_2),$
$(a \cdot c, b_1) \cdot (a \cdot c, d), (a \cdot c, b_2) \cdot (a \cdot c, d), (a \cdot c, b_1) \cdot (a \cdot c, b_2) \cdot (a \cdot c, d)\}$
where $(a \cdot c, b_1), (a \cdot c, b_2) \leq (a \cdot c, b_1 \cdot (a \cdot c, b_2)$
and $(a \cdot b_1), (a \cdot c, d) \leq (a \cdot c, b_1) \cdot (a \cdot c, d)$
and $(a \cdot c, b_2), (a \cdot c, d) \leq (a \cdot c, b_2) \cdot (a \cdot c, d)$
and $(a \cdot c, b_1) \cdot (a \cdot c, b_2), (a \cdot c, b_1) \cdot (a \cdot c, d), (a \cdot c, b_2) \cdot (a \cdot c, d) \leq$
$(a \cdot c, b_1) \cdot (a \cdot c, b_2) \cdot (a \cdot c, d)$

and the following are examples of products and extensions:

$$\begin{array}{l}
\mathsf{Ext}(a) = \{a\} \\
\mathsf{Ext}(b_1) = \{b_1\} \\
\mathsf{Ext}((a, b_1)) = \{(a, b_1), (a, b_1) \cdot (a, b_2), a\} \\
\mathsf{Ext}((a, b_1) \cdot (a, b_2)) = \{(a, b_1) \cdot (a, b_2), a)\} \\
\mathsf{Ext}((a, b_1) \cdot (c, d)) = \emptyset \\
\mathsf{Ext}((a, b_1) \cdot (a, b_1)) = \emptyset
\end{array}$$

▲

**4.4.10.** Definition. (discourse space)
Let $\mathcal{E} =< E, \cdot, 1_{\mathcal{E}} >$ be a space of wholes and let $\mathcal{A}_e =< A_e, \cdot, 1_e, \leq >$ for all $e \in E$ be adjacency lattices generated from a set of atomic objects $E^{\mathrm{At}}$ and a structure adjacency relation $R$ on $E^{\mathrm{A}}$. Now define $< O, \cdot_O >$ to be the associative and commutative monoid freely generated from $E^{\mathrm{At}} \cup \bigcup_{e \in E} A_e$ where

$$a \cdot_{\mathcal{O}} b = \begin{cases} a \cdot_{\mathcal{E}} b & \text{if } a, b \in E \\ a \cdot_{\mathcal{A}_e} b & \text{if } a, b \in A_e \text{ for any } e \in E \\ a \cdot_{\mathcal{H}} b & \text{otherwise} \end{cases}$$

where $\cdot_{\mathcal{E}}$ is the product of the space of wholes and $\cdot_{\mathcal{A}_e}$ is the product of the adjacency lattice of $e$ and $\cdot_{\mathcal{H}}$ is the product for the hybrid terms;

We will say that $\mathcal{O}_{\mathcal{E},R}$ is the *discourse space* generated by $E^{\mathrm{At}}$ and $R$. The rules for $\cdot_{\mathcal{E}}$ determine the product between whole objects, and the rules for $\cdot$ in $\mathcal{A}_e$ determine the behavior of the product between infons of one world (built of links) and the rules for $\cdot_H$ for the interaction between the different worlds. Again we are talking about behavior in terms of commutativity and idem-consumption and cloning. △

Given a collection of atomic objects and an adjacency relation over these, we can determine uniquely a discourse space. In a Kripke style model this combination is called a *frame*. A collection of objects together with an adjacency relation constitutes what we will call a *discourse frame*. We will use the spaces generated from this frame to define models.

**4.4.11.** Definition. (discourse frames)
Given a collection $E^{\mathrm{At}}$ and an adjacency structure relation $R$ over $E^{\mathrm{At}}$; we will say that $\mathcal{F} = (E^{\mathrm{At}}, R)$ is a *discourse frame*. Note that $E^{\mathrm{At}}$ and $R$ determine a space of wholes , the adjacency lattices for all objects in the space of wholes and a discourse space. △

**4.4.12.** Definition. (Valuations)
Let $\mathcal{F} = (E^{\mathrm{At}}, R)$ be a discourse frame and $\mathcal{O}_{\mathcal{E},R} =< O, \cdot >$ be the discourse space

determined by $\mathcal{F}$. A *pre-valuation* $\nu : Q_{\mathsf{UE}} \mapsto P(\mathcal{E})$ is a function that assigns to each propositional variable of $L_{\mathsf{UE}}$ a set of (whole) objects. We extend $\nu$ to take arbitrary formulas of $L_{\mathsf{UE}}$ and extend its range to $\mathcal{O}_{\mathcal{E},R}$ (links and objects) by:

$$\begin{array}{lcl}
\nu(P \sqcup Q) & = & \mathsf{Ext}(\nu(P) \cup \nu(Q)) \\
\nu(P \sqcap Q) & = & \mathsf{Ext}(\nu(P) \cap \nu(Q)) \\
\nu(P * Q) & = & \mathsf{Ext}(\nu(P) \cdot \nu(Q)) \\
\nu(\neg P) & = & \mathsf{Ext}[E - \mathsf{Ext}(\nu(P))] \\
\nu(\Diamond P) & = & \mathsf{Ext}(\{(q,p) | (q,p) \in R, p \in \nu(P)\}) \\
\nu(\top) & = & O \\
\nu(\bot) & = & \emptyset \\
\nu(\mathbf{1}) & = & \{1_{\mathcal{E}}\}
\end{array}$$

A *valuation* $V : Q_{\mathsf{UE}} \mapsto P(\mathcal{E})$ is defined as:

$$V(P) = \nu(P) \cap E$$

△

**4.4.13.** Definition. (Models for categorial graphs)
A pair $(\mathcal{F}, V)$ consisting of a discourse frame $\mathcal{F}$ and a valuation $V$ is called a *discourse model*. The notions of truth and validity given a (collection of) model(s) are defined as usual:

Let $\mathcal{M} = (\mathcal{F}, V)$ be a discourse model, where $\mathcal{F} = (\mathcal{E}, R)$ is a discourse frame, $\mathcal{E} =< E, \cdot, 1 >$ is a space of wholes and $a \in E$ is an object. A formula $\phi \in \Phi(M_{\mathsf{UE}})$ is *true* at object $a$ in model $\mathcal{M}$, notation $\mathcal{M}, a \models \phi$ if $a \in V(\phi)$. △

To end this section we will write down a very simple example of a model for the categorial graph language. This example illustrates the whole process of obtaining semantics for a categorial graph.

**4.4.14.** Example. Look at the very simple categorial graph $G$ of figure 4.1, and its instance (object graph) $O$. For $G$ we get:

$$A \Rightarrow \Diamond B$$

For $O$ we have the following model:

$$\begin{array}{rl}
\text{space of wholes} & < \{a, b, 1\}, \cdot_E, 1 > \\
\text{links} & \{(a,b)\} \\
\text{adjacency spaces} & \left\{ \begin{array}{l} < \{(a,b),(a,1)\}, \cdot_a, (a,1) > \\ < \{(b,1)\}, \cdot_b, (b,1) > \end{array} \right. \\
\text{adjacency structure mapping} & \left\{ \begin{array}{l} f_R(a) = (a,b) (= (a,b) \cdot (a,1) \cdot (a,1)) \\ F_R(b) = (b,1) \end{array} \right. \\
\text{adjacency structure relation} & R = \{(a,(a,b)), (a,(a,1)), (b,(b,1))\} \\
\text{discourse space} & < \{a, b, 1, (a,b), (a,1), (b,1)\}, \cdot_O > \\
\text{pre-valuation} & \left\{ \begin{array}{l} \nu(A) = \{a\} \\ \nu(B) = \{b\} \end{array} \right.
\end{array}$$

▲

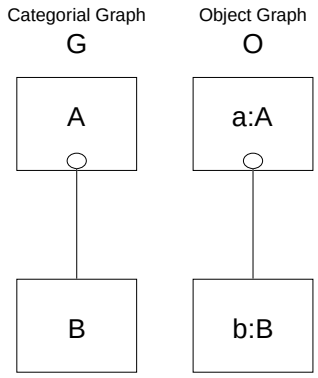



Figure 4.1: A very simple example of a categorial graph and a model

## 4.5 Summary

In this chapter we presented the semantics for the language of categorial graphs. The language together with its semantics provides a 'direct' formalization of expressions and concepts from the practical world of object oriented information systems. This system will be subject to further theoretical (logical) analysis in the subsequent chapters.

# Part III

# Logical Aspects

# Introduction

The chapters in this part present logical aspects of the theory of object oriented information systems.

- In chapter 5 we discuss the approach to semantics we have taken in this thesis which is based on *logic*. We will relate the semantics of object oriented information systems to research on semantics in computer science. Moreover we will place the major artifact from our semantical research -the logic of categorial graphs- in its logical context.
- In chapter 6 we will discuss some important logic-theoretic aspects of the logic of categorial graphs. We will talk about soundness and completeness, and will take a look at the computational complexity of the logic that catches the object oriented concepts.

# Chapter 5

# Methodology: semantics, logic and applications

*Computer science is not only a study of basic theory, and it is not just the business of making things happen. It's actually a study of how things happen.*

*Robin Milner*

What is the semantics of object oriented information systems good for? In this chapter we describe the *methodology* of semantic research underlying the investigations on object orientation in this thesis. Formal semantics is a curious form of mathematics but provides important insights when done properly. The formal semantics also form a valuable basis to research applications of object oriented information models. In this chapter we validate our research on its methodology and point to some gains of the research done so far.

The formal semantic studies on object orientation in this thesis delivers one major artifact: *the logic of categorial graphs*. We use the arsenal of modern logic to provide a proper semantics for the concepts studied. We will place the logic of categorial graphs in its logical context. We give its roots and its logical motivation, and will discuss matters that are important for this kind of logic, focusing on its use in semantical studies.

The theory we developed in this thesis for object oriented information models, of course, gives more then insight only. It provides a basis for research in various application domains that use object oriented information models. One nice application that is a direct application of this research, is the use of the logic for categorial graphs for data mining algorithm analysis ([HaasAdriaans99]). We will elaborate on this application and other possible applications in the second half of this chapter.

## 5.1 Formal Semantics

It can not be stressed enough that designers, developers and serious users of modeling, database or programming languages need a complete and accurate understanding of the semantics (or intended meaning) of those languages. A rigorous mathematical theory of semantics of a language is needed to support correct description and implementation of its meaning. Furthermore it is indispensable for verification and systematic development of programs written in the language. Of course semantics can be constructed in many different ways. Below we give an overview and describe the approach to semantics for object oriented information systems we have taken in this thesis.

### 5.1.1 Semantics in computer science

A formal semantics[1] for a language is a mapping from that language to some mathematical structure that models the universe of discourse. In this setting the considered language is called "object language". The mathematical structure is the "semantic domain" and should capture the meaning of the constructs and concepts we want to express in the object language. The elements of the semantic domain have mathematical properties, which exploit the behavior of these elements. In order to do mathematics with the semantic domain we also need a "meta language" with which we can reason about the semantic domain.

In computer science 'semantics' became an issue when the controversy and confusion over how to interpret the definition of ALGOL-60 started to become embarrassing. Some means had to be found of identifying the meaning of a programming language in a precise and unambiguous way. The, now classical, solution of Scott and Strachey ([ScottStrachey71]) was to associate a mathematical denotation to each phrase of the object language. They discovered Church's $\lambda$-notation suitable for this task and constructed a mapping from ALGOL-60 phrases to a $\lambda$-language. In this $\lambda$-language, the meta-language, one could specify the meaning of the object-language, ALGOL-60. But for precise comprehension this meta-language also needed a semantic definition, which Scott and Strachey provided for in the shape of reflexive domains. Semantics constructed in this manner are generally called denotational semantics.

The similarity between the notion of (denotational) semantics in the programming linguistics as described above, and the notion of semantics in mathematical logic has led to a strong influence of the latter in the field of computer science

[1] The definition of the concept "semantics" given here is tailored for computer science. We are very well aware that this concept if often used in a much broader setting or in different fields of science.

semantics. In mathematical logic a semantic domain together with a homomorphism from some syntax to that domain is called an 'interpretation'. Given a notion of 'truth' in the domain, an interpretation is a 'model' of a logic if all of the axioms of the logic theory are true in the interpretation and all the inference rules preserve truth. In the case of programming languages we can think of the (denotational) semantics as a model of the language its execution on some (abstract) machine (Floyd-Hoare theory). We only need to explicify some notion of truth in the semantic domain to obtain a proper logic from the interpretation. For example in the case of $\lambda$-calculus models we might use realizability as our notion of truth, in which case all that really needs to be checked is that the conversion rules are valid.

From the above it should be clear that in doing semantics by following the techniques of Scott and Strachey one is concerned with giving mathematical *models* for programming languages. This is in contrast to the *axiomatic* approach of other major frameworks of semantics such as Hoare Logic [Cousot90] and Structured Operational Semantics [Milner90] in which the execution behavior of programs is formalized by inventing axioms and rules for the basic programming language primitives. In this way one obtains a 'syntactic' system in which one can deduce programming expressions to be equivalent. Nevertheless one is -in a roundabout way- giving (via the axioms and rules) the linguistic primitives behavior that the interpreting objects should have when we would build a model for the execution[2]. It should be clear that the connection between the two kinds of semantics can be tight.

### 5.1.2 Semantics for categorial graphs

In this thesis we designed a 'concrete' discourse model for categorial graphs. This model is abstract enough to have different 'realizations', keeping its core notions intact. By realizations we mean that we can translate the discourse models to actual UML object schemas, real database incarnations, or runs of a program coded in an object oriented programming language.

Because of the appealing abstractness of the semantics domains in the field of logic we constructed the semantics for object oriented information systems in a logical setting. Moreover the logic and the semantics are specifically tailored for the domain of object orientation, and did not choose to code the semantics in a universal language like first order logic with is standard relational semantics. In this regard we take a route that witnesses a relatively new development in logic and theoretical computer science. Universal languages like first order or second order logic become less popular and there is a trend to construct different specific logics for different applications. Let us, for example (or for intimidation), cite

[2]and would have a 'direct' interpretation function

the famous logician Yuri Gurevich from his manifest of 'logic and the challenge of Computer Science' ([Gurevich88]):

> [...]
> *But the new applications call, we believe, for new developments in logic proper. First order predicate calculus and its usual generalizations are not sufficient to support the new applications.*
> [...]
> *It seems that we (the logicians) were somewhat hypnotized by the success of classical systems. We used first-order logic where it fits and where it fits not so well. We went on working on computability without paying adequate attention to feasibility. One seemingly obvious but nevertheless important lesson is that different applications may require formalizations of different kinds. It is necessary to "listen" to the subject in order to come up with the right formalization.*
> [...]

So to formalize a specific collection of notions, specific logics are used. Many exotic systems like Horn clause logic, temporal logic, second order polymorphic lambda calculus, dynamic logic, order sorted logic, modal logic, infinitary logic, intuitionistic higher order type theory, continuous algebra, intensional logic and linear logic have been proposed to handle notions of concurrency, time, overloading, exceptions, non-termination, program construction and even natural language. As examples of new application specific logics include various variants of 'linear logic', let us cite some of its ideology ([Girard87]):

> [...]
> *For logic, computer science is the first real field of application since the applications to general mathematics have been too isolated. The applications have a feedback to the domain of pure logic by stressing neglected points, shedding new light on subjects that one could think of as frozen into desperate staticism, as classical sequent calculus or Heyting's semantics of proofs.*

An example in the tradition of modal logic of application specific modal logics is 'arrow logic' of van Benthem ([Benthem93], [Venema94]) This logic is designed to deal with transitions, and therefore transitions (arrows) are intrinsic objects in the logic. Let us quote some of its motivation([Benthem93]):

> *The current interest in logic and information flow has found its technical expression in various systems of what may be called 'dynamic logic' in some broad sense. But unfortunately, existing dynamic logics based on binary transition relations between computational states have high complexity. Therefore, it is worthwhile rethinking the choice of*

> *a relatively simple dynamic base system forming the 'computational core' that we need, without getting entangled in the complexity engendered by the additional 'mathematics of ordered pairs'.*
> [...]
> *This may be seen by developing an alternative, namely a modal logic of 'arrows', which takes transitions seriously as dynamic objects in their own right.*

Of course there are enormous differences between these logics mentioned above. The only similarity lies in the fact that these logics are designed to talk 'conveniently' about a specific set of notions, by incorporating these notions on a high level in the formal system.

## 5.2 The roots of the logic of categories

The logic of categorial graphs that is designed to provide a semantics for object oriented information system languages has been built using the tools of modern mathematical logic. The logic has a very archetypical modal operator in the *adjacency* operator, but also the *aggregation* and the *empty type* could be seen as modal operators. On the other hand the *aggregation* or composition operator is very well known from substructural logics. Below we take a look at these roots of the logic of categorial graphs.

### 5.2.1 The categorial graph logic as a modal logic

Intuitively the adjacency modality has its roots in modal logic. The adjacency operator has a property in common with the archetypical modal operators: the adjacency operator gets a meaning only when we know in which world (object) in the model a formula with the adjacency operator is evaluated. To get an idea, we list some basic characteristics of modal logic.

The language of modal logic can be seen as the language of propositional logic to which some operators have been added. Modal languages are interpreted in so called *relational structures*. Modal logic has been very successful in providing expressive languages and reasoning calculi for various application domains that could be modeled in relational structures. The abstractness (and therefore general) of relational structures makes them very suited for such a task. A relational structure is a nonempty set of items on which a number of relations has been defined. With a modal language one can speak about such relational structures by interpreting the modal operators by the relations. To make things more concrete we give the definitions.

**5.2.1.** Definition. (relational structure) A relational structure is a tuple F, whose first component is a non-empty set $U$ we call *universe* or *domain* of F, and whose remaining components are *relations* on $U$. △

For example a set of objects $U$ together with a relation $R$ that relates two objects when one object is the adjacent of the other, is a relational structure.

**5.2.2.** Definition. (modal language) Let $O$ be a nonempty set of operators $\Diamond$, where each operator has arity $\rho(\Diamond)$ (the pair $(O, \rho)$ is called a *similarity type*), and let $\mathsf{Prop}$ be a set of proposition letters, then the modal language $L$ for $(O, \rho, \mathsf{Prop})$ is defined as follows:

$$L := \mathsf{Prop}|L \sqcap L|L \sqcup L|\neg L|\Diamond(L_1, \ldots, L_{\rho(\Diamond)})$$

Where $\Diamond$ ranges over the modal operators in $O$ and $L_i$ is an $L$ formula (with an index). For non-nullary modal operators $\Diamond$ we define its dual $\Box$ as follows: Let $A_1, \ldots, A_{\rho(\Delta)}$ be formulas in $L$ then

$$\Box(A_1, \ldots, A_{\rho(\Diamond)} := \neg\Diamond(\neg A_1, \ldots, \neg A_{\rho(\Diamond)})$$

△

The archetypical modal language is the language with one unary modal operator $\Diamond$. An example of a modal formula in this archetypical language is $\Diamond P_1 \sqcup \Diamond(P_1 \sqcap \Diamond P_3)$ (where $P_i \in \mathsf{Prop}$).

**5.2.3.** Definition. (model) Let $L$ be a modal language An $L$-frame is a relational structure $\mathcal{F}$ with the following ingredients:

1. a non-empty universe $U$,
2. for each modal operator $\Diamond$ of arity $\rho(\Diamond)$ in $L$, a $\rho(\Diamond) + 1$-ary relation $R_\Diamond$.

Given a *pre-valuation* $V : \mathsf{Prop} \mapsto 2^U$ mapping proposition letters to a set of elements in $U$, we can define an $L$-model as a $L$-frame with a pre-valuation:

$$\mathcal{M} = (\mathcal{F}, V)$$

In the model we say that $P$ is true in $x$ (or $x$ satisfies $P$) when $x \in V(P)$. △

The model for the archetypical modal language with one unary model operator $\Diamond$ will contain a universe $U$, a binary relation $R_\Diamond$ on $U$ and a valuation mapping proposition letters of the modal language to sets of elements in $U$. The relational structure with objects and a relation that relates adjacent objects, together with a valuation would be an appropriate model for this archetypical language.

**5.2.4.** DEFINITION. (semantics) Let $L$ be a modal language and let $\mathcal{M}$ be an $L$-model as above. Then we define the notion of a formula $A$ being *satisfied* in model $\mathcal{M}$ in element $x \in U$ (notation $\mathcal{M}, x \models A$) as follows:

$$\begin{array}{rll}
\mathcal{M}, x \models P & \text{iff} & x \in V(P) \\
\mathcal{M}, x \models \bot & & \text{never} \\
\mathcal{M}, x \models \neg A & \text{iff} & \text{not } \mathcal{M}, x \models A \\
\mathcal{M}, x \models A \sqcap B & \text{iff} & \mathcal{M}, x \models A \text{ and } \mathcal{M}, x \models B \\
\mathcal{M}, x \models A \sqcup B & \text{iff} & \mathcal{M}, x \models A \text{ or } \mathcal{M}, x \models B \\
\mathcal{M}, x \models \Diamond(A_1, \ldots, A_n) & \text{iff} & \text{for some } y_1, \ldots, y_n \in U \text{ with } xR_\Diamond y_1 \ldots y_n \\
 & & \text{we have } \mathcal{M}, y_i \models A_i \ (1 \leq i \leq n) \\
 & & (\text{for all modalities } \Diamond \text{ of } L)
\end{array}$$

Now we say that the *interpretation* of a formula $A$ in model $\mathcal{M}$ is the set $\{x | \mathcal{M}, x \models A\}$. △

Modal logics have axiomatics. The axioms and rules for the modal operators that are interpreted in relations structures all have a small number of axioms and rules in common. For unary modalities these are the axioms and rules for the minimal normal modal logic $K$. For the modalities with arity greater then 1 similar minimal systems exist.

**5.2.5.** DEFINITION. (Axiomatics for the minimal modal logic $\mathsf{K}$)

- all axioms and rules of propositional logic for the logic with $\sqcap$, $\sqcup$ and $\neg$
- rules for the modalities in the minimal modal logic

$$\begin{array}{ll}
(\Diamond\text{Distribution}) & \Diamond(A \sqcup B) \to (\Diamond A \sqcup \Diamond B) \\
(\Diamond\text{Necessation}) & \dfrac{\neg A}{\neg\Diamond A}
\end{array}$$

There is also a well known alternative formulation of the rules for $\mathsf{K}$ based on the $\Box$ (the dual modality of $\Diamond$) which we sometimes will use.

- rules for the modalities in the minimal modal logic based on $\Box$

$$\begin{array}{ll}
(\Box\text{Distribution}) & \Box(A \to B) \to (\Box A \to \Box B) \\
(\Box\text{Necessation}) & \dfrac{A}{\Box A}
\end{array}$$

△

Modal logics that are complete for relational structures in which the structures have some specific behavior have addition axiomatics that correspond to the behavior. The behavior of the relations (and the interactions between different relations in a relational structure) is normally defined by *constraints*. These constraints on the relations have corresponding axioms and rules in the axiomatics[3].

There are numerous examples of logics that talk about complex structures using a modal operator. The tree logic ([BlackburnEtAlii93]) talks about complex tree structures, and arrow logic ([Benthem93]) talks about transition systems.

In the next chapter we will do logical analysis of the categorial graph logic in a pure modal setting. This analysis will both give more insight into the categorial graphs, as well as deliver interesting languages and domains for logical research.

### 5.2.2 The categorial graph logic as a substructural logic

In the categorial graph logic of chapter 4 we have seen an aggregation operator '$*$'. The structural behavior of the domain in which this operator is interpreted is justified in the calculus with which we can reason about the domain by the presence and absence of structural rules for this operator.

In Gentzen style sequential formalisms a *substructural logic* shows itself by the absence of (some of) the so-called structural rules. Examples of such logics are relevance logic [Dunn86], linear logic [Girard87] and BCK logic [OnoKomori85]. Notable is the substructural behavior of categorial logic[4], which in its prototype form is the Lambek calculus. Categorial logics are motivated by its use as grammar for natural languages. The absence of the structural rules changes the abstraction of *sets* in the semantic domain to *structures*, where elements in an aggregation can have position and arity, while in a set they do not.

In figure 5.1 we list the axiomatics of the first order propositional sequent calculus[5], with the axioms , the cut rule, rules for the connectives and the structural rules for exchange, weakening and contraction.

**5.2.6.** Example. In a domain of sets the following 'expressions' are equivalent, while they are not necessarily so in the domain of stuctures:

$$a, a, b, a \approx a, b, b$$

In a calculus with all the structural rules the features 'position' and 'arity' are irrelevant in the semantic domain, because aggregates that differ in these features

[3]It is not necessarily so that each constraint corresponds to an axiom or rule. It is more the case that a collection of constraints correspond to a collection of axioms and rules that characterize the set of constraints.

[4]See here the inspiration for the name of our basic language building blocks!

[5]Note that in the variant we use here we have a special case of the R$\wedge$ rule.

$$\text{(Ax)}\ A \Rightarrow A \qquad \text{(Cut)}\ \frac{\Gamma \Rightarrow A, \Delta \;\; \Gamma', A, \Rightarrow \Delta}{\Gamma', \Gamma \Rightarrow \Delta', \Delta}$$

$$\text{(L}\wedge\text{)}\ \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta} \qquad \text{(R}\wedge\text{)}\ \frac{\Gamma \Rightarrow A, \Delta \;\; \Gamma' \Rightarrow B, \Delta}{\Gamma, \Gamma' \Rightarrow A \wedge B, \Delta}$$

$$\text{(L}\vee\text{)}\ \frac{\Gamma, A \Rightarrow \Delta \;\; \Gamma, B \Rightarrow \Delta}{\Gamma, A \vee B \Rightarrow \Delta} \qquad \text{(R}\vee\text{)}\ \frac{\Gamma \Rightarrow A, \Delta}{\Gamma \Rightarrow A \vee B, \Delta} \;\; \frac{\Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \vee B, \Delta}$$

$$\text{(Ex)}\ \frac{\Gamma, A \wedge B, \Gamma' \Rightarrow \Delta}{\Gamma, B \wedge A, \Gamma' \Rightarrow \Delta}$$

$$\text{(Weak)}\ \frac{\Gamma \Rightarrow \Delta}{\Gamma, A \Rightarrow \Delta}$$

$$\text{(Contr)}\ \frac{\Gamma, A, A \Rightarrow \Delta}{\Gamma, A \Rightarrow \Delta}$$

Figure 5.1: First order propositional sequent calculus

can be proved equivalent with the structural rules. To see this, observe that the left side of the above equation can be transformed to the right side by performing the following operation:

$a, a, b, a$

*contract* $a, a$ in first two positions to $a$

$a, b, a$

*exchange* $b, a$ in last to positions to $a, b$

$a, a, b$

*contract* again $a, a$ in first two positions to $a$

$a, b$

*weaken* expression $b$ in last position to $b, b$

$a, b, b$

In the logical analysis in the next chapter we will also analyze the categorial graph logic from a purely substructural logic point of view. This will give the logic of categorial graph a 'more explored' foundation, and will enable us to list some interesting characteristics of fragments of the categorial graph logic. We will also see that there is a natural transition from the modal way of looking at categorial graphs and the substructural way. This will become clear in chapter 6.

### 5.2.3 Related logics of information systems

The logic of categorial graphs is not the only approach to formalize the world of object oriented information systems using logic. Two influential proposals are 'O-Logic' of Maier ([Maier86], extended in [KiferWu93]) and 'F-Logic' of Kifer and Lausen ([KiferLausen89]). In these proposals the notions of object orientation are embedded in first order (predicate!) logic (O-Logic 'contains' first order predicate logic), and even higher order logic (F-Logic). The strong languages enables one to express all possible features easily and elegantly, and can be used to reason in these strong systems about the features expressed. However, because of the wealth of expressive power, these systems are less suited for logical analysis of the features themselves. Moreover the reasoning task is computationally undecidable, because the systems of first order predicate logic and higher order logic are computationally undecidable.

There are also related systems of Information logics, that were inspired from a more general (not particularly object oriented) conception of information and knowledge. Examples of these logics are 'description logics' ([Baader96], [Areces00]), 'feature logics' ([Rounds97]), logic of Information flow ([BarwiseSeligman97]).

The approach we take to object oriented information system logics distinguishes itself from the other proposals by the fact that we use thoroughly the achievements of modal and substructural logic. This enables us to analyze the notions from a perspective that is 'lower' in generality than most other approaches to object orientation, that are first order and higher order theories. Our approach enables us to analyze the notions of object orientation without carrying the burden of the generality of first order logic, which includes over-expressiveness (one can say much more in an over-expressive language than actually needed to capture the notions) and computational intractability (first order theories are in general undecidable). In our approach we can analyze the notions more directly (in a language tailored very closely to the notions) and we use theories with computationally nicer behavior. We will see some nice achievements from this approach in the next chapter.

In a logical sense our approach relates more closely to the mentioned information logics. These logics are tailored to reason about information (although not necessarily object oriented). It is therefore an interesting exercise to map the features we came up with to the frameworks of these information logics. This exercise is subject to future work.

## 5.3 Applications

Next to applications in the practice of object oriented information systems and logic, there are interesting applications for the logic of categorial graphs in fields that need a good understanding about the structure of object oriented information. One such field that really needs insight in information structures and that has both practical and theoretic involvement is data mining.

### 5.3.1 Applications in object oriented information system practice

The most direct applications can be found in the field of model checking of the large and complex information models that information system designers construct in practice. Modern software development methodologies (e.g. UP, see chapter 1) invite designers to specify parts of the systems in small pieces (components) from many different views (in different use cases, and in different level of abstractness -requirements, analysis, design and implementation). The scattered model should be checked on consistency and validity (for example in the sense that it should be satisfiable). Automatic support for this task is still very rudimentary and far from complete for object oriented models. A system like the logic of categorial graphs give a theory to handle the checking of the more involved constraints that are formulated for the information models.

### 5.3.2 New computational applications

The 'tailored' logical approach to object oriented information taken here suggests new ways of looking at other tasks that use object oriented information structures and benefit from logical systems. An example of a new task is the use of inductive logic for *data mining* purposes. In data mining one processes complex structured data (often modeled in OO languages like UML) and benefits from inductive reasoning models; i.e. from logic.

In [HaasAdriaans99] we proposed a framework for data mining algorithms based on the logic of categorial graphs. The research was initiated by observing interesting connections between data mining, inductive logic programming and grammar induction ([AdriaansHaas99]). The logic of categorial graphs facilitates the design of efficient Data mining algorithms using techniques from the field of inductive logic programming. This framework is based on the following idea:

The processing of more complex structured data calls for new algorithms. The inductive algorithms used for data mining on tabular (relational) structures are theoretically impaired with the computational complexity of full first order predicate logic. This is a problematic issue if one designs algorithms for induction on more complex structures than flat relations. This observation suggests that we need to found the structures on which we do data mining on a system with much

better computational properties. This complies with the broader trend in logic in computer science that suggests that one needs to find logics tailored more closely to the application domain than a general language as full first order predicate logic.

Complex data in modern information technology practice is often written down in the industry standard language UML. A logic that effectively bears the main features of object oriented languages like UML seems ideal for the task to support the design of inductive algorithms for finding patterns, because it has both

1. the ability to express complex patterns in complex data structures,

2. and a calculus that enables one to perform correct induction steps on discovered patterns.

A strongly related[6] new task that could benefit from our approach is the task of *learning from a logical perspective*. There is a strong relationship between a learning strategy, its formal learning framework and its logical representational theory. This relationship enables one to translate learnability results from one theory to another. Moreover if we go from a classical logic theory to a more specialized logic like the logic of categorial graphs or a substructural logic theory, we can transform learnability results of logical concepts to results for information models or string languages. In [AdriaansHaas00] we demonstrated such a translation by transforming the Valiant learnability result for Boolean concepts to a learnability result for a class of string pattern languages.

### 5.3.3 Logical and philosophical repercussions

In the light of the logical context the system for categorial graphs also puts forward some interesting questions for logical research. These questions will be asked in the next chapter. Furthermore the clear-cut analysis of the domain, involving 'objects' and 'partial descriptions', have surprisingly tight connections to issues in philosophy. When one realizes that information system modeling is a task of accurately and precisely modeling a part of reality, using basic notions like 'object', 'property' and 'description', applications in philosophy are certainly possible. A formal theory of information can provide a way to explicify issues in philosophical considerations or debate. We will show such an application in chapter 7.

[6] Data mining is learning patterns from data.

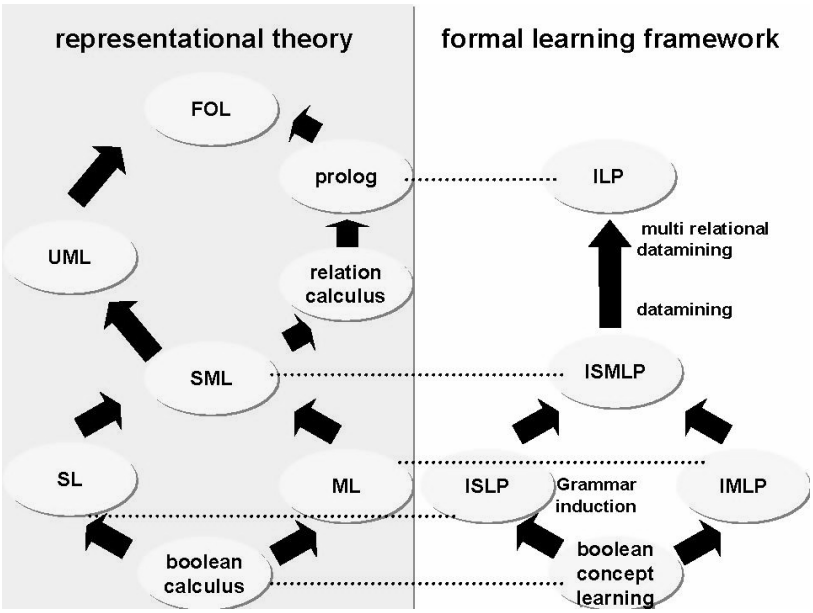



Figure 5.2: Research context where SL = substructural logic, ML = modal logic, SML = substructural modal logic, ISLP = inductive substructural logic programming, IMLP = inductive model logic programming, ISMLP = inductive substructural modal logic programming.

## 5.4 Summary

In this chapter we sketched the research domains that we are touching in the analysis of object orientation. We have mentioned research on computer systems semantics and mathematical logic. In the following chapter we will analyze the logic of categorial graphs exploiting the logical roots that were pointed out here.

# Chapter 6

# Logic of object oriented information

*1.1 Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge.*
*2.034 Die Struktur der Tatsache besteht aus den Strukturen der Sachverhalte*
*2.04 Die Gesamtheit der bestehenden Sachverhalte ist die Welt*

*(Ludwig Wittgenstein, Tractatus logico-philosophicus)*

The informal practice of object-oriented information systems has been analyzed in depth in Chapters 3 and 4. This culminated in our eventual proposal for precise mathematical 'discourse models'. These came with two languages streamlining the usual object-oriented discourse: a mixed graphical-symbolic one of categorial graphs and its more traditional 'meta-language', both reflecting the key features of adjacency, parts and products. Moreover, we gave a substructural calculus of resource-conscious proof rules that fits informal reasoning about specified constraints on objects and their properties. This package of formal languages, semantic structures, and proof calculi may be viewed on its own as our proposal for a 'precisified practice' in the area. But also, in Chapter 5, we pointed out how such a model can serve as a basis for new tasks, not yet studied systematically in the literature, such as *learning* resource-sensitive object-oriented structures. In this sixth chapter, however, we want to use our model in another mode, namely, for analyzing the logical properties of object oriented programming practice. In particular, we will show how the framework that we have developed fits with a number of independent developments in categorial and modal logic. For this purpose, we develop a new modal 'streamlined version' of the earlier systems which facilitates the comparison. This correspondence allows us to draw some precise conclusions about expressiveness, axiomatization, and complexity of the object-oriented paradigm. But also conversely, the resulting system presents some interesting challenges to modal and categorial logicians, as we shall point out in due course.

As usual, any logical analysis involves two mathematical decisions, which we repeat at the outset:

1. Which *structures* do we use to capture the items from the real world that are at stake?

2. Which *languages* do we use to talk about these items?

The following two sections state our answers, being a recapitulation of what we did in the preceding chapters, but with some new twists.

## 6.1 Models for object-oriented information systems

Chapters 3 and 4 gave a semi-formal, but rigorous, description of practically plausible object-oriented models and their basic structure. We now sharpen this up in two ways:

- define an *intended model*, which is a fixation of the exact structure we would like to study; the archetype of the concrete models of chapter 4.

- define *abstract models*, which are abstract versions of the former, providing greater generality and easier access for logical theorizing.

### 6.1.1 Intended models

The 'discourse models' of chapter 4 were meant to directly describe the semantic intuitions behind object oriented information systems. Now we give a *concrete* definition for logical working purposes.

**6.1.1.** Definition. (Intended model) There are two components:

1. One domain with a set of 'whole objects' $E$ , and products of whole objects ('aggregates') in set, multiset, and list flavors.

2. Another family of domains $\{A_e | e \in E\}$, consisting of 'partial descriptions' and their products.

These two domains are related as follows:

1. a 'structure adjacency relation' $f_R$ relates whole objects to their partial descriptions[1]

[1] In the concrete model, the partial descriptions are precisely the substructures of the total adjacency structure.

2. the 'extension relation' $\mathsf{ext}$ relates a partial description to other 'more informative' partial descriptions, and whole objects.

More precisely, an intended model gives us the following:

1. *Whole objects*: Atomic objects $E^{\mathrm{At}}$ and all the sets, multisets, and lists of atomic objects (seen as various sorts of aggregate objects) together with the appropriate 'aggregation' operations $*$ (set union[2], multiset union and list concatenation), and the 'substructure' relations $\leq$ (subset[3], submultiset, sublist) for the different flavors. *Thus we allow all three options at the same time, in one multi-sorted domain of objects with partially defined product operations.* This domain of whole objects is denoted by $E$.

2. *Partial descriptions*: For each $e \in E$, we have a domain of partial descriptions $A_e = e \times E$ (the 'labeled object domain'), again with its aggregation operations[4] $*_e$ and substructure relations[5] $\leq_e$.

3. *Adjacency mapping*: A function $f_R : E \mapsto \bigcup_{e \in E} A_e$ such that $f_R(e) \in A_e$, which maps each whole object to its largest partial description.

4. *Extendibility relation*: $\mathsf{Ext} : E \cup \bigcup_{e \in E} A_e \mapsto \mathcal{P}(E \cup \bigcup_{e \in E} A_e)$ where $\mathsf{Ext}(e) = \{e\}$ for all $e \in E$ and $\mathsf{Ext}(a) = \{b | a \leq_e b \leq_e f_R(e)\} \cup \{e\}$ for all $a \in A_e$ where $a \leq_e f_R(e)$.

$\triangle$

An attentive reader may have noted that we left out the empty object '1' and the empty descriptions '$(e, 1)$' that were present in the models for categorial graphs in the original analysis in chapter 4. These empty entities were introduced in the algebraic semantics for categorial graphs for convenience. It enabled us to take monoids as basic structures, and therfore we were able to elegantly express the algebraic properties. As we mentioned at its introduction in chapter 3, the empty edge is not a necessary artifact, even though we could give it a proper meaning. The models in this chapter have a more minimal nature, to enable the analysis of the object oriented structures. We could introduce the empty edge here without a problem, but then we will need some additonal rules and axioms to enforce its behavior, while we actually are not very interested in this artifact for the study of object orientation. An artifact that is convenient in one setting (the algebraic) may prove to be a nuisance in another setting (the logical).

[2] In our notation, set union will be $*^{\mathrm{set}}$, multiset union $*^{\mathrm{multiset}}$ and list concatenation $*^{\mathrm{list}}$. Using the $*$ by itself, we abstract over the choice of structure type.

[3] As notation then, subset will be $\leq^{\mathrm{set}}$, submultiset $\leq^{\mathrm{multiset}}$ and sublist $\leq^{\mathrm{list}}$. By using $\leq$ we abstract over the choice of structure type.

[4] Again, we actually have several: $*_e^{\mathrm{set}}$, $*_e^{\mathrm{multiset}}$, $*_e^{\mathrm{list}}$.

[5] And once more, we have several: $\leq_e^{\mathrm{set}}$, $\leq_e^{\mathrm{multiset}}$, $\leq_e^{\mathrm{list}}$.

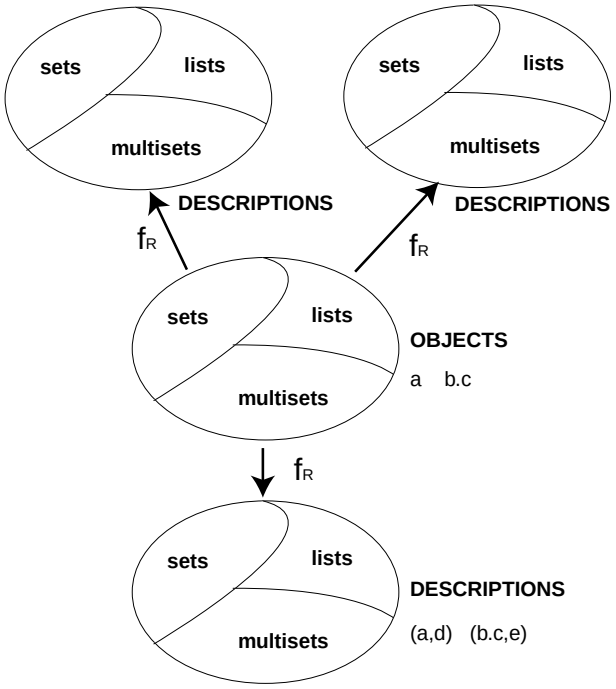



Figure 6.1: Topology of the intended model

### 6.1.2 Abstract models

Next, we take a more abstract viewpoint, highlighting what we see as the essential features of the preceding models. In particular, these are: the use of two kinds of entities: whole objects and partial descriptions, each with their own product operations, but living in harmony through suitable 'connection relations'. Getting a bit ahead of things, what follows are typical relational models for *modal logic*, satisfying some suitable constraint:

**6.1.2.** Definition. (Abstract model)

1. First, we have a domain $U$
2. Then there are two families of ternary relations $\mathcal{Q}^E, \mathcal{Q}^A$, where each family consists of three relations $Q^{\mathrm{set}X}, Q^{\mathrm{multiset}X}, Q^{\mathrm{list}X}$. When we abstract over the precise choice of $Q^X$, we sometimes write $Q^{\mathrm{struct}X}$.
3. Finally, we have binary relations $R_1$, $R_2$, and $S$.

In compact form an abstract model $\mathcal{M}$ is written as follows:

$$\mathcal{M} :=< U, Q^{\mathrm{set}E}, Q^{\mathrm{set}A}, Q^{\mathrm{multiset}E}, Q^{\mathrm{multiset}A}, Q^{\mathrm{list}E}, Q^{\mathrm{list}A}, R_1, R_2, S >$$

△

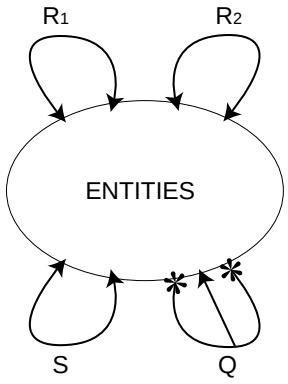



Figure 6.2: Topology of the abstract model

The motivating interpretation for this similarity type is the following. We have objects and descriptions living together in the total domain, each with ternary 'composition relations' of the form '$aQbc$' : $a$ is a composition (of one of our various sorts) of $b$ and $c$, in that order. This ternary style makes sure that the product operations are not necessarily total (relieving us of the duty to interpret every weird 'hybrid product') while also remaining uncommitted on the issue of whether forming aggregates is a single-valued partial function, rather than a multi-valued one. Moreover, 'communication' is arranged as follows. Objects are related to their descriptions by the relation $R_1$, while in the opposite direction, $R_2$ takes descriptions back to the objects figuring in them, either as the 'leading object' or as another one involved in the relevant property of the former. Finally, $S$ is the extension relation, which is characteristic for (after all) *partial* descriptions, but allowing 'culmination' in the 'main object' of the given description.

As usual, the move toward abstract models is not just a trick to make the theoretician's life easier. The above format may also suggest new applications, and new ways of looking at 'object oriented information systems'. In this chapter, we gain two things by working this way:

1. a clear view of the semantics for categorial graphs,

2. transfer of results from (and to) established modal logic.

### 6.1.3 Representation

What is needed to represent an abstract model as a concrete one? On these abstract models we will formulate constraints such that the abstract models have (much of) the same characteristics as a concrete model. In the abstract models the elements and relations are abstract, but 'secretly' we imagine them to have some intended meaning. The domain $U$ contains, in our intended meaning, both

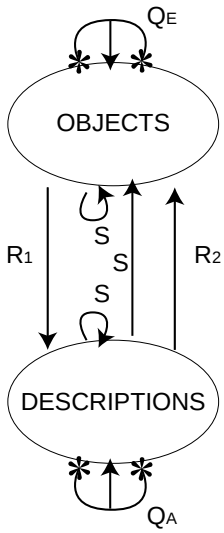



Figure 6.3: Target topology of the abstract model when constraints and characterizing formulas are defined

the whole objects and the partial descriptions. When enough structure is given in the abstract model we will be able to distinguish whole objects from partial descriptions. In the abstract view, when no characterizations are given, we just say our abstract domain contains elements. At the start of our analysis all the relations are also without any restrictions, but we will further on constrain relations in $\mathcal{Q}^E$ to model the aggregation (product) on whole objects (separate relations for sets, multisets, and lists) and the relations in $\mathcal{Q}^A$ to model aggregation on partial descriptions (also separate relations for sets, multisets, and lists). Furthermore $R_1$ will model adjacency relating whole objects to its partial descriptions; $R_2$ will model the relation between a partial object and the whole object that takes part in the adjacency. Finally $S$ will model extendibility. In the coming sections we will encounter the constraints that make all this happen.

## 6.2 Modal languages

To talk about the objects, the partial descriptions, and their interactions, we start with a modal language. The modal language is a slight extension to the meta language of categorial graphs in three ways:

- the *adjacency* modality is split up in *two* modalities, the first relating an object to its partial description, and the second relating the partial description to the actual adjacent object

- the *aggregation* modality will be accompanied by two additional modalities that look at the aggregation from the two other possible perspectives. More-

over we will have separate aggregation modalities for the different structures 'lists', 'multisets', and 'sets' and the different entities 'objects' and 'partial descriptions'.

- The notion of *extendibility* in the model will get a modality in the language to enable us to talk about it.

In the modal language we will omit the constant **1** for the empty object, for the reason we explained above.

The relations of the abstract model from above are now interpreted by the following modalities:

- unary $\Diamond_1, \Diamond_1^{\cup}, \Diamond_2, \Diamond_2^{\cup}$ modalities for the adjacency relations
- unary $\circ, \circ^{\cup}$ modalities for the extendibility relation
- binary modalities $*_1^{\mathrm{list}E}, *_1^{\mathrm{multiset}E}, *_1^{\mathrm{set}E}, *_2^{\mathrm{list}E}, *_2^{\mathrm{multiset}E}, *_2^{\mathrm{set}E}$ $, *_3^{\mathrm{list}E}, *_3^{\mathrm{multiset}E}, *_3^{\mathrm{set}E}$ for the aggregations (products) on whole objects (for all types of structures three for the 'triple view' of a ternary relation)
- binary modalities $*_1^{\mathrm{list}A}, *_1^{\mathrm{multiset}A}, *_1^{\mathrm{set}A}, *_2^{\mathrm{list}A}, *_2^{\mathrm{multiset}A}, *_2^{\mathrm{set}A}$ $, *_3^{\mathrm{list}A}, *_3^{\mathrm{multiset}A}, *_3^{\mathrm{set}A}$ for the aggregations (products) on partial descriptions (for all types of structures three for the 'triple view' of a ternary relation)

The modalities will be interpreted by the relations in the abstract model. The modalities embedded in a modal language will enable us to talk about complex objects, precisely like we did with the meta language for categorial graphs.

### 6.2.1 Definition language and semantics

**6.2.1.** Definition. (Modal language of categorial graphs) Let $\mathsf{Prop}$ be a set of propositional variables. Then we define the *modal language of categorial graphs* $L$ as follows:

- Boolean: $L = \mathsf{Prop}|L \sqcap L|L \sqcup L|\neg L$
- Adjacency: $\Diamond_1 L|\Diamond_1^{\cup} L|\Box_1 L|\Box_1^{\cup} L|\Diamond_2 L|\Diamond_2^{\cup} L|\Box_2 L|\Box_2^{\cup} L$
- Extension: $\circ L|\circ^{\cup} L$
- Aggregate on objects: $L = L *_1^{\mathrm{struct}E} L|L *_2^{structE} L|L *_3^{\mathrm{struct}E} L$ where $struct \in \{\mathsf{set}, \mathsf{multiset}, \mathsf{list}\}$
- Aggregate on partial descriptions: $L = L *_1^{\mathrm{struct}A} L|L *_2^{structA} L|L *_3^{\mathrm{struct}A} L$ where $struct \in \{\mathsf{set}, \mathsf{multiset}, \mathsf{list}\}$

△

The modal language of categorial graphs is interpreted in the abstract model as follows

**6.2.2.** Definition. (Semantics) Let $V : \text{Prop} \mapsto 2^U$ be a valuation that assigns subsets of $U$ to atomic symbols. Let $P \in \mathsf{Prop}$ and $A, B \in L$. Furthermore let $s, t, u \in U$. Then abstract model $\mathcal{M}$ interprets $L$ as follows:

- Boolean (standard as in section 5.2, but given here for completeness):

$$\begin{array}{lll} \mathcal{M}, s \models P & \text{iff} & s \in V(P) \\ \mathcal{M}, s \models A \sqcap B & \text{iff} & \mathcal{M}, s \models A \text{ and } \mathcal{M}, s \models B \\ \mathcal{M}, s \models A \sqcup B & \text{iff} & \mathcal{M}, s \models A \text{ or } \mathcal{M}, s \models B \\ \mathcal{M}, s \models \neg A & \text{iff} & \mathcal{M}, s \not\models A \end{array}$$

- Adjacency (interpreted by $R_1$ and $R_2$):

$$\begin{array}{lll} \mathcal{M}, s \models \Diamond_1 A & \text{iff} & \exists t(sR_1t \text{ and } \mathcal{M}, t \models A) \\ \mathcal{M}, s \models \Diamond_1^{\cup} A & \text{iff} & \exists t(tR_1s \text{ and } \mathcal{M}, t \models A) \\ \mathcal{M}, s \models \Box_1 A & \text{iff} & \mathcal{M}, s \models \neg\Diamond_1\neg A \\ \mathcal{M}, s \models \Box_1^{\cup} A & \text{iff} & \mathcal{M}, s \models \neg\Diamond_1^{\cup}\neg A \\ \mathcal{M}, s \models \Diamond_2 A & \text{iff} & \exists t(sR_2t \text{ and } \mathcal{M}, t \models A) \\ \mathcal{M}, s \models \Diamond_2^{\cup} A & \text{iff} & \exists t(tR_2s \text{ and } \mathcal{M}, t \models A) \\ \mathcal{M}, s \models \Box_2 A & \text{iff} & \mathcal{M}, s \models \neg\Diamond_2\neg A \\ \mathcal{M}, s \models \Box_2^{\cup} A & \text{iff} & \mathcal{M}, s \models \neg\Diamond_2^{\cup}\neg A \end{array}$$

- Extension (interpreted by $S$):

$$\begin{array}{lll} \mathcal{M}, s \models \circ A & \text{iff} & \exists t(sSt \text{ and } \mathcal{M}, t \models A) \\ \mathcal{M}, s \models \circ^{\cup} A & \text{iff} & \exists t(tSs \text{ and } \mathcal{M}, t \models A) \end{array}$$

- Aggregate on objects (interpreted by the relations in $\mathcal{Q}^E$):

$$\begin{array}{lll} \mathcal{M}, s \models A *_1^{\text{struct}E} B & \text{iff} & \exists tu(sQ^{\text{struct}E}tu \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \\ \mathcal{M}, s \models A *_2^{\text{struct}E} B & \text{iff} & \exists tu(tQ^{\text{struct}E}su \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \\ \mathcal{M}, s \models A *_3^{\text{struct}E} B & \text{iff} & \exists tu(tQ^{\text{struct}E}us \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \end{array}$$

- Aggregate on partial descriptions (interpreted by the relations in $\mathcal{Q}^A$):

$$\begin{array}{lll} \mathcal{M}, s \models A *_1^{\text{struct}A} B & \text{iff} & \exists tu(sQ^{\text{struct}A}tu \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \\ \mathcal{M}, s \models A *_2^{\text{struct}A} B & \text{iff} & \exists tu(tQ^{\text{struct}A}su \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \\ \mathcal{M}, s \models A *_3^{\text{struct}A} B & \text{iff} & \exists tu(tQ^{\text{struct}A}us \text{ and } \mathcal{M}, t \models A \text{ and } \mathcal{M}, u \models B) \end{array}$$

△

**6.2.3.** Example. Recall the running example of chapter 3 (example 3.1.1) and its formulation in the meta language for categorial graphs of example 4.2.2. As a brief illustration, we show how to express the basic graph expressions in the modal logic for categorial:

$$\begin{array}{l} \mathsf{pilot} \Rightarrow \Diamond_1\Diamond_2\mathsf{name} *_1 \Diamond_1\Diamond_2\mathsf{empno} *_1 \Diamond_1\Diamond_2\mathsf{qualif} \\ \mathsf{pilot} \Rightarrow \Diamond_1\Diamond_2\mathsf{roster} \end{array}$$

In the sections below we will first define a logic that handles adjacency, a logic that handles extendibility, and a logic that handles aggregates, before we present a combined system that can talk about the combination of all those things. Such a logic consists of a language (a fragment of the language from above), a model (a part of the abstract model from above), an interpretation of the logical connectives and modalities, and a collection of axioms and rules that characterize certain constraints on the relations of the abstract model.

### 6.2.2 Adjacency logics

In the logic of adjacents we focus on a very important feature of a language for object oriented information systems: the ability to construct complex objects using the adjacency operation. In the logic of adjacents we analyze two types of interactions between whole objects and partial descriptions. Firstly we will be able to say things about *having* adjacents; i.e. we need to relate an object to the partial description that witnesses that this object has some other object as an adjacent. Secondly we need means to say that an object *is* an adjacent of some other object; i.e. when an object takes part in a partial description.

The two matters of adjacency are expressed using two unary modalities $\Diamond_1$ and $\Diamond_2$, and are interpreted by the relations $R_1$ and $R_2$ in the abstract model. The $R_1$ relation here relates a whole object to the partial descriptions that describe the fact of *having* some adjacents. The $R_2$ relation relates a partial description to the object that *is* the adjacent. In other words, $R_2$ intends to express that a partial description involves some object. This is a direct concept of the object oriented paradigm we have already seen in chapters 3 and 4 that says that all the entities can be looked at as objects in their own right.

Note that here we directly look at the interpretation of an adjacent to its partial description, and do not model the adjacency via an adjacency relation that relates whole objects to whole objects. The need for considering partial descriptions was discovered when formulating the concrete models in chapter 4.

For example, consider object $a$ and partial description of $a$ called $b$ that is the witness of $a$ having as adjacent object $c$. Then $aR_1b$ relates the $a$ object to its $b$ partial description and $bR_2c$ relates partial description $b$ to the object $c$ that is the content of the partial description of $a$.

There is also another possible view on the adjacency relations, the *inverse* of the relations $R_1$ and $R_2$. This view also provides some nice means of expression. One could say interesting things about the partial descriptions; for example 'the partial description $b$ partially describes object $a$'.

Note that we cannot choose $R_1$ and $R_2$ arbitrarily. The relation modeling adjacency should satisfy some constraints to comply with the intended meaning[6]. These constraints are as follows:

1. An object can have a partial description, but a partial description in turn, does not have a partial description anymore
2. A partial description is a witness to an object being an adjacent. An object itself, however cannot be such a witness
3. Sets of objects and partial descriptions are disjoint
4. The universe contains no other entities then partial descriptions and objects with partial descriptions
5. A partial description is always a partial description of an object

In other words the relations, $R_1$ and $R_2$ are at a maximum one step deep; the collections of partial descriptions and objects are disjoint and exhaustive, and all partial descriptions must be related to whole objects by both $R_1$ and $R_2$; i.e.

**6.2.4.** Definition. (Constraints for the adjacency logic)

1. $\forall a(\exists b\, aR_1b \;\rightarrow\; not\; \exists c\, bR_1c)$
2. $\forall a(\exists b\, aR_2b \;\rightarrow\; not\; \exists c\, bR_2c)$
3. $\forall a \;\; \nexists b, c\; (aR_1b \text{ and } aR_2c)$
4. $\forall a\; \exists b, c\; (aR_1b \text{ or } aR_2c)$
5. $\forall a(\exists b\, aR_2b \;\rightarrow\; \exists c\, cR_1a)$

△

Note that having an adjacent (i.e. being the left side of an $R_1$ relation) is characteristic for a whole object, and being a witness of an adjacent (i.e. being the left side of an $R_2$ relation) is characteristic for a description. Constraint 4 says that all objects have adjacents (remember the possibility for introducing an empty description!) and that all descriptions are witnesses to an object being an adjacent (remember the possibility for introducing empty object!).

The language for the adjacency logic becomes the following:

[6]Note that if we would construct a model with a domain with only whole objects and a relation that relates whole object to the whole objects that take part in the partial description. Then we would not have any necessary constraint on the single adjacency relation $R$ and would come up with the modal logic $\mathsf{K}$. We could, though, optionally put some constraints like irreflexivity or acyclicity or foundedness on $R$, with their well known rules in the logic. We would however not be able to express things involving the partial descriptions like the structure of the adjacency.

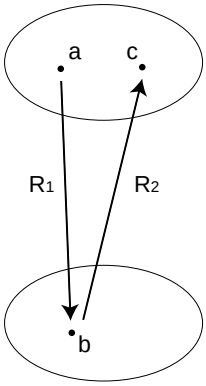



Figure 6.4: object $a$ with partial description $b$ witnessing that object $c$ is actually adjacent to $a$.

**6.2.5.** Definition. (Language for the adjacency logic)

$$L_I := \mathsf{Prop}|L_I \sqcap L_I|\neg L_I|\Diamond_1 L_I|\Diamond_2 L_I|\Diamond_1^{\cup} L_I|\Diamond_2^{\cup}|\Box_1 L_I|\Box_2 L_I|\Box_1^{\cup} L_I|\Box_2^{\cup} L_I$$

△

We can now interpret the adjacency modalities, their inverse modalities and all their dual modalities in the abstract model as indicated:

**6.2.6.** Definition. (Semantics for the adjacency logic)

$$\begin{array}{l} M, s \models \Diamond_1 A \text{ iff } \exists t(sR_1t \& M, t \models A) \\ M, s \models \Diamond_2 A \text{ iff } \exists t(sR_2t \& M, t \models A) \\ M, s \models \Diamond_1^{\cup} A \text{ iff } \exists t(tR_1s \& M, t \models A) \\ M, s \models \Diamond_2^{\cup} A \text{ iff } \exists t(tR_2s \& M, t \models A) \end{array}$$

As in standard modal logic, we introduce the dual operator of the $\Diamond_i^x$ by setting $\Box_i^x := \neg\Diamond_i^x\neg$. △

The modal logic for adjacency will contain at least the axioms and rules of the minimal modal logic $\mathsf{K}$ added with the principles that are needed to force the inverse modalities to be interpreted by the inverse relation. The most interesting principles come from the constraints we put on the relations $R_1$ and $R_2$ in the abstract model to force behavior that is similar to that of the concrete models.

**6.2.7.** Definition. (Axiomatics for the adjacency logic) First we construct the axioms and rules for the minimal logic for the adjacency modalities:

- all axioms and rules of propositional logic for the logic with $\sqcap$, $\sqcup$ and $\neg$

- rules for the modalities in the minimal modal logic

$$
\begin{array}{ll}
(\Diamond_1\text{Distribution}) & \Diamond_1(A \sqcup B) \rightarrow (\Diamond_1 A \sqcup \Diamond_1 B) \\
(\Diamond_2\text{Distribution}) & \Diamond_2(A \sqcup B) \rightarrow (\Diamond_2 A \sqcup \Diamond_2 B) \\
(\Diamond_1^{\cup}\text{Distribution}) & \Diamond_1^{\cup}(A \sqcup B) \rightarrow (\Diamond_1^{\cup} A \sqcup \Diamond_1^{\cup} B) \\
(\Diamond_2^{\cup}\text{Distribution}) & \Diamond_2^{\cup}(A \sqcup B) \rightarrow (\Diamond_2^{\cup} A \sqcup \Diamond_2^{\cup} B) \\
(\Diamond_1\text{Necessation}) & \dfrac{\neg A}{\neg\Diamond_1 A} \\
(\Diamond_2\text{Necessation}) & \dfrac{\neg A}{\neg\Diamond_2 A} \\
(\Diamond_1^{\cup}\text{Necessation}) & \dfrac{\neg A}{\neg\Diamond_1^{\cup} A} \\
(\Diamond_2^{\cup}\text{Necessation}) & \dfrac{\neg A}{\neg\Diamond_2^{\cup} A}
\end{array}
$$

- then we need to add the rules to relate the modalities to their inverse modalities:

$$
\begin{array}{ll}
(\Diamond_1\text{Inverse}) & A \rightarrow \Box_1\Diamond_1^{\cup} A \\
(\Diamond_2\text{Inverse}) & A \rightarrow \Box_2\Diamond_2^{\cup} A
\end{array}
$$

- finally we need to add rules and axioms that reflect constraints on the adjacency relations. In the adjacency logic this amounts to the following principles:

$$
\begin{array}{ll}
(\Diamond_1\text{Step}) & \Box_1\Box_1\bot \\
(\Diamond_2\text{Step}) & \Box_2\Box_2\bot \\
(\text{Disjoint}) & \neg(\Diamond_1\top \sqcap \Diamond_2\top) \\
(\text{Exhaustive}) & \Diamond_1\top \sqcup \Diamond_2\top \\
(\text{Description Requires Object}) & \Diamond_2\top \rightarrow \Diamond_1^{\cup}\bot
\end{array}
$$

△

The adjacency logic enables us to say things about the domain that have to do with adjacency. Now we have appropriate means to express whether an object is a whole object, or whether an object is a partial description:

- ▶ an entity $x$ is a whole object iff $x$ satisfies $\Box_2\bot$
- ▶ an entity $x$ is a whole object iff $x$ satisfies $\Diamond_1\top$
- ▶ an entity $x$ is a partial description iff $x$ satisfies $\Box_1\bot$
- ▶ an entity $x$ is a partial description iff $x$ satisfies $\Diamond_2\top$
- ▶ an entity $x$ is a partial description iff $x$ satisfies $\Diamond_1^{\cup}\top$
- ▶ an entity $x$ is a partial description of an $A-$ object iff x satisfies $\Diamond_1^{\cup}A$

Note that when we want to connect an object to another object that takes part in its adjacency structure, like in the original definition of the categorial graph meta language, we can simply traverse via a partial description. Suppose we want to say that an object $x$ has an $A$-adjacent object. We expressed this in original language by $\Diamond A$. In this more refined setting we will say $\Diamond_1\Diamond_2 A$. If we want to express constraints directly on the object adjacency relation (i.e. treat it as a normal relation) we can formulate the corresponding principles using the composed $\Diamond_1\Diamond_2$ modality.

There are also some nice things to say from a modal logic perspective about the axiomatics of the adjacency logic. There are nice principles deducible from the system. For example: *it is necessary for an object to have an adjacent object via a partial description*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Diamond_1\top \sqcup \Diamond_2\top \text{ (Exhaustive)}}{\Box_1(\Diamond_1\top \sqcup \Diamond_2\top)}\ (\Box_1\text{Necessation})}{\Box_1(\neg\Diamond_1\top \to \Diamond_2\top)}}{\Box_1\neg\Diamond_1\top \to \Box_1\Diamond_2\top}\ (\Box_1\text{Distribution})}{\dfrac{\Box_1\Box_1\bot\ (\Diamond_1\text{Step}) \qquad \Box_1\Box_1\bot \to \Box_1\Diamond_2\top}{\Box_1\Diamond_2\top}\ (\text{modus ponens})}$$

Another observation is that there is only *one* principle that introduces an asymmetry between the objects and the partial descriptions. This principle is the *Description Requires Object* axiom. This axiom requires that each complete chain of objects and partial descriptions connected alternatively by $R_1$ and $R_2$ relations always start with an object. I.e.:

$$\circ \to_1 \circ \to_2 \circ \to_1 \circ \to_2 \cdots$$

is an admissible structure in a model for an adjacency structure, but the following is not:

$$\circ \to_2 \circ \to_1 \circ \to_2 \circ \to_1 \cdots$$

An other interesting observation is that we have taken a *positive existence* property in axioms '*Disjoint*' and '*Exhaustive*' to characterize that some entity in the model is an object or a description; namely respectively $\Diamond_1\top$ and $\Diamond_2\top$. An interesting alternative is to take the *negative existence* properties $\Box_2\bot$ and $\Box_1\bot$. The axioms then become:

$$\begin{array}{ll} \text{(Disjoint)} & \neg(\Box_2\bot \sqcap \Box_1\bot) \\ \text{(Exhaustive)} & \Box_2\bot \sqcup \Box_1\bot \end{array}$$

In the negative formulation we allow entities in the model that are not connected through $R_1$ or $R_2$. Such entities can be either an object or partial description (it actually does not matter which). A hybrid formulation is also a plausible

alternative; for example $\Box_2\bot$ to characterize objects and $\Diamond_2\top$ to characterize descriptions. In a more general setting this matter will be touched in the section on 'further logical considerations' below.

### 6.2.3 Extendibility logics

In the extendibility logic we isolated the extendibility operation that can talk about extending partial descriptions and objects to more (or equal) informative descriptions or (in the limit) the object itself.

With the categorial graphs of chapter 3 we defined a language that enables one to talk about object oriented information systems from the perspective objects. The analysis for the interpretation in chapter 4 motivated us to introduce partial descriptions and the extendibility relation. The reason that the extendibility found its way in the concrete model is because it models a natural intuition[7]. Coming to the conclusion that these are important features of models for object oriented information systems, it is only fair that we provide constructs for talking from the perspective of partial descriptions as well.

The extendibility logic should capture the constraints we want to put on the relation $S$ in the abstract model to enforce extendibility behavior. As we saw in chapter 4, the extendibility relation is reflexive, and transitive, as it related a description or object to itself and all the descriptions that are more informative, with in the limit the whole object itself. Reflexivity and transitivity are well known constraints in modal logic, and the minimal modal logic of S4 describes such a system. In the isolated case (where we have no adjacency and aggregation) the only intriguing constraint we put on $S$ is that it has a top element when we look at each element in the relation. This top element is the whole object that, in the limit, a proper partial description (and the object itself) can be extended to. i.e.

- A partial description $a$ is extendable to all partial descriptions that partially describe the whole object that $a$ partially describes, and $a$ is also extendable to the most informative description of this object, the object itself.

This means that each element in the $S$ relation relates to itself and to all objects in the $S$-chain above, and that each element relates eventually to a unique local top element; i.e. then:

[7]Note In fact we could have made up the concrete model with the adjacency alone. But then we would have lost that intuition.

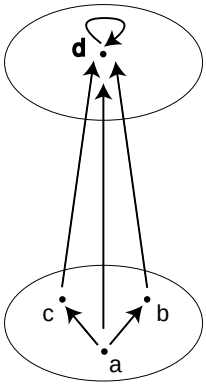



Figure 6.5: The constraints for extendibility logic

**6.2.8.** Definition. (Constraints for the extendibility logic)

| | |
|---|---|
| (reflexivity) | $\forall a\ aSa$ |
| (transitivity) | $\forall a, b, c(aSb\ \&\ bSc\ \rightarrow\ aSc)$ |
| (top) | $\forall a \exists b\ (aSb \& \forall c\ (aSc\ \rightarrow\ cSb))$ |
| (unique) | $\forall a, b, c[(aSb \& aSc)\ \rightarrow\ \exists d(bSd \& cSd)]$ |

△

The language for the extendibility logic becomes:

**6.2.9.** Definition. (Language for the extendibility logic)

$$L_{II} := \mathsf{Prop} | L_{II} \sqcap L_{II} | \neg L_{II} | \circ L_{II} | \circ^{\cup} L_{II}$$

△

We will abbreviate $\triangle := \neg \circ \neg$ to have convenient notation for the dual of $\circ$. The interpretation is:

**6.2.10.** Definition. (Semantics for the extendibility logic)

$$M, s \models \circ A \text{ iff } \exists t(sSt \& M, t \models A)$$

$$M, s \models \circ^{\cup} A \text{ iff } \exists t(tSs \& M, t \models A)$$

△

The 'top' constraint we have formulated for $S$ is exactly expressed by the so-called McKinsey axiom, while the 'unique' constraint is the constraint that enforces the well known Church-Rosser property. The accompanying logic for both constraints (together with the reflexivity and transitivity) are respectively the S4.1 and the S4.2 modal logics ([HughesCresswell68]). The logic for extendibility will therefore be an S4.1+2 system.

**6.2.11.** Definition. (Axiomatics for the extendibility logic)

- all axioms and rules of propositional logic for the logic with $\sqcap$, $\sqcup$ and $\neg$
- rules for the modalities in the minimal modal logic

$$\begin{array}{ll} (\circ\text{Distribution}) & \circ(A \sqcup B) \to (\circ A \sqcup \circ B) \\ (\circ^{\cup}\text{Distribution}) & \circ^{\cup}(A \sqcup B) \to (\circ^{\cup} A \sqcup \circ^{\cup} B) \\ (\circ\text{Necessation}) & \dfrac{\neg A}{\neg \circ A} \\ (\circ^{\cup}\text{Necessation}) & \dfrac{\neg A}{\neg \circ^{\cup} A} \end{array}$$

- then we need to add the rules to relate the modalities to their inverse modalities:

$$(\circ\text{Inverse}) A \to \neg \circ \neg \circ^{\cup} A$$

- finally we need to add the axioms that reflect constraints on the extendibility relation (reflexivity, transitivity and McKinsey axiom).

$$\begin{array}{ll} (\text{T}) & A \to \circ A \\ (4) & \circ \circ A \to \circ A \\ (\text{M}) & \triangle \circ A \to \circ \triangle A \\ (\text{C}) & \circ \triangle A \to \triangle \circ A \end{array}$$

$\triangle$

It is common knowledge in the field of modal logic that when we introduce the axioms for reflexivity ($T$) and transitivity (4) for the modality $\circ$, then transitivity and reflexivity can be deduced for its inverse modality $\circ^{\cup}$. Thus we do not need to repeat the axioms $T$ and 4 for $\circ^{\cup}$. The extendibility modality enables one to say important things about partial descriptions and objects; e.g.

- ▶ an entity $x$ is a description of an existing object (or an object itself) iff $x$ satisfies $\circ \top$
- ▶ an entity $x$ is a description of an $A$ object (or an $A$ object itself) iff $x$ satisfies $\circ A$

### 6.2.4 Aggregate logic

We have seen in the presentation of the language for categorial graphs that it is a core language element to talk about taking things together. This is called *aggregation*. In the aggregate logic we will shape the domain such that one can talk about aggregates. In other words we will be able to say things about things taken together. The structures of things taken together can be a number of things, among which are sets, multisets, and lists.

In our abstract model, the $Q$ relations model the aggregation. We do not constrain the behavior of the $Q$ relations yet. Curiously, this is not necessary because most things we want to say with a $*$ operation about taking things together do correctly coincide with the minimal intuition we could have about such a $*$ operation interpreted by a relation $Q$. Things will get more complicated when we start to require behavior of the different structures that are the result of the aggregation: sets, multisets, and lists. Then we need constraints that force associativity, multiplicity and order when the language enables us to say things about these matters. In the analysis these constraints on the abstract model will be optional.

For the aggregation we have different relations to interpret the different aggregation operations. We have separate families of relations for aggregations of objects ($\mathcal{Q}^E$) and aggregations of partial descriptions ($\mathcal{Q}^A$); and then for each type of structure, set, multiset and list again separate relations within each family ($Q^{\mathrm{set}E}$, $Q^{\mathrm{multiset}E}$, and $Q^{\mathrm{list}E}$ for $\mathcal{Q}^E$ and $Q^{\mathrm{set}A}$, $Q^{\mathrm{multiset}A}$, and $Q^{\mathrm{list}A}$ for $\mathcal{Q}^A$)

**6.2.12.** Definition. (Optional constraints for the aggregate logic) Let $Q$ be an arbitrary aggregation relation, then we can formulate the following constraint on $Q$ forcing associativity:

$$\text{(associativity) } \forall x\ (\exists y(xQyc\ \&\ yQab) \leftrightarrow \exists y'\ (xQay'\ \&\ y'Qbc))$$

Furthermore, we have constraints that are specific for certain structures, the sets, multisets, or lists. We already saw these constraints in the concrete models in chapter 4.

| | |
|---|---|
| (idem-consumption/cloning) | $\forall a\ aQaa$ |
| (commutativity) | $\forall abc(aQbc \rightarrow aQcb)$ |

The $Q$-relations for sets ($Q^{\mathrm{set}E}$ and $Q^{\mathrm{set}A}$) need to satisfy (at least) both idem-consumption/cloning and commutativity. The $Q$-relations for multisets ($Q^{\mathrm{multiset}E}$ and $Q^{\mathrm{multiset}A}$) need to satisfy (at least) commutativity and the negation[8] of idem-consumption/cloning. Finally the $Q$-relations for lists ($Q^{\mathrm{list}E}$ and $Q^{\mathrm{list}A}$) need to satisfy (at least) both the negations of idem-consumption/cloning and the negation[9] of commutativity. △

[8]Note, however, that it is not always necessary to require the negation of idem-consumption/cloning. Not requiring idem-consumption/cloning is enough to worry about multiplicity. Only when one strictly needs multisets, the negation of idem-consumption/cloning should be included. Cf. trace theory where there is a concept of partial commutativity.

[9]Note, however, that it is not always necessary to require the negation of commutativity. Not requiring commutativity is enough to worry about order. Only when one strictly needs lists, the negation of commutativity should be included.

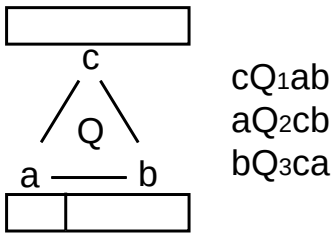



Figure 6.6: A category is the aggregation of two other categories.

Although there are a lot more constraints needed to force the behavior of sets (such that aggregation is set-union), or multisets (such that aggregation is multiset-union), or lists (such that aggregation is list concatenation), the constraints from above are powerful enough to analyze logical systems for object oriented information systems that have the ability to express things about arity (counting) and order; two abilities we consider basic for a language for information systems in our analysis.

Let us now turn to the language for the aggregation logic. We introduce (dyadic) modal operators to talk about the (ternary) aggregation relations. In general there are *three*[10] ways to talk about one ternary relation with a dyadic modal operator. We list the three cases for the aggregation relations:

1. you want to express that an object is the aggregation of two objects of a certain kind

2. you want to express that an object is the left part of an aggregation

3. you want to express that an object is the right part of an aggregation

The three modalities that express just these things form a 'versatile triple' and are studied in [Venema91] and [Benthem2000a]. A view like this is very relevant in the context of a graphical modeling language like the language of categorial graphs. When we depict a type (category) to be the aggregation of two other types (categories) we can visually 'see' all three ways to express a property for an aggregation (see figure 6.6). So it is only fair that we give the ability to express these different ways in the logic that is the meta language in which we are able to express all the things from the graphical language.

The language for adjacency logic now will look as follows:

[10]Compare this with the *two* ways to talk about a binary relation with a monadic modal operator: the modality that traverses the relation from left to right, and its inverse, that traverses a relation from right to left.

**6.2.13.** Definition. (Language for the aggregate logic)

$$\begin{array}{lll} L_{III} & := & \mathsf{Prop}|L_{III} \sqcap L_{III}|L_{III} \sqcup L_{III}|\neg L_{III}| \\ & & L_{III} *_1^{\mathsf{struct}X} L_{III}|L_{III} *_2^{\mathsf{struct}X} L_{III}|L_{III} *_3^{\mathsf{struct}X} L_{III} \\ \mathsf{struct} & := & \text{set}|\text{multiset}|\text{list} \\ X & := & E|A \end{array}$$

△

The interpretation for the three modalities for the product then are interpreted as follows:

**6.2.14.** Definition. (Semantics for the adjacency logic)

$$\begin{array}{lll} M,s \models A *_1^{\mathsf{struct}X} B & \text{iff} & \exists t,u(sQ^{\mathsf{struct}X}tu \;\&\; M,t \models A \;\&\; M,u \models B \\ M,s \models A *_2^{\mathsf{struct}X} B & \text{iff} & \exists t,u(tQ^{\mathsf{struct}X}su \;\&\; M,t \models A \;\&\; M,u \models B \\ M,s \models A *_3^{\mathsf{struct}X} B & \text{iff} & \exists t,u(tQ^{\mathsf{struct}X}us \;\&\; M,t \models A \;\&\; M,u \models B \end{array}$$

△

For a reader that is familiar with the Lambek calculus, it may be enlightening to remark that there is a tight connection between the operations here and the operations of the Lambek calculus ([Lambek58]). In the Lambek calculus we also have three dyadic operations: one '$*$' is the normal aggregation (i.e. like $*_1$), and two operations '$\backslash$' and '$/$' that are respectively 'left and right searching'. The latter $A/B$ expresses the fact that *if* an entity gets aggregated to the right side with an entity of some type $B$ they together (as an aggregate) form an object of some type $B$. It is even true that we can define the Lambek slashes with the operations of the versatile triple and vice versa. E.g.

$$A/B := \neg(\neg A *_2 B)$$

(i.e. it is not the case that I am aggregated to the right with a $B$ entity and together we form an entity that is not of type $A$)

$$A *_2 B := \neg(\neg A/B)$$

(i.e. it is not the case that I am an entity that when concatenated to the right with a $B$ entity, we form an entity that is not of type $A$)

For the axiomatics of the aggregate logic we need a minimal modal logic for all the dyadic operators, together with the principles that connects the triples of related modalities. Moreover, we need to find the rules for the constraints mentioned at the beginning of this section.

**6.2.15.** Definition. (Axiomatics for the aggregate logic)

- all axioms and rules of propositional logic for the logic with $\sqcap$, $\sqcup$ and $\neg$
- rules for the dyadic modalities in the minimal dyadic modal logic (for all $\mathsf{struct} \in \{\text{set}, \text{multiset}, \text{list}\}, X \in \{E, A\}, i \in \{1, 2, 3\}$)

| | |
|---|---|
| ($*_i^{\mathsf{struct}X}$LeftDistribution) | $(A \sqcup B) *_i^{\mathsf{struct}X} C \leftrightarrow A *_i^{\mathsf{struct}X} C \sqcup B *_i^{\mathsf{struct}X} C$ |
| ($*_i^{\mathsf{struct}X}$RightDistribution) | $(A *_i^{\mathsf{struct}X} (B \sqcup C \leftrightarrow A *_i^{\mathsf{struct}X} B \sqcup A *_i^{\mathsf{struct}X} C$ |
| ($*_i^{\mathsf{struct}X}$LeftNormal) | $\neg(\bot *_i^{\mathsf{struct}X} A)$ |
| ($*_i^{\mathsf{struct}X}$RightNormal) | $\neg(A *_i^{\mathsf{struct}X} \bot)$ |
| ($*_i^{\mathsf{struct}X}$LeftNecessation) | $\dfrac{\neg A}{\neg(A *_i^{\mathsf{struct}X} B}$ |
| ($*_i^{\mathsf{struct}X}$RightNecessation) | $\dfrac{\neg B}{\neg(A *_i^{\mathsf{struct}X} B}$ |

- then we need to add the rules to relate the modalities of each versatile triple. These are the axioms to ensure that the three modalities can be interpreted by one $Q$ relation from the different views. i.e. for all $\mathsf{struct} \in \{\text{set}, \text{multiset}, \text{list}\}, X \in \{E, A\}$:

| | |
|---|---|
| ($*_1^{\mathsf{struct}X}$ $*_2^{\mathsf{struct}X}$ coherence) | $A \sqcap (B *_1^{\mathsf{struct}X} C) \rightarrow (B \sqcap (A *_2^{\mathsf{struct}X} C)) *_1^{\mathsf{struct}X} C$ |
| ($*_1^{\mathsf{struct}X}$ $*_3^{\mathsf{struct}X}$ coherence) | $A \sqcap (B *_1^{\mathsf{struct}X} C) \rightarrow B *_1^{\mathsf{struct}X} (C \sqcap (A *_3^{\mathsf{struct}X} B))$ |
| ($*_2^{\mathsf{struct}X}$ $*_1^{\mathsf{struct}X}$ coherence) | $B \sqcap (A *_2^{\mathsf{struct}X} C) \rightarrow (A \sqcap (B *_1^{\mathsf{struct}X} C)) *_2^{\mathsf{struct}X} C$ |
| ($*_2^{\mathsf{struct}X}$ $*_3^{\mathsf{struct}X}$ coherence) | $B \sqcap (A *_2^{\mathsf{struct}X} C) \rightarrow A *_2^{\mathsf{struct}X} (C \sqcap (A *_3^{\mathsf{struct}X} B$ |
| ($*_3^{\mathsf{struct}X}$ $*_1^{\mathsf{struct}X}$ coherence) | $C \sqcap (A *_3^{\mathsf{struct}X} B) \rightarrow (A \sqcap (B *_1^{\mathsf{struct}X} C)) *_3^{\mathsf{struct}X} B$ |
| ($*_3^{\mathsf{struct}X}$ $*_2^{\mathsf{struct}X}$ coherence) | $C \sqcap (A *_3^{\mathsf{struct}X} B) \rightarrow A *_3^{\mathsf{struct}X} (B \sqcap (A *_2^{\mathsf{struct}X} C$ |

  This first coherence axiom relating $*_1$ to $*_2$ says that:

  - *when* an entity is of type $A$ and also ($\sqcap$) is an aggregation ($*_1$) of a $B$-entity and a $C$-entity, *then* this entity is an aggregation ($*_1$) of two entities:
    - one entity that is both a $B$-entity and ($\sqcap$) an entity that is the left-hand part of an aggregation ($*_2$) of $A$-entity that has a $C$-entity as the right-hand part
    - and another entity that is a $C$-entity

  The other axioms relating the modalities of the versatile triple can be explained similarly.

- finally we formulate the axioms that reflect the optional constraints on the aggregation relations.

| | |
|---|---|
| ($*_1^{\mathsf{struct}X}$associativity) | $A *_1^{\mathsf{struct}X} (B *_1^{\mathsf{struct}X} C) \leftrightarrow (A *_1^{\mathsf{struct}X} B) *_1^{\mathsf{struct}X} C$ |
| ($*_1^{\mathsf{struct}X}$cloning) | $A \rightarrow A *_1^{\mathsf{struct}X} A$ |
| ($*_1^{\mathsf{struct}X}$idem-consumption) | $A *_1^{\mathsf{struct}X} A \rightarrow A$ |
| ($*_1^{\mathsf{struct}X}$commutativity) | $A *_i^{\mathsf{struct}X} B \rightarrow B *_1^{\mathsf{struct}X} A$ |

These constraints clearly correspond to the constraints we listed above. For example the ($*_1^{\mathsf{struct}X}$commutativity) axiom says that *when* an object is an aggregation ($*_1$) of an $A$-entity and a $B$-entity *then* this object is also an aggregation ($*_1$) of a $B$-entity and an $A$-entity.

△

The language of the aggregate logic enables us to state a lot of powerful things about entities in our models. For example:

- atomicity: $x$ is atomic iff $x$ satisfies $\neg(\top *_1^{\mathsf{struct}X} \top)$
  (actually the conjunction of this formula for all the different types of aggregation; i.e. $\neg(\top *_1^{\mathrm{set}E} \top) \sqcap \neg(\top *_1^{\mathrm{multiset}E} \top) \sqcap$
  $\neg(\top *_1^{\mathrm{list}\ E} \top) \sqcap \neg(\top *_1^{\mathrm{set}A} \top) \sqcap \neg(\top *_1^{\mathrm{multiset}A} \top) \sqcap \neg(\top *_1^{\mathrm{list}\ A} \top)$ )
- membership: $x$ is a member of a $\mathsf{struct}$ structure iff $x$ satisfies
  $\top *_2^{\mathsf{struct}X} \top \sqcup \top *_3^{\mathsf{struct}X} \top$
- $x$ is a member of an $A-$ type $\mathsf{struct}$ structure iff $x$ satisfies
  $A *_2^{\mathsf{struct}X} \top \sqcup A *_3^{\mathsf{struct}X} \top$
- $x$ is a member of a structure with an $A-$ member iff $x$ satisfies
  $\top *_2^{\mathsf{struct}X} A \sqcup \top *_3^{\mathsf{struct}X} A$

When we have *exchange* for the structures then it suffices to say for the membership:

- $x$ is a member of a (multi)set iff x satisfies $\top *_2^{(\mathsf{multi})\mathsf{set}X} \top$
- $x$ is a member of an $A-$ (multi)set iff $x$ satisfies $A *_2^{(\mathsf{multi})\mathsf{set}X} \top$
- $x$ is a member of a structure with an $A-$ member iff x satisfies
  $\top *_2^{(\mathsf{multi})\mathsf{set}X} A$

The above examples show that we can express some interesting *local* properties of entities in our model. The logic is even strong enough to express 'global properties'.

- existential: $\top *_2^{\mathsf{struct}X} A$ holds when $A$ holds somewhere
  (Actually the disjunction of this formula for all the different types of aggregation; i.e. $(\top *_2^{\mathrm{set}E} A) \sqcup (\top *_2^{\mathrm{multiset}E} A) \sqcup (\top *_2^{\mathrm{list}\ E} A)$
  $\sqcup (\top *_2^{\mathrm{set}A} A) \sqcup (\top *_2^{\mathrm{multiset}A} A) \sqcup (\top *_2^{\mathrm{list}\ A} A)$ )
- universal: $\neg(\top *_2^{\mathsf{struct}X})\neg A$ holds when $A$ holds everywhere
  (Again actually the disjunction of all the different types of aggregation)

Note that the logic of aggregation is surprisingly expressive when we realize that once we have these global expressions we can express the powerful principle of *induction* ([Benthem2000a]). We have the possibility to talk about atomicity and do existential and universal statements. Now let us denote 'atomicity' by $\mathsf{At}$

($\mathsf{At} := \neg(\top *_1^{\mathsf{struct}X} \top)$, 'Existential' by $\mathsf{Ex}$ ($\mathsf{Ex} := \top *_2^{\mathsf{struct}X}$) and 'Universal' by $\mathsf{Un}$ ($\mathsf{Un} := \neg\mathsf{Ex}\neg$), then we can express induction as follows:

$$[\mathsf{Un}(\mathsf{At} \to A) \sqcap \mathsf{Un}(A *_1^{\mathsf{struct}X} A \to A)] \to \mathsf{Un}A$$

The universal and existential modalities are well known from modal logic and we get them here for free (i.e. without axiomatizing them, but expressing them using the aggregation modalities). An interesting exercise for instance would be to deduce the $\mathsf{S5}$ axioms for the defined existential and universal modalities from the earlier axioms for aggregation.

### 6.2.5 The combined system

In this system we combine all the features of the above logics. This means that the language contains all the operators mentioned above, and that the abstract model will inhabit all the constraints that the relations had in their isolated analysis. In the combined system we need to add the constraints and accompanying principles that regulate the *interaction* between the different subsystems. This means to regulate interaction between aggregation and adjacency, extendibility and aggregate and extendibility and adjacency.

Between aggregation and adjacency there are two important constraints that we also saw in chapter 4: *regularity* and *extentiality*.

Regularity says that the result object of taking two objects together should be consistent with the adjacents of the objects taken together. In a stronger version this constraint says that whenever we have objects $a$ and $b$ that have partial descriptions involving respectively objects $c$ and $d$, then the aggregation of $a$ and $b$, say $e$ should have partial descriptions that also involve objects $c$ and $d$.

Extentiality says that whenever an object has a partial description that is an aggregation of two other partial descriptions, then this object also has the two partial descriptions separately as partial descriptions.

**6.2.16.** Definition. (Constraints on the combined system for aggregation and adjacency)

| | |
|---|---|
| (weak regularity) | $aR_1f \;\&\; fR_2c \;\&\; bR_1g \;\&\; gR_2d \;\&\; eQ^{\mathsf{struct}E}ab \;\to$ $\exists h,i(eR_1h \;\&\; hR_2c \;\&\; eR_1i \;\&iR_2d)$ |
| (strong regularity) | $aR_1f \;\&\; fR_2c \;\&\; bR_1g \;\&\; gR_2d \;\&\; eQ^{\mathsf{struct}E}ab \;\to$ $\exists h,i,j(jQ^{\mathsf{struct}A}hi \;\&\; eR_1j \;\&\; hR_2c \;\&\; iR_2d)$ |
| (extentiality) | $eR_1j \;\&\; jQ^{\mathsf{struct}A}hi \;\to\; eR_1h \;\&\; eR_1i$ |

△

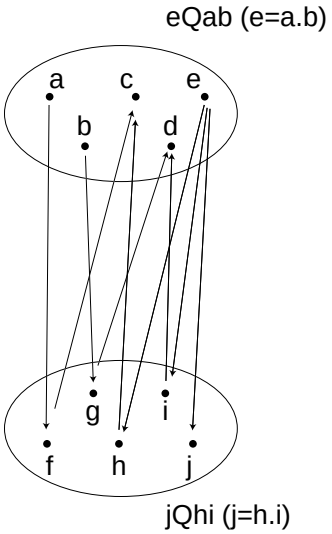



Figure 6.7: The constraints for aggregation and adjacency

The weak regularity just states the existence of the appropriate partial descriptions when objects are taken together. The strong regularity also states that these partial descriptions should be aggregated, and by that relating the product of objects to the product of partial descriptions via the adjacency. This strong version is needed when we deal with resource conscious structures like multisets and sets, because then we need to ensure that the multiplicity is handled appropriately by the partial descriptions as well. Moreover we observe that the constraints 'strong regularity' and 'extistentiality' clearly imply 'weak regularity', making it a redundant constraint (but it most directly translates to the textual -intuitively stated- regularity constraint).

Between the aggregation (product) of partial descriptions and the extendibility there is also some interaction. If an aggregation of two partial descriptions extends to some object, then it is a more informative description of the object than the partial descriptions it is an aggregation of. In general extendibility relates proper[11] descriptions to more (or equal) informative proper descriptions or the object itself. This means that if a partial description $a$ is an aggregation (product) of two partial descriptions $b$ and $c$; and $a$ extends to an entity (object or partial description) $d$, then also the partial descriptions $b$ and $c$ extend to $d$

**6.2.17.** Definition. (Constraints on the combined system for aggregation and extendability)

$$\text{(more informative extendibility) } aQ^{\mathsf{struct}^A}bc \,\&\, aSd \;\rightarrow\; bSd \,\&cSd$$

[11]Proper means that the description is really a description of an object, i.e. it is a substructure of the adjacency structure of the object or it is the object itself; and thus not some aggregation of partial descriptions that is overstating the properties of the object it intends to describe.

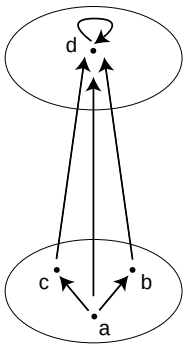



Figure 6.8: $a$, $b$, $c$, and $d$ (itself) all extend to $d$

△

Between Extendibility and adjacency there is the well expected constraint that says that if $b$ is a partial description of $a$ via the adjacency relation $R_1$, then $b$ is extendable to $a$.

**6.2.18.** Definition. (Constraints on the combined system for adjacency and extendibility)

$$\text{(extendable adjacents)} \quad aR_1b \to bSa$$

△

The language of the combined system includes the full set of modalities as we defined them in the beginning of this section (see definition 6.2.1). And also their interpretation is the one we started off with (see definition 6.2.2). The interesting question now is what are the axioms and rules relating to the combination of the various features.

**6.2.19.** Definition. (Axiomatics of the combined system)

- axioms and rules of the adjacency logic
- axioms and rules of the extendibility logic
- axioms and rules of the aggregate logic
- axioms and rules for combining adjacency and aggregation:

$$\begin{array}{ll}
\text{(weak regularity)} & A \sqcap \Diamond_1(F \sqcap \Diamond_2 C) *_1^{\mathsf{struct}E} \\
& \quad B \sqcap \Diamond_1(G \sqcap \Diamond_2 D) \to \Diamond_1\Diamond_2 C \sqcap \Diamond_1\Diamond_2 D \\
\text{(strong regularity)} & A \sqcap \Diamond_1(F \sqcap \Diamond_2 C) *_1^{\mathsf{struct}E} \\
& \quad B \sqcap \Diamond_1(G \sqcap \Diamond_2 D) \to \Diamond_1(\Diamond_2 C *_1^{\mathsf{struct}A} \Diamond_2 D) \\
(\Diamond_1\text{extentiality}) & \Diamond_1(F *_1^{\mathsf{struct}A} G \to \Diamond_1 F \sqcap \Diamond_1 G
\end{array}$$

- axioms and rules for combining aggregation and extendibility:

$$(*_2^{\mathsf{struct}A}\text{extendibility}) \quad \top *_2^{\mathsf{struct}A} \circ A \to \circ A$$
$$(*_3^{\mathsf{struct}A}\text{extendibility}) \quad \top *_3^{\mathsf{struct}A} \circ A \to \circ A$$

  To explain the correspondence between the axioms and the constraints, look again at figure 6.7 illustrating the constraints between aggregation and adjacency. For convenience, we formulated the axiom such that we can read the picture in such that object $a$ in the picture is of type $A$, and object $b$ in the picture of type $B$ etc.. The 'weak regularity' axiom now says: " *if* I have an object that is the aggregation $(*_1^{\mathsf{E}})$ of two objects

  - one object that is of type $A$ and $(\sqcap)$ that is an object that has a partial description $(\Diamond_1)$ that is both of type $F$ and $(\sqcap)$ is an adjacency witness $(\Diamond_2)$ for an object of type $C$
  - another object that is of type $B$ and $(\sqcap)$ that is an object that has a partial description $(\Diamond_1)$ that is both of type $G$ and $(\sqcap)$ is an adjacency witness $(\Diamond_2)$ for an object of type $D$

  *then* this object has a $C$-object as adjacent $(\Diamond_1\Diamond_2)$ and $(\sqcap)$ has a $D$-object as adjacent $(\Diamond_1\Diamond_2)$". The other correspondences can be explained similarly.

- axioms and rules for combining extendibility and adjacency:

$$(\Diamond_1 \circ \text{overlap}) \quad \Diamond_1^{\cup} A \to \circ A$$

△

From this system we can deduce non trivial principles for the categorial graphs. For example "partial descriptions of an $A$ object extend to an $A$ object":

$$\dfrac{A \to \Box_1\Diamond_1^{\cup}A \ (\Diamond_1\text{Inverse}) \qquad \dfrac{\dfrac{\Diamond_1^{\cup}A \to A \ (\Diamond_1 \circ \text{Overlap})}{\Box_1(\Diamond_1^{\cup}A \to A)} \ (\Box_1\text{Necessation})}{\Box_1\Diamond_1^{\cup}A \to \Box_1 \circ A} \ (\Box_1\text{Distribution})}{A \to \Box_1 \circ A} \ (\text{cut})$$

The combined system gives us the full power of the categorial graph meta language of chapter 4, and even a little more, because we introduced additional operators that take a look at things from another perspective. These are:

- We added extendibility also in the language
- split the adjacency in two steps
- added inverse modalities for the unary modalities of adjacency and extendibility

| description | meta language | modal language |
|---|---|---|
| conjunction | $\sqcap$ | $\sqcap$ |
| disjunction | $\sqcup$ | $\sqcup$ |
| negation | $\neg$ | $\neg$ |
| adjacency | $\Diamond$ | $\Diamond_1\Diamond_2$ |
| aggregation | $*$ | split into collections of operations $\{*_1^{\mathsf{struct}} \vert_{\mathsf{struct}} \in \{\text{set}, \text{multiset}, \text{list}\}, X \in \{E, A\}\}$ |
| reflection | self | ... |

Figure 6.9: Correspondence between the categorial graph meta language and the combined modal language

- we introduced the modalities for the dyadic aggregation that look at the aggregation from a different perspective (versatile triple).

Correspondence between the categorial graph meta language of chapter 4 and the pure modal language are summarized in the figure 6.9.

### 6.2.6 Conclusion

We are now in the business of modal logics, and can import all kinds of techniques: decidability, axiomatization, frame correspondence, bisimulation. We say that the natural constraints on the abstract model correspond nicely with modal formulas. This enables one to study frames corresponding to interesting principles like $\Box_1\Box_1\bot$, or $\Box_1\top \leftrightarrow \Diamond_2\bot$. This way we give something back to the modal logic community: the motivation to study certain principles based on the very practical case of object modeling.

## 6.3 Other logical formalizations

Starting from the modal logic developed in this chapter we can traverse to other logical formulations. The modal logic community has seen several nice correspondences with more general and more specialized logics. In this section we will explore such paths.

### 6.3.1 Translation

Modal languages are part of first order logic (speaking generally; we could however add some higher operations involving fixed-points). The language constructs of modal logic can be translated into general first order logic using a standard translation. This translation will show that all the principles we defined for the modal logic for categorial graphs are first order definable. The modal language of categorial graphs can be translated to the following first order language:

**6.3.1.** Definition. (First order language for categorial graphs) Recall the definition of the modal language in definition 6.2.1. the first order language for categorial graphs $L^{\mathrm{FO}}$ contains the following:

- For all propositional variables in the modal language L, $P \in \mathsf{Prop}$, we have a unary predicate $P^{\mathrm{FO}}$ in $L^{\mathrm{FO}}$
- For all modalities $m$ of arity $i$ we define precicates $T_m$ of arity $i+1$ in $L^{\mathrm{FO}}$; i.e.

| | |
|---|---|
| $T_{\Diamond_1}$ | of arity 2 for $\Diamond_1$ |
| $T_{\Diamond_2}$ | of arity 2 for $\Diamond_2$ |
| $T_{\Diamond_1^{\cup}}$ | of arity 2 for $\Diamond_1^{\cup}$ |
| $T_{\Diamond_2^{\cup}}$ | of arity 2 for $\Diamond_2^{\cup}$ |
| $T_{\circ}$ | of arity 2 for $\circ$ |
| $T_{\circ^{\cup}}$ | of arity 2 for $\circ^{\cup}$ |
| $T_{*_i^{\mathsf{struct}}}$ | of arity 3 for $*_i^{\mathsf{struct}}$ |
| | ($\mathsf{struct} \in \{\mathrm{set}, \mathrm{multiset}, \mathrm{list}\}, i \in \{1,2,3\}$) |

- all other standard things to complete first order language including variables, $\wedge$, $\vee$, and $\neg$

△

The (standard) translation for this language now looks as follows:

**6.3.2.** Definition. (standard translation) Let $x, y, y_1, y_2$ be individual variables in the first order language $L^{\mathrm{FO}}$. The standard translation $ST_x$ taking formulas from $L$ into $L^{\mathrm{FO}}$ is defined as follows:

$$
\begin{aligned}
ST_x(P) &= Px \\
ST_x(\bot) &= x \neq x \\
ST_x(\top) &= x = x \\
ST_x(\neg A) &= \neg ST_x(A) \\
ST_x(A \sqcap B) &= ST_x(A) \wedge ST_x(B) \\
ST_x(A \sqcup B) &= ST_x(A) \vee ST_x(B) \\
ST_x(\Diamond_1(A)) &= \exists y(T_{\Diamond_1} xy \wedge ST_y(A)) \\
ST_x(\Diamond_2(A)) &= \exists y(T_{\Diamond_2} xy \wedge ST_y(A)) \\
ST_x(\Diamond_1^{\cup}(A)) &= \exists y(T_{\Diamond_1^{\cup}} xy \wedge ST_y(A)) \\
ST_x(\Diamond_2^{\cup}(A)) &= \exists y(T_{\Diamond_2^{\cup}} xy \wedge ST_y(A)) \\
ST_x(\circ(A)) &= \exists y(T_{circ} xy \wedge ST_y(A)) \\
ST_x(\circ^{\cup}(A)) &= \exists y(T_{\circ^{\cup}} xy \wedge ST_y(A)) \\
ST_x(A *_i^{\mathsf{struct}X} B) &= \exists y_1, y_2(T_{*_i^{\mathsf{struct}X}} xy_1y_2 \wedge ST_{y_1}(A) \wedge ST_{y_2}(B)) \\
&\quad (\text{where } y_1, y_2 \text{ are fresh variables})
\end{aligned}
$$

△

As an example look at the following modal formulas and their translations

$$\begin{array}{lcl} ST_x(\Diamond_1^{\cup} p \to \circ p) & = & (\exists y(T_{\Diamond_1^{\cup}} xy \wedge Py)) \to (\exists y(T_{\circ} xy \wedge Py)) \\ ST_x(\Box_1 \Box_1 \bot) & = & \forall y_1(T_{\Diamond_1} xy_1 \to \forall y_2(T_{\Diamond_1} y_1 y_2 \to y_2 \neq y_2)) \end{array}$$

We can optimize the translation from above by translating related modalities like $\Diamond_1$ and $\Diamond_1^{\cup}$ or a versatile triple $*_1^{\mathrm{set}E}, *_2^{\mathrm{set}E}, *_3^{\mathrm{set}E}$ to the same predicate with different order in the variables. i.e.

$$\begin{array}{rcl} ST_x(\Diamond_1(A)) & = & \exists y(T_{\Diamond_1} xy \wedge ST_y(A)) \\ ST_x(\Diamond_1^{\cup}(A)) & = & \exists y(T_{\Diamond_1} yx \wedge ST_y(A)) \\ ST_x(A *_1^{\mathsf{struct}E} B) & = & \exists y_1, y_2(T_{*_1^{\mathsf{struct}E}} xy_1y_2 \wedge ST_{y_1}(A) \wedge ST_{y_2}(B)) \\ ST_x(A *_2^{\mathsf{struct}E} B) & = & \exists y_1, y_2(T_{*_1^{\mathsf{struct}E}} y_1xy_2 \wedge ST_{y_1}(A) \wedge ST_{y_2}(B)) \\ ST_x(A *_3^{\mathsf{struct}E} B) & = & \exists y_1, y_2(T_{*_1^{\mathsf{struct}E}} y_1y_2x \wedge ST_{y_1}(A) \wedge ST_{y_2}(B)) \end{array}$$

### 6.3.2 First Order approach

In this setting we can go *up* in terms of generality to the first order level. This means that we can do simple first order logic by translating our modal formula's into first order formulas using the translation from above, and then interpret these first order formulas in standard Tarski models for first order logic. If we translate all the principles from the modal logic into the first order logic, we have a proper first order logic for categorial graphs. In order to make the Tarski models more concrete we can introduce sorts, a sort for objects, and a sort for partial descriptions. Then we can study the first order properties in a well known type of models: two sorted Tarski models. We will use this fact to say something about the completeness and complexity in future sections.

### 6.3.3 Resource approach

Probably more relevant is going *down* in terms of generality. We will gain in specificness by considering calculi in which we can talk directly about resources. For this we will use sequent calculi and leave out (or diversify on) the structural rules. In other words we will use a substructural calculus. When, in the abstract case, we leave in all the operations we actually gain in expressiveness, and therefore in complexity as well. We will look, instead, at tractable fragments of modal logics found in categorial or linear logic using the extra expressiveness. We will specifically look at a minimal language that has product and conjunction and adjacency.

**6.3.3.** Definition. (Substructural Adjacency language) Let $L_{SA}$ be the language called *substructural adjacency language* that talks about structures and adjacency; i.e. about aggregating (taking things together) objects and partial descriptions in a resource conscious manner.

$$L_{SA} = \mathsf{Prop} | L_{SA} * L_{SA} | L_{SA} \sqcap L_{SA} | \Diamond_1 L_{SA} | \Diamond_2 L_{SA}$$

△

The calculus is a simple substructural one. The language of structures consists of *formula's*. In a sequence calculus, like the one below, we deal with *terms*, which are comma separated lists (possibly empty) of formulas.

**6.3.4.** Definition. (Calculus for the substructural adjacency logic)

$$(AX) \qquad A \Rightarrow A$$

$$(CUT) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma', A \Rightarrow B}{\Gamma, \Gamma' \Rightarrow B}$$

$$(L\sqcap) \qquad \frac{\Gamma, A \Rightarrow C}{\Gamma, A \sqcap B \Rightarrow C} \qquad \frac{\Gamma, B \Rightarrow C}{\Gamma, A \sqcap B \Rightarrow C} \qquad (R\sqcap) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \sqcap B}$$

$$(L*) \qquad \frac{\Gamma, A, B \Rightarrow C}{\Gamma, A * B \Rightarrow C} \qquad (R*) \qquad \frac{\Gamma \Rightarrow A \quad \Gamma' \Rightarrow B}{\Gamma, \Gamma' \Rightarrow A * B}$$

For the modalities we add the following:

$$(\Diamond_1 I) \qquad \frac{A \Rightarrow B}{\Diamond_1 A \Rightarrow \Diamond_1 B} \qquad (\Diamond_2 I) \qquad \frac{A \Rightarrow B}{\Diamond_2 A \Rightarrow \Diamond_2 B}$$

△

We could now say things resource consciously:

- In Model $\mathcal{M}$ an $A$ object has (at least) two partial descriptions, a $B$ and a $C$ one $\mathcal{M} \models A \Rightarrow \Diamond_1 B * \Diamond_1 C$
- In Model $\mathcal{M}$ an $A$ object has (at least) two adjacents, a $B$ and a $C$ one $\mathcal{M} \models A \Rightarrow \Diamond_1 \Diamond_2 B * \Diamond_1 \Diamond_2 C$

More specifically about structures one can say:

- in model $\mathcal{M}$ the structures are order *un*conscious for $A$ items $\mathcal{M} \models A * \top \Rightarrow \top * A$
- in model $\mathcal{M}$ the structures are order *un*conscious for $A$ and $B$ items $\mathcal{M} \models A * B \Rightarrow B * A$
- in model $\mathcal{M}$ the structures *could sometimes be* counting conscious for $A$ items $\mathcal{M} \not\models A \Rightarrow A * A$

We could take it a step further and introduce more reasoning connectives plus the remaining modalities: negation, disjunction and extendibility. To be complete (and very similar to the calculus of the concrete model of chapter 4) we list the remaining language items and rules.

**6.3.5.** DEFINITION. Let $L_{FCA}$ be the language called *full substructural categorial language*

$$L_{FCA} = L_{SA} | L_{FCA} \sqcup L_{FCA} | \neg L_{FCA} | \circ L_{FCA}$$

△

**6.3.6.** DEFINITION. (Calculus for the full substructural adjacent language)

The rules of the substructural adjacency language plus the following:

$$(L\sqcup) \quad \frac{\Gamma, A \Rightarrow C \quad \Gamma, B \Rightarrow C}{\Gamma, A \sqcup B \Rightarrow C} \qquad (R\sqcup) \quad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \sqcup B} \quad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \sqcup B}$$

$$(L\neg) \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta} \qquad (R\neg) \quad \frac{\Gamma, A \Rightarrow \Delta}{\Gamma \Rightarrow \neg A, \Delta}$$

For the modalities we add the following:

$$(\circ I) \quad \frac{A \Rightarrow B}{\circ A \Rightarrow \circ B}$$

$$(\Diamond_1 Distribution) \quad \Diamond_1(A \sqcup B) \Rightarrow \Diamond_1 A \sqcup \Diamond_1 B$$

$$(\Diamond_2 Distribution) \quad \Diamond_2(A \sqcup B) \Rightarrow \Diamond_2 A \sqcup \Diamond_2 B$$

$$(\circ Distribution) \quad \circ(A \sqcup B) \Rightarrow \circ A \sqcup \circ B$$

△

The logics in these examples capture quite a bit of the expressive power we want to have for talking about objects. Such fragments therefore are valuable for doing specific reasoning in a computationally more tractable setting then the more general approaches. We will talk about reasoning and complexity matters in a moment.

## 6.4 Axioms and completeness

Here we will investigate the completeness of the axiomatics with respect to the abstract models and the intended models that are presented above. Why would we bother to investigate completeness? In other words what does completeness mean for our languages and our models?

If we have a semantically specified logic, then completeness w.r.t. some calculus means that we have found a calculus that syntactically (and exactly) characterizes this logic. Also if we have a syntactically specified, then logic completeness w.r.t. a semantics means that we have found a semantic characterization of this logic.

### 6.4.1 The first-order case

For the first order logic for categorial graphs (the logic we get after translation) we can derive some results from the general framework of first order logic when it comes to matters of axiomatization and completeness.

For the abstract models we know that all the logics we presented above are *effectively axiomatizable*, because the constraints we put on the relations of the abstract models are all first order definable (we gave these definitions above). Completeness then is trivial. On the other side we also know that the logic system will most likely be *undecidable*, unless we are really so lucky that the set of constraints mitigated the undecidability of first order logic in the general (unconstrained) case. We do not believe that this is the case, and are of the opinion that a lot more can be gained in this matter when we look *down* to less general frameworks.

For the intended models we have bad news in the first order case right away. The logics of categorial graphs will be at least as complex as 'True Arithmetic' when we have the aggregations in our model as a structure domain (product structure). This follows directly from a result of Quine ([Quine46], [Benthem91]) that states that the first order theory of simple syntax[12] is equivalent to True Arithmetic. To be precise quote the result:

**6.4.1.** THEOREM. *(Quine) Consider the model $\mathcal{M}\{a,b\}$ containing a binary operation of concatenation*[13] *and all finite strings from the two-symbol alphabet $\{a,b\}$. The first order theory of $\mathcal{M}\{a,b\}$ is equivalent to 'True Arithmetic' ($\Omega(\mathbb{N},+,\cdot)$).*

True Arithmetic is known to be undecidable, non-axiomatizable and of very high complexity by Gödel's and Tarski's classical theorems. The reason for such absurdly high complexity is that we have already too much structure when we have a structure domain (or simple syntax for that matter) when we allow the full first order language to talk about it. This structure can be 'abused' (with the expressive power of first order language) to code numbers and do arithmetic in it. We say 'abused', because the structures were not meant to code arithmetic but only complex information structures.

[12] Syntax that can be concatenated to form syntax again; i.e. what we have in a structure domain.

[13] or for that matter, a ternary relation of concatenation

### 6.4.2 The modal case

In the presentation of the different logics we were quite easy about the correspondence between the constraints and the accompanying axioms. This has a reason, because from the field of modal logic we *know* that such correspondences are generally valid when the modal axioms have a suited syntactical form; namely the *Sahlqvist form* ([BlackburnRijkeVenema01]).

**6.4.2.** DEFINITION. (Very Simple Sahlqvist Formula[14])

- An occurrence of a proposition letter $P$ is a *positive* occurrence if it is in the scope of an even number of negation signs. A modal formula is *positive* if all occurrences of its proposition letters are positive.
- A *very simple Sahlqvist antecedent* in a modal language is a formula built up from $\top$, $\bot$, and proposition letters, using only $\sqcap$, and existential modalities (e.g. the $\Diamond$ but not its dual $\Box$).
- A *very simple Sahlqvist formula* is an implication $A \to B$ where $A$ is a very simple Sahlqvist antecedent and $B$ is a positive formula.

$\triangle$

For example it is easy to see that that the ($*_1^{\mathsf{struct}X}$ $*_2^{\mathsf{struct}X}$ coherence) axiom is a very simple Sahlqvist formula. When we look at

$$A \sqcap (B *_1^{\mathsf{struct}X} C) \;\to\; (B \sqcap (A *_2^{\mathsf{struct}X} C)) *_1^{\mathsf{struct}X} C$$

we see that the antecedent is built from proposition letters $A$, $B$, and $C$, and the connective $\sqcap$ and the existential dyadic modality $*_1^{\mathsf{struct}X}$, hence it is a very simple Sahlqvist antecedent. Moreover all occurences of $A$, $B$ and $C$ in the conclusion are positive.

**6.4.3.** THEOREM. *(Sahlqvist correspondence theorem) Let $A$ be a Sahlqvist formula for a modal language $L$. Then $A$ corresponds to a first order condition $c_A$ on the models of $L$. Moreover $c_A$ is effectively computable from $A$.*

General modal logic provides an even stronger result. In the light of axiomatics and completeness these Sahlqvist formulas have the following property.

**6.4.4.** THEOREM. *(Salhqvist completeness theorem) Given a set of Sahlqvist axioms $\Sigma$, the minimal normal modal logic $K$ extended by the axioms of $\Sigma$ is complete with respect to the models that satisfy all the corresponding first order conditions.*

[14] Actually a broader syntactically characterized collection of formulas, the *Sahlqvist formulas*, will have the same desirable properties as mentioned in the succeeding theorems as these very simple ones. The 'very simple' subset suffices for our purposes.

For all but one of the axioms for the logics of categorial graphs from above we can easily establish that they are Sahlqvist formulas. Thus for the logics restricted to those axioms and corresponding conditions we have that

- the axioms correspond to the accompanying first order constraints
- the axiomatization is complete with respect to the abstract model

All the constraints that we encountered for the categorial graph logic correspond to Sahlqvist formulas, except one: the McKinssey axiom

$$(M) \neg \circ \neg \circ A \rightarrow \circ \neg \circ \neg A$$

corresponding to the local top condition

$$\text{(top)}\ \forall a \exists b\ (aSb \& (\forall c\ aSc\ \rightarrow\ cSb))$$

A normal modal logic with the McKinsey axiom added (i.e. the modal system S4.1) , however, *is* a complete axiomatization with respect to a relational structure with the 'top' condition. For proving completeness for the combined systems now the question remains whether or not adding Sahlqvist formulas to the system with the McKinsey axiom (S4.1) frustrates the completeness (note that adding Sahlqvist formulas to complete normal logics that already contain Sahlqvist formulas does leave completeness intact). To our knowledge this question has not been answered yet, meaning that we cannot solve this concrete matter with standard results of modal logic. Therefore it remains an open question here, of which we strongly believe in the positive answer.

For the intended models the situation is more complex (and we can far less rely on the known results in modal logic). Similarly, as in the first order case, the danger of too much structure resulting in an inherently incomplete system is possible. The structure of the domain is as complex as in the first order case. The modal language, on the contrary, is normally less expressive as the full first order language, so there is still a chance to get completeness here. Nevertheless there are a lot of very powerful things that one can say in the modal language about the mathematical structure of the intended model (for a thorough investigation in these matters see [Benthem2000a]). An example that we have seen in the presentation of the aggregation logic is that we can express the *induction principle*. This indicates that the completeness issue could be a very tricky one, because we can express very complicated mathematical structures using induction.

So how are the chances for interesting completeness theorems here? This is not known yet and could be an interesting open question for logicians.

### 6.4.3 The resource case

The previous analysis points out that our models can give rise to completeness proofs that are more complex than in traditional modal logic. But it can be the other way around: completeness proofs for our models can also turn out to be a lot simpler then what we normally see in modal logic. We will demonstrate this by the results for the resource fragments of the logics for categorial graphs. Now we again take a step further *down* in generality and see what we can say about completeness for the substructural fragments we have seen above.

For the *abstract* models we can, in relation to the resource language and sequent calculus that is present in the resource case, prove completeness using the 'minimalistic' strategy of proving completeness for resource logics that is investigated by Kurtonina ([Kurtonina95]), Buzkowski ([Buszkowski86]), and Dosen ([Dosen88], [Dosen89]). We will prove completeness for the calculus of the substructural adjacency language $L_{SA}$ with the adjacency modalities $(\Diamond_1, \Diamond_2)$, conjunction $(\sqcap)$ and aggregation $(*)$.

**6.4.5.** THEOREM. *The calculus of $L_{SA}$ is complete with respect to the abstract models $\mathcal{M} =< U, \mathsf{Q}, \mathsf{R}_1, \mathsf{R}_2, V >$*

**Proof:** To prove completeness we construct a canonical model as follows:

1. Universe $U = \{A | A \text{ is a formula of } L_{SA}\}$
2. Product relation $Q$ is defined as
$$A\mathsf{Q}CD \text{ iff } \forall E, F(C \Rightarrow E \& D \Rightarrow F \text{ then } A \Rightarrow E * F)$$
3. Adjacency relations $\mathsf{R}_1$ and $R_2$ are defined as
$$A\mathsf{R}_1C \text{ iff } \forall D(\vdash C \Rightarrow D \text{ then } \vdash A \Rightarrow \Diamond_1 D)$$
$$A\mathsf{R}_2C \text{ iff } \forall D(\vdash C \Rightarrow D \text{ then } \vdash A \Rightarrow \Diamond_2 D)$$
4. Valuation $V$ is defined as $V(P)\{A | A \rightarrow P\}$ for propositions $P \in L_{SA}$

For the canonical model $\mathcal{M}$ we prove the truth lemma:

$$\mathcal{M}, A \models B \text{ iff } \vdash A \Rightarrow B$$

- CASE $B = P$ for a proposition $P$: directly from the definition of valuation $V$
- CASE $B = C \sqcap D$:

| | | |
|---|---|---|
| $\vdash A \Rightarrow C \sqcap D$ | iff | $\vdash A \Rightarrow C$ and $\vdash A \Rightarrow D$ |
| | iff (by induction) | $\mathcal{M}, A \models C$ and $\mathcal{M}, A \models D$ |
| | iff | $\mathcal{M}, A \models C \sqcap D$ |

- CASE $B = \Diamond_1 C$:

  $\vdash A \Rightarrow \Diamond_1 C$ then $A\mathsf{R}_1 C$ because $\forall D$ if $\vdash C \Rightarrow D$ then

  $$\dfrac{A \Rightarrow \Diamond_1 C \quad \dfrac{C \Rightarrow D}{\Diamond_1 C \Rightarrow \Diamond_1 D}\ \Diamond_1\text{Necessation}}{A \Rightarrow \Diamond_1 D}\ \text{CUT}$$

  then $\mathcal{M}, A \models \Diamond_1 C$ (by definition semantics)

  $\mathcal{M}, A \models \Diamond_1 C$ then $\exists D A\mathsf{R}_1 D$ and $\mathcal{M}, D \models C$
  then $\forall E[\vdash D \Rightarrow E$ then $\vdash A \Rightarrow \Diamond_1 E]$ (by definition $\mathsf{R}_1$ and AX)
  and $\vdash D \Rightarrow C$ (by induction)
  thus $\vdash A \Rightarrow \Diamond_1 C$

- CASE $B = \Diamond_2 C$: Similar to the previous case

- CASE $B = C * D$:

  $\vdash A \Rightarrow C * D$ then $A\mathsf{Q}BC$ because $\forall E, F$ if $\vdash C \Rightarrow E, \vdash D \Rightarrow F$

  $$\dfrac{A \Rightarrow C * D \quad \dfrac{C \Rightarrow E \quad D \Rightarrow F}{C * D \Rightarrow E * F}\ \text{L}* \text{ and R}*}{A \Rightarrow E * F}\ \text{CUT}$$

  then $\mathcal{M}, A \models C * D$ (by definition semantics)

  $\mathcal{M}, A \models C * D$ then $\exists E, F[A\mathsf{Q}EF\ \&\ \mathcal{M}, E \models C\ \&\ \mathcal{M}, F \models D$
  then $\vdash A \Rightarrow E * F$ (by definition $\mathsf{Q}$ and AX)
  and $\vdash E \Rightarrow C$ and $\vdash F \Rightarrow D$ (by induction
  then $\vdash A \Rightarrow C * D$ (by L$*$, R$*$ and CUT)

Hence we have completeness □

If we add more connectives (i.e. disjunction and negation) this simple construction will not work as it stands, and the canonical model will become more complex. We could then, for example, use the The Dosen strategy ([Dosen89]) which introduces an additional relation $\leq$ over the aggregation structure to handle the aggregation in relation to the disjunction[15]. This relation is a technical one, which we can not really give a proper meaning, because in the abstract model it enables us to characterize objects that are of indefinite type[16]. We could extend our ternary relation $\mathsf{Q}$ that models the aggregation such that it covers this $\leq$ to handle indefinite objects as well. This is realized by letting $\mathsf{Q}$ model $a \cdot b \leq c$ instead of $a \cdot b = c$ as before. Note however that with this technicality we drift further away from our intended model, because there we do *not* have these indefinite objects.

[15] if we aggregate something of type $A \sqcup B$ with something else, we cannot just look at the cases where we either have an $A$ object or a $B$ object in the aggregation.

[16] i.e. we need an object $x$ that is of type $A \sqcup B$ in order to canonically interpret $(A \sqcup B) * C$ , but in the canonical model $x$ is neither of type $A$ or of type $B$ (it is indefinite).

Let us now look at the *intended* models. In the literature there is only one result, to our knowledge, that proves completeness in a resource logic (i.e. products) w.r.t. a concrete model. This is the famous completeness proof of Pentus ([Pentus93]) for the (multiplicative) Lambek calculus. Completeness for a concrete model of strings for the rules of the Lambek calculus follows from the following:

**6.4.6.** THEOREM. *(Pentus) The* recognizing power *of the (multiplicative) Lambek calculus is precisely the class of all context free languages.*

Pentus proves this by showing that for every given Lambek grammar (i.e. a collection of formulas recognizing all the formulas we can infer from them, seen as strings) we can effectively construct a coinciding categorial grammar and vice versa. This means that the Lambek calculus is complete for a model of strings. This proof is very involved, even though the Lambek calculus itself it quite limited. And thus it seems, at this time, extremely hard to establish a similar result for the concrete and more involved models for categorial graphs. Pentus's result however, implies that the basic logic of aggregation is complete for intended sequence models, so a completeness result for a richer language is a serious possibility. We hope that the challenge to find it will be taken up by the logic community[17].

### 6.4.4 What does this mean for our object models?

We have seen a number of calculi for our languages for categorial graphs, and nice completeness results for these calculi with respect to the abstract models. These abstract models properly show the behavior we are interested in when talking about information objects. In this respect we have 'good' calculi for reasoning about these interesting matters. There is, however, still a gap between the abstract models and the intended models, and this gap is not bridged by a completeness proof of a calculus with respect to the intended models. Judging by the literature the completeness issue for intended (rich) structures is a very hard one and has only a few positive results (we already mentioned the completeness proof of Pentus for the Lambek calculus with respect to languages). The completeness issue for object models raised here is yet another interesting challenge to logicians who like to think about a real structure of current interest.

[17]Even better would be a Sahlqvist-like result for resource logics. Then we would have a similarly strong tool for solving completeness and defninability questions as for normal modal logic. We have seen above that the Sahlqvist theorems for normal modal logic were very helpful for logical engineering of object oriented intuitions.

## 6.5 Complexity

In this part we analyze the complexity of the logics for categorial graphs. The complexity gives a measure for the 'hardness' of computational tasks using the expressive power of the systems presented here for object orientation.

### 6.5.1 Benchmark tasks

There are several 'benchmark' tasks for which computational analysis is done on logics. These tasks are the basis of most of the important algorithmic solutions for computing with the logics analyzed, and thus by assuming generality of these logics, they are indicative for many nontrivial computational tasks of the application domain the logic talks about. These tasks are:

- *Model checking.* Given a model $\mathcal{M}$, and an entity $x \in \mathcal{M}$, and a formula $A$ in the logical language $L$, then the task of model checking consists of checking whether $\mathcal{M}, x \models A$. This task is measured in the amount of computational steps in terms of the size of $\mathcal{M}$ plus the size of $A$.
- *Satisfiability (SAT).* Given a formula $A$ in $L$, does there exist a model $\mathcal{M}$ and an entity $x \in \mathcal{M}$ such that $M, x \models A$? The complexity of this task is measured in terms of the size of $A$.
- *Inference.* Given formulas $A, B$ in $L$, can we proof $A \vdash B$ from the calculus of $L$? This task is measured terms of the size of $A$ plus the size of $B$.

We will analyze these tasks for the logics of categorial graphs below. In our case the complexity of these benchmark tasks are indicative for many of the non-trivial computational tasks in object oriented models.

### 6.5.2 Model checking

Model checking is a common task in working with information systems. It is the question whether an expression (say constraint) is satisfied in some part of the information system.

The complexity of this task is easily proven to be in $\mathsf{P}$ (class of polynomial time computable tasks) for all the abstract modal logics of categorial graphs we presented here. For model checking we need to verify that a formula $A$ is satisfied in a given entity $x$ of our model $\mathcal{M}$. This involves (worst case)

1. when $A$ is a proposition letter a check of the valuation in $x$ which is bound by the size of $\mathcal{M}$
2. when $A$ is $B \sqcap C$ the check for $\mathcal{M}, x \models B$ followed by the check for $\mathcal{M}, x \models C$ which by induction on the structure of the formula are polynomial time computable in the size of $\mathcal{M}$ plus the size of $A$

3. when $A$ is $\neg B$ the check for $\mathcal{M}, x \models B$ followed by the inverse conclusion, which by induction on the formula structure is polynomial time computable in the size of $\mathcal{M}$ plus the size of $A$

4. when $A$ is $\Diamond B$ (where $\Diamond$ is any of the monadic modalities) then $\forall y(xRy)$ (where $R$ is the relation interpreting the modality $\Diamond$) we need to check $\mathcal{M}, y \models B$. The individual checks are, again by induction on the structure of the formula, polynomial time computable. The number of checks (i.e. number of $y$'s) is bound by the size of $\mathcal{M}$ (number of accessible worlds). Hence we are again polynomial time computable in the size of $\mathcal{M}$ plus the size of $A$.

5. when $A$ is $B * C$ (where $*$ is any of the dyadic modalities) then $\forall y, z(xQyz)$ (where $Q$ is the relation interpreting $*$) we need to check $\mathcal{M}, y \models B$ and $\mathcal{M}, y \models C$. The individual checks are by induction on the structure of the formula, polynomial time computable. The number of checks is bound by the square of the size of $\mathcal{M}$ (all possible pairs of accessible worlds of $\mathcal{M}$). Hence we have again polynomial time computability in the size of $\mathcal{M}$ plus the size of $A$ for model checking.

For the intended models and the resource models we need to take some extra care in analyzing the complexity of model checking. These models are of *infinite* size, because they contain free product structures (the domain structures). For example we have objects $x \cdot \ldots \cdot x$ of arbitrary length in our models. This complicates the complexity analysis as we defined it (and as it is commonly defined) for model checking, because we measure complexity in the size of the model. The solution to analyze these infinite systems is to measure their complexity in the size of a finite generator of the models. The infinite product structures (for aggregation) are generated from a finite set of atomic entities, and also the condition of *regularity* enforces that this base of atomic entities really completely generates the model. We discussed the generator of the intended models already in chapter 4. The generator consists of the atomic objects of the model together with the relations that specify the adjacency structure. Summarizing, we will measure the complexity of model checking in the size of the formula $|A|$ plus the size of the generator $|\mathsf{gen}(\mathcal{M})|$ of the model $\mathcal{M}$ and the size of the entity $x$ of $\mathcal{M}$ in which we do the model checking (we need to add $x$ here because unlike in the former case $x$ (in general) is not part of $\mathsf{gen}(\mathcal{M})$ and therefore can obfuscate the complexity analysis when it is taken very large).

The complexity for model checking for the intended models and the resource models is harder then P-time. We will prove that it is NP-hard. The reason for this increase is that an object can be split into parts in a number of ways that is exponential to the size of the object.

**6.5.1.** THEOREM. *Model checking for the resource logic for categorial graphs ($L_{FCA}$) is* NP*-hard*

**Proof:** We will proof NP-hardness by a reduction from the '*exact cover*' problem to model checking for the resource logic. The 'exact cover' problem is well known to be *NP*-complete ([GareyJohnson79]). The 'exact cover' problem is the following:

> Given a set $X$ with $x$ elements and subsets $A_1, \ldots, A_n$ of $X$, is there a collection of $k$ subsets $A_{i_1}, \ldots, A_{i_k}$ that *exactly covers* $X$ (i.e. the union of the $A_i$'s contain all elements of $X$ precisely once)?

We can reformulate this problem in terms of model checking for the resource logic as follows:

1. We express that a set $X$ has elements $p_1, \ldots, p_x$ by the formula $P_1 *_1 P_2 * \cdots * P_x$. Let us abbreviate this formula by $\overline{X}$. To make this scheme work we need to require that in our model all $P_i$ are satisfied in different entities[18]

2. Similarly we can express that a subset $A_i$ contains certain elements; i.e. $\overline{A_i} := P_{i_1} * \cdots * P_{i_{j_i}}$.

3. Now we can express the set $X$ is exactly covered by the $k$ subsets $A_i$ by stating
$$\overline{X} \quad \sqcap \quad [(\overline{A_{i_1}} \sqcup \cdots \sqcup \overline{A_{i_n}}) * \cdots * (\overline{A_{i_1}} \sqcup \cdots \sqcup \overline{A_{i_n}})]$$
$$\llcorner \ldots k \text{ times } \ldots \lrcorner$$

Let us now choose a model $\mathcal{M}$ with an element $X$ that is the aggregate of all its $x$ members of atomic objects, and let us choose $n$ subsets characterized by $A_1, \ldots A_n$. Now it is the case that $X$ has an exact cover of $k$ subsets $A_{i_1}, \ldots A_{i_k}$ if and only if

$$\mathcal{M}, X \models \quad \overline{X} \quad \sqcap \quad [(\overline{A_{i_1}} \sqcup \cdots \sqcup \overline{A_{i_n}}) * \cdots * (\overline{A_{i_1}} \sqcup \cdots \sqcup \overline{A_{i_n}})]$$
$$\llcorner \ldots k \text{ times } \ldots \lrcorner$$

□

## 6.5.3 Satisfiability

In the context of information systems using the theory of categorial graphs, satisfiability answers questions for the situation where one wants to know whether a modeling activity (resulting in a theory with additional constraints on all kinds

[18] We can also force it in the formula by stating $P_i \sqcap \neg P_1 \sqcap \cdots \sqcap \neg P_{i_{-1}} \sqcap \neg P_{i_{+1}} \cdots \sqcap P_x$ for ever $P_i (1 \leq i \leq x)$. This is not necessary to proof the reduction though, because we may without loss of correctness require satisfiable constraints on the models.

of information objects) is *consistent*. i.e. whether there are models (i.e. instances of information systems) that can satisfy all the constraints that were formulated during the modeling activity. This is a basic task for information processing.

Satisfiability for propositional logic is the archetypical case of satisfiability that is well known to be NP-complete. This means that for a formula $A$ it takes 'exactly'[19] a non-deterministic algorithm a polynomial number of steps (in terms of the length of $A$) to compute whether there is a model $\mathcal{M}$ and an entity $x$ in the model that satisfies $A$.

This fact constrains the results of the analysis of the complexity of the logics of categorial graphs, because all of these logics[20] contain propositional logic. This means that satisfiability for these logics for categorial graphs will at least have an NP-hard satisfiability task.

For the modal logics in general it is the case that most modal satisfaction tasks are not solvable in NP, but are at least PSPACE-hard. These tasks are solvable by a computational algorithm using only *polynomial space*. For example the minimal normal modal logic K and the modal logic S4 are PSPACE-complete. One way to get below PSPACE is when we can prove that the model that we need to construct to satisfy $A$ is at most polynomial in size of $A$. This *polynomial size model property* can be proved for models in which the constraints ensure that the model is compact[21]. Most models, however, can be used to simulate *binary trees* (i.e. exponential branching) and then this polynomial size model property fails and, moreover, proves *PSPACE*-hardness for the satisfiability task (cf. PSPACE-hardness criteria in [Spaan93]).

The situation sketched above implies some clear results for the individual logics for categorial graphs.

**6.5.2.** Theorem. *The satisfiability task for the fragments of the adjacency logics where we have only one type of modality, either type 1 ($\Diamond_1, \Diamond_1^{\cup}$) or type 2 ($\Diamond_2, \Diamond_2^{\cup}$) are in* NP.

**Proof:** This is directly implied by the constraint that the individual adjacency relations are at most only one step deep (forced by $\Diamond\Diamond\bot$). This means that we

[19] 'exactly' now means that is not easier than this; i.e. every task to which satisfiability of propositional formulas can be translated needs at least such an algorithm, and cannot be solved by an algorithm that is of lower computational complexity.

[20] Actuall all save one: when we take out the Boolean connectives, like in the example $L_S$ we could get lower complexity.

[21] When we have a polynomial size model property, we can let a non-deterministic algorithm guess a polynomial size model $\mathcal{M}$ and entity $x$. Now we need to do model checking for a polynomial number of times to verify that $\mathcal{M}$ really is a model that satisfies the constraints that are put on the system (these constraints correspond to formula for our logics) and then do one other time model checking for $A$ in $\mathcal{M}, x$. Model checking is P in $\mathcal{M} + |A|$, and thus also polynomial in $|A|$. Hence we have an NP algorithm.

can branch over the individual $R_i$ relations only once, so we need at most $2 \times |A|$ entities to cover all the entities that $A$ can say something about. □

**6.5.3.** THEOREM. *Satisfiability for the full adjacency modal logic is* PSPACE*-hard*

**Proof:** (Sketch) We can code the trees in two steps: a node is an $R_1$ source, an edge is an $R_2$ source, similar to where we presented the edge graphs formulation of a normal graph in chapter 3. □

**6.5.4.** THEOREM. *Satisfiability for the extendibility logic is* PSPACE*-hard*

**Proof (Sketch):** The extendibility logic is, in its isolated shape, the logic S4.1 combined with the logic S4.2. The logic S4 is known to be PSPACE-hard, and this is proved by showing that its models can simulate trees[22]. The restriction of a local top of the McKinsey axiom (M) clearly does not frustrate this tree structure (we are certain to have a beautiful tree with one top). □

An interesting corollary of the above result that requires a less sketchy proof, and exemplifies the fruits of the expressiveness of the logics for categorial graphs is the following: When we take a step further towards the combined logic add the $\Diamond_1$ modality to the extendibility logic, we have a system in which we can define the 'tops' of the extendibility relations by $\Diamond_1\top$ (tops are the objects!). Now an $S4$ formula $A$ is satisfiable if and only if $A$ relativized[23] to $\neg\Diamond_1\top$ is satisfiable in the extendibility logic with the $\Diamond_1$ modality. Hence the extendibility logic plus the $\Diamond_1$ modality is *PSPACE*-hard.

The above results dash every hope to get the combined modal system in to NP, which coincides with the intuition that reasoning about complex objects is strictly more complex than simple propositional reasoning. On the other hand we have seen in the previous section that we do not have the burden of undecidability, that some models (like the standard associative models as we indicated above) have; and this implies that the satisfiability problem for the combined logic and its abstract models is also decidable. This shows the intriguing and surprising balance of languages and model classes. We gained the interesting insight here that reasoning about complex objects does not require full computational power of first order logic (i.e. undecidable). This is rather intuitive, but we note that in practice most reasoning algorithms are based on heuristics for full first order (undecidable) languages. Using the logics of categorial graphs as a basis would most likely give better results, because we then would build on a theory with better computational characteristics.

[22] and therefore can do the well known PSPACE-complete Quantified Boolean Formula's task.

[23] we say that a formula $A$ is relativized to a formula $\neg\Diamond_1\top$ when all modal subformula's $\circ B$ of $A$ are replaced by $\circ(\neg\Diamond_1\top \rightarrow B)$. This enables one to 'place' a top on the S4 model, such that it becomes a model for the extendibility logic plus the $\Diamond_1$.

For the abstract resource models and substructural languages for categorial graphs we have a good case for the substructure adjacency fragment $L_{SA}$. These models have the finite model property for the substructural languages. The reason is the following: we can only talk resource consciously about the objects and their partial descriptions (which are entities in the model!), and thus all the specifications in the formula only have one entity in the model. Moreover, because this fragment does not contain *negation*, we do not need any combinatorics to construct a satisfying model. We can simply construct a satisfying model by introducing an entity for every building block of the formula: an atomic object for each propositional variable, an aggregate for each $A * B$ subformula, and a union of (sub) models for every conjunction $A \sqcap B$. Then we only need to check this one constructed model, because when it fails, then the formula cannot be satisfied by any model. This is evident by the lack of conflicting combinations in the valuation (only the relations may or may not satisfy the constraints of the logic, but the relations are fixed by the formula that we need to satisfy). We state the result without proof.

**6.5.5.** THEOREM. *The substrcutural language $L_{SA}$ with respect to the abstract resource models have a* P*-time satisfiability task.*

The complexity most likely becomes less tractable when we do take into account the negation. The fragment with[24] $\Diamond_1, \Diamond_2, *, \neg$ will still have the finite model property (hence NP satisfiability), but the fragment including $\Diamond_1, \Diamond_2, \sqcap, \neg$ will *not* have the finite model property (hence PSPACE-hard satisfiability). The reason lies in whether or not the fragments are able to encode a binary tree shaped model.

**6.5.6.** THEOREM. *1. The satisfiability task of the fragment of substructural $L_{FCA}$ with $\Diamond_1, \Diamond_2, *, \neg$ is* NP

*2. The satisfiability task of the fragment of substructural $L_{FCA}$ with $\Diamond_1, \Diamond_2, \sqcap, \neg$ is* PSPACE*-hard.*

**Proof:** The first statement is proven by showing that every formula $A$ in the fragment can be satisfied by a model of maximal polynomial size in the length of $A$. We prove this by induction on the structure of $A$. It suffices to prove it for the $*, \neg$, and $\Diamond := \Diamond_1\Diamond_2$, because only chains of alternating $\Diamond_1\Diamond_2$ can be arbitrarily nested (remember the $\Box_i\Box_i\bot$ principle forcing each individual modality to be one step deep only). Let $\mathcal{M}, a \models A$, and $\mathcal{M} =< U, Q, R_1, R_2, V >$ with universe $U$, relations $Q, R_1, R_2$ and valuation $V$ as usual.

- suppose $A = P$ (where $P$ a proposition): take $\mathcal{M} =< \{a\}, \emptyset, \emptyset, \emptyset, V >$ with $V(P) = \{a\}$, and $VP' = \emptyset$ for all propositions $P'$ other then $P$.

[24] This fragment is obtained from $L_{SA}$ by adding negation but removing conjunction.

- Suppose $A = B_1 * B_2$: by induction we have polynomial size models $\mathcal{M}_1, \mathcal{M}_2$ such that $\mathcal{M}_1, b_1 \models B_1$ and $\mathcal{M}_2, b_2 \models B_2$. Now construct $\mathcal{M}$ by taking the union[25] of (wlog supposed disjoint) models $\mathcal{M}_1, \mathcal{M}_2$ and extend $Q, V$ in $\mathcal{M}$ such that $aQb_1b_2$. We have now by definition of the interpretation that $\mathcal{M}, a \models A$ and size is simply $|\mathcal{M}_1| + |\mathcal{M}_2| + 1$.

- suppose $A = \Diamond B$. By induction we have a polynomial size model $\mathcal{M}'$ such that $\mathcal{M}', b \models B$. We extend $\mathcal{M}'$ to $\mathcal{M}$ by adding an object $a$ to the universe of $\mathcal{M}'$ and setting $aR_1R_2b$. We have now by definition of the interpretation that $\mathcal{M}, a \models A$ and size is simply augmented by a constant.

- suppose $A = \neg B$: By induction we have a polynomial size model $\mathcal{M}'$ such that $\mathcal{M}', b \models B$. We can transform $\mathcal{M}' =< U, Q, R_1, R_2, V' >$ to $\mathcal{M} =< U, Q, R_1, R_2, V >$ by altering the valuation as follows: For every entity $e$ in the universe of $\mathcal{M}'$ we invert the valuation with respect to the propositional variables that occur in $B$. This means that in $\mathcal{M}$ for all $P$ that occur in $B$ we put $V(P) = U - V'(P)$. It is clear that the size stays the same. We need to show that this (relatively cheap) construction works to ensure that $\mathcal{M}, a \models A$. This is seen by the following observation: For a model constructed in the process of this proof it holds that every subformula is local to *one* entity in the model. i.e. every modal subformula is satisfied by an atomic entity in $\mathcal{M}$ that has one $R_1R_2$ adjacent only (we do not have a conjunction to hold in the same entity, we only have aggregation that is satisfied by two independent entities). Spelling it out: Let $\overline{a}$ refer to an object in the inverted model $\mathcal{M}$ while $a$ refers to the object in model $\mathcal{M}'$. Now flipping the valuation to satisfy negation works for an object $a$ satisfing $P$, because $P$ will not hold in $\overline{a}$ anymore. For $B_1 * B_2$ that is satisfied in an aggregation object $a$ where $aQb_1b_2$, we also have that it does not hold in $\overline{a}$ anymore, because $B_1$ and $B_2$ do not hold anymore in their inverted subobjects $\overline{b_1}, \overline{b_2}$, and we are not allowed to misuse $\overline{b_2}$ to satisfy $B_2$ or misuse $\overline{b_1}$ to satisfy $B_1$, because we do not have exchange in the substructural calculus. Finally when $\Diamond B$ holds in $a$ because $B$ holds in $b$, where $aR_1R_2b$, then it will not hold anymore in $\overline{a}$, because there is only one $R_1R_2$ adjacent to $\overline{a}$, which is $\overline{b}$, and there $B$ is not satisfied.

We prove the second statement of the theorem by showing that we can force the model to have a tree shape, using a formula that is logarithmic in the size of the tree using the techniques developed by Hemaspaandra ([Spaan93]).

Let $q_0, \ldots, q_m$ and $p_1, \ldots, p_m$ be propositional variables. We will use the $q_i$ variables to encode the level in the tree[26], and $p_i, \neg p_i$ to force the branching. We

[25] Union of the universes and the relations.

[26] Note we use objects as nodes in the tree and partial descriptions as edges. Thus we need to state $\Diamond_1\Diamond_2 p_i$ to force that $p_i$ holds one level deeper in the tree. The conditions $\Box_1\Box_1\bot$ and

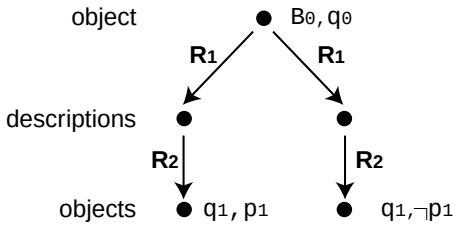



Figure 6.10: A model satisfying the branching formula $B_0$

abbreviate a *branch* formula $B_i$ as follows[27]:

$$B_i := q_i \rightarrow (\Diamond_1\Diamond_2(q_{i+1} \sqcap p_{i+1}) \sqcap \Diamond_1\Diamond_2(q_{i+1} \sqcap \neg p_{i+1}))$$

Now we force branching[28],by:

$$\Box_1\Box_2 B_i$$

Now we need a formula that *sends* the truth values assigned to $p_i$ and its negation one level down in the tree. This way we get the situation that once $B_i$ has forced a branching in the model by creating a $p_i$ and $\neg p_i$ its truth values are sent down in the tree.

$$S(p_i, \neg p_i) := (p_i \rightarrow \Box_1\Box_2 p_i) \sqcap (\neg p_i \rightarrow \Box_1\Box_2 \neg p_i)$$

to force a send we again use a necessation:

$$\Box_1\Box_2 S(p_i, \neg p_i)$$

To force a tree of $m$ levels we need to force $m$ branchings; i.e. $B_0 \sqcap \Box_1\Box_2 B_1 \sqcap \Box_1\Box_2\Box_1\Box_2 B_2 \sqcap \cdots \sqcap (\Box_1\Box_2)^{m-1}$ and to force down the truth values all levels $m$ we need the conjunction of $m^2$ formulas $(\Box_1\Box_2)^j(S(p_i, \neg p_i)$ (i.e. for all the $m$ $S(p_i, \neg p_i)$ we need then on on each level $j$).

Now we have a formula of size that is polynomial in the numbers of level of the tree it enforces; i.e. the formula forces a model that has exponential size with respect to its length. Hence we need at least PSPACE to compute satisfiability. □

For the concrete models we have the danger of the not-completely-axiomatizable system due to the richness of structure. Remember the undecidability result for the associative string calculus. In such case the satisfiability task is undecidable.

$\Box_2\Box_2\bot$ are essential to force that we cannot encode a tree with partial and whole descriptions playing the same role in the tree. When we can do that, we could use the standard encoding of trees in the model, where edges are not objects in the model but simply encoded in the relations

[27]Standard branching formula is :$B_i := q_i \rightarrow (\Diamond(q_{i+1} \sqcap p_{i+1}) * \Diamond(q_{i+1} \sqcap \neg p_{i+1}))$.

[28]In the standard encoding this is $\Box B_1$

### 6.5.4 Inference

*Inference* answers another, strongly related, question. It will compute whether a system of constraints from a modeling activity infers some other constraint. This task is the basis for verifying certain properties that are not explicitly forced by a system, but should follow from the theory and constraints that are actually implemented in a system.

For languages that have a (classical) negation connective and that have a complete calculus (w.r.t. their interpreting domain) the complexity results of the inference task and the satisfiability task are exactly the same. This is a consequence of the following statement:

$$\vdash A \text{ if and only if } \neg A \text{ is not provable}$$

For characteristics of the inference task for the resource calculi we need to do some work. We have no negation in $L_{SA}$, so we cannot use the results from the satisfiability task. Nevertheless the inference task is also expected to be P-time like the satisfiability task, due to the limited reasoning one can do in the language.

## 6.6 Extensions

In chapters 3 and 4 we briefly mentioned two extensions for the language of categorial graphs: the self and the ! (bang). These extensions are not part of the 'core' object intuition, but interesting extensions that enhance the ability to express constraints. In this section we will briefly discuss these extensions of the logic of categorial graphs.

In the object oriented paradigm, a statement (e.g. a constraint) is formulated from the point of view of an object. Formulating such a statement, it can be valuable to be able to refer to the object itself, i.e. to refer to the 'here and now' from the point of view where the statement is formulated. This can be accomplished by a modal constant self, which is interpreted to be true only in the evaluation point of the whole formula. One could then, for example, express that an object has itself as an adjacent:

$$\Diamond_1\Diamond_2\mathsf{self}$$

The self modality is not completely new. It is already studied in modern modal logic in the context of so-called *hybrid languages* ([BlackburnSeligman95]). The self modality fits into a nice extension of modal logic, where one has next to variables also so-called *nominals*. These nominals interpreted such that they are

true in exactly one object. The self modality is a special case. For this extension there exists a sound and complete axiomatization with respect to abstract relational semantics. Moreover, the complexity of the system is like most other modal systems: the satisfiability task is PSPACE-complete. We will not recite the axiomatics here. We argue that fruitful extensions from modal logic can be added[29] to the core modal system of categorial graphs.

In the field of resource logics there is a well known modality that enables one to introduce the structural rules in a controlled manner: The '!' (bang). The weakening, contraction, and exchange rules are introduced only for formulas labeled with the bang; e.g.

$$!\text{CONTRACTION} \quad \frac{\Gamma, !A, !A \Rightarrow \Delta}{\Gamma, !A \Rightarrow \Delta}$$

The bang enables one to explicitly type an object that is an arbitrary long aggregation of objects of some kind. Structures like sets, multisets and lists are such kind of objects. In a setting like this (i.e. with the necessary bang-rules) an object that (itself) is a set of $A$-type objects can be typed as $!A$. In linear logic the bang is well studied. However, introducing the bang has quite some influence on the complexity of the system. The full propositional system (i.e. with $*, \sqcap, \sqcup, \neg$) with the bang for all the structural rules (together this is full propositional linear logic) is undecidable. From the computational point of view this modality should be introduced with care.

## 6.7 Further logical considerations

The main line of this chapter is a modal-substructural elaboration of our object-oriented information models. Its main ideas of adjacency and object-description duality also suggest other logical directions, however, even closer to classical first-order logic.

### 6.7.1 'Object'/'type' duality

One example is the 'object'/'type' duality found throughout standard logic. E.g., the information structures of Barwise & Seligman ([BarwiseSeligman97]) consist of sets of objects which can 'satisfy' or 'belong to' types, which one can think of as propositions, or sets in the extensional case. This is like our valuation $V$. The main logical structure imposed by these authors is the following:

$$T \leq_{\text{type}} T' \text{ if every object satisfying } T \text{ also satisfies } T'$$

[29] Another nice, but less object-oriented flavored extensions worth looking at is the modal logic of inequality.

This is the usual implication ordering, and one may, or may not, require closure of the types under the Boolean operations. Dually, there is also an object inclusion:

$$o \leq_{\text{object}} o' \text{ if every type } T \text{ that holds of } o \text{ also holds of } o'$$

This is like the 'specialization ordering' among points in topological spaces. Another close analogy is 'Chu Spaces', as studied extensively by Vaughan Pratt ([Benthem2000b]).

It is of some interest to compare this inclusion structure with our orderings: neither inclusion $\leq$ is *exactly* what we had, though we could certainly define these additional relations.

$$T \leq_{\text{type}} T' \text{ if } \Diamond_1^{\cup} T \to \Diamond_1^{\cup} T'$$

Note that we have a richer domain and need to take into account descriptions *and* objects, so simple implication ordering does not suffice. We need extendibility to make sure that descriptions that do not follow from each other, but accidentally describe the same (of a subset of each others) objects, are properly ordered. In information technology terms one could think about descriptions of the same objects from another perspective; e.g. the descriptions 'morning star' and 'evening star' do not 'include' each other but do describe the same object 'Venus'.

$o \leq_{\text{object}} o'$ if every description $d$ of $o$ there is a description $d'$ of $o'$ that is a witness of the same type of adjacent; i.e.
$\forall d\ oR_1 d \exists d'\ o' R_1 d' (\exists a, a'\ dR_2 a \& d' R_2 a' \& a \models T \text{ iff } a' \models T)$

Although this theory differs from ours, we can prove properties of it in our logics. Moreover our approach assumes that the objects themselves come with some prior structure, namely, the product construction. This richer structure, not found with Barwise & Seligman, then interacts with the inclusions: e.g., are products inclusion-monotone w.r.t. their components? This extension seems worth exploring.

### 6.7.2 Treating 'facts' as first-class citizens

But perhaps a still closer analogy to our view of information models lies right inside first-order predicate logic. The duality between objects and descriptions amounts to treating *facts* as first-class citizens in their own right, in the spirit of our discussion in Chapter 5. Our approach shows how one might do this in Tarski semantics. In addition to the ordinary universe of objects, take a second domain of 'descriptions', consisting of all the positive atomic facts that are true in the model. Facts can be of any arity, assigning properties to objects.

Normally the fact that, say, $P$ holds between $o$ and $o'$ is modeled by putting the ordered pair $< o, o' >$ in the set of pairs that forms the interpretation of $P$. But

this 'reduction' is not necessary: we can say that $o$, $o'$ 'participate' in the fact in some more abstract -and yet more intuitive- way. Then it seems reasonable to say that our earlier relation $R_1$ holds between those objects and the fact "$Poo'$". And $R_2$ is just its converse, linking a fact to the objects participating in it. Thus, our earlier analysis may be viewed as an analysis of the mutual ties between objects and facts, yielding an alternative ontology for predicate logic, and another locus for 'logical structure': $R_1$, $R_2$ are now basic *logical* items.

Now this would make $R_2$ just the converse of the relation $R_1$. This choice is natural. E.g., the composition $R_1; R_2$ will hold between any two objects that occur together in at least one positive fact of the model. This is the so-called 'Gaifman order' of a first-order model, which has various model-theoretic uses. Moreover, the idea of co-occurrence in a fact is precisely the main idea of the *guarded fragment* of predicate logic, an avant-garde development in modern modal logic ([AndrekaBenthemNemeti96], [RijkeVenema95]). Restricting quantification to guarded tuples of objects makes quantification 'local', and leads to decidability of the language.

On the other hand, our original intuition about object orientation was still a bit different. We were thinking of facts *about certain objects* as protagonists, with the others involved as auxiliary characters in the fact. This is why $R_1$ and $R_2$ are not inverses. This showed in our 'tagging' of facts to object: one atomic statement "$Poo'$" could be two facts: one about $o$ : viz. $< o, Poo' >$ and one about $o'$, viz. $< o', Poo' >$. This is another take on the same semantic setting - but we leave it to the logicians, or philosophers, to decide whether this additional 'aboutness' of facts is part of their essential structure.

Finally, *if* one reorganizes predicate-logical semantics in this way, then it makes sense to rethink the language as well. Should we not dualize everything, and allow quantification over facts? Our point with the present excursion is more modest, however. Far from being an exotic structure, object-description models in our sense might also be an interesting style of modeling basic logical structures, and one can think of our various logical systems then as axiomatizations for the 'mechanisms' that drive these new models.

To add some flesh to the considerations we describe three modal logic versions of the above idea:

- version A: We take a domain of objects $O$ and a domain of *facts about adjacency* between objects $R_{\text{Adj}}$. Between these two domains we have two relations $\pi_1$, $\pi_2$ that relate an adjacency fact to, respectively, the object and its adjacent

- version B: We generalize the domain of facts and allow *arbitrary facts* $T(a), U(a, b), V(a, b, c) \ldots$ about the objects in $O$. We have a relation $\pi$ between the facts an the objects, relating a fact to and object when the object occurs in the fact.

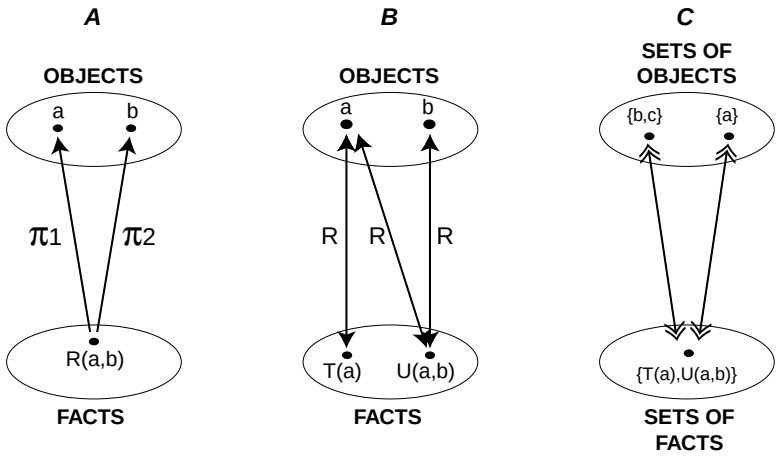



Figure 6.11: Version A,B and C of the object/fact models

- version C: one domain of entities consists now of sets of basic objects that occur in facts; and similarly we take the domain with sets of facts (hey, we are taking things together like in the object oriented calculus!). A set of facts $Y$ relates to a set of objects $X$ when *at least one* of the objects in $X$ occurs in *at least one* of the facts in $Y$.

Version A is actually the adjacency logic where $R_1$ is the converse of $\pi_1$ and $R_2$ coincides with $\pi_2$. The axiomatization therefore could be similar. However, from this point of view a symmetric set of axioms for modalities for $\pi_1$ and $\pi_2$ seems more natural. Let $< \pi_1 >, < \pi_2 >$ be the modalities interpreted by $\pi_1, \pi_2$ respectively, and let $[\pi_1], [\pi_2]$ be their dual modalities and $< \pi_1 >^{\cup}, < \pi_2 >^{\cup}$ their inverses. To force proper behavior of the $\pi_i$ relations in the light of objects and adjacency facts we then typically get (next to the normal modal principles) the following principles in the logic for adjacency facts:

| | |
|---|---|
| (Disjoint) | $\neg((< \pi_1 > \top \sqcap < \pi_2 > \top) \sqcap (< \pi_1^{\cup} > \top \sqcup < \pi_2 >^{\cup} \top))$ |
| (Exhaustive) | $(< \pi_1 > \top \sqcap < \pi_2 > \top) \sqcup (< \pi_1^{\cup} \top \sqcup < \pi_2 >^{\cup} \top))$ |
| ($\pi_1 -$ is a function) | $< \pi_1 > A \rightarrow [\pi_1]A$ |
| ($\pi_2 -$ is a function) | $< \pi_2 > A \rightarrow [\pi_2]A$ |
| (First Order) | $[\pi_1][\pi_1]\bot \sqcap [\pi_2][\pi_2]\bot$ |

Note that we cannot express that when two facts have their projections to the same objects we actually are talking about the same fact; i.e. if

$$f\pi_{\mathbf{1}}o_1 \& f\pi_{\mathbf{2}}o_2 \& f'\pi_{\mathbf{1}}o_1 \& f'\pi_{\mathbf{2}}o_2$$

then $f = f'$. Although this uniqueness is an important property of facts, we can ignore the matter using this logic for analyzing these facts, because all general models for our logic of adjacency facts are bi-similar to a special model that does have this uniqueness property.

Version B reveals interesting principles for a full predicate model with facts as first-class citizens in modal terms. Let $R$ be the relation between objects and facts where $oRf$ if object $o$ occurs in fact $f$ (e.g. $f = T(o)$). Let $\Diamond_\downarrow$ be the modal operator that is interpreted by $R$ with its dual $\Box_\downarrow$, its inverse $\Diamond_\uparrow$, and its inverses dual $\Box_\uparrow$ . The logic for the first-class-facts typically looks as follows[30]:

| | |
|---|---|
| (Disjoint) | $\neg(\Diamond_\downarrow \top \sqcap \Diamond_\uparrow \top)$ |
| (Exhaustive) | $\Diamond_\downarrow \top \sqcup \Diamond_\uparrow \top$ |
| (First order) | $\Box_\downarrow \Box_\downarrow \bot$ |

This system forces every object to play a role in a fact (i.e. there are no uninteresting objects where we know nothing of). An alternative that loosens this constraint (and introduces some asymmety between facts and objects) has the following alternative rules:

| | |
|---|---|
| (Disjoint$'$) | $\neg(\Diamond_\uparrow \bot \sqcap \Diamond_\uparrow \top)$ |
| (Exhaustive$'$) | $\Diamond_\uparrow \bot \sqcup \Diamond_\uparrow \top$ |

Version C translates our intuition on aggregating descriptions and objects to an intuition on aggregating facts and objects. Let $\mathbf{R}$ be the relation between sets of objects and sets of facts where $O\mathbf{R}F$ if some object $o \in O$ occurs in some fact $f \in F$. Let $\Diamond_{\downdownarrows}$ be the modal operator that is interpreted by $\mathbf{R}$ with its dual $\Box_{\downdownarrows}$, its inverse $\Diamond_{\upuparrows}$, and its inverses dual $\Box_{\upuparrows}$. Also let $\cup_1, \cup_2, \cup_3$ be the versatile triple that models an abstract relation[31] for union $\mathbf{U}$; i.e. $Z\mathbf{U}XY$ relates $Z$ to $X$ and $Y$ when $Z$ is the union of $X$ and $Y$. Now we can say some things about monotonicity of the relation between sets of objects and sets of facts. The logic for the sets of first class facts typically has next to the normal modal principles for unary and dyadic modal operators the following principles:

| | |
|---|---|
| (Disjoint) | $\neg(\Diamond_{\downdownarrows} \top \sqcap \Diamond_{\upuparrows} \top)$ |
| (Exhaustive) | $\Diamond_{\downdownarrows} \top \sqcup \Diamond_{\upuparrows} \top$ |
| (First order) | $\Box_{\downdownarrows} \Box_{\downdownarrows} \bot$ |
| (Downward monotonicity) | $(X \sqcap \Diamond_{\downdownarrows} F_1) \cup_1 Y \ \rightarrow \ \Diamond_{\downdownarrows} F_1$ |
| (Upward monotonicity) | $(F_1 \sqcap \Diamond_{\upuparrows} X) \cup_1 F_2 \ \rightarrow \ \Diamond_{\upuparrows} X$ |

In the combined system of categorial graphs we have seen more complex monotonicity principles: the regularity principles. For these principles we had stronger versions too. The stronger equivalents in this logic relate 'aggregating' sets of objects to 'aggregating' sets of facts (cf. the regularity axioms for the combined system for categorial graphs in definition 6.2.19):

| | |
|---|---|
| (Downward regularity) | $(X \sqcap \Diamond_{\downdownarrows} F_1) \cup_1 (Y \sqcap \Diamond_{\downdownarrows} F_2) \ \rightarrow \ \Diamond_{\downdownarrows}(F_1 \cup_1 F_2)$ |
| (Upward regularity) | $(F_1 \sqcap \Diamond_{\upuparrows} X) \cup_1 (F_2 \sqcap \Diamond_{\upuparrows} Y) \ \rightarrow \ \Diamond_{\upuparrows}(X \cup_1 Y)$ |

[30] Note that we can leave out the principle for requiring a source and targets for the facts

[31] We can not force all the axioms of set theory by modal principles for the set-union relation.

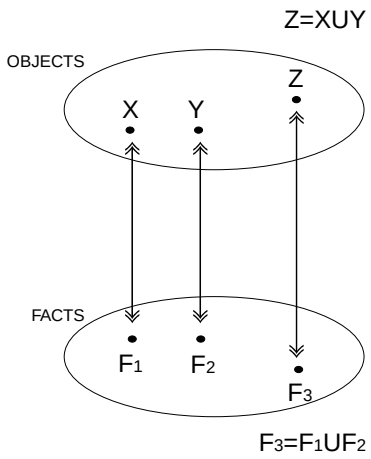



Figure 6.12: Monotonicity for version C (logic of sets of first class facts)

What we have seen here is that intuitions on how to talk about object oriented structures can be mapped on the setting of relational structures that were invented to interpret first order predicate logic. This seems promising for the intuition for the categorial graphs when we realize that the language of categorial graphs was intended to talk about the real world (modeled in object oriented structures) while first order predicate logic in turn is a language that intends to talk about phenomena in the real world (and is interpreted in relational structures).

Moreover, it shows that our intuitions on object orientation could possibly have consequences for standard logic, because it introduces a new view on objects and facts. This view could be beneficial to the field of logic itself.

## 6.8 Summary

In this chapter we analyzed the concepts of the system of categorial graphs of the previous chapters from a perspective of modern logic. We have disclosed logical properties of the individual core notions and have given account of their axiomatics and complexity, using results from the field of modal and substructural logic. The results provided us with a clear view on the core concepts of object orientation. On the other hand we provided the logicians with a concrete system that bears some interesting logical questions. Moreover, we proposed a different view on logic itself, using an intuition from object oriented practice. We strongly belief this scientific cross-fertilization bears even more fruits that are be beneficial for both the computer modeling field and logic. Here lies a challenge to be taken on in further research.

# Part IV

# Philosophical Backgrounds

# Introduction

The modern languages for information systems talk about complex objects. These objects have *identity* and can have a *complex signature*, and can be subject to *complex constraints*. In order reach the expressiveness needed to specify the behavior of an object, we need to be able to talk about the whole object and about partial descriptions of a complex object. Preferably we should be able to talk about objects taken together and also have the ability to count objects.

Although it seems quite natural to combine sentences that say things about an aspect of an object and sentences that say things about the whole object, this combination raises some difficult questions. The questions involve the way we interpret the language constructors that combine propositions on the whole object and propositions on its aspects. We will argue that these questions arise from problems involved with dealing with the notion of identity when speaking about parts of the world. These problems are well known in philosophy and have proven to be far from trivial. Therefore it is only natural we encounter these problems when formalizing the languages of information systems that talk about parts of the world.

In this part of the thesis we discuss philosophical issues of modeling information systems and the object oriented concepts. If effect we take a step back from the conceptual world we sketched out in the previous chapters and concentrate separately on some core notions that are important in the field of modern information systems modeling. In the coming chapter we will discuss four philosophical issues in modeling parts of the real world and we will discuss these issues in the light of the semantical study of the language of categorial graphs. We will show how these issues appear in the context of the categorial graph language. We do not propose a solution for the philosophical problems raised, but show that we can deal with several interpretations of the problematic issues in a clear way.

# Chapter 7

# Four philosophical issues

As we already observed in our analysis on object orientation, the large popularity and numerous occurrences of object notions in modeling, database and programming languages suggest that *it underlies an important intuition on how to model parts of the (real) world*. In essence the modeling activity consists of stating sentences[1] about objects in a world. These sentences represent knowledge about these objects. Even though there surely exits a natural intuition on how to model parts of the world, it is, by far, not unproblematic to give a rigid description of the modeling concepts and tools (language). This phenomenon has been subject to intensive study in philosophy in the last 2500 years.

In this chapter we will take four problematic issues in dealing with complex objects from the history of philosophy, and show their relation to concepts of information system modeling. We will see that these problems arise naturally when constructing and formalizing modeling languages using complex object notions. In most practical languages, like UML, these problems are hardly recognized, because these languages often have not been subject to formal semantical investigation. We think it is important to recognize these problematic issues, when dealing with a modeling language. Especially because these problematic issues, how natural they may be, really matter when one, as an information analyst, models parts of the world building an information system[2].

[1] These sentences are in the context of the logic of categorial graphs technically propositions and rules.

[2] We note that there has been done some work that connects object oriented information system languages to philosophical languages. An example of this is a paper that connects the concepts of object orientation to Aristotelian logic ([RaysideCampbell00]). Actually this paper shows an effort to clarify the notions from object orientation, like we did (in a completely different manner) in the previous chapters by providing a formal semantics. The paper does not try to uncover problematic issues that can occur while doing modeling, which is the intention of this chapter exemplified by four philosophical issues.

We will discuss the four philosophical issues in the light of the semantical study of the language of categorial graphs. We will show that the semantical investigations can provide us with a clear view on the philosophical issues raised, and sheds light on how to solve some nasty ambiguities in the information modeling practice. We, of course, do not claim we have a solution for the philosophical problems, nor do we suggest that we provide a thorough list of philosophical issues that touch the field of information systems modeling. We merely present four examples from philosophy -two from ancient philosophy and two from the beginning of modern philosophy- and show a way to handle different approaches to the issues at stake in the context of our semantic study in information systems modeling. The different approaches to the philosophical issues are still subject to their philosophical criticisms.

## 7.1 Examples from 2500 years of modeling information systems

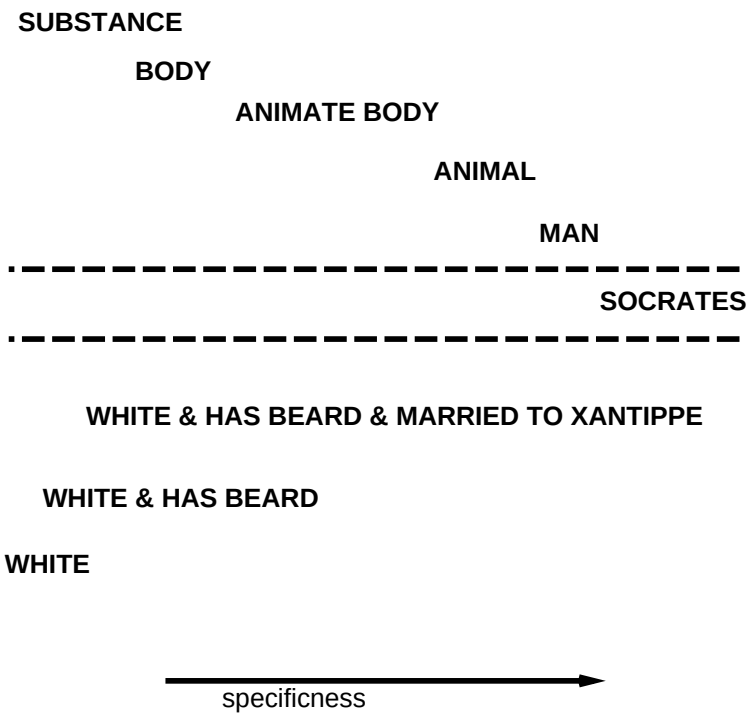
**I.** In western philosophy Plato started to use 'logical rules' to specify knowledge, and his pupil Aristotle continued this in even a more rigid way in his essay 'The Categories'[3]. In this essay Aristotle starts with a modest list of notions used to model things from the world, among which the ability to talk about a whole object and about part(s) of an object[4] and about aggregates[5]. In an attempt to clarify the concepts in his list Aristotle rigidly analyses the precise semantics of the concepts and discovers a lot of deep and far from trivial matters that need to be solved to fully understand the concepts. The same kind of problematic matters come up when we start from a shopping list of concepts needed in a modeling and database language. An important observation made by Aristotle is that sen-

[3]For more extensive philosophic considerations about the notion of *categories* from Aristotle in relation to information engineering we gladly refer to [Adriaans92] citing from it that "we still cannot think of a better introduction to the problems of knowledge engineering and theory of knowledge then this small essay ['The Categories'] of Aristotle".

[4]Categories(1a20) *Of things there are:(a) some are* said of *a subject but are not* in *any subject. For example, man is said of a subject, the individual man, but is not in any subject. (b) Some are in a subject but are not said of any subject. (By 'in a subject' I mean what is in something, not as a part, and cannot exist separately from what it is in.) For example, the individual knowledge-of-grammar is in a subject, the soul, but it is not said of any subject; and the individual white is in a subject, the body (for all color is in a body), but is not said of any subject. (c) Some are both said of a subject and in a subject. For example knowledge is in a subject, knowledge-of-grammar. (d) Some are neither in a subject nor said of a subject, for example the individual man or individual horse-for nothing of this sort is either in a subject or said of a subject. Things that are individual and numerically one are, without exception, not said of any subject, but there is nothing to prevent some of them from being in a subject-the individual knowledge-of-grammar is one of the things in a subject* [Ackrill63]

[5]Categories(1a16) *Of things that are said, some involve combination while others are said without combination* [Ackrill63]

tences that talk about partial descriptions of objects are treated fundamentally different from sentences that talk about the whole object. For example the sentence *Socrates is a man* says something about the whole object (we reference to by the name) Socrates, while the sentence *Socrates is white* is a sentence about an aspect (partial description) of Socrates (his color) The sentence on a whole object can vary from very general to very specific. For example, one could say of Socrates that he is a *substance*, or more specifically he is a *body*, even more specific he is an *animate body*, which in turn is implied by the sentence that he is an *animal*, which is made even more specific by proposing that he is a *man*, etc.. When talking about aspects, one can get more specific also, but the intent is very different, because one only gives partial descriptions of the whole object. For example one can say *Socrates is white*, which is less specific then *Socrates is white and has a beard*, or *Socrates is white, has a beard and is married to Xantippe*. If one would consider an ordering in information content there will be a switching point in information content when one exactly characterizes the whole object. From a sentence that exactly characterizes the whole individual object one can further talk about the whole object in more general terms (less specific, with less information content). And also from a complete description of the whole object one can talk about less and less 'aspects' of the object describing only partially the object. In a picture:




**II.** In his essay 'De Interpretatione' ([Ackrill63]) Aristotle makes very clear that *combining* predicates of different kinds is not a trivial matter. Combining predicates that say something about the whole object and predicates that say some-

thing on aspects (partial descriptions) of the object cannot be done uniformly[6]. For example consider the following predicates on partial descriptions of an object: 'white' (on some objects color) and 'musical' (on some objects ability); and consider the following predicates on a whole object: 'man' and 'animal'. Look now at the following combinations:

1. a man is white
2. white is a man
3. white is musical
4. musical is white
5. a man is an animal
6. an animal is a man

A predicate on a whole object and a predicate on a partial description of an object can be naturally combined in a sentence when the 'evaluation point' of the sentence is the whole object (see 1), while it becomes problematic when the 'evaluation point' is that of the partial description (see 2). For a combination of the predicates on partial descriptions the naive way of combination is problematic (see 3 and 4). Note however that in the combinations 3 and 4 above the 'evaluation point' is not a whole object, but in both cases an 'aspect'. A combination of two predicates that talk about a whole object (see 5 and 6) appears to have a clear interpretation[7].

Concluding we can say that from Aristotle's analyses it follows that we can distinguish (at least) two kinds of predicates: predicates that talk about the whole object, and predicates that talk about a partial description (or aspect) of an object. These two types of predicates turn out to have fundamentally different behavior.

[6]Aristotle makes a more subtle distinction then we propose here. He talks about qualities that are accidental (we call them 'predicates on aspects of the object') and qualities that are not accidental but essential (we call them 'predicates on the whole object') (De Interpretatione, 21a7):*"Of things predicated, and things they get predicated of, those which are said accidentally, either of the same thing or of one another, will not be one. For example, a man is white and musical, but 'white' and 'musical' are not one, because they are both accidental to the same thing. And even if it is true to say that the white is musical, 'musical white' will still not be one thing; for it is accidentally that the musical is white, and so 'white musical will not be one. Nor, consequently, will the clobber who is (without qualification) good, though an animal which is two-footed will (since this is not accidental).* [Ackrill63]

[7]However Aristotle had some doubts about the meaningfulness of a combination of a predicate of a whole object and a generalization of this predicate, because in his view of semantics such a combination is either false or superfluous (De Interpretatione 21a7)

**III.** An other valuable notion in modeling is the ability to *count*. There has been a lot of analysis on the meaning of numerical statements, when talking about objects. An interesting quarrel on this subject is one between Husserl and Frege. In his 'Philosophie der Arithmetik' [Husserl70] Husserl argues that numerical statements[8] on objects taken together can be expressed quite adequately by means of the conjunction 'and'. A numerical statement thus would be of the form "A and B and C and $\cdots$ and Q is $n$". E.g. *Berlin and Dresden and Munich are 3*. Frege[9] objects to this point of view by arguing that the conjunction is used in numerical statements only in the context of *identity* statements and not for indicating complex numerical structures. In the context of identity statements only the numbers 1 and 2 are meaningful: 1 to indicate existence, and 2 to indicate that two objects are different. For example Frege says that the statement *Berlin and Dresden and Munich are 3* is meaningless without an implicit statement on the identity of the objects in the statement, because it gives no information. It does not say whether Berlin is different from Dresden nor whether Berlin exits. Frege claims that in everyday speech however the numbers 1 and 2 are used with implicit statements on the identity of the objects[10]. Then the statement *Berlin and Dresden are 2* would express that Berlin is different from Dresden, and *Berlin is 1* would express that Berlin exists. Although Husserl admitted that Frege's criticism on his concept of numerical statements was founded, it remains a natural intuition that a statement like '*In this village we have a baker, a butcher and a grocer*' has some numerical content. But do we know about how many people we are talking? Of course one can object that the statements that express difference with a numerical are incomplete, and that the statement *Berlin and Dresden are 2* should be rephrased to *Berlin and Dresden are 2 and Berlin is*

[8]For accurateness we need to mention that in his philosophical studies Husserl talks about 'judgments' and not about 'sentences' like we do in our (logic) semantics. This means that the truth bearer of Husserl (judgment) is philosophically different from ours (sentence). Frege on the other hand does talk about sentences. This probably is one underlying reason for the difference of opinion on this matter. Moreover it is a reason to defend the position of Husserl that judgments on numerics are connected to the 'act' of counting (taking together), while Frege his point of view only takes an objective truth value (the sentence is true or false, no justification needed) into account and therefore rejects an 'act' as the source of the numerics. This discussion is a deep philosophical one and far out of our scope. Whatever the philosophical differences are, they do no harm the case we want to make here, because we merely exemplify different approaches and relate them to our semantics of information systems. It is actually very interesting that we can reflect these total different approaches in the semantics of information systems.

[9]in his review of Husserl's *Philosophie der Arithmetic* (C.E.M. Pfeffer, Leipzig 1891) in *Zeitschrift für Philosophie und phil. Kritik, vol. 103 (1894),pp.313-332)*; translated in [GeachBlack52])

[10]In ([GeachBlack52]) (translation of 'on numerical statements') Frege claims: "*I find that it is really used only in two cases: first, with the numerical 'two,' to express difference (*Rübsen und Raps sind zwei*'-rape-seed and rape are two (different things)'); secondly, with the numerical 'one,' to express identity-'I and the Father are one.'*

*different from Dresden.*

**IV.** In his essay *Über Sinn und Bedeuting* [Frege1892] Frege analyses further and encounters an even more problematic matter in modeling objects with identity. Frege has taken on the following problematic matter. Suppose you talk about two objects; you talk about one object stating *it is the morning star*, and about one object stating *it is the evening star*. As both the 'morning star' and the 'evening star' are the planet Venus, the sentences have the same *meaning* according to Frege. The predicates 'morning star' and 'evening star' differ, according to Frege, not in their meaning but in their *sense*. Frege distinguished three levels of being different[11]:

1. two sentences about an object only describe a different *idea* on the same object (the same object plays a different role in the different sentences),

2. two sentences differ in their *sense*, but not in their meaning (two objects with different intention only happen to be the same)

3. two sentences have really a different *meaning* (the two objects are really different).

This matter raises difficult questions when one wants to interpret the sentences in a formal model for information systems. For example do we need to be able to distinguish somehow in our system between sentences with the same meaning but with a different sense? Moreover if we talk about *the morning star 'and' the evening star*, do we interpret it as one object, two times the same object, or two different objects.

This subject has also quite recently been the subject of intensive philosophical research ([Linsky71]. For example assume you give always the same meaning to sentences that happen to have the same meaning but have a different sense (are different like in 2 above). Then the sentences *The morning star is visible in the morning*, *The evening star is visible in the morning* and *Venus is visible in the morning* all should mean exactly the same. This attitude becomes even more problematic when one is allowed to use modalities in a modeling language. E.g. the following sentences should then have the same meaning: *Raphael believes that the morning star is different from the evening star* and *Raphael believes that the morning star is different from the morning star* and even *Raphael believes that Venus is different from Venus*. While the first sentence may be true in a model, the second and third one very likely are not.

[11] [Frege1892], p 30: *Wir können nun drei Stufen der Verschiedenheit von Wörtern, Ausdrücken und ganzen Sätzen erkennen. Entweder betrifft der Unterschied höchstens die Vorstellung, oder den Sinn, aber nicht die Bedeutung, oder endlich auch die Bedeutung*

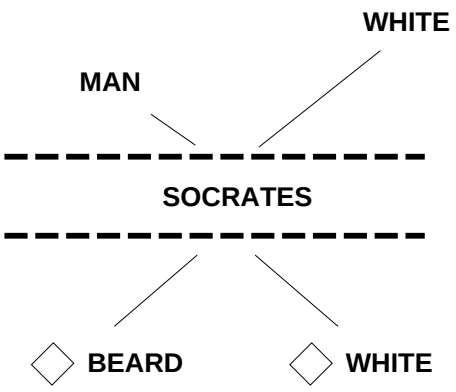
The matters above all give rise to different ways of giving semantics to (combinations of) sentences on objects, all of which can be subject to some philosophical criticism. In the next section we will formulate the above four issues in the categorial graph meta language we developed in the previous chapters to talk about information using the concepts from object orientation. We illustrate a range of possible interpretations one can give to the issues raised above. In practical information system modeling languages one can very quickly be victim to unclearness in the describing situations involving matters from above. The semantics developed in this thesis, however, can give an analyst or developer of an information system more insight into the interpretation of the sentences in the modeling language that touch these problematic issues. We have no intention whatsoever to solve the philosophical problems we mentioned above. We simply aim to show its implications for interpreting a language for modeling information systems.

## 7.2 The philosophical issues in terms of categorial graphs

Below we give a short list relating properties of the semantic domain of the categorial graph languages to the philosophical dilemmas described above.

**I.** *in and of a substance.* Suppose there is a specific name that refers to a particular object. Then there are sentences that talk about the whole object in more general terms, and sentences that talk about aspects of the object. In our language we have the ability to talk about whole objects, which is simply stating a proposition or a formula, but we can also talk about parts of an object, using the adjacency modality described above. Suppose the name Socrates refers to some object in our model[12]. Then we can talk about Socrates by stating more general predications on the object like man. On the other hand we can talk about things *in* our object. For example that the object has a beard: $\Diamond$beard. It becomes more interesting if we want to state that Socrates is white. We have two possibilities: either we predicate white or we predicate $\Diamond$white.

[12]Note that in the analysis of Aristotle Socrates is called a 'substance' and not a predicate. In a logical theory you have only predicates, of which some can be so special that there is only one object in the model that can interpret it.




Aristotle states in his 'categories'[13] that in this case we are talking about *something that is in a subject*, and that ◊white would be the most appropriate. This would actually also be the way which most complies with the Object Oriented modeling paradigm, in which an object is of some (most specific in the modeled type system) class, and all properties are modeled via attributes. In our meta language this would be stated as Socrates → ◊white[14]. The alternative would be choosing to have a type (class) white of which the class Socrates is a subtype (i.e. class Socrates inherits from class white) from; i.e. in our meta language Socrates → white.

**II.** *Evaluation points in whole objects.* The adjacency operator enables one to talk about aspects of objects. This adjacency operator really is a modal operator, because, as all modal operators, it gets a meaning (describing an 'aspect') only when we know in which object in the model a formula with the modal operator is evaluated. For example the formula stating ◊beard ('having a beard') can be interpreted as an 'aspect' only when we have a whole object with a beard in our model. This way of modeling, again, complies with the paradigm of object orientation, in which we model the world relating our knowledge on the world to objects only. A more interesting example is the following: we can talk about the object white of type white, which in a modeling language is nothing more than a value (but in a pure object oriented dogma, even a value is an object), and we can talk about an object having the white-object of type white as an aspect or partial description[15]: ◊white. The distinction made by Aristotle on things that are not said of a subject but are *in* a subject on one hand and things that are said of a subject but *not in* a subject, is now not dependent on what is said (e.g. white or man), but on *how* it is said: ◊white, or white. With that ability in the language, it is the task of the information system designer to find out 'what' are the objects in his model and 'what' are the aspects.

[13] see [Ackrill63] Categories $1^a20$

[14] this has of course the intuitive graphical representation in the categorial graph language

[15] i.e. to partially describe the whole object

**III.** *Counting with resource conscious connectives: wholes 'and' wholes, partial descriptions 'and' partial descriptions, wholes 'and' partial descriptions.* Due to the fact that predicates on wholes and predicates on partial descriptions have different semantical behavior when they take part in a complex formula, we need different rules for combining these kinds of predications[16] in a formula. We need rules that specify the combining of predicates on wholes and partial descriptions. A very important, and very non-trivial way of combining these predicates is in a context of resource conscious conjunction ('and').

When one talks resource-consciously about *a* $\mathsf{man}$ *and a* $\mathsf{man}$ ($\mathsf{man} * \mathsf{man}$), one most likely talks about '2 man'. This '2 man' is in our model an aggregation of two objects, each being a man. Alternatively one could say that the above sentence can be interpreted by one man: either twice the same man or (if one rejects the ability to count whole objects with a conjunction) just one man.

When one talks resource-consciously about something *having a* $\mathsf{beard}$ *and being* $\mathsf{white}$ ($\Diamond\mathsf{beard} * \Diamond\mathsf{white}$), one could talk about several things: first one could talk about *one* object that has both a beard and is white colored. On the other hand one could also be talking about an aggregate of two objects, one having a beard, and one being white colored; or even about an aggregate of 10 objects among which at least one object has a beard and one object is white colored.

When one talks resource-consciously about something *being a* $\mathsf{man}$ *and having a* $\mathsf{beard}$ ($\mathsf{man} * \Diamond\mathsf{beard}$), we again have several choices. One could talk about one man having a beard, about a man aggregated with something that has a beard, or about a man aggregated with a number of objects of which at least one has a beard.

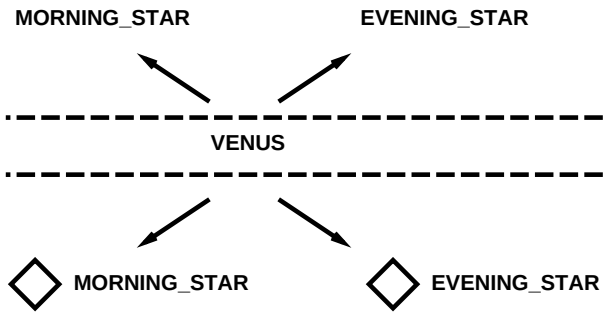
In most cases one does not want to have all the choices from above. Especially because some of them run into philosophical debate. One can formulate rules for the logic of the categorial language, and corresponding rules for the object structure interpreting this language that will enable or forbid some of the interpretations from above. These rules can enable or disable the strict counting for parts and wholes, and regulate the way we combine resource-consciously the predicates on wholes and parts. One can think of many different configurations. We could for example construct a world in which we can count the parts of an object, but not the whole objects, and not combinations of whole objects and parts. This would entail that $\Diamond\mathsf{beard} * \Diamond\mathsf{beard}$ (having at least two beards) would mean something different than $\Diamond\mathsf{beard}$ (having at least one beard), while $\mathsf{beard} * \mathsf{beard}$ would mean the same as $\mathsf{beard}$ (being a beard). And $\mathsf{man} * \Diamond\mathsf{beard}$ would not necessarily be two objects (one being man and one having a beard).

**IV.** *Roles, purpose and meaning.* In modern modeling languages it is very

[16]Note that in our modal context setting the predications are technically propositions that are or are not satisfied in a object (world). In philosophical terms they are nevertheless predicates because they are stating qualities of an object.

common to draw objects taking part in a relation in a certain role. This means, for example, that some object Socrates can relate to some object Xantippe in the role labeled by *husband*, while it relates to the object Plato in the role labeled by *friend*. This means that in the different relations one has a different view on the object. In the language of categories this can be achieved by using labeled adjacency modalities. The labeled modalities require an interpretation in a domain in which objects have an ordered adjacency structure, so we can distinguish between the different roles an adjacent plays in its relation to that object.

For objects having a different sense the story is much more difficult. Although the matter on *Sinn und Bedeutung* remains difficult, we can express matters on this problem quite clearly in our language. Suppose we have the predicates evening-star, morning-star and venus. One could either say that evening-star and morning-star are predicates about the 'whole' object or about 'aspects' of the object (compare to the matter on white from above). In other words either evening-star and morning-star say something about a whole object like the predicate man does about a particular person, or evening-star and morning-star say something about an aspect of an object i.e. about its appearance[17]. Suppose now that we have an object that is precisely characterized (as a whole) by the name venus, we can illustrate that choice with the following picture[18]:




The different styles of modeling the fact of being a morning or evening star, i.e. as aspect or as whole, imply different behavior of these predicates (propositions in a modal logic setting) when giving meaning to it. For example in a model in which one can count the the formula

$$\text{morning-star} * \text{evening-star}$$

will be interpreted as the aggregation of *two* objects, one being the morning star, and one being the evening star. These objects, however, need not be different, in which case the interpretation would be two times the object referenced by venus. In the same model, i.e. when one can count, the formula

$$\Diamond\text{morning-star} * \Diamond\text{evening-star}$$

[17]We would assume the latter choice is the most preferred one if we take the Aristotelian view on talking about objects

[18]One should read the arrows in the picture as implications.

can be interpreted by *one* object, having (at least) *two* aspects, being one appearance as morning star and one appearance as evening star. In this case the interpretation may be one time the object referenced by venus[19].

In the modal case the *Sinn* of the formula ◊morning-star is *the possibility to appear as the morning star*. The interpretation (meaning) of such a formula is just dependent on the model; i.e. if there is an object, say venus, in our model that has as an aspect the appearance as a morning-star, then venus is a possible interpretation of this formula. More elaborate: if there is one object that has both aspects morning-star and evening-star, then the propositions ◊morning-star and ◊evening-star and ◊morning-star∗◊evening-star are all true in this same object. Moreover the aspects themselves (i.e. both kind of appearances) will appear in the model as different objects, both being adjacent to the venus.

In the non-modal case, the 'whole' object is characterized (more or less specific) by the predicate morning-star. This means that morning-star is a characterization of the whole object, just like man is a characterization of the object Socrates. The *Sinn* of such a formula is *being a morning star*. The interpretation of such a formula can only include an object being 'morning star'. This could, in a model, possibly be the same object as that being the 'evening star'.

Both ways of modeling the evening and morning star in our language can validate a model with one object (venus) being the interpretation of both. Note however that the way in which the interpretation of the expressions is 'computed' is fundamentally different. In the modal case (◊) structural properties (aspects) of the object in the model are checked, while in the non-modal case only intrinsic properties of the whole object (type assertions) can be checked.

## 7.3 Summary

In this chapter we touched four philosophical issues that arise naturally when modeling information. We showed that these issues really can be problematic in general. We also showed that we can quite clearly form sentences involving the problematic matters in the logic of categorial graphs. The general nature of the logic of categorial graphs and the fact that it is a formal system enabled us to deal with the problematic issues in a clear way, avoiding nasty ambiguities.

[19] or, for the sake of completeness, this one object venus aggregated with some other objects, as the aggregation of venus with something else still has the two mentioned parts.

# Part V

# Conclusion

# Chapter 8

# Categories for Profit

*Free advice is seldom cheap.*

*Rule #59 from the Ferengi rules of acquisition ([Behr95])*

Let us assume that the best thing that has happened to the field of information systems is the remarkable insight which Edgar F. Codd had in the summer of 1969, when he wrote a research report for his employer IBM, in which he suggested that database technology should have a *formal mathematical foundation*. The resulting data-model, the relational model, has become very well developed, assumingly because of its clear and rigorous mathematical foundation. It is true, however, that when one asks the users of relational information systems which advantage of the relational information systems they think is the most prominent, hardly anyone mentions its mathematical foundation. Even among developers, we know, the importance of the rigorous foundation is underestimated. We still conjecture that this foundation is the most important factor in developing the relational database to the maturity it has now. Similarly important is a mathematical foundation for object oriented information systems. However, the situation for object oriented information systems differs considerably from the situation for relational information systems. While the concepts of relational systems were based on the mathematical notion of 'relation', the notions of object oriented information systems evolved from practical use. This situation provides a challenge that is not unfamiliar to theoreticians: trying to capture notions from practice in such a manner that it provides a rigorous and precise understanding of the notions[1].

So why is it only a few value the mathematics as very important?. The reason probably lies in the supposition that for understanding and working with an

[1]Consider, for example, research on formal linguistics, capturing notions of natural languages

artificial language it is not necessary to know the full rigorous mathematical semantics, but only some simpler informal easy-to-deal-with intuition. We agree with this statement. But this does not imply that the mathematical semantics is not important, nor that it is the *most* important. We need some hard ground on which to built the 'easy-to-deal-with' intuition. We will give an analogy[2] to explain this claim.

When one asks a mathematician what complex numbers are, (s)he will explain that they are constructed from ordered pairs of downward closed sets of rational numbers (Dedekind cuts), i.e. the hard thing. When one asks an engineer, (s)he will explain to you the geometric intuition which can be conveyed about the complex plane; the simple thing. This is a robust intuition for which there is an elegant calculus and a nice axiomatization. Dedekind cuts are studied for foundational concerns, i.e. to confirm the correctness of the logic. For working with complex numbers Dedekind cuts are not necessary.

Although one can do many things with an intuition that is not totally rigorous, in the mathematical sense, one still needs it to verify or even build this intuition. Furthermore the (possibly less formal) intuition should in some sense be *robust* and have features like a calculus for reasoning or even a (not necessarily complete, but at least sound) axiomatization for proving some properties. For designing semantics it is necessary to construct a high level system that forms the basis for the intuition. This system should not be very involved, i.e. by not using many esoteric mathematical constructions, but by giving a direct account of the concepts that are to be understood for using the considered language. For building this system one may use as much heavy mathematical artillery as one wishes.

This chapter we summarize what has been researched in this thesis. We will discuss the analysis and the mathematical foundation we propose for object orientation. Moreover, we will answer the question: what did we gain from all of this? In other words: what is the value of the artifacts from this thesis for the information system analyst, what is the value of these artifacts for the theoretician, and which problems did we solve?

In this thesis we have performed seven tasks:

1. We analyzed the practical context in which notions of object orientation are used.

2. We analyzed object oriented concepts themselves.

3. We constructed a general and formal language for object oriented information systems.

[2]This analogy is borrowed from Dana Scott, explaining the value of his work in semantics.

4. We defined a formal model (semantics) for this formal language of object orientation.

5. We introduced themes from theoretical computer science and logic for analysis of the formal language and its accompanying model.

6. We analyzed logical properties of our formalization of the concepts from object orientation.

7. We analyzed four philosophical issues using the constructed formal (and therefore precise) language and model for object oriented information.

These seven tasks are summed up in the sections below.

## 8.1 The object oriented development practice

The analysis of the practical context in chapter 1 provides us with a number of important insights into how object oriented languages are used in software development practice. A software analyst starts with labels that have no meaning, and evolves to a model with objects and types that carry structure and meaning, but the objects may be partially, or even non-wellfoundedly, specified. This insight has a large impact on the way we should interpret the object oriented languages that the analyst uses.

For the practicing software analyst this analysis is nothing more than an interesting view on his daily work. For a theoretician, on the other hand, this analysis provides important requirements on the model for object orientation he wants to construct. We have seen the influence of this insight in the mathematization of the object oriented concepts in this thesis. Notable mathematical concepts that relate to these insights are the notion of a 'link' (or 'aspect' or 'infon'), and the 'extendibility' notion.

## 8.2 Concepts of object orientation

The analysis of object oriented concepts in chapter 2 gives an overview of the main concepts used in object oriented technology. Because these concepts arose from use in practice, they can be interpreted in several ways, and sometimes are not very precise. We have pointed to potential problems with some of the concepts, and provided an interpretation that is the basis for the formal model of object orientation in this thesis.

For the software analyst this overview provides a thorough view on the notions he uses in practice, and can make him aware of potential problems. For the theoretician this overview can form a basis for his model, as it did for our model.

## 8.3 A generalized language for object oriented information systems

The major artifact of this thesis is the general language for object oriented information we defined in chapter 3: the language of categorial graphs. This language has both graphical and textual elements, and has a formal syntax. The formal syntax enables one to precisely define semantics for the language constructs and the notions expressed in this language. Moreover we solved the non-trivial problem of 'exploding' and 'imploding' in a graphical language.

A formal language is a necessary artifact for a theoretician to do formal analysis. In practice, the language of categorial graphs gives a designer of an object oriented language[3] the possibility to map his language to a formal one which is suited for formal analysis. Moreover, an information analyst can translate the expression he writes down in his OO modeling language to expressions in the categorial graph language in order to compute mathematical properties of his model. He can, for example, then compute 'satisfiability' of the model he defined with his expressions.

## 8.4 A semantics for object oriented information systems

In chapter 4 we built a model for object oriented information systems; more specifically, we built a semantics for the language of categorial graphs. This model captures the behavior of entities in object oriented information systems, and thereby 'makes concrete' the notions of object orientation. It is the realization of our analysis. Important new mathematical concepts are the notions of 'object', 'infon' and 'extendibility'. These notions, for example, realize a driving slogan of object orientation: 'every property of an object is an object in its own right'.

A formal semantics is the target artifact of the OO theoretician. It is the construction that rigorously captures his intuition and enables deeper analysis of the notions involved. The model gives the practitioner the possibility to do formal (and thus automated) model checking, using the formal interpretation of the language of categorial graphs. Moreover it enables one to infer properties of a constructed model using the sound syntactic calculus that accompanies the model.

[3]Note that even the standard language UML is evolving. UML version 2.0 is bound to be released!

## 8.5 Methodology: semantics, logic and applications

Chapter 5 displays an overview of the scientific context in which the research that has resulted in this thesis took place. Here we have introduced for the reader the scientific tools we used for formal analysis; these are 'computer science semantics', 'modal logic' and 'substructural logic'. Moreover we discussed related research and pointed to applications of the theory that has been developed in this thesis.

Such introductions and references are good practice in science.

## 8.6 Logic of object oriented information

The analysis of the formal language and model of object oriented information systems took place in chapter 6. This analysis has built the hard ground under our intuitions on object oriented concepts. It gives insight into the axiomatics and complexity of notions from object orientation. Moreover we propose an interesting view on how to do logic, using the intuition of object orientation for a general logic.

For the logician these logics provide insight into the logical properties of an interesting domain with practical relevance. Moreover, these logics have purely logical relevance as well. For the software analyst this is the theory that validates his intuition.

## 8.7 Four philosophical issues

In chapter 7 we analyzed four philosophical issues in the light of the object oriented language for specifying information developed in this thesis. A long history in philosophy has shown that describing parts of the real world is very complex and subject to serious issues. Because the aim of object oriented modeling is also capturing parts of the real world; it is therefore only natural that in object oriented modeling we encounter the same issues. Four of these issues were elaborated.

Most practitioners of object oriented modeling probably do not realize that such issues lie in wait for their models. It is very valuable, however, to realize they do, because these issues really pose problems of consistency on their models. We show that the formal system of categorial graphs gives a clear insight into these problematic issues, which enables one to avoid misinterpretations. Hereby we show its value for analyzing complex matters of describing information in an object oriented manner.

# Bibliography


[Abiteboul90] S. Abiteboul, *Towards a deductive object-oriented database language*, Data & Knowledge Engineering, vol.5, 1990, pp. 263-287

[AbiteboulHull87] S. Abiteboul & R. Hull, *IFO: A Formal Semantic Database Model*, ACM ToDS, vol. 12, no. 4, 1987, pp. 525-565

[Ackrill63] J.L. Ackrill, *Aristotle: 'Categories' and 'De Interpretatione'*, Clarendon Press, Oxford, 1963

[Adriaans90] Pieter Adriaans, *Categoriale modellen voor kennissystemen*, Informatie, pp.118-126, 1990.

[Adriaans92] Pieter Adriaans, *Language Learning from a Categorial Perspective*, Academisch proefschrift, Universiteit van Amsterdam, 1992.

[AdriaansHaas99] Pieter Adriaans & Erik de Haas, *Grammar Induction as Substructural Inductive Logic Programming*, Proceedings of the workshop on Learning Language in Logic (LLL99), Bled, Slovenia, June 1999, pp. 117-126 (to appear in Springer LNAI series)

[AdriaansHaas00] Pieter Adriaans & Erik de Haas, *Learning from a substructural perspective*, Proceedings of the 4th conference on Computational Natural Language Learning and of the 2nd Learning Language in Logic (LLL) workshop, September 13th-14th 2000, Lisbon, Portugal, pp. 176-183, September 2000.

[AndrekaBenthemNemeti96] Hajnal Andreka, Johan van Benthem & Istvan Nemeti, *Modal Languages and Bounded Fragments of Predicate Logic*, Journal of Philosophical Logic, 27(3), pp. 217-274, 1998

[AndriesEngels94] Marc Andries & Gregor Engels, *Syntax and Semantics of Hybrid Database Languages*, in H.J. Schneider & H. Ehrig (eds.), Graph transformations in Computer Science, Springer LNCS 776, 1994, pp. 19-36

[Areces00] Carlos Areces, *Logical Engineering: The Case of Description and Hybrid Logics*, Dissertation, Institute of Logic, Language and COmputation (ILLC), University of Amsterdam, 2000

[ArnoldEtAlii00] Ken Arnold, James Gosling, David Holmes, *The JAVA Programming Language, third edition*, Addison-Wesley, 2000

[AtkinsonEtAlii89] M. Atkinson, F. Bancilhon, D. DeWitt, D. Maier, K.Dittrich, S.Zdonik, *The Object-Oriented Database System Manifesto*, Proc, of the First Int. Conf. on Dedictive and Object-Oriented Databases (DOOD 1989), Kyoto, 1989, pp. 223-240

[Baader96] F. Baader, *A formal definition for the expressive power of terminological knowledge representation languages*, Journal of Logic and Computation, No.6, 1996, pp. 33-54

[Barendrecht84] H.P. Barendrecht, *The Lambda Calculus: its syntax and semantics*, revised edition, North-Holland, Amsterdam, 1984

[BarwiseSeligman97] Jon Barwise & Jerry Seligman, *Information flow: the logic of distributes systems*, Cambridge Tracts in Theoretical Computer Science 44, Cambridge University Press, 1997

[Behr95] By Quark as told to Ira Steven Behr, *The Ferengi Rules of Acquisition*, Kangaroo pocket books, 1995.

[Benthem91] Johan van Benthem, *Language in Action: Categories, lambdas and Dynamic Logic*, North-Holland, 1991

[Benthem93] Johan van Benthem, *A Note on Dynamic Arrow Logic*, in J. van Eijck (ed.), Logic and Information Flow, Kluwer 1993

[Benthem2000a] Johan van Benthem, *Categorial Grammar and Modal Logic*, draft paper for the Workshop on Computational Linguistics and Logic, UiL OTS, University of Utrecht, Sept. 6 2000

[Benthem2000b] Johan van Benthem, *Information Transfer Across Chu Spaces*, Logic Journal of the IGPL, vol. 8, no. 6, Nov 2000, pp. 719-731

[BlackburnEtAlii93] P. Blackburn, C. Gardent, W. Meyer-Viol, *Talking about Trees*, in: Proceedings of the 6th Conference of the European Chapter of the Association for Compuational Linguistics, Utrecht, 1993, pp. 21-29

[BlackburnRijkeVenema01] P. Blackburn, M. de Rijke, Y. Venema, *Modal Logic*, Cambridge Tracts in Theoretical Computer Science. vol. 53, Cambridge University Press, 2001.

[BlackburnSeligman95] P. Blackburn & J. Seligman, *Hybrid Languages*, Journal of Logic Language and Information, vol. 4, 1995, pp. 251-272

[Booch94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Redwood City BA, Benjamin/Cummings, 1994

[Bucalo94] Anna Bucalo, *Modalities in Linear Logic Weaker than the Exponential "of Course": Algebraic and Relational Semantics*, Journal of Logic, Language, and Information, No. 3, 1994, pp. 211-232

[Buszkowski86] W. Buszkowski, *Completeness results for Lambek syntactic calculus*, Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, No. 32, 1986, pp. 13-28

[Cardelli84] Luca Cardelli, *A Semantics of Multiple Inheritance*, in: G. Kahn, D.B. MacQueen, G. Plotkin (eds.), *Semantics of data types*, Springer LNCS 173, 1984, pp. 51-67

[CardelliWegner85] Luca Cardelli, *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, vol. 17, n0o. 4, December 1985, pp. 471-522

[Cattell94] R.G.G. Cattell, The Object Database Standard: ODMG-93, Release 1.1, Morgan Kaufmann Publ., 1994

[Cattell97] R.G.G. Cattell, The Object Database Standard: ODMG 2.0, Morgan Kaufmann Publ., 1997

[CattellEtAlii00] R.G.G. Cattell & D.G. Barry, The Object Data Standard: ODMG 3.0, Morgan Kaufmann Publ., 2000

[CoadYourdan91a] P. Coad & E. Yourdan, *Object Oriented Analysis*, Yourdan Press, Englewood, 1991

[CoadYourdan91b] P. Coad & E. Yourdan, *Object Oriented Design*, Yourdan Press, Englewood, 1991

[Codd70] E.F. Codd, *A relational model of data for large shared data banks*, Communications of the ACM, no. 13, 1970, pp. 377-387

[ColemanEtAlii94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, *Object Oriented Development: The Fusion Method*, Prentice Hall, 1994

[Comm.ACM91n10] *Communications of the ACM*, no. 10, 1991

[Cousot90] P. Cousot, *Methods and logics for proving programs*, in: J. van Leeuwen (ed.), Handbook of theoretical computer science, Vol. B, North-Holland, 1990, pp. 841-993

[DarwenDate95] H. Darwen & C.J. Date, *The third Manifesto*, ACM SIGMOD Record no.3 1995.

[Dastani98] Mehdi Dastani, *Languages of Perception*, Dissertation, Institute of Logic Language and Computation (ILLC), University of Amsterdam, 1998

[Davis58] M. Davis, *Computability and Unsolvability*, McGraw-Hill, 1958

[deChampaeuxEtAlii93] D. deChampeaux, D. Lea, P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993

[Denneheuvel90] Sieger van Denneheuvel & Peter van Emde Boas, *The rule language RL/1*, in: A.M. Tjoa & R. Wagner (eds.), Proc. of the Int. Conf. on Database and Expert Systems Applications (DEXA'90), Vienna, Austria, Springer, 1990, pp. 381-387

[Dosen88] Kosta Dosen, *Sequent-Systems and Groupoid Models I.*, Studia Logica, vol. 47,No. 4, 1988, pp. 352-386

[Dosen89] Kosta Dosen, *Sequent-Systems and Groupoid Models II.*, Studia Logica, vol. 48, No. 1 1989, pp. 41-65

[Dummett73] Michael Dummet, *Frege's Philosophy of language*, Duckworth, 1973, pp. 545

[Dunn86] J. Dunn, *Relevance Logic and Entailment*, in¿ D. Gabbay & F. Günther (eds.), Handbook of Philosophical Logic III, D. Reidel, 1986, pp. 117-224

[EmdeBoas96] Peter van Emde Boas, *Computerspelen en de identificatie van objecten*, in: Kwartaalschrift van de Universiteit van Amsterdam, nummer 8, December 1996

[EmdeBoas98] Peter van Emde Boas, *Formalizing UML; Mission Impossible?*, (Position paper at OOPSLA'98 workshop #9; Formalyzing UML; Why?; How?), X-1998-03, Institute of Logic Language and Computation (ILLC), 1998

[FowlerScott00] Martin Fowler & Kendall Scott, *UML Distilled second edition, a brief guide to the standard object modeling language*, Addison-Wesley, 2000

[Frege1892] G. Frege, *Über Sinn und Bedeutung*, Zeitschr. f. Philos. u. philos. Kritik, NF 100, 1892, pp 25-50

[GanterWille99] Bernard Ganter & Rudolf Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer, 1999

[GareyJohnson79] M. Garey & D.S. Johnson, *Computers and Intractibility; A guide to the theory of NP-completeness*, W.H. Freeman and Co., San Francisco, 1979

[GeachBlack52] Peter Geach & Max Black (eds.), *Translations from the philosophical writings of Gottlob Frege*, Basil Blackwell Publisher, 1952

[Girard87] J.-Y Girard, *Linear Logic*, Theoretical Computer Science, No. 50, pp. 1-102

[Gurevich88] Yuri Gurevich, *Logic and the Challenge of COmputer Science*, in: Egon Börger (ed.), Trends in Theoretical Computer Science, Computer Science Press, 1988, pp. 1-57

[Haas91] Erik de Haas, *Object Oriented Application Structuring and its Semantics*, Master Thesis, FWI, Universiteit van Amsterdam, October 1991

[HaasEmdeBoas93] Erik de Haas & Peter van Emde Boas, *Object Oriented Flow Graphs and their Semantics*, in : Andrzej M. Borzyszkowski, Stefan Sokolowski (Eds.), *Mathematical Foundations of Computer Science 1993*, 18th International Symposium, MFCS'93, Springer, Lecture Notes in Computer Science 711, 1993, pp. 485-494

[Haas94] Erik de Haas, *Categorial Graphs: The Logic*, in: Arthur Nieuwendijk (Ed.), *Accolade '94*, Dutch Graduate School on Logic (Onderzoekschool Logica, OzsL), 1994, pp. 103-119

[Haas95] Erik de Haas, *Categorial Graphs*, in: Horst Reichel (Ed.), *Fundamentals of Computation Theory*, 10th international conference, FCT'95, Springer, Lecture Notes in Computer Science 965, 1995, pp. 263-272

[HaasAdriaans99] Erik de Haas & Pieter Adriaans, *Substructural Logic: A framework for second generation data mining algorithms*, in Paul Dekker (ed.) Proceedings of the twelfth Amsterdam Colloquium (AC99), University of Amsterdam , December 18-21 1999, pp. 121-126

[Haas01] Erik de Haas, *Adding dynamics to categorial graphs*, Manuscript, 2001

[HendersonEdwards90] B. Henderson-Sellers, J.M. Edwards, *The Object Oriented Systems Life Cycle*, Communications of the ACM, vol. 33, no. 9, 1990, pp. 142-159

[HopcroftUllman79] J.E. Hopcroft & J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979

[HughesCresswell68] G.E. Hughes & M.J. Cresswell, *An Introduction to Modal Logic*, Methuen and Co Ltd, 1968

[Husserl70] Edmund Husserl, *Philosophie der Artithmetik*, (Herausgegeben von Lothar Eley), Martinus Nijhof, Den Haag, 1970

[Jacobson85] Ivar Jacobson, *Concepts for modeling large real time systems*, Dissertation, Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985

[JacobsonEtAlii92] I. Jacobson, M. Christerson, P. Jonsson, P. Övergaard, *Object Oriented Software Engineering*, Addison-Wesley, 1992

[JacobsonEtAlii99] Ivar Jacobson & Grady Booch & James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999

[KiferLausen89] M. Kifer & G. Lausen, F-Logic: A Higher-order language for reasoning about objects, inheritance, and scheme, in: Proc. of the ACM SIGMOD, 1989, pp. 134-146

[KiferWu93] M. Kifer & J. Wu, *A Logic for Programming with Complex Objects*, Journal of Computer and System Sciences, No. 47, 1993, pp. 77-120

[Kurtonina95] Natasha Kurtonina, *Frames and Labels. A Modal Analysis of Categorial Inference*, Dissertation, Institute of Logic Language and Computation (ILLC), University of Amsterdam, 1995

[Lambek58] J. Lambek, *The mathematics of sentence structure*, The American Mathematical Monthly, No. 65, 1958, pp. 154-170

[Leeuwen93] Jacques van Leeuwen, *Identity: Quarrelling with an unproblematic notion*, LP-93-04, Institute of Logic Language and Computation (ILLC), Amsterdam, 1993

[LewisPapadimitriou81] Harry R. Lewis & Christos H. Papdimitriou, *Elements of the theory of computation*, Prentice-Hall, 1981

[Linsky71] Leonard Linski (ed.), *Reference and Modality*, Oxford University Press, 1971

[Maier86] D. Maier, *A Logic for Objects*, in: Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington D.C., 1986, pp. 6-26

[Marshall00] Chris Marchall, *Enterprise modeling with UML, designing successful software through business analysis*, Addison-Wesley 2000.

[MartinOdell92] J. Martin, J. Odell, *Object Oriented Analysis and Design*, Draft manuscript, 1992.

[Milner90] J. Milner, *Operational and algebraic semantics of concurrent processes*, in: J. van Leeuwen (ed.), Handbook of theoretical computer science, Vol. B, North-Holland, 1990, pp. 1201-1242

[OnoKomori85] H, Ono, Y. Komori, *Logics without contraction rule*, The Journal of Symbolic Logic, No. 50, 1985, pp. 169-201

[Pentus93] Mati Pentus, *Lambek grammars are context free*, Proc. of the 8th Ann. Symp. on Logic in Computer Science, 1993, pp. 429-433.

[PomykalaHaas93] Janusz A. Pomykala & Erik de Haas, *A Note on Categories of Information Systems*, in: Wojchiech P. Ziarko, *Rough Sets, Fuzzy Sets and Knowledge Discovery*, proceedings of the international workshop on rough sets and knowledge discovery (RSKD'93), Springer, Workshops in Computing, 1993, pp. 149-156

[PomykalaHaas94] Janusz A. Pomykala & Erik de Haas, *A Note on Categories of Information Systems*, Demonstratio Mathematica, vol. XXVII, No. 3-4, 1994, pp. 641-650

[PomykalaHaas96] Janusz A. Pomykala & Erik de Haas, *A Note on Categories of Information Systems*, Fundamenta Informaticae vol. 2, No. 2/3, 1996, pp. 221-227

[Pooley87] R. Pooley, *Introduction to programming in Simula*, Alfred Waller ltd., 1987

[Quine46] Willard V. Quine, *Concatenation as a basis for arithmetic*, Journal of Symbolic Logic, 11(4), pp. 105-114, 1946

[RaysideCampbell00] Derek Rayside & Gerard T. Campbell, *An Aristotelian Understanding of Object-Oriented Programming*, in Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2000, ACM Sigplan Notices, Vol. 35, No. 10, October 2000, pp. 337-353

[Reynolds80] J.C. Reynolds, *Mathematical Semantics*, in: B.W. Arden, What can be Automated? The Computer Science

and Engineering Research Study, MIT Press, 1980, pp. 261-188

[RijkeVenema95] Maarten de Rijke & Yde Venema, *Sahlqvist's theorem for Boolean algebras with operations*, Studia Logica, no. 95, pp. 61-78, 1995

[RistTerwilliger95] Robert Rist & Robert Terwilliger, *Object Oriented programming in Eiffel*, Prentice Hall, 1995

[Ross77] D. Ross, *Structured Analysis (SA): A Language for communicating idea's*, IEEE Transactions on Software Engineering, vol. 3, no. 1, pp. 16-34, January 1977

[Ross85] D. Ross, *Applications and Extensions of SADT*, IEEE Computer, vol. 18, no. 1, pp. 25-34, April 1985

[Rotterdam96] Ernest Rotterdam, *OORL*, Manuscript, Institute of Logic, Language and Computation, University of Amsterdam, 1996.

[Rounds97] W. Rounds, *Feature Logics*, in: J. van Benthem, A. ter Meulen (eds.), Handbook of Logic and Language, North-Holland, 1997, pp. 475-533

[RumbaughEtAlii91] J. Runbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object Oriented Modeling and Design*, Prentice-Hall, 1991

[ScottStrachey71] D.S. Scott and C. Strachey, *Toward a mathematical semantics for computer languages*, in: J. Fox (ed.), Proc. Symposion on Computers and Automata, Polytech Institute of New York, 1971, pp. 19-46

[ShlaerMellor88] S. Shlaerm S.J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdan Press, 1988

[Smith95] David N. Smith, *IBM Smalltalk: The language*, Benjamin/Cummings Publishing Co., 1995

[Spaan93] Edith Spaan, *Complexity of Modal Logics*, Dissertation, Department of Mathematics and Computer Science, University of Amsterdam, 1993

[SQL3] SQL-3, ISO-ANSI working draft, X3H2-93-359 and MUN-003, 1993

[Stroustrup91] Bjarne Stroustrup, *The C++ Programming Language, second edition*, Addison-Wesley, 1991

[StonebrakerEtAlii90] M. Stonebraker et alii, *Third generation Database Systems Manifesto*, ACM SIGMOD Record 19,3, September 1990

[Tailvalsaari96] Antero Taivalsaari, *On the notion of Inheritance*, ACM Computing Surveys, Vol. 28, No. 3, September 1996, pp. 438-479

[Troelstra92] Anne S. Troelstra, *Lecture Notes on Linear Logic*, CSLI Lecture Notes No. 29, 1992

[Ullman88] J.D. Ullman, *Principles of database and knowledge-base systems*, Computer Science Press, 1988

[Ullman91] J.D. Ullman, *A Comparison between Deductive and Object-Oriented Database Systems*, Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD'91), 1991, pp.263-276

[UML97] *UML 1.1 Specification*, OMG documents ad970802-ad970809, www.omg.org 1997.

[UML99] *UML 1.3 Specification*, OMG documents ad990606-ad990609, www.omg.org 1999.

[Venema91] Yde Venema, *Many-Dimensional Modal Logic*, Dissertation, Department of Mathematics and Computer Science, University of Amsterdam, 1991

[Venema94] Yde Venema, *A crash course in arrow logic*, Logic Group preprint series 107, Department of Philosophy, Utrecht University, 1994

[WarmerKleppe99] Jos Warmer & Anneke Kleppe, *The Object Constraint Language, Precise Modeling with UML*, Addison-Wesley, 1999

[WijngaardenEtAlii76] A. van Wijngaarden et alii, *Revised Report on the Algorithmic Language Algol 68*, Springer, 1976

# Samenvatting

Dit proefschrift bevat een onderzoek naar formele aspecten van object oriëntatie. Het bevat een semantische beschrijving van een generieke taal voor modelleren en specificeren van object geörienteerde informatiesystemen. Deze semantische beschrijving geeft inzicht in de constructies die gebruikt worden bij het object geörienteerd modelleren. Dit inzicht kan voor een aantal doeleinden worden gebruikt, bijvoorbeeld voor het ontwikkelen en verfijnen van geautomatiseerde object geörienteerde software ontwikkelsystemen of voor het ontwikkelen van optimalisatie technieken voor algoritmen die object geörienteerde data verwerken.

In de praktijk van object geörienteerd modelleren en object geörienteerde databases hebben de gebruikte talen meestal geen formeel wiskundig fundament. Belangrijke voorbeelden van zulke talen zijn UML (de industriële standaard taal voor object geörienteerd modelleren) en ODMG (een voorgestelde standaard voor object geörienteerde database talen). Ondanks dat deze talen geen formele basis hebben word er toch een aantal 'semi-formele' taken mee uitgevoerd. Voor UML bestaan er zogenaamde 'code-generatie' algoritmen, die UML expressies vertalen naar expressies in object geörienteerde programmeertalen. Ook bestaan er algoritmen voor query optimalisatie op object geörienteerde database modellen, die zijn opgeschreven in UML of ODMG. Omdat deze talen geen formele basis hebben, zijn deze systemen verdacht voor inconsistenties en ambiguïteiten. Daarom kunnen bijvoorbeeld sommige UML expressies niet worden vertaald. Onderzoek naar de mathematische fundamenten van zulk soort talen beoogt de ontwikkeling van systemen die object geörienteerde talen gebruiken te ondersteunen, door het wegnemen van onduidelijkheden en het voorzien in een consistente interpretatie van deze talen.

Het onderzoek naar semantische aspecten van object oriëntatie is ook interessant vanuit een ander oogpunt. De concepten die belangrijk zijn in object oriëntatie komen uit de praktijk, en zijn daar ontwikkeld om informatie analisten en software ontwikkelaars te helpen op een nette en precieze wijze hun informatie modellen op te schrijven. Deze informatie modellen zijn afspiegelingen van een

werkelijke situatie. In deze zin raakt dit onderzoek thema's uit de filosofie. In de filosofie is het immers een belangrijk doel om op precieze wijze aspecten van de werkelijkheid te beschrijven.

In dit proefschrift worden de volgende zeven onderwerpen behandeld:

1. In hoofdstuk 1 is een analyse gemaakt van de praktijk context waarin object geörienteerde talen worden gebruikt. Deze analyse is belangrijk, omdat deze voorwaarden stelt aan de (mathematische) interpretatie van de talen. Een informatie analist begint normalerwijze aan zijn modelleertaak met het benoemen van objecten. Dit benoemen resulteert in eerste instantie alleen in 'labels', omdat er aan het begin van de analyse nog geen kennis zal zijn van de structuur en het gedrag van de objecten (daar moet de analist juist achter zien te komen). Toch zal de analist reeds in het begin delen van zijn model moeten opschrijven om hierover te kunnen communiceren. In de loop van zijn analyse zullen de objecten meer betekenis en structuur gaan krijgen, maar op elk moment zal het zo kunnen zijn dat objecten slechts partiëel, of zelfs 'ongefundeerd' beschreven zijn. De taal en de interpretatie daarvan zal met deze 'partialiteit' en 'ongefundeerdheid' om moeten kunnen gaan.

2. Hoofdstuk 2 bestaat uit een analyse van de concepten die over het algemeen als belangrijk voor het object geörienteerde paradigma worden bestempeld. Omdat deze concepten uit de praktijk, zonder formele basis, zijn geboren, bestaan er vaak meerdere invullingen van de concepten. In de analyse wijzen wij op mogelijke problemen met het interpreteren van de concepten en doen wij een voorstel voor het (informeel) interpreteren van deze concepten. Deze analyse vormt de basis van de formalisatie in de opvolgende hoofdstukken.

3. In hoofdstuk 3 introduceren wij de formele taal die wij ontwikkeld hebben voor de mathematisering van object geörienteerde talen. Deze taal heeft zowel grafische als tekstuele componenten en wordt aangeduid met: "de taal der *categoriale grafen*". Deze taal is een generieke taal waarop de meeste basiscomponenten en constructoren uit de object geörienteerde modelleer en database talen kunnen worden afgebeeld. In dit hoofdstuk worden syntactische eigenschappen van deze taal behandeld, en wordt er een probleem opgelost dat te maken heeft met het imploderen en exploderen van grafische taalcomponenten.

4. Het formele model voor object geörienteerde informatiesystemen wordt gedefiniëerd in hoofdstuk 4. Dit model vormt het semantisch model voor de taal der categoriale grafen. Daarnaast definiëren wij een formele interpretatie van deze formele object geörienteerde taal in dit model, en een

syntactische calculus voor de taal. In andere woorden: wij definiëren een *logica* voor object oriëntatie.

5. In hoofdstuk 5 schetsen wij de wetenschappelijke context van de logica voor object oriëntatie die wij zojuist hebben geconstrueerd. Hierin behandelen wij de notie van formele semantiek in de informatica, en de logica theorieën die ten grondslag liggen aan onze logica voor object oriëntatie. Dit zijn *modale logica* en *substructurele logica*. Verder laten wij een aantal aan onze logica gerelateerde systemen de revue passeren. Aan het eind van dit hoofdstuk wijzen wij nog op een tweetal toepassingen van onze logica: op het gebied van object geörienteerd software ontwikkelen, en op het gebied van 'data mining' op object geörienteerde data.

6. In hoofdstuk 6 vindt de logische analyse plaats van onze logica voor object geörienteerde informatiesystemen. In dit hoofdstuk worden de concepten ontleed in opzichzelfstaande logica's en de thema's 'axiomatisering', 'compleetheid' en 'computationele complexiteit' behandeld. Hierbij geeft de logica inzicht in de concepten die een rol spelen bij object geörienteerde systemen. Daarnaast doen wij ook iets terug voor de logica: wij doen een voorstel om op een bepaalde manier logica te bedrijven, die gebaseerd is op intuities voor object oriëntatie.

7. Hoofdstuk 7 bevat de analyse van vier filosofische vraagstukken, waarbij wij gebruik maken van de formele taal die in dit hoofdstuk ontwikkeld is. In de filosofie is het bekend dat reeds beschrijvingen van basale concepten uit de werkelijkheid tot lastige vraagstukken kunnen leiden. Aangezien object geörienteerd modeleren tot doel heeft zaken uit de werkelijkheid te beschrijven, is het niet meer dan natuurlijk dat wij daar dezelfde lastige vraagstukken tegen kunnen komen. In de praktijk wordt echter meestal over deze lastige problemen heengestapt, omdat de informele talen uit de praktijk vaak geen heldere interpretatie geven. Dat gebeurt terwijl deze vraagstukken toch voor inconsistenties kunnen zorgen en daardoor tot fouten kunnen leiden, omdat de expressies in de informele taal steeds voor verschillende interpretaties vatbaar zijn. Wij laten met behulp van vier voorbeelden zien dat je door gebruik te maken van onze formele taal deze problemen helder kunt krijgen. Zodoende kun je een bewuste keuze maken in het interpreteren van expressies waarin lastige filosofische vraagstellingen schuilgaan. Hiermee tonen wij aan dat onze logica voor object oriëntatie behulpzaam is voor het analyseren van complexe object geörienteerde informatiesystemen.

*Titles in the ILLC Dissertation Series:*

ILLC DS-1996-01: **Lex Hendriks**
*Computations in Propositional Logic*

ILLC DS-1996-02: **Angelo Montanari**
*Metric and Layered Temporal Logic for Time Granularity*

ILLC DS-1996-03: **Martin H. van den Berg**
*Some Aspects of the Internal Structure of Discourse: the Dynamics of Nominal Anaphora*

ILLC DS-1996-04: **Jeroen Bruggeman**
*Formalizing Organizational Ecology*

ILLC DS-1997-01: **Ronald Cramer**
*Modular Design of Secure yet Practical Cryptographic Protocols*

ILLC DS-1997-02: **Nataša Rakić**
*Common Sense Time and Special Relativity*

ILLC DS-1997-03: **Arthur Nieuwendijk**
*On Logic. Inquiries into the Justification of Deduction*

ILLC DS-1997-04: **Atocha Aliseda-LLera**
*Seeking Explanations: Abduction in Logic, Philosophy of Science and Artificial Intelligence*

ILLC DS-1997-05: **Harry Stein**
*The Fiber and the Fabric: An Inquiry into Wittgenstein's Views on Rule-Following and Linguistic Normativity*

ILLC DS-1997-06: **Leonie Bosveld - de Smet**
*On Mass and Plural Quantification. The Case of French 'des'/'du'-NP's.*

ILLC DS-1998-01: **Sebastiaan A. Terwijn**
*Computability and Measure*

ILLC DS-1998-02: **Sjoerd D. Zwart**
*Approach to the Truth: Verisimilitude and Truthlikeness*

ILLC DS-1998-03: **Peter Grunwald**
*The Minimum Description Length Principle and Reasoning under Uncertainty*

ILLC DS-1998-04: **Giovanna d'Agostino**
*Modal Logic and Non-Well-Founded Set Theory: Translation, Bisimulation, Interpolation*

ILLC DS-1998-05: **Mehdi Dastani**
*Languages of Perception*

ILLC DS-1999-01: **Jelle Gerbrandy**
*Bisimulations on Planet Kripke*

ILLC DS-1999-02: **Khalil Sima'an**
*Learning efficient disambiguation*

ILLC DS-1999-03: **Jaap Maat**
*Philosophical Languages in the Seventeenth Century: Dalgarno, Wilkins, Leibniz*

ILLC DS-1999-04: **Barbara Terhal**
*Quantum Algorithms and Quantum Entanglement*

ILLC DS-2000-01: **Renata Wasserman**
*Resource Bounded Belief Revision*

ILLC DS-2000-02: **Jaap Kamps**
*A Logical Approach to Computational Theory Building (with applications to sociology)*

ILLC DS-2000-03: **Marco Vervoort**
*Games, Walks and Grammars: Problems I've Worked On*

ILLC DS-2000-04: **Paul van Ulsen**
*E.W. Beth als logicus*

ILLC DS-2000-05: **Carlos Areces**
*Logic Engineering. The Case of Description and Hybrid Logics*

ILLC DS-2000-06: **Hans van Ditmarsch**
*Knowledge Games*

ILLC DS-2000-07: **Egbert L.J. Fortuin**
*Polysemy or monosemy: Interpretation of the imperative and the dative-infinitive construction in Russian*

ILLC DS-2001-01: **Maria Aloni**
*Quantification under Conceptual Covers*

ILLC DS-2001-02: **Alexander van den Bosch**
*Rationality in Discovery - a study of Logic, Cognition, Computation and Neuropharmacology.*

ILLC DS-2001-03: **Erik de Haas**
*Logics For OO Information Systems: a Semantic Study of Object Orientation from a Categorial Substructural Perspective*

ILLC DS-2001-04: **Rosalie Iemhoff**
*Provability Logic and Admissible Rules*