Formal Description of Anything in the Universe

Arve Meisingset

I have not been able to find an academic publisher for this book, as they all operate within strict academic disciplines, and do not publish cross overs. I have also studied Open Access publishing, but these publishers require evaluation by a selected academic team, where I will have to cover all costs. This implies that a sponsoring organization is needed, which I do not have after retirement.

From the above, I have chosen to publish the book on my own web. This does not give the recognition, visibility and marketing that a professional publisher can give. I believe that the book gives substantial contributions to mathematical philosophy and computer science.

One famous publisher wrote:

“frankly spoken, if I had seen a possible publication platform, I would have suggested it:

- for a philosophical book, it is too much on computer science
- for a mathematical book, again it seems too close to computer science
- for a data base book, it is too philosophical and theoretical
- for a theory book, it is too philosophical
- and of course, it's not a coffee table book either.”

To my mind, this rejection is the most fantastic review I can get. I have tried to build bridges between these disciplines. When having crossed a bridge, I have watched the discipline with the critical eyes of a foreigner. I have studied the subjects carefully, modified and used some, and thrown other stuff into the dustbin.

Oslo 15.03.2017

Arve Meisingset

Please send comments to arve.meisingset@movinpics.com.

Formal Description of Anything in the Universe

by

Arve Meisingset

Drafted 01.01.2014 - 01.04.2014
Reviewed and improved 01.04.2014 - 15.03.2017

Published on movinpics.com

Type of document:	Report

Preface

Purpose

This book addresses

- What exists?
- How is it described?

These topics are seen through the eyes of a computer scientist. However, the book is a cross over between several subjects. The book touches upon philosophy of knowledge, philosophy of mathematics, philosophy of nature, foundation of computer science, theology, logic, linguistics, cosmology, engineering and design. Hence, readers of many subjects have shown interest into reading.

The book is written for the benefit of people who want to choose an approach for mathematical modelling or specification of IT systems. Should the developers use relational mathematics, predicate calculus or conceptual structures when writing their specifications? We recommend use of none of them.

The developer should specify the structure of data as they will appear at the human-computer interface of an IT system for management of data about the particular Universe of Discourse. This structure of data has some similarity to the Entity-Relationship Model for database applications, but is also different. We will use the headings at the screens as class labels of the variable fields, and will introduce no other variable or constant term. We will see the sequences of fields on the screen, including their role labels, as statements about their Universe of Discourse. We will study design of human-computer interfaces as design of statements and languages, representing a view of the Universe of Discourse.

End user data appear in the context of other end user data, and they cannot be mapped to concepts or phenomena without this context. You may use relational databases for implementation, but we discourage the use of any context free approach for the specification.

The specification of syntax of end user data has impact on worldviews.

Most human developers of mathematical models and specifiers of IT systems think they are observing their Universe of Discourse from the outside. This outside view leads to inappropriate notions of entities, relationships and roles. Therefore, this book takes a different view. It considers the observer to be inside his Universe of Discourse, and it considers the specifier to be inside his descriptions. The observer is moving inside its Universe of Discourse while recording and interpreting it. Hence, phenomena are observed in context, and data are defined in context. This approach leads to a worldview of roles having conditions, but leaves out both entities and relationships.

This book considers the observer and the specifier to be a non-human. What exist for the observer appears on its input media. We will not rely on humans' imaginations about predicates, sets, strings, interpretations and abstractions. The insider view of the universe and the view of a non-human observer constitute the fundamental perspectives of this book.

In this book, we will see the universe from the perspective of a database application. If the database application cannot observe the phenomenon, then the phenomenon does not exist. The application sees the inside of its interfaces. The book is not about databases, but about their applications, architectures, languages and design methodologies. The book is not about

programming, but its perspectives can be implemented by programming. You may see a practical implementation in Annex G.

The universe and its phenomena cannot be more general than the language we use to represent them. Hence, this book reasons about the universe and its phenomena by reviewing the languages for their representation. This study belongs to mathematical philosophy. We do not need to study physics in order to develop a Grand Unified Theory of everything.

We argue and show that both the phenomena and their description can be built up by three-dimensional lists of lists:

- The first dimension is the direction of observation from phenomena observing other phenomena
- The second dimension is made up of phenomena being observed in parallel
- The third dimension contains algorithmic functions on the phenomena

The three dimensions must appear at the cosmological scale via the human scale down to the Planck length. Phenomena at any scale are represented by only one kind of particle, colon (:), in all three dimensions. There is no need for more symbols - terms and operators – to express anything. Nametags may be attached to the colons. We reduce the fundamentals of language theory to become identical to the geometry of the physical universe.

The colons make up a three dimensional tree of threads. The threads do not use any alphabet. These treads are not visible when using truth-value logic or when using a one-dimensional Turing tape of 1-s and 0-s. The colons are the phenomena within the observer. They are not the truth-values of propositions about the phenomena.

We claim and show that this basic language is practical in use, leads to clear specifications and efficient implementations. In doing so, we abandon the Relational model for databases and Set theory, and we abandon truth-value logic.

Humans may introduce the Universe of sets to explain the appearance of the physical universe. However, the Universe of sets and the physical universe are very different from each other, they are disjoint, and there does not seem to be any observable connection between them. For a human brain inside the physical universe, it is hubris to think that it can map to the disconnected Universe of sets outside the physical universe. This non-crossable bridge is known as the mind-body duality. In this book, we stick to observed physical phenomena and their data representations. Hence, we abandon abstractions like sets, points, lines, infinities and continuity.

In classical logic, you map from similar inscriptions to their common string, which is mapped to a set. We do not use strings and sets. We map between inscriptions at several layers. In the External terminology layer of an observer, no two inscriptions can denote the same phenomenon, even if the inscriptions look the same. Hence, our framework for interpretation of statements is radically different from classical logic, and we provide a new foundation.

We treat classes as prototype inscriptions, which are copied into instances. By copying inscriptions, we do away with variables.

When using rewriting grammars, the final statement contains leaf nodes of the syntax tree. We introduce the notion of an attachment grammar, where the final statement contains the entire syntax tree. The syntax tree shows the context. Hence, much will be different from your previous learning.

The book addresses the following questions:

- What does the physical universe look like when being observed?

- How appropriate is classical logic for describing the physical universe?
- What structure and capabilities should the observer possess?
- What language is needed to express the observer's view?
- What is the impact of using the proposed observer framework and language?

This book contains five parts, addressing these questions in the given sequence.

Readers' guide

Every section starts with an ingress explaining its purpose.

The last subsection of each section contains an Outlook from the current text.

Annex B contains a summary of the technical contents of the book, but it gives neither motivations nor explanations.

Annex G shows a simple implementation of the approach, architecture and language proposed in this book.

Background

Until the mid 1970-ies, I considered mathematical logic to be as safe and correct as if it were God's own language. Then I developed several large database applications for management of the telecommunication network in Norway, and had to define the data. This was before I knew the Entity-Relationship model and Relational model for database definitions. I saw that my formulations resembled Set theory; so I tried to apply it. However, I wanted to define the terms exactly as the users would see them on the screens. The users use significant duplicates and terms that are local to other terms. This violated the requirement of having globally unique constants as of Set theory. The Set theory did not work, as I had reused the same term in different contexts with different meanings. I would need means to manage contexts, not only of variables, but also of constants. As I started to explore this, I lost my faith in classical mathematical logic. I have been an atheist, skeptic and rationalist long before this, and still remain so. Now, I also became a skeptic in classical mathematical logic.

In my subsequent studies, I realized that the syntactical difficulties came from the Type-token principle, which affects all of classical logic. I additionally met difficulties with entity migration, as if I associate information with entities or relationships, these associations may have to be changed when adding new information. The entity migration difficulties could be avoided by associating information with roles only, and not with entities and relationships. So, I would need a worldview where roles are observed from roles, and discard entities and relationships. This worldview showed to be closely related to the mentioned syntactical challenges.

My interest in using mathematics to describe an observer's world may be seen already in my diploma thesis at the Norwegian Technology University, Trondheim, Norway, where I developed "Atferdsmodell for fisk" (Behavior model for fish, in Norwegian only, January 1973). I developed a mathematical model for how the fish can navigate in the dynamic action-discomfort image of its world. The thesis was written under my master study of Technical Cybernetics. The permissible actions of the fish defined the dimensions of the space. A weighted sum of stimuli relative to goals of the fish measured the discomfort associated with each point in the space. Goals and sub-goals were established by correlations, which correspond to the transfer function of the nerve cells. The fish navigates towards minimum discomfort, which will change with time as the fish adapts to stimuli. This was a

continuous model of an observer's perception of its world. In the current book, I develop an entity/role based model of an observer's perception of its Universe of Discourse.

During the early 1980-ies, I participated in the ISO work on Concepts and Terminology for the Conceptual Schema and the Information Base, see Annex A [36]. The ambition was no less than to define a language for defining and describing everything in the entire Universe. The reason for this goal was that in databases we want to manage data about anything in the entire Universe. Hence, we need a data definition and manipulation language that is general enough for describing anything.

Several formalisms were proposed, but the most influential one from a philosophical point of view was Interpreted Predicate Logic. The philosophy behind the notations of the report was naive realism. I did not agree in this choice, as I by then had become an extreme nominalist and phenomenologist. However, I did not have much influence on the work, but I learned a lot.

In 1986, I was a key contributor to defining a study within Telematics at the University Studies at Kjeller, and I drafted a compendium on Specification Languages and Environments, see Annex A [110]. Thereafter, I have been teaching and supervising many master and doctor students on my views on formal languages. These ideas have been used in a large set of Recommendations on data definitions within the International Telecommunication Union Standardization Sector (ITU-T).

For several decades, I have been the Telenor technical coordinator towards ITU-T, and have been following the development of a large set of programming and specification languages. See the ITU-T Z-series Recommendations. The Unified Modelling Language, UML, has been influenced and adopted by ITU-T. UML may not be characterized as a conceptual schema language, but may be said to belong to the same school of thinking. UML has also been applied in the work on Open Distributed Processing (ODP). See the ITU-T X-900 series Recommendations. For many years, I have been the liaison officer from ITU-T to ISO on ODP, but my world-view and language preferences have been different.

My standards work has only been part time or free time. Most of my career I have been working as a research scientist, developing large database applications, methods, tools, data definitions and IT architectures. Data design has been a key to my approaches. During recent years, I have been working as an IT consultant in Telenor operations world wide.

If you read the list of references of the compendium, you will see that when authoring it, I have studied a large set of literature on formal languages and mathematical philosophy. See Annex A Bibliography. Mathematical philosophy became my key interest, but practical projects in Telenor did not allow me to dive into this subject, until now after retirement. During the work on the compendium, I started to think on developing a book on my views on formal languages, the language architecture and philosophy that explicitly or implicitly follow the choice of language. The result is the current book, which I have drafted during the first three months of my retirement. I have spent much more time waiting for comments and improving the draft.

Remarks on the manuscript

I have received one request to compare my proposals with the Vienna Development Method (VDM). This is not done in this book. I note that the world view, language architecture and languages are very different from the various versions of VDM. My proposals are very different from the specification language UML, as well.

One reader has asked for footnotes, as terminologies from several disciplines are used throughout the book. I think that any reader can look up these words on the internet.

I have also been asked to tell what is original work, and what is established knowledge. This book is not a presentation of the state of the arts, where I make some incremental contributions to existing arts. The book tries to make a disruption, where I review and propose changes to the fundamentals of the arts. Therefore, I do not make references to very recent results in any field of research. Most of my references are to the old classical texts.

During the 1980-ies, I have made presentations in international fora on every part of the book. The Bibliography tells when I have published papers on each topic. The archive from Telenor Research contains still earlier working papers. This book gives discussions, perspectives and explanations that are not found in the publications.

Finally, I have been asked to explain the functioning of human thinking and consciousness. I claim that you cannot understand the functioning of humans if you do not understand the functioning of automata. However, to understand the general functioning of automata is not sufficient to understand the particular functioning of humans. This book is about languages of automata. Humans have to understand these languages in order to fully understand their own languages.

Most philosophers think that understanding of consciousness is a hard problem, and that it requires notions of something immaterial. I do not think so. I think it only requires the ability to build a virtual space and to place the observer himself into it. This follows from my diploma thesis where a living organism is defined to be a process that can observe its own energy transformation and attempts to maintain it. In this book, I refer to consciousness as a transaction handler.

I use my computer science language throughout the book, as I believe that computer science perspectives will give contributions to philosophy, logic, transhumanity and humanities.

Acknowledgements

Professor emeritus Peter Linington, School of Computing University of Kent, United Kingdom and Professor Birger Møller-Pedersen, Department of Informatics University of Oslo, Norway have been reviewing parts of the book. I want to thank both for their help. Also, I want to thank Master of Science in Applied Mathematics/Computer programming, Cecilia Irgens, and Bachelor of Science in Mathematics, David Ludlam, for their help to improve my English and the presentation.

The draft book has been presented in two colloquia for doctor students and professors in Informatics at the University of Oslo.

Contents

Introduction

The ambition of this book is no less than developing a notation for describing everything in the entire physical universe. The notation appears to be practical, and is already used to define data for huge database applications. I admit that the expressions may become somewhat cluttered, as you explicitly have to navigate into the right context. However, simple applications may become simple. See Annex G.

The expressions using the notation and its interpretation appear inside the physical universe, and hence the notation cannot be more general than the physical universe itself. By exploring the most general notations, we explore the limitations set by the physical universe, and hereby develop a general Theory of everything in this universe. Therefore, by studying formal languages, we can develop a theory of the functioning of the physical universe, without studying the physical properties of this universe. This follows from the observation that the physical universe cannot be more general than the language used to describe it. The physicist is constrained by the language we give him. There is no valid thinking outside the language tool for describing and reasoning.

It has been an accepted truth that Predicate calculus – maybe Higher order Predicate calculus – can be used to describe everything in the entire Universe. A Universal Turing automaton is capable of executing everything formulated in Predicate calculus.

The first claim is known already from Wittgenstein's writings. The second claim is known as the Church-Turing thesis [33].

This book questions the notion of describing something the way it has been done in classical mathematical logic, e.g. by use of Predicate calculus [20], [21]. This notion is questioned by others, eg. Quine, and is modified in his Free logic [28], [29]. The discussions in this book start with questioning the so-called Type-token principle [8], [9], [10], which comes before definition of any notation and of any logic.

I will stress that the objective of this book is not to question the deductive capabilities of various notations in classical mathematical logic. The attacks are on subjects like name spaces, denotations, truth, worldviews, associations, language architecture, infinities and paradoxes. I find Set theory to be very different from my understanding of the physical universe, everything in it, and from my needs of how to describe them.

The book is organized into five parts.

- Part 1 introduces the Perspective.
- Part 2 discusses use of Classical logic.
- Part 3 proposes a Language architecture, and formulates requirements on the contents of the architecture.
- Part 4 introduces the proposed Language notions.
- Part 5 discusses the impact of the language proposal.

Already in the first part, sections 2 to 5, we introduce two perspectives on describing a system: An outside view, and an inside view. The outside view is the traditional approach. The inside view comes from moving inside the system when observing and describing it. I argue that the entities and associations observed this way will be different from those observed from the outside. I introduce and recommend a context dependent, phenomenological [7] and nominalistic [6] worldview as an alternative to the traditional context free, realistic [1], [2], [3], [4] or conceptualistic [5] worldviews.

There is another important difference to classical logic. We start in sections 2 and 3 with inspecting a graph which has no nametag. In section 4, we attach nametags to every node in the graph. We do not, as in Model Theory, interpret statements against an abstract structure of sets. Rather, our approach ensures that the statements have the same structure as of the phenomena, with some additions, like the nametags.

In part two, sections 6 to 9, we introduce languages of classical logic. These are adapted and used in computer science. Rather than taking these languages and their notions at face value, this section discusses their foundation with a computer scientist's eyes. Does classical logic provide a proper foundation for describing everything in the entire physical universe? The text concludes that it does not. We will abandon use of truth-value logic and of Set theory.

Part three, sections 10 to 13, introduces a software architecture for interpreting and executing language statements. Software provides a much richer environment for doing this than when creating imaginations about statements on paper. We introduce a Data transformation architecture that translates data in one language to data in another language. The translation takes place at two levels of transformation from concrete to abstract syntax for each language. These transformations are replacing the Model theoretic interpretations. The transformations apply inscriptions only, do not use strings and sets, and not variables and quantifiers.

Part four, sections 14 to 18, introduces the proposed language notions. First, we introduce a language for normalized expressions of the External terminology layer. Next, the language notions are extended for expressing composite non-normalized statements of the Contents layer. Also, sections on Layout, Semantics and Execution are included. For the External terminology layer and the Contents layer, we use what we call an Existence logic, where the existence of some data affects the creation or deletion of other data.

Part five, sections 19 to 22, discusses possible extensions of the current book, and applies the notions introduced to discuss More on existence, applies the same notions on some Paradoxes, and contains a Post script.

Finally, there are Annexes on Bibliography, Popular summary, Graphic notation, Documentation, Detailed requirements, Entity migration and Example implementation.

I believe that each part of the book, except part five, may be read in isolation, without reading the previous parts, but all parts contribute to proposing a language theory for everything.

In this book, we will use the term Universe to denote the set of everything in the physical universe. A Universe of Discourse is the subset addressed in a particular application.

In this Introduction, we also need to introduce the terms Notation, Language and Formal, as they are used hundreds of times throughout the book.

The mapping from a phenomenon to the symbols used to describe it is called a Notation. Also, a collection of symbols and an organization of symbols we call a Notation, even if we have no mapping from phenomena.

The symbols together with the grammatical rules for combining them are called a Language. The symbols may be called terms. In most language theories, the terms are defined in a glossary, and the grammar is defined as a set of production rules over the terms. The production rules are used to define a syntax tree, where the terms appear as leaf nodes. In part four of this book, we will allow the terms to occupy every node in the syntax tree, and they cannot be defined or left alone independently of the syntax tree.

The grammatical rules make one Language different from another. If Norwegian and Swedish had only been using different words, but the grammars were identical, then we might have called them two versions or Notations of the same Language. However, the grammars are slightly different, as well, so we treat them as being two different Languages.

The syntax tree of an alphanumeric notation and a graphical notation may be identical, except for the symbols/terms occupying the leaf nodes. Therefore, we will call them two different Notations of the same Language. Also, when showing actual examples, we talk about Notations, and not Language, even if we could have used the expressions Example of language or Example of Statement within a Language.

A Formal definition of a Language requires that you create a complete definition of the Language using some other Language, eg. using classical logic. This will not be done in this book. Rather, we discuss what a universal Language should look like. Hence, this book may be understood as a contribution to mathematical philosophy rather than to mathematical logic. However, the proposed Language will be sufficiently well specified for implementation in an IT system. This observation serves as a defense for use of the term Formal.

We will compare different Formal languages throughout the book.

Part 1

Perspective

- What does the physical universe look like when being observed? -

1. Two perspectives

What will a world look like when it is observed without being described?

In classical logic, we typically start with statements that are interpreted against some structure. We choose in this book to start with discussing the structure first, without having any statements or identifiers of entities.

This section discusses the differences between an outside and an inside view of a graph. Will the outsider and the insider see different images of the same graph? Will the recording structures of images be different? Also, the insider's perception of where he is and what he sees is discussed. How can an insider distinguish different views of the same node, or similar views of different nodes? Can he count surrounding nodes? May it help to add details, such as colours of each node? Can time variation, such as insertion or deletion of nodes help or clutter the recognition? What happens if we shift from observing idealized nodes in an abstract graph to observing physical entities?

You may respond that it is impossible to give right answers to all these questions. However, this is the kind of challenges we as humans are meeting every day when navigating in our environment. Hence, we need some practical answers, even if we know that we may come short in the general case, eg. when trying to follow the individual cars in a busy street or individual balls in a pool game.

God's perspective

When a mathematician is studying a graph, he typically depicts it or imagines it to be seen from the outside. The graph is his Universe of Discourse, and he is God, who perceives all the nodes and edges between the nodes in his entire Universe, all at once. This God-given worldview we call a context free worldview. In this perspective, the mathematical God may even perceive several disconnected graphs – several Universes - simultaneously.

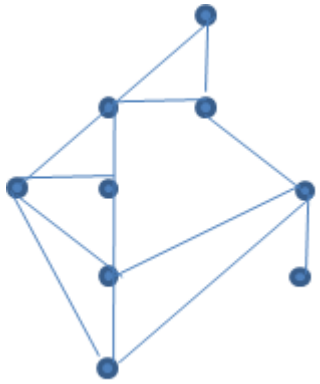


Figure 1.1. Example graph as being seen by an outside observer

The graph may represent a telecommunication network, persons, owned cars, houses, stars, elementary particles or anything, and relationships between these. The nodes represent individuals, and we are in the first parts of this book not discussing classification of the individuals. Hence, one node represents Person 1, another represents Person 2 etc., and we

are not representing the class Person. During the discussion, we may zoom in on particular nodes and edges, and may find sub-graphs in each.

Most books on philosophy and linguistics discuss classes and not individuals, which are focused in this book. The treatment of classes in the latter part of the book will be different from most other texts, as in this book each class will look identical to its individuals.

An Ant's perspective

When the mathematician wants to inspect the graph, he works in a different manner. He starts at one node, crosses over an edge to another node, and is parsing the graph as if he is an Ant. He crawls like an Ant over the edges to neighbour nodes, and he is creating paths through the graph by crawling from node to node.

When using this Ant behavior, the mathematician has no way to cross over to another disconnected graph. In order to switch over to the other graph – to the other Universe – the mathematician has to turn to God's context free worldview. He can then land on a node in the other graph and there continue his tedious inspection. This tracing of a graph from node to node inside it, inside the Universe, we call a context sensitive worldview. Sometimes we will call this a context dependent worldview. What nodes to be seen, and how to perceive them depend on which node the mathematical Ant is coming from.

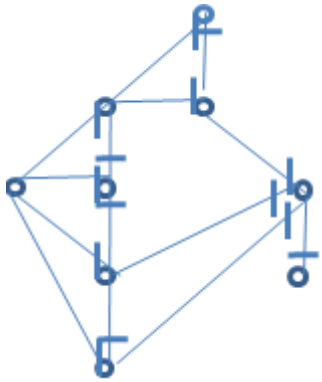


Figure 1.2. Example graph as being seen by an inside observer; thick short lines indicate the observed phenomena; the observer starts at the leftmost node, and only one observation is made along each edge

Differences

From the previous subsections, it is evident that a mathematician who takes a context sensitive worldview only, cannot see everything that can be seen from a context free worldview. By traversing between nodes inside a Universe you cannot pass over to another Universe. By parsing and observing over the edges in the context sensitive worldview, you cannot simultaneously observe all the nodes perceived in the context free worldview.

The Ant capabilities

We assume that the Ant

- can observe images of neighbour nodes in some order
- can visit these images in that given order, or reverse to the already visited images
- can move forward to (an image of) a particular neighbour node

- can backtrack from an image of a particular neighbour node to the previous observation position

We assume that the Ant

- can distinguish between forward movements and backtracking
- can remember the observation position(s) from where it came from in forward movements
- cannot identify individual phenomena or nodes by tags or other means; they all look alike

The Ant is acting like a mountain climber who is rappelling down a wall with a tight rope without loops. The nodes are pins or bolts, which may bend the rope. The Ant knows how to come up, and sideways from each position. It does not remember all the positions it has visited. This deficiency is not due to its limited mental capacity, but lack of tags to distinguish the positions. Hence, the Ant may visit the same pin several times via different routes, but does not know that it has been there. We will discuss how it can reason about the various pins and various routes to the pins.

The Ant is observing neighbour pins when it is at a location. It has no prior map or description of the wall. However, the mountain climbing analogy is not right in every detail, as the Ant may go down many legs to subordinate pins, and then reach a pin that may be reached by fewer legs along another route from the root pin. The reached pin may even be the root. Subordinate pins to the left and right is a good analogy for the ordering of images.

Nodes with walls and roof

Suppose the nodes are three-dimensional buildings. Then God would only see the roofs of the buildings, while the Ant would see the walls only. The walls and roof correspond to images, or projections, of the building. God's single image of the house is not identical to the Ant's many different images of the house. God and the Ant do not know of each other images.

In the above distinction between views, we have assumed that both God and the Ant are physical beings, obeying physical laws, and can only observe photons coming directly towards them. We do not study a God who can break the speed limit of light, intercept every light ray in the entire Universe, along the entire path of each simultaneously, without disturbing its direction, speed or energy. A God that is outside physics and logic is outside our minds.

The Ant is positioned at one particular node when observing a particular wall of a neighbour node. Hence, it has only a two-dimensional view of this wall. Likewise, we assume that God is sitting still in his chair when observing the roof of this node, and he does not move freely around in space to get a glimpse of the Ant's view. He cannot create a three-dimensional view of the building. God has a flat worldview, while the Ant has a richer worldview, as it has many images of God's single image.

God has only one image of each node. This image of the roof he perceives to be the node itself. Therefore, God may be an absolute and naïve realist, thinking that the nodes exist independently of him observing them, and of how they are observed.

The Ant may have no knowledge of God's worldview. The Ant has only a set of images of walls of the buildings. The contents of the images depend on from which direction, which edge, the Ant is coming. We assume that there is one image per wall, and one edge per wall and image.

The walls represent the roles of the buildings as seen from neighbour nodes. However, the Ant has no direct means of finding which walls belong together, to make up a building.

The environment

The Ant may zoom in or move to a wall. However, when being sufficiently close, the Ant may not be able to see details that distinguish one wall from another.

The Ant may look out along several edges towards walls of neighbour nodes. One of these is the wall or node from which it came from.

When looking out, the Ant may realize that the image of the surroundings is identical to the view from another wall. Hence, the Ant may choose to group, or associate, the images from which it may observe the same environment – the same number of images. If you were God, you would say that all the associated images belong to the same roof, being the observation platform, but the Ant does not know about the roof. Therefore, the Ant will associate all the wall images having a common surroundings. These images are the walls of the same building.

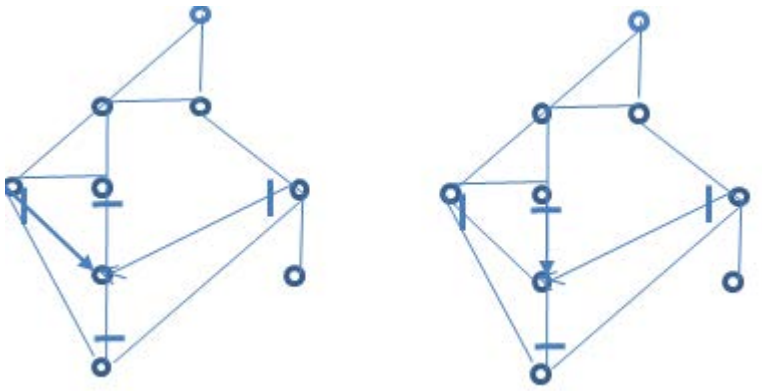


Figure 1.3. The Ant may come along different edges (indicated by different arrows in the two figures) to the same node

The only way to recognize the observed node in Figure 1.3 is to look at its surroundings (we see four neighbour nodes). If the nodes have no tag, one environment cannot be distinguished from another having the same number of neighbor nodes. The graph contains four nodes from which you can see four other nodes. Hence, the identification of a particular node is not unique by using the approach from this subsection. The building under the roof and behind the walls is neither observed directly by God nor by the Ant. However, both God and the Ant may derive or create other entities, such as buildings, behind, under or above their images.

Becoming a phenomenologist

Through associating images (of the same node), the Ant acts as a phenomenologist. The Ant considers the images to be its phenomena, to be all there is, and it associates the phenomena. It creates an association of images for each building, but does not know of buildings.

If the phenomena are images of the walls, then different detailed properties or characteristics may be associated with each phenomenon, e.g. the walls may have colours. These characteristics may be fundamentally different from the characteristics associated with the

roof, or with the node in an absolute realistic worldview. For example, the walls may have colours, the roof may have tiles, and the building may have volume.

In case the nodes are buildings, we have seen that God may perceive the roof, and the Ant may perceive the walls. A mathematical God does not perceive the same phenomena as of a mathematical Ant.

The colours, tiles and volume are details, which do not appear in our original graphs in Figure 1.1 to 1.3. In order to find which walls belong together to make up a building, the Ant will have to observe the common surroundings of these walls. If the surrounding is the same for a set of walls, these walls are at the same place, and the Ant may conclude that these walls belong together (to be walls of the same building).

More details

The graph provides little details to distinguish the surroundings, as several nodes may have the same number of neighbour nodes. The picture becomes more precise if we include images of neighbours of neighbour nodes. By doing so recursively, we may get different and distinct pictures of each surroundings.

At a next level of precision, we may consider dynamic movement, creation and destruction of nodes and their accompanying images. If we focus along an edge, we may see that an individual phenomenon disappears.

If we observe one neighbour node at a time, the image of each node is undistinguishable from other images of the same or other nodes.

When making a new observation of the set of neighbour nodes, the number of images of neighbour nodes may change, but we may not know which one appeared or disappeared. However, when observing neighbour of neighbour nodes recursively, we may get enough information to distinguish the individuals. See on functional dependencies in section 3.

When observing neighbour of neighbour nodes recursively, we may even see that a node has changed position in the list of neighbour nodes. The notion of time comes from observing change of relative positions of the phenomena in a list of phenomena. If the positions do not change, then there is no notion of time. If we introduce space, with angles and distances between the nodes, we will get a more precise knowledge of positions and movements of the phenomena.

The above discussion may help to distinguish phenomena in a practical application, but there is in the general case no foolproof method to distinguish phenomena in graphs.

Edges

We are now ready to consider the role of the edges between the nodes. Edges are elementary paths on which the Ant may cross over to other nodes. The edges may also be considered being lines of interaction, communication, or observation between two nodes. The Ant makes observations along an edge in order to observe a neighbour node. Hence, the Ant may not be observing the edge itself; it is observing the neighbour node through or by the edge. The edge is the transmission medium or empty space through which the image of the other node is conveyed.

If the edge is no physical medium, which can be perceived from the outside, then the outside observer, i.e. the mathematician taking God’s context free worldview, cannot observe it,

because there is nothing to observe. The mathematician may insert an entity to represent the edge, because this is the only way he has available to represent the Ant's view of the other node.

When the mathematician takes the Ant role, he may just see an image of another node, and may see no edge.

As we have indicated with nodes being buildings, the Ant's image of the other node, and God's image of the same node may not be identical. They are not seeing each other's images. This is also the case for the edges. The Ant and God may have different images of the same edge, or none of them may see any edge even when the nodes are somehow connected.

God may need to invent an edge in order to explain the Ant's behavior, but the Ant may observe the neighbour node and cross over to it without having an edge to bridge between the two nodes. Hence, the edges may only be created in God's imagination. The Ant may only see images of nodes. The Ant sees neither edges nor nodes.

Note that neither God nor the Ant has yet produced a description of their Universe. They only have their images, and derived data are created in the images. If God can see graphs, he may imagine edges everywhere, even when the Ant sees none.

Note that even the recognition of nodes and walls requires derived data. These may be processed and stored inside the image itself or among the images, and do not require a description of the images.

Stars

Suppose the nodes are stars on the heaven. We readily accept that God may know of more stars than what the Ant may see with its naked eye. The stars seen directly by the Ant are the neighbour nodes. The Ant perceives no edge between itself and these stars. The stars are all it sees, and it does not perceive all stars in God's Universe. God also perceives the stars, and he may perceive the Ant's behavior depending on what stars it perceives.

God may deduce that the Ant is perceiving a particular star, but God does not perceive any edge between the star on which the Ant is located and the star at which the Ant is looking. The photons of the light from the stars to the Ant do not reveal themselves to an outside observer, like God.

God is introducing artificial edges to tell which star is observed from (an Ant at) which star. If the Ant were a space traveler, it could travel to its neighbour stars and make observations of still more stars. God may keep track of all these observations and travels, and create an edge for every jump between stars. The edges are not directly observed by him; they are only deduced from observing the Ant's behavior, and are added to God's image.

Associated images

From the discussion above, we started out with a graph of nodes connected by edges in a context free worldview, and have replaced the graph by a set of images and associations between them, maybe without any edges, in a context sensitive worldview. The number of images will most often be far larger than the number of nodes. The images will be dependent on in which direction the observer is looking – along the edge.

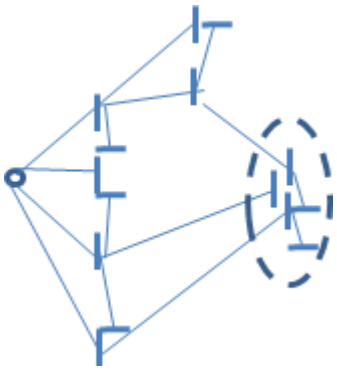


Figure 1.4. The inside observer's graph of images;
the root node is depicted as a small ring.

Note in Figure 1.4 that similar images (short thick horizontal lines down to the right in the ellipse) are seen from two different images (up to the left in the ellipse) of the same node, indicating that the images (down to the right) are the same as seen from one and the same node (up to the left in the ellipse). The reference to each individual image is somewhat clumsy in this text, because we have not and will not yet introduce identifiers of each image.

The Ant cannot distinguish the edge from the phenomenon. The Ant sees the phenomenon through the edge. Hence, the T symbols in Figure 1.4 represent the phenomena and their context. The top of the T represents what is observed, and the foot represents the phenomenon.

The foot of the T symbol points at the context from which the phenomenon is observed. The phenomenon does not have an existence independently of any observer and its position, like in a context free world. The phenomenon only exists as seen from an observer at a given position – and time.

We may combine the context free and context sensitive worldviews through associations, but for the time being, we will keep them apart and investigate the characteristics of each.

Duplicates

In the above discussion, we have assumed that there is no duplicate node. Two duplicate nodes are two nodes that are connected to the same set of neighbour nodes. This means that from two duplicate nodes we see similar sets of images; we have no way to distinguish the two sets.

An Ant at the Moon and an Ant at the Earth may have no way to distinguish their surrounding stars. If the Ant made observations of the stars as seen from the Earth, then was put to sleep and moved to the Moon, making new observations of the stars, it would not know where it was, and could think that it remained on the same earth. Its view of the heaven remained unchanged after the move. Hence, the Ant cannot decide if the Earth and the Moon are two different entities.

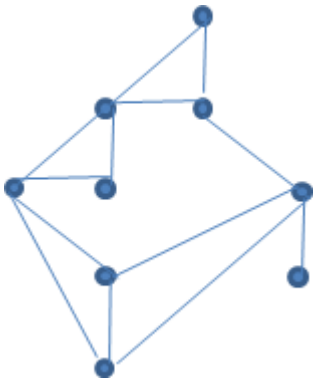


Figure 1.5. Duplicate nodes;
the two nodes towards the center and bottom of the graph are duplicates

An Ant at one of the stars may perceive the Moon and the Earth to be two separate phenomena. If the Universe were static, the Ant may not be able to decide if the Earth and the Moon are two separate entities or two images of the same entity.

If the Ant moves from a star to the Earth or the Moon, its images of the heaven seen from these two places may become identical. The Ant concludes that the Earth and the Moon are duplicate nodes. This means that they are two different phenomena – as seen from the stars -, but they are indistinguishable concerning their characteristics.

Parallel edges

So far, we have assumed that there is only one edge between two nodes. Suppose there are many edges, such that an Ant may see several images of the same node. If moving to any of these images, the Ant would not be able to tell if it has arrived at the same or different nodes. This is the challenge that astronomers meet when light is bent via several paths through the Universe. The astronomer may infer that there are several images of the same star, e.g. by comparing variability of the images. In a static universe, the astronomer may not have this option.

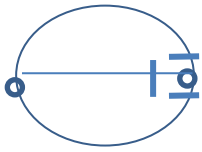


Figure 1.6 Three images of the same node seen via multiple edges

If the Ant can observe the node under its feet, it may observe that the place on Earth and the place on Moon are not the same. However, now we have extended the Universe to include entities on Earth and on Moon in order to make the observation of surrounding entities different. We have extended the original graph, such that Earth and Moon are no more duplicate nodes. Colours on walls and tiles on roofs have already served a similar purpose.

Recapture

We started out with a graph where the nodes were considered to be point-like entities without any internal structure. In order to motivate for the different views of each node, we have extended the node notion to be of buildings with one surface for each edge to each node. This

is not what the classical mathematician will accept. He considers the graph to be a conceptual structure, a creation of pure thought, consisting of point-like nodes, and immaterial edges are connecting the nodes in his context free universe. The classical mathematician is a conceptualist. The representation of the graph on paper or other media, he considers to be imperfect, while his own abstraction has no flaw.

The classical mathematician thinks that there cannot exist different images of the same node. All images of a node along different edges to the node are all identical. The classical mathematician thinks that he is acting according to his divine rights when he creates his abstractions. The practical applications of mathematics prove that he was right from the start of time. He does not accept anyone questioning his convictions. However, we will question his position by exploring the definition of the predicate Exist, in a section 4.

Absolute realism and conceptualism are very similar worldviews, where matter or concepts may exist independently of the observer, his position and properties. However, conceptualism takes a more idealistic and abstract view than of realism. Phenomenology is different, as here the observer, his position, properties and instrumentation chain are taken into account when creating images and descriptions of the Universe. The Ant is a phenomenologist.

In phenomenology, the phenomenon is the basic notion. There is no entity behind or outside the phenomenon. The abandoning of entities is known from the philosophy of structural realism. Here the relationships and their grouping into relations are the basic notions, and not entities.

In relational mathematics, a relation may have roles in each end. In phenomenology, we abandon the relation as well, and keep the role as the basic notion. The role is a phenomenon, and is not a phenomenon of something, like an entity. The phenomenon is all there is as seen from another phenomenon. However, we may correlate and associate phenomena, and will come back to this.

Finally, we have from the very start, assumed that the phenomena, like stars, planets and elementary particles, are distinguishable from their surroundings. This distinction from the surroundings may not be obtainable for fluids and wave functions. In theory, these are continuous phenomena, but any observer will perceive them as discrete phenomena. If they cannot be discretized, the observer will not know of them.

The discussions in this section introduce the topics, and indicate a direction of thought, and the discussions herein are not indicating a complete answer to how phenomena are distinguished, associated or identified.

In this book we will have to take a closer look at the convictions of the classical mathematician, and discuss if they are appropriately founded, or we should replace them by a worldview which depends on our context of observations.

The above introductory text tries to motivate for the view that when observing a Universe from the inside or from the outside, you may not get the same information about this Universe. A context free worldview of a Universe is not identical to, and does not convey the same information as of a context sensitive worldview of the same Universe.

The historical view of our Earth and its surrounding universe was that the Earth had a reserved position, and the Universe was circulating around it. This worldview was replaced by the great astronomers who saw that the Earth was just one of many heavenly bodies, and the Earth had no reserved or elevated position.

The classical mathematicians have considered themselves to have an elevated position outside their Universe, imagining that they can observe and describe their Universe in a

context free way. In this book we will explore, discuss and challenge this worldview. We will investigate if this worldview can be replaced by a context sensitive worldview, and what benefits this worldview can give. Can and should a flat earth view of the graph be replaced by a context sensitive structure of phenomena? What would such a structure look like, and how would it behave? In this section, you have got a taste of the answers.

The Grand Unified Theory of Everything

The task of finding a general language for describing anything in the Universe is equivalent of finding a Grand Unified Theory of everything (GUT).

Suppose the Universe has ten dimensions, but you with your instruments can observe and experience only three particular dimensions. You cannot observe or experience the others. You will then not know where you are or what phenomena you are encountering in these other dimensions. You could certainly create a description with ten free variables, but you would have no means to map the variables to the ten dimensions. Therefore, I believe that physicists have made a mistake on the mappings between free variables and physical dimensions. I believe they are very different notions. I believe it makes no sense to claim that the Universe has more dimensions than what we can observe. Therefore, I believe that our notion of the Universe has to be constrained to what is observable.

We would like to have a language theory that can span everything in the entire Universe. Since we cannot map to something outside our observable Universe, the theory cannot be more general than the Universe.

Some thinkers believe they have cognitive capabilities to map outside the physical universe. I do not think so. I believe that the statements have to be made by physical means inside the Universe, the mappings to phenomena have to be made by physical means, and the Universe itself is physical. If we observe a phenomenon, we classify this phenomenon as a lump of matter or as being a relation between phenomena of matter.

Suppose we want to define our theory by using logical equations. Then each equation will state a constraint on what is permissible. However, we do not want any constraint, as we want to allow for everything in the entire Universe. Therefore, we have to remove all the constraints. Hence, we get our GUT by stating nothing. Total silence is the answer to our quest. This book is about the contents of this nothingness.

This is not so exceptional. Most books are about nothing. They make assumptions, which are giving limitations, and have to invent means to add to or bypass the limitations. We will explore how we can avoid any limitation.

Silence is the answer if you use logical equations. However, we may not use logical equations to express the GUT. Physicists think they have to study physics to find the GUT. Then they have to express their theory in some kind of formal language. Hence, the physicists' theory cannot become more general than their language theory. We will explore the limitations that are made implicit or explicit in the language theory.

Outlook

In this section we have been discussing how a graph is observed from the outside and from the inside without having any identifier on the nodes and edges.

We have seen that the graph will have to contain some variation for the observer to recognize a place being visited via several routes. The place must maintain some similarity during

several visits. If changes happen to the graph, they need to be permanent through several visits as well.

The last subsection on The Grand Unified Theory of Everything provides a first discussion on what is the most general theory of the physical universe, what language is needed for describing it, and what is the mapping between the statements and the phenomena of the physical universe.

2. Phenomenology

What does an observer's image of his world look like?

In this section, the observer will use a camera. The images will appear as photos or structures within photos within the camera. The photo is used as a reference frame for positioning of the phenomena, their identification and organization structure.

The phenomena are on the photo inside the camera, and are not entities out in some real world.

Like in the previous section, we will discuss observations and recognitions without introducing nametags on the phenomena. Hence, the phenomena will be made up of pixels. We will study a world that is an organization of pixels.

God being a Cyborg

We have in the previous section discussed how a mathematician may perceive a graph from the outside or inside. Each image of a node we call a phenomenon. How does the mathematician distinguish the phenomena? Suppose the phenomena are images of stars on the heaven.

We start by assuming that the mathematician is God, and he observes the graph.

The mathematician may fix the images to a reference frame by shooting a photo of them, and place a coordinate system along the edges of the photo. Then each phenomenon may be assigned a coordinate. This way, within the context of this photo, the mathematician has a means of positioning the image of each star. The mathematician considers his photo, camera angle, time and location to be a reserved position in his God-given worldview. If these parameters are not God-given, he has to refer them to some other references, which are commonly known and agreed with other gods.

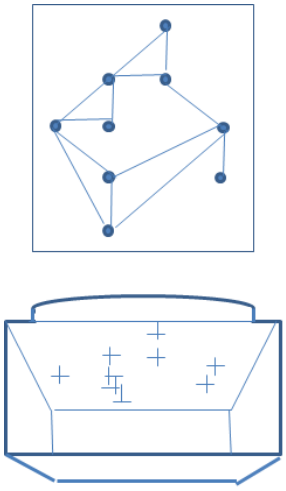


Figure 2.1 God's photo of the graph.
God may or may not see the edges

We see that this photo worldview is not as elevated and constant as the mathematician would like to believe. The photo, angle, time etc. may change, and when being confronted with other photos, the mathematician is easily put down on earth to the Ant's worldview.

The photo is an image inside the camera being controlled by the observer. Hence, the star phenomena in the photo are inside the camera. The phenomena are indicated by the feet of the T-s in the previous section, and objects being depicted are indicated by the tops.

When a star fills up most of the space inside the photo, the observer claims that he is close to the star. When the star takes up just one pixel of the photo, the observer claims to be far away from it. Here far and close refer to the size of the camera viewing angle being taken up by the observed entity. By zooming in on an entity, you are close to it. Hence, the observer has no understanding of the physical distances to the entities.

> The camera metaphor is not just speculation. See use of the Virtual Travels technology on movinpics.com. Here you are travelling through series of still pictures. In each still picture you zoom in and animate along an S curve in perspective up to the next photo. This is a particular use of the Ken Burns effect, which is put together into a near continuous interactive real time video. The principle is patented by the author in Norwegian patent 323509, European Patent EP1924971, and United States Patent US 8,896,608 B2. You will see that the angles are distorted as you zoom in, as the viewing angle is reduced compared to that of a real travel. It is not likely that the human eye can do better than Virtual Travels, as you can only see your thumb nail sharply when you stretch out your arm. The rest of the image in your eye is very blurred. The difficulty with the Virtual Travels technology is that the human user may look anywhere in the still picture. Hence, the animation has to come close to the real world, and not only to the human perception of it. The animation has to integrate perceptions in many directions.

This description of God's use of a camera will be useful to explain the Ant's view of the world. The camera may create 2D photos of a 3D world, it may create 3D photos, it may create animations over time, like some kind of video, or it may only create a one dimensional list of all observed nodes, as being seen from the current node.

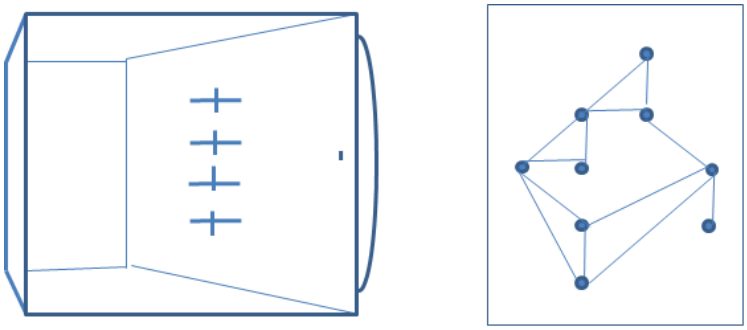


Figure 2.2 The Ant's view of the nearest stars as a list, being seen from the leftmost star/node

The Ant's extended capabilities

The Ant is integrated with a camera

- The Ant shoots a photo of each neighbour node
- The Ant organizes all its images into one integrated image

See more on the Ant's capabilities in the previous section.

The Ant being a Cyborg

We have learned how God may observe and record the graph in his context free worldview. We will now describe how the Ant may do a similar task.

The Ant starts at a node. It picks up its camera and shoots a photo of all surrounding neighbour nodes.

Next, the Ant moves to an image of a neighbour node, and shoots a new photo from this new position. This way, the Ant may traverse the entire graph, shoot photos and traverse to one of the images in it. This way, the Ant is creating a graph as depicted in Figure 1.4 in the previous section.

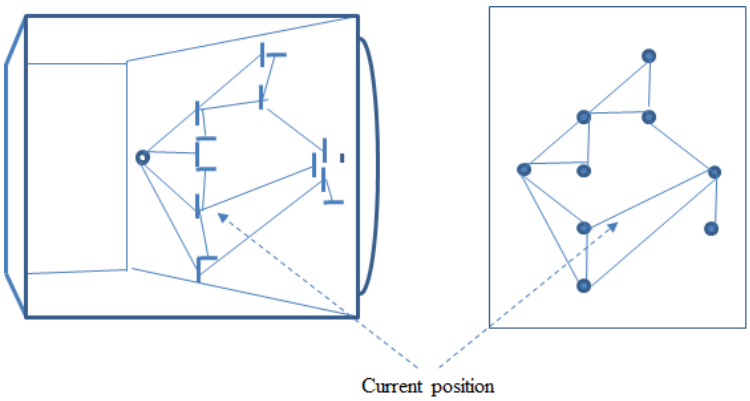


Figure 2.3 The Ant's composed photo of the graph

Suppose that the Ant has a means of backtracking, such that it knows that it is at a particular node at which it has been before; then it can move along another edge and explore the entire graph through multiple paths.

However, the Ant may without backtracking come to nodes at which it has been before. How does it know? The only way to know is to compare photos and sequences of photos, to settle if they are identical to images and sequences of images it has covered before. This is though not a foolproof test, as the graph may very well have similar regions. However, to test similarity is the only means available to the Ant.

In order to check for similarities, the Ant or its camera need to have a capability to store information about all parsed paths, and be able to search through and test for similarities. The Ant or its camera will have to build an internal picture of images of all the nodes in the graph; and there may be several images of the same node dependent on the edge over which the observation is made. This is illustrated in the ellipsis in Figure 1.4.

The Ant is positioned at an arbitrary node. Each photo provides a 360 degrees view of all the neighbour phenomena. Each phenomenon is an image of a neighbour node. Depending on the functionality of the camera, these images may make up a list of phenomena. Alternatively, the images are placed in a space, e.g. the flat area of a two-dimensional photo. In this case, the Ant may scan the entire area of the photo, and create a one-dimensional list of phenomena, as well.

The Ant may move to the neighborhood of any of these images, and shoot a photo from this new position. This way, it may continue until photos are shot from all nodes along all edges. There will be two phenomena, one in each direction, along every edge.

When comparing sub-graphs, the Ant may compare more than neighbor phenomena; it may compare neighbours of neighbours recursively. Comparison of sub-graphs does not give a guarantee for identity, but similarity of larger sub-graphs increases the likelihood for identity within a finite study.

If the Ant is forwarding and backtracking in the list of lists of images in a photo, it will know where it has been, and will be able to navigate to the next image not being visited yet, or to go back to any of the images that already have been visited.

If the graph is purely conceptual, where the edges out of a node have no ordering, the outside positioned God will have no means to know which edges have been visited, and which have not. However, if he can attach nametags to the edges, he may know.

If the Ant creates a one-dimensional list of images as being seen from each node, the Ant may come back to any phenomenon in its static universe.

When having done a tour around in the graph, shooting photos from each new node, and coming back to a node that has already been visited, the Ant faces challenges. How does it know that it has already been here? Every tour can be a loop, and how does it know? The Ant can only compare photos and sequences of photos. If they are alike, the Ant may assume that it has come back to a place where it has been before. However, then the Ant has disregarded the option that there may be several similar sub-graphs of the graph. One place in the graph may be indistinguishable from the other, and you may go back and forth without knowing. The Ant cannot see what God in his wisdom can see, and it will never know. This is the case for static graphs. If the graph were dynamic, where edges and nodes may disappear and pop up, the Ant's reasoning about the graph become even more challenging.

The only strategy that the Ant can use to map the graph is to traverse the graph through a tree of paths from the start node, put all the paths into one integrated photo, and then compare if branches of the tree look identical, and to merge them if they do. This way, the Ant is creating a minimal tree, which is satisfactory for explaining all paths through the graph. This minimal tree is described in Part 4 section 13 External terminology language.

The challenges with comparing sub-trees and deciding if they are the same or different are known from everyday life. If you have a cyclic time understanding, you may perceive the same time of the day to reappear each day with some variation, you may perceive the seasons of the year to reappear, and you may perceive life spans of humans to reappear, like in reincarnation. In a linear time perspective, each event is different from every other event even if they have similarities.

Figure 1.4 in the previous section shows a tree of phenomena. Here only phenomena in one direction along each edge are depicted; there may be phenomena in both directions.

The Ant may observe one phenomenon in each direction along an edge. If the graph contains n nodes and e edges, the maximum number of phenomena is 2e. The number of phenomena is fully independent of the number of nodes. Hence, the items in the Ant's graph of phenomena are very different from the items in God's graph of nodes. However, given God's graph, you may derive all possible Ant graphs. God may only know which graph of phenomena is used by a particular Ant by studying the Ants' behavior.

If different Ants choose different start nodes, there will be one structure for each start node. We will have to study dynamic graphs in order to find common worldviews of several Ants. See on functional dependencies in the next section.

Discussions

We have outlined how an external mathematician, called God, may see an internal mathematician, called the Ant, who is positioned in a graph and moves around in the graph. The Ant does not see the graph the way God sees it. The Ant is equipped with a camera, and its phenomena appear inside the camera. Neither God nor the Ant have identifiers on the phenomena that they observe.

The Ant will traverse paths in what it perceives being a tree of phenomena. The Ant may find commonalities between the branches and state references between the commonalities.

Somehow, God has to do the same as the Ant, when he states that an edge is connecting two nodes, and the same nodes are involved in other connections. However, God's worldview is flat, while the Ant's worldview is a recursive list of lists with references between the items. The Ant's view is like graphs at many levels with references between items at the same or different levels.

The Ant's camera does not need to be a closed box, which nobody but the owner can look into. The camera may be open, and the various images may be spread out on a table. Hence, God may see the Ant's images. However, now God's universe no longer contains only the original graph; his universe is extended by the Ant's camera, photo, images, position etc.

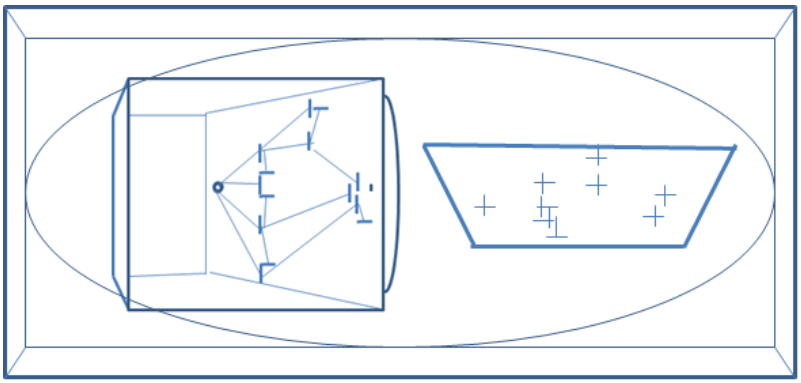


Figure 2.4 God's view of the graph (to the right) and of the Ant's view of the graph (to the left); both being seen by God

Not only may God see the Ant's images, but other Ants may do, as well. In particular, the Ant may see its own previous images when shooting new ones. This association between images as seen from different nodes is what is depicted in Figure 2.4, by placing the images close to each other around the node that they are images of. Note that the images are not out in the real world of the original graph; they are inside the Ant or its camera, placed at a neighbour node, at the foot of the T symbol. We learn that the Ant may create its Universe of images by recursively observing them from other images.

As an example; the road authority may see a vehicle, the car registration authority may see a car, and the car owner may see his owned car when observing the same node out in the real world. Neither vehicles, cars nor owned cars exist out in the real world; they are all images within the observers - road authority, car registration authority and car owner, respectively. In a graph of these phenomena, we may associate images that are images of each other. Maybe, an owned car can only exist if it corresponds to a car, and maybe the car can only exist if it corresponds to a vehicle. This topic we will come back to, but note in this example, that we do not need an Ant to observe the phenomena. Each phenomenon is itself an observer of all its neighbour phenomena. The Universe consists of phenomena observing each other.

Each image is placed at a foot of a T. The top of the T indicates what is depicted. The trunk indicates the mapping from the object to the image. This mapping is implemented by an instrumentation chain. Satellites may shoot photos of a star. The photos are transferred as bit strings by radio communication to earth, by optical fibers to computers, then processed in several computers to finally be presented to a researcher. The quality of the entire instrumentation chain has to be ensured, but the chain may use multiple technologies for the transfer and processing. This communication is the topic of measurement and instrumentation theory, very well combined with information and computation theory.

As indicated by the car example, the observer may not be a closed box, as being indicated by the camera metaphor. The observer may very well be a community, like a tribe of Ants, an organization, a set of computers, a country etc. But we will need to be clear about what is inside and outside the observer, what is the observer, what are his characteristics, and from where he is observing. This is the additional complexity that comes with context sensitivity.

We additionally have to keep track of time. The time when events take place at a star may be light years away from the recording of the image on the earth. This distinction of timings is the cause of the famous dispute between realists and phenomenologists. A realist may claim that elementary particles did exist before we had instruments to observe them. A phenomenologist may say that the elementary particles started to exist when we observed them. He means that they existed in the past when seen from us after the means of observation have been created. For people living before this creation, the particles did not exist. The realist will claim that entities, like particles, may exist by themselves independently of observers. This topic we will explore in the section on Existence. However, the reader should by now not be surprised about my position.

Outlook

In this section, God represents an outside observer of a graph. An Ant positioned on a node inside the graph represents the inside observer. We have equipped the observers with a camera each. All phenomena appear as images inside the cameras.

God in his context free worldview observes or imagines nodes and edges. The Ant in its context dependent worldview observes phenomena through God’s edges, but sees no edge. The phenomena are roles of God’s nodes, but the Ant sees no node.

An observer may be open, such that God may see the images inside the Ant’s camera. The observer may even be a community, where its images are spread out for anyone to see.

We need neither God nor Ants to observe the phenomena. Phenomena may observe other phenomena recursively. A government may observe a registration authority that may observe a registered car. The government and the registration authority are the observers.

We will note that the phenomena are called roles in relational mathematics. However, roles in relational mathematics are associated with relationships between entities, while we in this book argue that phenomena may have their own existence; they do not need to be phenomena of something. Phenomena are first class citizens.

The phenomena are organizations of pixels inside the observer. A pixel is an elementary data item, and it needs not have a nametag. The pixel notion may indicate observation of light only. I do not constrain the observation to light. What a dendrite of a nerve cell can observe at a given moment, I call a pixel. Using several dendrites, the observer can locate the source of a signal at its body or outside it, and associate its feelings with this location. The feelings are observations of internal states of the observer.

A reader of philosophy will know that phenomenology is through its history associated with many other kinds of worldviews than what has been presented in this section. We do not care about the history of phenomenology; we redefine it within our own technical perspective and for our own technical use.

3. Nominalism

How can we describe an observer's world of images?

We have in the previous sections discussed how a mathematician may perceive a graph from the outside or inside.

In this section, we will discuss how the observer can associate nametags with the phenomena in the graph. We will discuss the organization of these nametags, and discuss how they map to the structure of phenomena. We will introduce relative distinguished names, which map to the insider's view of phenomena.

God being a clerk

When taking God's context free perspective, the mathematician may have two boxes with nametags. One box with numbers, and one box with words made up of letters. He may have decided to have just one instance of each number, and one instance of each word. Hence, they are all globally unique.

One way to create globally unique numbers is to use a calculator, and add one for each used number. The mathematician may similarly consider the alphabet to be a number system and add one, to get a new word. The number and word order tells the usage sequence. This makes the numbers and words unique.

The two collections of numbers and words are called name spaces. The mathematician holds one name space in each box. We have chosen to design the name spaces such that their contents are disjoint. This may not be strictly needed.

The mathematician then assigns a word to each node, and a number to each edge. He may do so by gluing one nametag to each node and to each edge. This way, he denotes all entities in his Universe, and he can refer to each entity by a number or a word. If he knows which entity is an edge and which entity is a node, he might have used the same name space for both, as the combination with the category terms, edge or node, would make the identifiers unique and give the same information as when having separate name spaces.

The numbers and words are the identifiers within his Universe. The associations between identifiers and real world entities – the pasting of the tags on to the entities - are called denotations. The associations the other way, from entities to identifiers, are called notations.

There is no rule for sorting of the identifiers, so the alphabetical or numerical sorting states nothing about the structure of the Universe. The identifiers may even be spread over several Universes, and still the identifiers are globally unique within each Universe.

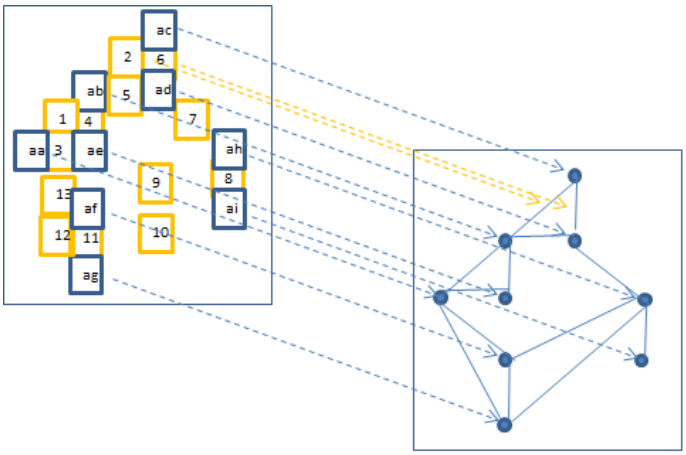


Figure 3.1 God's denotation of nodes and edges.
Denotation mappings are shown with dashed arrows; only two denotations of edges (no. 2 and 6) are shown, but all nametags are depicted

Note that the fixation of the identifier to the real world entity is essential. If we do not know this correspondence between identifiers and real world entities, we do not know which entity is denoted by which identifier. The management of the correspondences between identifiers and entities is called identity management.

In everyday language, people are often using terms for which they have no denotation. Father Christmas may be such a term, and hence people are free to associate the term with anything or nothing. This is the root of superstition. Use of unbound quantifiers, like all and some, is another example. People may use these without having any means of pointing out the set that is to be denoted. Hence, they do not know what they say. This explains why denotations and identity management are so important.

If we want to track Father Christmas to ensure that he is one and the same all the time, and not a fake one, we will have to put a nametag on him. We will have to ensure that the nametag is unique, and put in a control regime, such that the tag cannot be replicated or moved to another entity. If we do not put this identity management regime in place, we will be fooled by all the fake saints or gnomes, who will love to fool you. By using identity management means, we ensure that we relate to the same entity all the time, but of course, we do not ensure that this entity has super powers, or is the one that others have been talking about. Maybe, they are each talking about different entities, and some may be talking about someone who does not exist.

There may be synonym nametags on the same entity, like Santa Claus, Sinterklaas, Saint Nicholas and Grandfather Frost. The denotations may point out the same entity, different entities, no entity, and the interpretation may be culture dependent.

We may have terms that do not denote anything, but still are useful. One example can be the split of a person's name into several parts. Each part may not denote a person or anything else; still you treat each as a separate term.

Some terms may be introduced to give an overview of other terms. A circuit in the telecommunication network may be such an example. We may never be able to observe the circuit in its totality; we only observe its components at various sites. Sometimes the circuit goes through open air as a pulse coded radio signal. The circuit term tells what is connected or affected, but its physical existence as a phenomenon may be questioned. We may change

the components and routing of the circuit, while the terminations remain the same, and we may have several circuits between the same terminations, because they have different routings.

We want in this book to be explicit about what is denoted by each term, and which terms do not denote anything.

Note that the term entity is ambiguous, as it may mean a conceptual entity, a real world entity, a phenomenon or a data entity. Much of the literature on identity management seems to speak about identification of real world entities, like a person or a subscriber, which they believe are out in the real world; what they actually achieve is to identify data entities, like a telephone number in a telecommunication network. The person or subscriber may own this telephone number, but they may not themselves be denoted by the telephone number. Fingerprints and other biomarkers may be copied and faked. Hence, you identify only the data, and not the real world entity.

To manage the denotation mapping between identifiers and entities is no trivial task. If the graph is large, the mathematician may have difficulties with ensuring that each entity has got an identifier.

Suppose that the nodes are the visible stars on the heaven. The mathematician may fix images of them to a reference frame by shooting a photo of them, and place a coordinate system along the edges of the photo. Then each star may be assigned a coordinate and an identifier. Within the context of this photo, the mathematician has a means of identifying the stars. The mathematician considers his photo, camera angle, time and location to be a reserved position in his God-given worldview. If these parameters are not God-given, he has to refer them to some other references, which are commonly known and agreed.

We have learned how God may assign identifiers in his context free worldview. We will now outline how the Ant may do a similar task.

The Ant's clerical capabilities

- The Ant creates a tree of nametags, which is isomorphic to its tree of phenomena
- The Ant creates a one-to-one mapping between the nametags and the corresponding phenomena
- Each nametag is unique within the scope of its superior nametag only

Isomorphic means a one-to-one correspondence.

The expression superior nametag indicates a nametag which is closer to the root of the tree, as the tree is turned upside down with the root at the top.

The Ant being a clerk

The Ant starts at a node. It picks up its camera and shoots a photo of all surrounding neighbour nodes. It assigns an identifier to each image of a neighbor node in this photo. The identifiers are unique as seen from the start node.

Next, the Ant moves to a neighbour node, and shoots a new photo from this new position. Now the Ant takes a new name space, and assigns a unique identifier to the image of each neighbour node. The name space may have identical contents as of the first name space. Therefore, the identifiers may overlap. Now the identifiers are only unique within each

position of the Ant and its camera. These positions correspond to nodes, or more precisely to images of nodes being shot along a particular edge.

The Ant may this way traverse the entire graph, shoot photos and assign identifiers to each image of a node as seen via a particular edge. This way, the Ant is creating a graph as depicted in Figure 3.2.

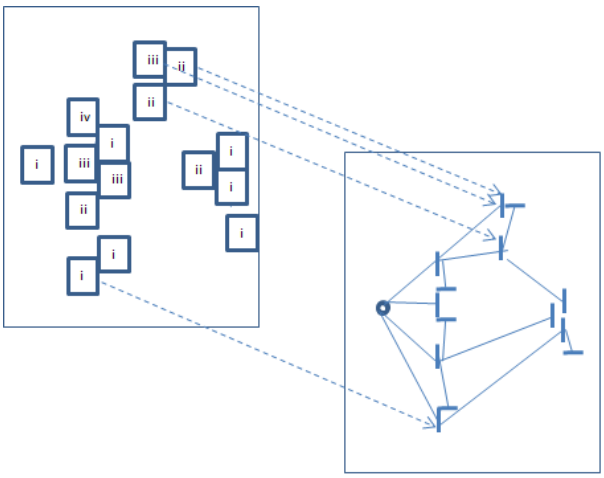


Figure 3.2 The Ant's nametags on all its phenomena; not all denotation mappings are shown

The assigned identifiers are no longer globally unique. They are only unique within the scope of the nametag of the previous phenomenon. These identifiers we therefore call relative distinguished names. The nametag of the previous phenomenon is the root of each name space of relative distinguished names.

If we concatenate the relative distinguished names from the nametag of the start phenomenon, we will get a globally unique identifier of the image of a node as being observed via a particular sequence of edges. This concatenation of relative distinguished names we call a global distinguished name.

The Ant may assign the relative distinguished name <i> to the first image being seen from a phenomenon, then find the next image, assigns <ii> etc. until all phenomena within this photo from a particular phenomenon are assigned a relative distinguished name.

If the Ant is forwarding and backtracking in the list of lists of images in a photo, it will know where it has been, and will be able to navigate to the next image not having received an identifier, or to go back to any of the images that already have been assigned an identifier.

If the graph were purely conceptual, where the edges out of a node have no ordering, the outside positioned God will have no means of knowing which edges have been visited, and which have not, other than using the nametags. Edges or nodes being assigned a tag have been visited, and those without tags have not been visited. However, the conceptualist disregards this kind of trivial time consuming assignment of identifiers. He says that there shall be identifiers, and then all nodes have identifiers. For the conceptualist, there exists no node that is not identified.

However, the phenomenological Ant is struggling with its assignment of identifiers. If it were a realist, it could have glued the tags to the entities in the real world. This is like putting up a flag on each land, which it claims to hold. Being a phenomenologist, the Ant can only glue the tags to the phenomena in its photo. When having the phenomena in just one photo, this

works fine. The Ant may navigate inside the photo, assign an identifier to each phenomenon and reason about them. The Ant may come back to any phenomenon in its static universe.

Every item in the minimal tree of phenomena may be assigned a relative distinguished name, and a corresponding global distinguished name as seen from the start phenomenon.

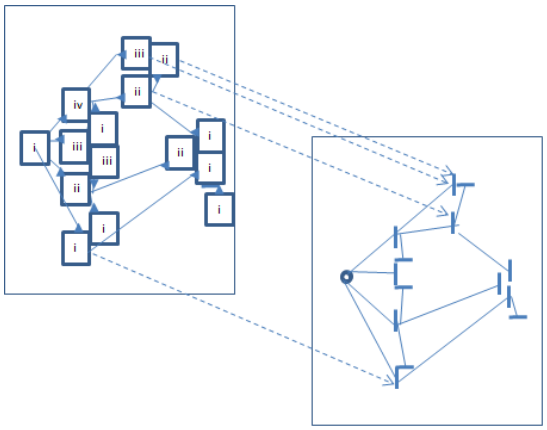


Figure 3.3. The Ant's naming tree of all its nametags.
Reversed arrow heads show containment

Since the items are ordered within each local name space, the Ant's depiction of the graph is not a tree, but is a list of lists. God's flat worldview of the graph is replaced by the Ant's recursive structure of lists of lists. This list structure is depicted in Figure 3.4.

Note that both phenomena and nametags are data inside an observer. Phenomena and data are not made up of different stuff. The phenomena, nametags and the mappings between them are all made up of data. This is called a nominalistic worldview.

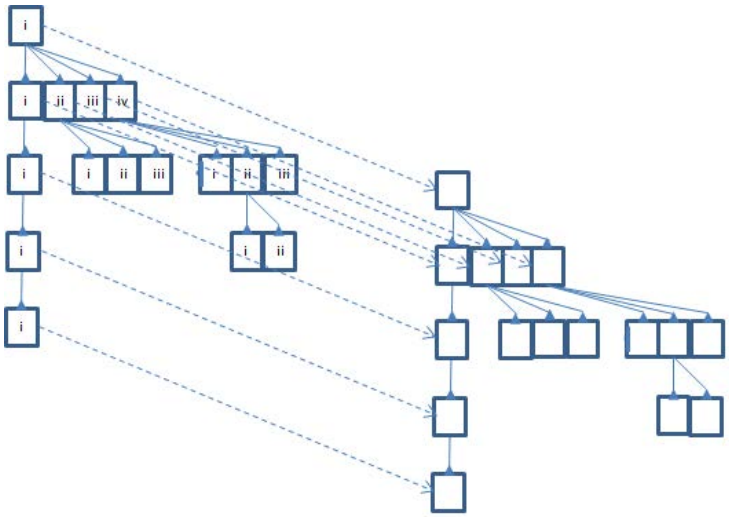


Figure 3.4 The Ant's recursive lists of data of naming tree and phenomena.
Only some of the denotation mappings are illustrated

Figure 3.4 indicates a formal representation of the phenomena and their naming tree, but there is much more to add. Each nametag will contain three data items: Entity class, Identifier attribute class, and value. The denotation mapping will contain a Denotation item and a

condition to the appropriate phenomenon. Additionally, we will see extensive use of significant duplicates. This is illustrated in Figure 3.5 for a few nametags.

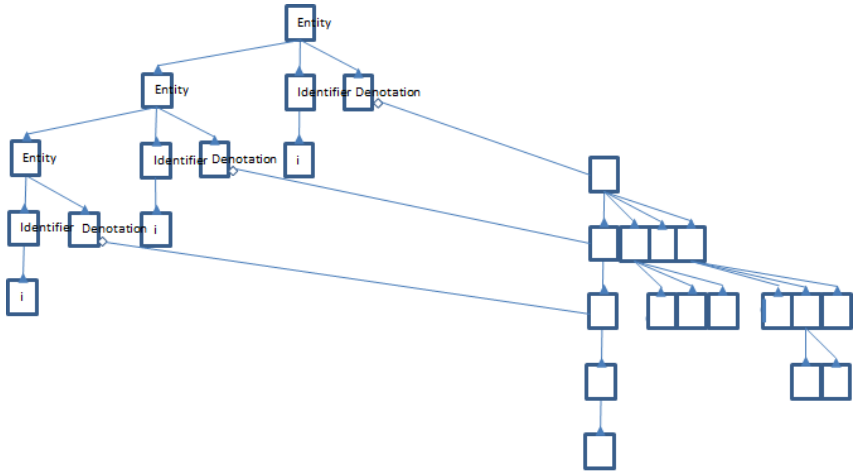


Figure 3.5 A more complete depiction of nametags

In Figure 3.5, the three Entity nametags are containing Identifier attribute nametags, which are containing value nametags. Each Entity nametag is additionally containing a Denotation tag, which refers to a Phenomenon, and can only exist if the Phenomenon exists This is expressed by the condition symbol attached to the Denotation box.

The organization of nametags, shown in Figure 3.5, is very different from the interpretations used in classical logic, and we will explain this difference in subsequent sections.

Figure 3.5 depicts Denotations from the Entities having the global distinguished names <i>, <i, i> and <i, i, i>, respectively. Note that the Denotations are not stated from the Identifiers and their values, but from the Entities containing these Identifiers and values.

Note that the Denotation mapping from nametags to phenomena is isomorphic, ie. there is a one-to-one correspondence. This means that if a nametag denotes a phenomenon, then the superior nametag will denote the superior phenomenon.

However, the mapping is not onto either way. There may be phenomena that are not denoted, and some nametags may not denote any phenomenon.

The denotation mapping is similar to a synonymity mappings between terms, and may appear at several levels, i.e. a denotes b which denotes c etc. However, here we only illustrate one level.

If different Ant's choose different start nodes, there will be different structures for each start node. We will have to study dynamic graphs in order to find common worldviews of several Ants. See on functional dependencies below.

When creating the minimal recursive list of lists, the Ant must state that the item reached one way somehow corresponds to an item reached another way. The Ant will have to state references between the items in its structure of items. This topic is deferred to Part 4, Language notions, of the book.

Somehow, God has to state that an edge is connecting two nodes, and the same nodes are involved in other connections. However, God's worldview is flat, while the Ant's worldview

is a recursive list of lists with references between the items. The Ant's view is like graphs at many levels with references between items at the same or different levels.

Note that the Identifier-s introduced in this section are not strictly needed. The Ant may create the structure of Entities and Denotation-s without using Identifier-s. The Ant's name spaces will then consist of significant duplicates, where each Entity in Figure 3.5 denotes a different Phenomenon even if they have no Identifier, which distinguishes them from other Entity data items. Identifier-s may be good to have, but are not strictly needed when data are organized in lists.

All phenomenon data are made up of pixels, which are organized in lists. Also, all other data are made up of pixels organized in lists. Even the letters may be defined by pixels, even when being defined via splines or vectors. Data are distinguished by the structure of their lists.

Data classes may be defined by lists, as well. A pixel is the only atom of phenomena and of other data.

You may have read about Nominalism in texts on the War of universals. Our Nominalism is in the details different from, or we should say that it is a more extreme variant of Nominalism known from philosophy. The structures of data indicated in the current section, may serve as an introduction to and motivation of the languages to be proposed in Part 4 of this book.

Functional dependencies

When studying a static graph, all its nodes, edges and phenomena exist simultaneously. From each node, the Ant can only observe phenomena of its neighbour nodes.

When the graph become dynamic, nodes, edges and phenomena may appear, disappear and move. Then the appearance and movement of some phenomena may depend on each other:

1. The limbs of a person can normally only exist on this body, and cannot normally be moved to another person
2. The cable pairs of a cable can only exist inside the cable, and cannot be moved to another cable
3. The trees within a field cannot normally be moved to another field
4. The registered cars exist only within one authority
5. etc.

Observed at a distance, you may only see the person's body, the cable, the field, the authority etc. Only when you move towards these phenomena, will you be able to see the limbs, pairs, trees and registered cars. The existence of each detailed item is functionally dependent on one of the previous items. The observer may choose to observe a detailed item from its superior item. Hence, the traversal of the graph of phenomena is depending on the functional dependencies between the phenomena.

Note in case (4) that it is the identifier of the registered car that is functionally dependent on the identifier of the authority. The authority observes only its registered cars, and not all cars. In this case, the phenomena are made functionally dependent due to the design and management of the data identifiers. In cases (1), (2) and (3) the functional dependency of phenomena come first, and the identifiers are made functionally dependent because of this. From a cable you can only observe its contained pairs, and not all pairs of any cable, etc.

In order to find functional dependencies, you may observe the movement of a phenomenon, and see what follows with it. You may move a person and see that his limbs follow with him, relative to the environment of other phenomena. Alternatively, you may move yourself

around the person, and see how the environment is rotating around him. This is already illustrated in Figure 1.4. Even if we started out with a static graph, we introduced a moving observer, the Ant, which is traversing the graph.

The movements of the phenomena and of the observer are essential means to find the structure of any image. Humans use triangulation from two eyes, which is a kind of movement. Movements, comparisons and filtering are means to find the essential parts of an image.

In addition to movements, the observer may observe colours, shades and other characteristics of the surfaces for identifying the bodies in an image. We will not go into further details on how to filter out the phenomena.

When doing the analysis of functional dependencies, we are often looking for independence rather than dependence. We will find that a registered car is a different phenomenon from a car. We will find that a person and his telephone number are independent phenomena, and that they are independent from his address. Telecom operators are often mixing up these phenomena in their customer management systems, and the implied confusions have great impact. Hence, these issues are not trivial.

In Part 2 of this book, you will read about relational mathematics, which provide the basis for the Relational model of databases. In the Relational model, we study functional dependencies between attributes. In this book we apply the functional dependency notion in a general way on all kinds of phenomena and of data.

The functional dependencies are used to design the naming tree of the Universe of Discourse. We have learned that in order to find and design the functional dependencies, we have to study both the phenomena and the data.

Outlook

We have outlined how to create a tree of data that is isomorphic to a tree of phenomena. Both phenomena and data, and the mappings between them, are made up of pixels only. Both phenomena and data are data inside an observer.

We have learned that it is the entity instance label that has a denotation mapping to a phenomenon. The identifier attribute and its value denote nothing. They are only being used for search, to find the right entity label instance. This treatment of denotations is very different from Classical logic. See Part 2 section 7 on Naïve Model Theory.

We have outlined how functional dependencies between phenomena may be found by movements of the phenomena relative to each other. Some functional dependencies are introduced among the data without being experienced among the phenomena.

4. Existence

What is the meaning of to exist?

Some people believe that Existence cannot be defined. They believe that to-exist is an atom of knowledge that is undisputable. By claiming so, they will not be able to distinguish Not-to-exist from Exists-but-we-do-not-know.

If we take a phenomenological worldview, to-exist means to be observed, and we are able to create a long list of requirements for accepting something to-exist. The notion of Existence is linked to an observer, and nothing can exist by itself.

A separate list of requirements applies for identity management. These requirements are in particular powerful, since we cannot claim that a particular individual exists without having observed him and having attached a name label on him at an earlier moment of time.

This section ends with a discussion of example words, which do not denote phenomena that do exist.

Absolute realism and absolute conceptualism

In a context free worldview, an entity may be considered to exist even if it is not being observed, or even cannot be observed. Both naive realism and absolute conceptualism may subscribe to such notions. A difficulty with these worldviews is that they miss the notion of an entity coming into existence and disappearing from existence. Hence, this is a static worldview for eternity.

In naïve realism, the entity is considered to exist out in the physical universe without being observed. Farther Christmas may be an entity that is claimed to exist out in the real world even when not being observed.

Naïve realism has much similarity to absolute conceptualism, even if in the first case, entities are considered to exist out in the physical universe, while in the second case they exist inside the observer. The observer may though be open, such that in Soviet philosophy, the concepts may exist out in the society, and not only inside human brains, as of classical Western philosophy. Hence, in Soviet philosophy, the society may be considered to be the observer.

In absolute conceptualism, an idea may be considered to exist even if you can never implement it, not even in data. Examples of such ideas are notions of infinities and of continuity. Notions of points and sets may belong to the same category. In Part 4 section 16 on Denotations, we will address not so absolute conceptualism.

Empirical physicists typically discard infinite solutions from their mathematical calculations, as they know that these solutions cannot exist in the observed world. To the physicist's thinking, maybe the result can become very large, but not infinite.

Existence for a phenomenologist

A phenomenologist deals with the phenomena as they are being observed by some observer with a certain set of properties and capabilities, at a certain position and time. The phenomenologist does not deal with the idealistic worldviews of the naïve realist or absolute conceptualist.

The phenomenologist

1. May claim: Phenomenon A exists
2. He points – e.g. at a screen - at the phenomenon image
3. He says: This is A
4. He records the place and time of the observation – at the screen
5. He delimits the phenomenon from its surroundings, e.g. by pointing
6. He has an instrumentation chain from the image at the screen to the entity being observed
7. He has validated the instrumentation chain
8. He records the place and time for the measurement at the entity side, or he calculates this from the known time for transfer from the entity side to the phenomenon side
9. Then he may add: The statement “Phenomenon A exists” is True.
10. If anything of the above is missing, then the statement “Phenomenon A exists” is False.

These ten elements define how the phenomenologist will use the predicate Exists. The predicate takes a phenomenon x as input and produces the values True or False:

Exists (x)==True, if all the first eight bullets are satisfied
Exists (x)==False, in all other cases

In the above ten bullets, we used A as a constant identifier of an individual. x is a variable that can range over constants, including A.

We see that the phenomenologist is a positivist; for something to exist, it must be observed. This has the radical implication that when something is not observed, it does not exist.

If we add “for the observer” to the statement, it all becomes clear, and it is no longer radical. The important thing is that the phenomenologist has no notion of existence by itself; for something to exist, it must be observed. Hence, Exists and Be-observed are not the same predicates, but to Be-observed, i.e. bullet 2, is the key element of the predicate Exists.

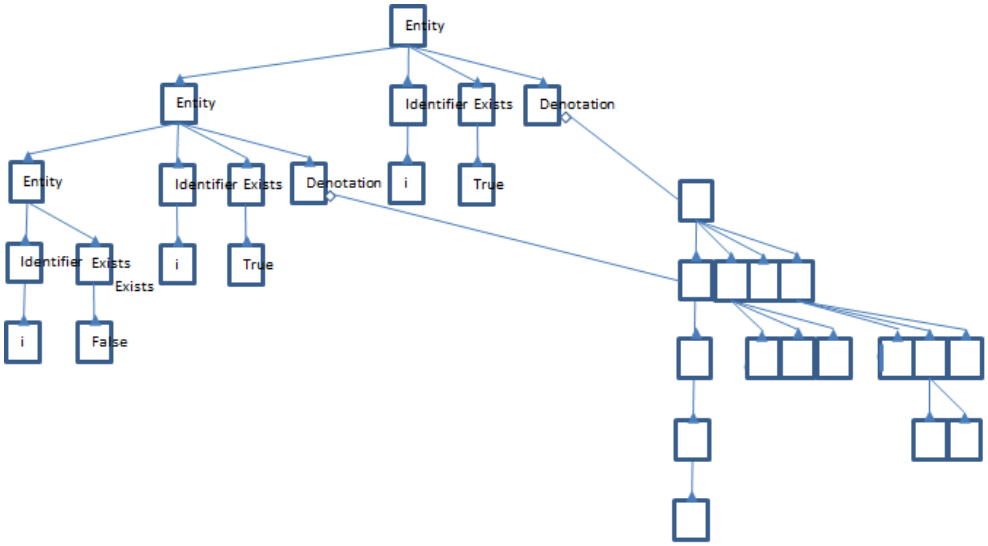


Figure 4.1 Depiction of Denotations of existing and non-existing individual phenomena. The diamond symbols state a condition from a Denotation role via a navigation line to a phenomenon. See more on this in Part 4 of the book.

In Figure 4.1, the Entity-s contain the Denotation references. The values, <i> and <i, i>, contain no Denotation.

The individual Entity-s containing the Identifier-s with values <i> and <i, i> exist, because they have a Denotes reference to some existing phenomena. An individual <i, i, i> - to the very left - does not exist, because it is lacking a Denotation reference. We have not included this logic - of checking if the Denotation reference exists - in the Figure; we only show the resulting truth value – True or False - of the execution.

Note that in the predicate formulation we use the equivalence operator (==) to assign a truth value (True or False) to the predicate Exists (x). Here a variable is used for the entity. We may also use a variable z for the truth value. When calculating the existence of a particular entity, the variable z is replaced by the constant True or the constant False, e.g. z=True.

In our data notation, depicted in Figure 4.1, we attach the values True or False subordinate to the attribute Exists. The attribute is contained in the Entity, i.e. Entity (Exists, and the values are contained in the attribute, e.g. Entity (Exists (True.

Use of equality (=) and equivalence (==) may lead to notions of conceptualism, which we will avoid. If you introduce equality, you will let the Devil out of the box, and he will do all kinds of harm.

Note that the definition of existence is about actual existence and not about potential existence. We may put in Farther Christmas for x into our definition, and find that Exists (Farther Christmas)==False. This is because we are not able to observe him now, i.e. he is not a phenomenon. However, we have not defined what Farther Christmas may look like; hence, we may denote any entity with the nametag Farther Christmas. After having made this baptizing of some entity, we may find that Farther Christmas does really exist; i.e. there is a phenomenon having this nametag.

If we have not tagged the entity, and have not put into place a regime for controlling the identity, then Farther Christmas cannot exist. We cannot claim the existence of an individual that we haven’t already pointed out, denoted and are able to recognize.

Identity management

The ten previous requirements for the Exists predicate to be True apply only at the time of observation and baptizing. For the entity to exist at a later event, we have to add the following requirements:

1. The observer puts in place a mechanism which ensures that the nametag A is unique within a name space having entity B as its root
2. The observer puts in place a mechanism which ensures that the phenomenon A does not change its identity before a second observation
3. The observer puts in place a mechanism which ensures that nobody else can copy or use the nametag A within the context of B before the second observation
4. The observer reads in a second observation the nametag A within the context B
5. The observer states: Phenomenon A (still) exists
6. Then he may add: The statement “Phenomenon A exists” is True in the second observation
7. If anything of the above is missing, then the statement “Phenomenon A exists” is False in the second observation

Note that the seven additional requirements are about persistence of a phenomenon, i.e. about existence of the same phenomenon instance across several observations. The nametag A in these lists of requirements may be an artificially assigned nametag, or it may be some naturally appearing property, like a fingerprint, DNA-sequence or other, that is expected to be unique within the given “name space” having B as its root entity.

Discussions

If you claim that Farther Christmas exists without having done proper baptizing of him, you are simply a fool. By checking that you have not done the proper baptizing, we can prove that your Farther Christmas does not exist. You cannot believe that Farther Christmas exists without having proper empirical knowledge of him. A claim of existence requires at least one prior observation, and you must have done your homework during that observation.

You cannot claim the existence of some unknown individuals of a class without some empirical knowledge of these individuals, e.g. observation of their excrements, tooth marks or other. You may claim potential existence of unknown individuals of a class if you define the class and explain how these individuals could be constructed and function. However, without empirical knowledge of the unknown individuals, the probability for their actual existence shall be set to zero. In this paragraph, the word unknown refers to some arbitrary individuals of whom you have no identification.

We may apply our definition of existence to the famous dispute about the tuberculosis bacteria; Did they exist at the time of the Pharaohs, one thousand years before Christ? Well, we did not have instruments to observe them until recent time. So they could not exist for any observer before this time. Today we can reason about tuberculosis at the time of the Pharaohs, but the Pharaohs themselves, or anyone around them could not. The tuberculosis bacteria came into existence for observers with the invention of the microscope. Combined with means to take samples from mummies, we are now able to reason about tuberculosis bacteria at the time of Pharaohs. The phenomenologist has to distinguish time of recording/sampling and time of observation of phenomena. With this distinction, we have resolved the dispute.

The expression “tuberculosis bacteria” refers to a class or to individuals of a set of individual bacteria. However, our definition of the predicate Exists only refers to a particular individual A. The predicate does not apply for classes or some arbitrary individual of a set. Therefore, the reasoning in the previous paragraph is informal.

Revised predicate

With the above discussions, we may replace the Exists predicate with the Existence predicate:

Existence (o, x, px, tx, ps, ts, y)

The predicate tells that observer o has observed the phenomenon x at place px at time tx, being sampled at place ps and time ts through the instrumentation chain y. The name tagging of o, x, y and of the places and timings are implicit in this definition.

We would have to add more parameters to state that x shall be constrained to a class X. x may take the value A, as within the above text on the Exists predicate. The place ps may take the value B.

Note that the definition of existence in this section is about phenomenon instances, and not about classes of phenomena. In Part 5 section 19, on More on Existence, we will address the existence of Classes.

More examples

We will now discuss a few other examples of use of the Exists predicate.

If a person leaves the room, such that you cannot see him for a while; does he exist when he is unseen? The answer is no. The person is not observed when he is out; hence, he does not exist for the observer. He may be destroyed while he is out. If he went out into a safe environment, you may calculate that it is very likely that he will reappear, i.e. come into existence again, but he does not exist when not being observed.

Does the future exist? You cannot denote a phenomenon that is not yet observed. Hence, the future or anything in it does not exist.

What about Schrodinger's cat? Does it exist? The answer is no. When physicists claim that the cat is both dead and alive simultaneously, they are taking a realistic or conceptualistic worldview. For a phenomenologist, the cat only exists when it is being observed, and then it is either dead or alive, never in a superposition of being dead and alive. The super-positioned state is just a hypothetical state that is used to reason about possible outcomes, but it will never appear as an observed state. It does not exist.

Just like it has been possible to observe both particle and wave properties of a photon simultaneously, I will not fully disregard the possibility to observe the probabilities of quantum states without finally settling a state. You way put the cat into a box, which you put on a scale. You can then record that the box contains the same mass as of the cat, but you do not know that the mass is still a cat. You may look into the box using x-rays, and will see that the mass has the same form as of a cat, but you have not sufficient knowledge to conclude that it is a cat, and so on. The observations may partly camouflage or expose the cat properties. Schrodinger's box is just a means to put in a sufficient delay between subsequent observations. You have the same considerations between two blinks of the eye. The phenomena only existed in the last observation. After the observation, they do not exist. In the future, they may come into existence again, but they do not now exist in the future.

What about infinities? Do they exist? The answer is no. What then about all the mathematics of infinite series? The results on that they may never exceed a certain limit are right. It is the notion of existence of an infinite number of elements in the series that is wrong. You may consider the notion of infinity as a hypothetical notion, which is used in the reasoning – like a function or functor. The infinity is not a result, but an intermediary in producing a finite result. Hence, pi and 1/3 are not numbers; they are functions for producing numbers. The output depends on how many decimals you use. When we in classical mathematics treat them as numbers, we deal with them as the maximum or minimum limits that we can produce – with a given resolution. These limits – with the given resolution – are numbers. Strictly speaking, pi and 1/3 should not be considered to be numbers according to our definition of the predicate Exists.

What then about Zeno's paradox about Achilles and the tortoise? How do you resolve the paradox without using infinities? Achilles and the tortoise are competing in a run. The tortoise starts 10 meters in front of Achilles, but Achilles runs ten times faster than the tortoise. So when Achilles runs up to the point where the tortoise started, the tortoise has been running one meter. Achilles runs up to this one meter, but then the tortoise has been

running another ten centimeters, and so on. Achilles will never bypass the tortoise even if the distance between them will decrease. This is seemingly a paradox, as we are certain that Achilles will bypass the tortoise since he is running ten times faster.

The classical mathematician may resolve the issue by using infinite series, and show that they have a finite sum. After an infinite number of steps in finite time, Achilles will reach up to the tortoise position and bypass it.

Is the mathematician right? If Achilles is running up to the tortoise's previous position, we will need an instrument to record this event. The recording has to be faster and faster as Achilles approaches the tortoise. The speed of the recording has to become infinite, and the recording will need to use an infinite amount of energy per time unit. The approach outlined in the problem formulation is simply impossible to implement. In reality, the speed of the instrument and its energy consumption will have to be finite; Achilles will have to stop and wait for the recording, and he will never come parallel to the tortoise, and will never outperform it. The tortoise is the winner. The mathematicians have been wrong since Zeno.

Achilles may certainly outperform the tortoise, but then the problem formulation has to be different. With the current problem formulation, the required observations will affect the result. Without these observations, Achilles may bypass the tortoise.

Do I exist? The answer is no. The Exists predicate is about objects, not about subjects. You as an object, being observed by me may exist, but I as a subject cannot exist according to the definition. I may observe my own body, and say that my body exists as an object. I cannot observe my own thinking in real time while I am doing the thinking. According to Shannon's sampling theorem, I would have to sample my own thinking minimum two times per pulse of my own thinking. Therefore, I would have to observe myself faster than my own speed of thinking. Because I cannot do so, my own thinking does not exist in real time. This does not prohibit that I can record my own thinking in a transaction handler and control my own thinking. This transaction handler is what people call consciousness. The consciousness about my thinking is always delayed compared with my thinking, but may put constraints on and directions to my future thinking.

Does the Universe exist? The answer is again no. Suppose that the Universe is an extensional set. An extensional set is defined by its contents, like a sealed bag of particular apples. If we replace an apple, it is not the same extensional set. We seal the bag in order to ensure that the set of apples is extensional. The physical universe is not like this; in the physical universe, particles pop-up from vacuum and disappears. The number of particles you see depends on temperature and of speed of the observer.

For the extensional set to exist, there must be an outside observer – a God who can see the Universe and denote it from the outside. For God to exist, there must be somebody to observe him. The Universe being an extensional set cannot exist for humans who do not know how to find it all, do not know if it is theoretically possible to find it all, people who know that it is not possible in practice to find all contents of the Universe, where everything is changing before you have been able to make a first scan of it, and you have no means to denote it all. The notion of the Universe being an extensional set makes no sense. Therefore, it does not exist.

The Universe may be an intensional set. An intensional set is defined by its boundaries, and not by its contents. A River is an example intensional set. We define the River by what we can see, e.g. the two hundred meters we can observe from your cottage. We may delimit the river by stones, trees, bends or other. The River is not defined by its contents, as the water in it is exchanged continuously. Some water runs out, and other water comes in. Our Universe is

similar. We may delimit our Universe by how far we can see by the best telescopes, ie. 46 billion light years, but we do not know what is in it, and cannot know. This intensional Universe, we call a Universe of Discourse (UoD). Different observers have different UoDs. Your UoD may be delimited by your room, by your country, or by a particular topic, like music. The UoD may be particular for one observer, or shared among many observers, who together form a collective observer.

Outside our UoD there is not a God, but more universe. This is just like our River; outside it there is no troll, but more river. Our UoD and our River depend on us, where we are, what observations we make, and how we have defined the boundaries. Our Universe of Discourse may exist, but the Universe does not exist. Therefore, neither do multiverses exist.

We may now look back on our graph. Does the Ant on the graph exist? It can only exist if there is someone to observe it, e.g. another Ant, and the first Ant has become a part of its UoD. Ants may recursively observe each other. Also, God needs to be observed in order to exist, as he cannot exist outside the context of anyone's UoD. We will not allow the mathematician to exist in his eternity; we will drag him down to the Ant's world.

Outlook

We have introduced a set of requirements for accepting a report on the existence of a phenomenon during one observation. We have introduced a set of additional requirements for accepting the same phenomenon to exist during recurring observations.

In the previous subsection, on More examples, we have discussed the existence of phenomena that are claimed to exist by most people, but which we do not accept the existence of.

Most people have only an intuitive notion of the mappings between terms and phenomena. In Part 4, Language notions, of this book, we will explicitly state the Denotation reference from each term instance to its phenomenon instance.

In Part 4 we will introduce a language stating conditions on terms, and delete or create terms depending on the outcome. Hence, the conditions on terms state conditions on the existence of these terms. Existence is the key notion. Data are organized in recursive lists of terms, and the conditions between terms indicate functions that may rewrite terms.

Part 2

Classical Logic

- How appropriate is classical logic for describing the physical universe? -

5. Naïve Set Theory

Can we use Set Theory to describe the universe?

In this section, we will introduce the conceptualistic world of sets, and show how terms are used to denote entities in this world. A more formalistic Set Theory will be presented in section 8.

The objective of this section is to introduce the fundamental notions, and to introduce some words on their use. We are concerned with notations and their interpretations. The objective is not to provide a textbook on the theories and their applications.

We end this section with expressing distrust in taking Set theory as a general theory of some Universe of Discourse. The discussion will go deeper in subsequent sections of this part of the book, where we will abandon Set theory altogether.

The reasons for discussing Set theory [9-17] are two-fold. The Relational model for databases is based on, but is not identical to relational mathematics [34]. Relational mathematics is based on Set theory. Databases are used to describe anything in the Universe, and we want to discuss the foundation of a language for describing anything. Secondly, Set theory may be used to interpret statements in formal languages, such as Predicate calculus [20]. We want to discuss the universality and validity of these interpretations.

A set is an aggregate of elements. The elements are themselves sets. A set may be an element of other sets.

Set theory has, from Cantor's very conception, been developed as a tool for studying infinities [18]. However, infinities are outside our scope of interest. We will study finite systems, and discuss how sets may or may not be used to describe these. Hence, we will perceive a set to be similar to a sealed bag of apples. The term bag indicates a finite size. The seal indicates a fixed contents; i.e. no apple can be removed, inserted or changed, and they do not rot.

Sets are extensional; i.e. their contents cannot change, and the set is completely defined by its contents, i.e. by its members.

Sets are unique; i.e. each set is a unique individual. If we want to allow for significant duplicates, we may use the Theory of bags, also called Multisets [36], and not Set theory. This means that when using Set theory to describe a sealed bag of apples, we consider each apple to be uniquely distinguishable from any other apple and not to be replaceable by any other apple in the bag.

When Set theory is used in mathematics, the mathematician typically starts with the empty set, i.e. a set that has no element. There is only one such set in his entire universe. Then he creates other sets by taking sets of sets recursively.

Set theory may be modified to allow for use of any ur-element, and not just the empty set. This is done in Quine's so-called New Foundation. If we want to use Set theory to describe anything in the Universe, we will need to distinguish individuals without having to define

them from the empty set. Hence, we will allow for having any number of ur-elements other than the empty set.

There are many versions of Set theory. Some of these may relate to topics discussed in this book, but will also be different.

Graphical notations

The notations used in this part of the book have no bearing on the notations used in the first or latter parts of the book. We will introduce the non-conventional Set graph notation to illustrate set structures, as Venn diagrams are only suitable to illustrate sets at two levels. Note that this limitation of Venn diagrams is not a limitation of Set theory; it only limits the diagramming capabilities within one diagram.

In Set graphs we use

- Rings to illustrate sets
- Hooked arrows to illustrate set memberships, with the arrow pointing at the element of the set

The following Figure contains two Set graphs.

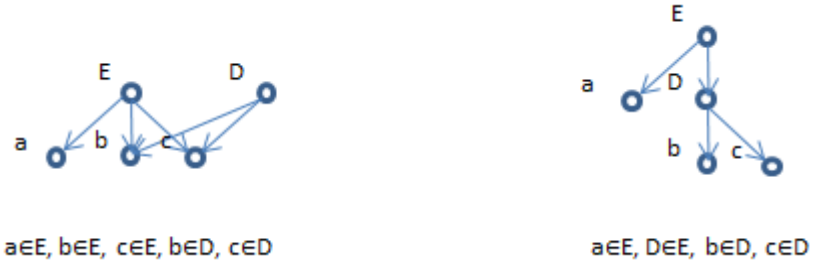


Figure 5.1 Example Set graphs

In Figure 5.1, the membership symbol (∈) and the set membership arrow have the tips and the hooks directed the same way.

The leftmost graph in the above Figure can be depicted in one Venn diagram. The rightmost graph requires two Venn diagrams. Globally unique names are needed to refer between identical sets in different Venn diagrams. This is illustrated in the following Figure.

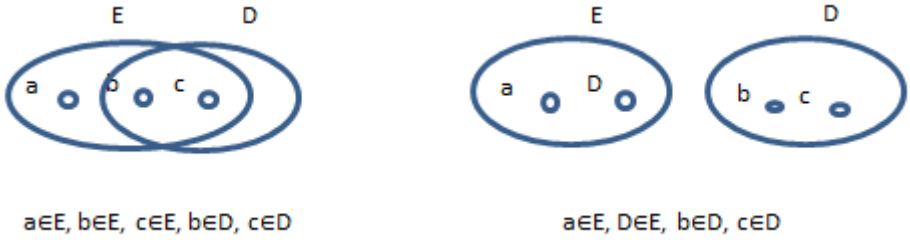


Figure 5.2 Example Venn diagrams

Venn diagrams can be understood as a constrained version of Euler diagrams [36]. We will not use either of them in this book, as we will use Set graphs as a more general and explicit tool to illustrate small set extensions.

Textual notations

Above, we have presented two diagramming techniques for sets. In the following, we will present three textual notations for defining sets.

Set membership notation

a, b, … A, B	Constant terms • constant names can denote sets only • different sets have different names • one set may have more than one name • all names are globally unique
ϕ	Name of the empty set • an empty set contains no member • there is only one empty set • the empty set may have multiple names
$\in$	Set membership; reads “is a member of”
Example	a $\in$ E, D $\in$ E; these are example statements
Comment	References between statements are made by using identical names, eg. E and E Different names do not exclude equality, eg. that a=D However, not-equality may be the intention behind using different Names. Equality must be explicitly stated.

Set extension notation

a, b, … A, B	Constant terms
{,}	Set extension is introduced to avoid using definite names
{}	Expression for the empty set, ϕ ={}
Example	{a, {a, b}}
Comment	Note that the sequence of members in a set is immaterial, i.e. a set has no ordering of its members {a, {a, b}}={{a, b}, a} ={a, {b, a}} Note that {a, b} is a member of {a, {a, b}}, and is not a subset of {a, {a, b}} Therefore, two Venn diagrams are needed to depict {a, {a,b}} The proper subsets of {a, {a,b}} are {a} and {{a,b}} Also, {a, {a,b}} is a subset of {a, {a,b}}

Set equality notation

a, b, … A, B	Constant terms
{,}={,}	The equality states that both sides denote the same set
A=B	The equality may be stated between two constants, two set expressions {,}={,}, or a constant and a set expression
Example	{a,b}={b, a, c}, a=c
Comment	For the first equality to be true, it implies that either a=c or b=c

If a=c, this does not prohibit that also b=c, which implies in this case that the expressions are reduced to {a}={a}, a=a, i.e. a tautology

Most presentations of Set theory include a set of operations on sets. We may call these operations a Set calculus. The Set calculus only introduces the operations, and does not introduce new notations of the sets. The equality symbol may be considered being an operator, as well, but we have included it in the list of notations, as it is a means to express sameness, i.e. both sides of the equality sign denote the same set.

In most usages, the operators are used to state relations between sets, and not to calculate on the sets. The term relation must here be understood in the colloquial sense, and not as a definition of relations in terms of sets of ordered pairs; see later. The Set calculus is used in relational mathematics.

The subject of notation is the mapping from phenomena to terms. The subject of denotation is the mapping from terms to phenomena. Sets are used to represent concepts, not phenomena. Set theory is a conceptualistic approach, and we will soon come back to the mappings between terms and sets. Set graphs will be used to illustrate these mappings.

We recapture the set operators here, but will not use them in this book:

- **Union** of the sets A and B, denoted $A \cup B$, is the set of all objects that are a member of A, or B, or both. The union of {1, 2, 3} and {2, 3, 4} is the set {1, 2, 3, 4} .
- **Intersection** of the sets A and B, denoted $A \cap B$, is the set of all objects that are members of both A and B. The intersection of {1, 2, 3} and {2, 3, 4} is the set {2, 3} .
- **Set difference** of U and A, denoted $U \setminus A$, is the set of all members of U that are not members of A. The set difference {1,2,3} \ {2,3,4} is {1} , while, conversely, the set difference {2,3,4} \ {1,2,3} is {4} . When A is a subset of U, the set difference $U \setminus A$ is also called the **complement** of A in U. In this case, if the choice of U is clear from the context, the notation A^c is sometimes used instead of $U \setminus A$, particularly if U is a universal set as in the study of Venn diagrams.
- **Symmetric difference** of sets A and B, denoted $A \vartriangle B$ or $A \ominus B$, is the set of all objects that are a member of exactly one of A and B (elements which are in one of the sets, but not in both). For instance, for the sets {1,2,3} and {2,3,4} , the symmetric difference set is {1,4} . It is the set difference of the union and the intersection, $(A \cup B) \setminus (A \cap B)$ or $(A \setminus B) \cup (B \setminus A)$.
- **Cartesian product** of A and B, denoted $A \times B$, is the set whose members are all possible ordered pairs (a,b) where a is a member of A and b is a member of B. The cartesian product of {1, 2} and {red, white} is {(1, red), (1, white), (2, red), (2, white)}.
- **Power set** of a set A is the set whose members are all possible subsets of A. For example, the power set of {1, 2} is { {}, {1}, {2}, {1,2} } .

The above list is copied from Set theory at [12]. Typically, Venn diagrams showing overlapping areas are used to illustrate these operations.

Statements about sets are typically expressed in Predicate calculus. Predicate calculus is a language for expressing any property of an individual set or a collection of sets. We are not yet ready for introducing statements in Predicate calculus and the mappings from the constituents of the statements to sets.

Relationship theory

We are now ready to introduce the notions of relationships.

A relationship between two sets, a and b, is denoted by an ordered pair, <a, b>, between the terms a and b. Note that the relationship belongs to the world of sets, and not to the world of terms and expressions about the sets. Note also, that the relationship has a direction, i.e

$$<a, b> \neq <b, a> \text{ if } a \neq b$$

If a=b, then <a, b>=<a, a> and <b, a>=<a, a>. Therefore, <a, b>=<b, a>, since <a, a>=<a, a>. Hence, there are not two different directed pairs between a and a, as there is between a and b if they are different.

Suppose we want the first position in the ordered pair to represent the role Lover and the second position to represent the Loved-one, then in case the person loves himself, the ordered pair will not be able to distinguish the two roles.

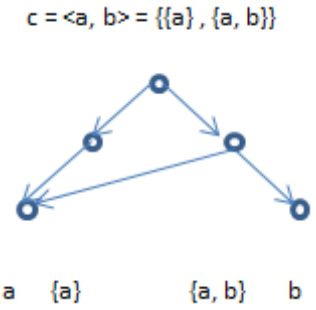
If we want to distinguish the two roles as separate entities a and b, we would have to distinguish the roles from the person p, relate the person to the two roles through <p, a> and <p, b>, and relate a to b by <a, b> where a and b are always different.

According to Kuratowski, an ordered pair is defined by using the non-symmetrical set expression

$$<a, b> = \{\{a\}, \{a, b\}\}$$

In case a=b, the set expression will be reduced to <a, b>={{a}}. This explains why the ordered pair breaks down to represent two different roles between the same entity. Both the notions of sets of sets and non-symmetrical sets do not look intuitive to represent relationships.

The following Figure depicts Kuratowski's ordered pair using Set graphs.



Fgure 5.3 Kuratowski's ordered pair

The next Figure depicts a recursive relationship from a to a. ie. a=b.

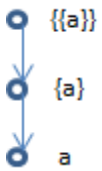


Figure 5.4 Recursive relationship

We see that the ordered pair degenerates to a set of a set in case a=b. If we would like to associate different properties to each role, this cannot easily be done.

In the next Figure we therefore have introduced separate entities a and b to distinguish between these two roles and the person p.

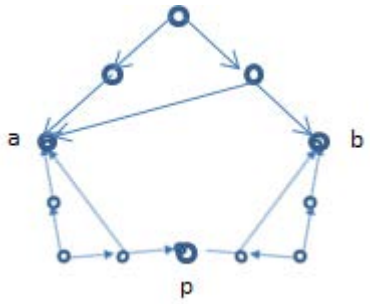


Figure 5.5 Use of explicit roles

In this case, we may easily associate different properties with each role, as they are separate entities. Figures 5.4 and 5.5 are examples of entity migration. We start with a simple structure, as of Figure 5.4. When we want to add information, we will have to replace the existing set (p) by the more complex sets a, b, and p, and the relationships between them. This easily becomes dirty in database applications. See Annex F on Entity Migration.

Above, we have introduced binary relationships. N-ary relationships are introduced by using nested ordered pairs. A ternary relationship is defined as follows:

<a, b, c>={{a}, {a, <b, c>}}={{a}, {a, {{b}, {b, c}}}}

Here we have been slightly informal in the middle expression, as we have used the ordered pair <b, c> to denote a set within a set parenthesis. The rightmost expression defines the ordered triple. This is illustrated in the next figure.

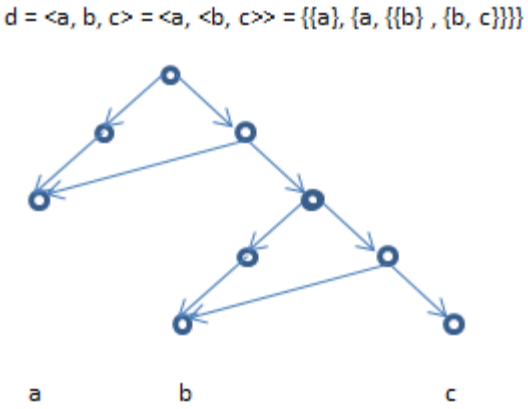


Figure 5.6 A ternary relationship

Relation theory

Using Set theory, a relation is defined as being a set of relationships.

R={<a, b>, <a, c>, … <b, a>, <b, b>, <b, c>, … }

Using Set theory, the relation may or may not include relationships in both directions. In practical applications, we may constrain the relations to be one-way only.

Note that a relation is a set, and hence unordered.

Note also that a relation is extensional, i.e. defined by its contents, and cannot change contents by insertion or deletion. It is important to note that a relation is not defined by its intention/purpose, e.g. seek marriage, or intension, e.g. future marriages; it is only defined by its current extension, i.e. the actual marriages therein.

In case we use Kuratowski's ordered pair to represent a recursive relationship, as shown above, we could associate a property both to the set {{a}} and to the role set Loves or Loved, and hence indicate to which role this property belongs.

If we use separate entities for the roles, as depicted in Figure 5.6, they will each either belong to the set Loves or the set Loved. Hence, the properties may be related to elements of these sets. The elements, eg. a and b, are treated as if they are strongly typed by either the Loves or Loved set.

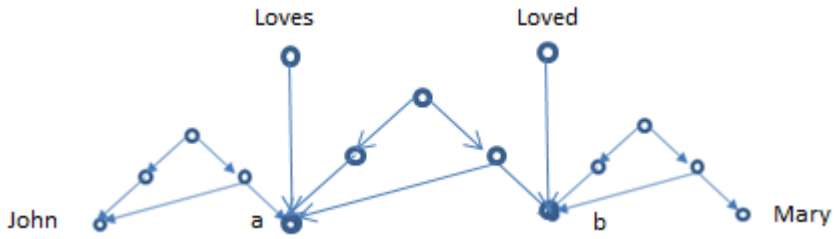


Figure 5.6 Roles as separate entities

Every relation has a fixed arity. This means that binary and ternary relationships cannot be grouped into the same relation.

Binary relations may have two cardinalities. If the Relation R relates a and b, the cardinality on the b role is the maximum number of b-s there can be for every a. For the same Relation R, the cardinality on the a role is the maximum number of a-s there can be for every b.

The positions within the ordered pairs are called roles.

Each role may span a subset of the elements in a set called the domain of this role. The domain spans the permissible elements of the role. The actual elements of the role may span less elements.

Relational mathematics is a calculus on relations, and is using the same operators as of Set calculus. These operators are used in practice in relational databases. This use in databases is a motive for this presentation of Set theory.

Function application theory

We will introduce a function application as a notion within Set theory. A function application f_1 is a many-to-one mapping from a set of arguments a, b, c … to a function value e.

The notation is as follows:

$$e=f_1(a, b, c)$$

Here e, a, b and c are constant terms, that are globally unique. Also, the function symbol f_1 is globally unique. The number of arguments in a function f_1 is fixed, and cannot vary. There are three arguments in this example.

We see that a function application is an n+1 relationship between n arguments and one function value. Both the function value and the arguments denote sets.

We recall that an n+1-ary relationship can be defined by nesting of Kuratowki's ordered pairs. Likewise, the function application may or may not be defined by nesting

$$e=f_1(a, b, c)= f_2(a, f_3(b, c))$$

Note that the relationship denoted by $<a, <b, c>>\neq<<a, b>, c>$,

The function application $f_1(a, b, c)= f_2(a, f_3(b, c))= f_4(f_5(a, b), c)$ as long as the function value e is the same. Hence, a function application may be mapped to one or more alternative relationships. However, suppose the partial functions f_2 and f_3 exist; this need not imply that the partial functions f_4 and f_5 exist, as the operations used to define f_1 may not be associative.

A function application and a relationship are not exactly the same notions.

Function theory

A function is a set of function applications.

The notation when using variables over the function values and arguments is as follows:

$$u=f_1(x, y, z)$$

A function is a relation having cardinality one at the function value side. The function values may be a subset of a set called the domain of the function.

The cardinalities at the argument sides may be any. In the above example, the argument cardinality of f_1 is three. Each argument may span a subset of a co-domain of the function.

The function may neither map from every element in the co-domain nor to every element of the domain.

As both functions and function applications are defined as sets, we may interpret the set parenthesis as being function applications or functions. Let us study Kuratowski's ordered pair in this perspective:

$$<a, b>=\{\{a\}, \{a, b\}\}$$

We may use the function symbol $<>_2$ for an ordered pair. For an ordered triplet we would write $<>_3$ etc. Hence

$$<a, b>=<>_2(a, b)$$

The function value e of the ordered pair function is defined as

$$e=<>_2(a, b)$$

Let us now do a similar rewriting of the right hand side of Kuratowski's definition. We write $\{\}_1$ for a set of one element, $\{\}_2$ for a set of two elements etc. We then get:

$$\{\{a\}, \{a, b\}\}=\{\}_2 (\{\}_1 (a),\{\}_2(a, b))$$

Note that $\{\}_n$ does not mean an empty set, $\{\ \}$. An empty set may be indicated by $\{\}_0$.

We may combine the above results into the following expression for Kuratowski's definition:

$$e=<>_2(a, b)=\{\}_2 (\{\}_1 (a),\{\}_2(a, b))$$

The right hand equality shows how the ordered pair is defined. Note that when using functions, ordered pairs and set expressions, we do not need to assign proper names to the function values.

The above text shows how ordered pairs and set expressions may be rewritten using function symbols $<>_2$, $\{\}_1$ and $\{\}_2$. Ordered pairs and set expressions are function applications.

Rather than using the various parentheses in the function symbols, we might have used letters like p and s, which might have improved the readability.

There are more general definitions of functions than what we will discuss in this text. Here we stick to the basic notions only, as this is what we need for the discussions herein.

Outlook

In Part 4 section 13 on External terminology language we will use the <> symbol to state conditions, and any term with a condition attached to it, we will consider to be a function. We do not have separate names and name spaces for functions. Hence, the notation used in the previous subsection is a taste of what will come. In the previous sub- section we have shown that both ordered pairs and sets may be defined as function values.

In the Relational model for databases, each role is assigned a name. The relation Loves between two Persons, may have a Lover and Loved role. Since these names are unique within the relation, the position of the Lover and Loved may be switched, and there is only one Loves relation between the two Persons.

In relational mathematics, the sequence of the roles is essential, and they cannot be switched. Hence, you may have two relations, e.g. Loves and Being-loved, and the roles in each are independent of the roles in the other relation, and they are normally not named. In this respect, the relational model and relational mathematics are different.

The use of Kuratowski's ordered pairs to define relationships between entities is not intuitive. For recursive relationships, we have shown that Kuratowski's ordered pairs breaks down.

If we want to define more than one relationship between a and b, we will need to introduce the roles as separate sets, a1, a2, a3, b1, b2, b3 etc, relate a to a1, a2, a3 by creating <a, a1>, <a, a2>, <a, a3>, relate the b-s in a similar way, and then relate <a1, b1>, <a2, b2>, <a3, b3>. We will have to identify and relate the roles, and not relate the entities a and b. Set theory is not prepared for the identification of roles and the associations between them.

In Part 1 section 3, on Nominalism, we have indicated a need for a list structure of phenomena. Set theory does not seem to be proper for this use.

Sets are static structures; and you cannot add or delete elements without the result becoming another set. For describing real world phenomena, we will need to manage dynamic structures, different from extensional sets.

In section 8 on Axioms of Set Theory, we will give a deeper discussion of the foundation of Set theory, and of the appropriateness of using Set theory for describing phenomena within the physical universe.

The motivation for this discussion is that the Relational model is based on Relational mathematics, and Relational mathematics is a branch of Set theory.

When using the Relational model, the relations are being normalized. Basically, functional dependencies are identified between the columns in the Relational tables. If the columns are functionally independent, they may be split on separate normalized tables. This shows that the Relational model is based on Function theory, and maybe Function theory could form a better foundation for a universal language without the use of Set theory.

6. Predicate calculus

Can we use Predicate calculus to describe the physical universe?

Predicate calculus [19-21] **is a language for stating facts about some world. This world may be a structure of sets according to Set theory.**

The mapping from statements in the language to sets will be discussed in the next section. To discuss use of name spaces and of mappings to sets is the only purpose of the presentation in the current and next sections.

The current section ends with contrasting Predicate calculus with a constructivist view on the language notions.

When considering the mappings from Predicate calculus statements to the sets being described by the Predicate calculus statements, we may call Predicate calculus a descriptive language.

When looking at the statement forms only, we may call Predicate calculus a declarative language.

In this section we will focus on First order predicate calculus. First order predicate calculus is the most applied formalism for stating facts in logic.

Horn clauses is a normal form of Predicate calculus statements that is used in the programming language Prolog.

Before presenting the First order predicate calculus, we will have to introduce a more basic form, the Zero order predicate calculus, which normally is called Propositional calculus.

Propositional calculus

A proposition is a state of affairs in some Universe of Discourse. Propositional calculus is a calculus on the truth values of statements about these propositions. Hence, Sentential calculus could have been a better term.

Individual statements are denoted by constants, like

- A
- Mary has red hair

Note that these two statements may or may not assert the same proposition.

Any statement may be denoted by a variable, like x and y. Sentential calculus contains no mean to limit the scope of the variables. Therefore, both constants and variables of statements have to be globally unique.

In Sentential calculus, each statement has a specific or unspecified truth value. The truth values are 0 or 1, or they may be denoted True or False, or other.

Statements may be combined by logical operators to form new statements. Typical logical operators are

- Negation, ¬

- Conjunction, ∧
- Disjunction, v
- (material) Implication, →
- Equivalence, ↔

Some of these logical operations may be defined by some of the others. The negation operator is unary, eg. ¬ p, while the others are binary, eg. p ∧ q. Also, it may be possible to extend the notation to allow for n-ary operators, like in the programming language APL, eg. ∧ (p, q, ….) to mean ∧p∧q ∧….

New statements may be formed by using these operators on given statements and parentheses.

An outside-positioned mathematician could codify his whole universe, using Sentential calculus only, e.g.

(((Mary has red hair) ∧ (Mary has not black hair)) ∧ (¬ (Mary has black hair))) ∧

(Mary is ten years old)

When replacing the statements with their truth values, we get

((1∧ 1) ∧ ¬ 0) ∧ 1

When calculating this, we get

((1 ∧ ¬ 0)) ∧ 1 ⊢ ((1 ∧ 1) ∧ 1 ⊢ 1 ∧ 1 ⊢ 1,

which tells that the total statement is true, even if one of its constituents, “Mary has black hair”, is false. The meta symbol ⊢ is used to state that what is written on its right side is inferred from what is written on its left side, before blank; line shifts may be used to indicate the same. The ⊢ symbol is used in deductions, and may be read as “infer” or “deduce”.

We observe that Sentential calculus is an approach for calculating the truth value of a given statement from the truth values of other statements. The other statements may be primitive, like

Mary is ten years old

or be axioms, like

Mary has red hair → ¬ Mary has black hair

Sentential calculus gives no guidance on how to structure information about Mary and her hair.

Predicate calculus

Predicate calculus is an expression of statements according to Sentential calculus. However, the contents of the primitive statements are expressed by

- Predicates, which apply on terms and map to truth values, e.g. the predicate statement Loves(John, Mary) has the truth value 0 or 1

Also, statements may be quantified by

- Universal quantifiers, like ∀ x ((x ∈ Persons) ∧ Loves(John, x) …), which reads: For all x ..
- Existential quantifiers, like ∃ x ((x ∈ Persons) ∧ Loves(John, x) …), which reads: There are some x .. . The expression “some x” means, “there is at least one x”.

Quantified statements are themselves statements, and map to truth values. Quantification applies on terms within a statement. The quantified statement is indicated by the outer parenthesis. Quantified statements can be nested.

The two kinds of quantifier expressions can be defined from each other and use of negations.

Predicates apply on terms, which can be

- Constants, which denote individual entities/sets, like a, b, 1, 2, John, Persons
- Variables, which can range over constants, like x, y, person
- Function values, like the output of daughter(Bill)

Function values can be the output of function applications or functions.

The following is an example predicate statement: Loves(John, daughter(Bill)),

Predicates are primitive statements of Sentential calculus. Predicates therefore assert propositions.

Terms denote entities, e.g. sets.

Constants, predicate symbols and function symbols are globally unique.

A variable term within predicates or functions is only unique within the scope of the quantification where it is declared, e.g. within $\forall$ x or $\exists$ x.

For a universally quantified statement to be true, it must be true for all permissible substitutions of the variable, i.e.

$$\forall x\ (x \in X) \wedge (X=\{a, b, c, .. \}) \wedge P(x) \leftrightarrow (P(a) \wedge P(b) \wedge P(c) \wedge \ldots))$$

For an existentially quantified statement to be true, it must be true for at least one of its permissible substitutions of the variable, i.e.

$$\exists x\ (x \in X) \wedge (X=\{a, b, c, .. \}) \wedge P(x) \leftrightarrow (P(a) \vee P(b) \vee P(c) \vee \ldots))$$

If the scope of the variable is stated within the quantified statement, eg.

$$\forall x\ (x \in X),$$

then the quantifier is said to be bound.

If the scope is not given, then the quantifier is said to be unbound.

Elaborations

If quantifiers are nested, the uniqueness of a variable applies within all quantifiers inside the quantifier where the variable is first used. Hence, the variable cannot be reused for other purposes within this larger scope.

Examples:

- $\forall$x (Loves(John, x) $\wedge$ Hair-colour(x, Red)); For all x whom John Loves and who has Red Hair-colour
- $\forall$x (Loves(John, x) $\wedge \exists$ y Hair-colour(x, y)); For all x whom John Loves and who has some Hair-colour

In the last example, we see that x is used to refer between the outer and inner quantifier. Hence, the inner quantifier has to use a different variable if the purpose is different.

The reader should note that all terms, all truth values, all predicates and functions stay static during the calculations. No change is permissible.

If change over time is wanted for the terms, this may be achieved by adding timing terms to the predicates and functions, such that the expressions are still static, but cover expressions for all relevant timings. If the behaviour of predicates and functions need to change, you add still more terms to control their behaviour; hence, the problem space is extended while the formulation remains static.

Scoping of quantifiers is an important issue. If all permissible values are explicitly listed, eg.

$$\forall x\ (x \in X) \wedge (X=\{a, b, c, .. \}),$$

then the scope is clear.

If the set X is not explicitly defined, the scope may be unclear. Suppose the scope is the set of Natural numbers, then we may write

$$\forall x\ (x \in N)$$

N is an infinite set. We will never be able to produce all elements of N, and some elements will be so large that we will not be able to produce these individual finite elements. Hence, a constructivist will claim that N does not exist, even if a recursive function for generating each number is defined. The constructivist will limit himself to finite natural numbers, finite rational numbers, etc. Any finite expansion of an irrational number is a rational number. Therefore, rather than talking about irrational numbers, we may talk about and characterize the functions used to produce them.

If the quantifier is unbound, the constructivist will not accept this as a meaningful expression.

We observe that predicates map from terms to truth values. A predicate is an elementary statement. Compound statements map from predicates, combined with logical operators or quantifiers, to truth values.

We may ask if truth value logic is needed in a universal language. In later sections we will conclude that truth value logic is not needed. We may do well with functions without use of predicates and truth values.

Outlook

Predicate calculus is a language for stating elementary and compound facts about a Universe of Discourse. Statements in Predicate calculus resemble natural language statements, but are very remote from the tree of data being isomorphic to the tree of phenomena, being requested in Part 1 section 3 on Nominalism.

Predicate calculus has no means of defining terms within the scope of superior terms. Quantifiers are insufficient means of scoping.

Predicate calculus provides a calculus on static structures only, while we need handling of dynamic structures for describing phenomena in a dynamic universe.

We may not need truth-value logic in a universal language, and may do better with function theory only. See also Comparison with Predicate calculus in Part 4 section 16 Denotations.

7. Naïve Model Theory

How do we interpret statements in Predicate calculus?

The Type-token principle [8-10] **is essential for the understanding of languages of classical logic. Unfortunately, many expositions of logic notations skip this essential topic. Here you get an introduction to the principle and its implications. Model theory** [22-24]**, gives a more complete theory for how to interpret logic statements.**

This section ends with a short discussion on how the Model theory notions are different from the notions introduced in Part 1 of this book.

Model Theory deals with interpretation of statements into some structure, eg. into sets.

A theory is a collection of statements expressed in a formal language. A model is a structure that satisfies the statements in that theory. This explains the expression Model Theory.

Note that the term model in this section denotes the result of an interpretation.

Model Theory is different from data models. In data models, it is the data that models something. In Model Theory it is the Set structure, defined by the data in statements, that makes up the model. The statements may be written on paper, but the model is not. In Model theory, the model is a semantic abstraction behind the paper, even if it is sometimes illustrated by graphs.

Computer science is constrained to finite structures. Hence, interpretation of statements in computer science is constrained to Finite Model Theory [37]. This will be our scope.

In Model Theory, you study what alternative models may satisfy a given theory, i.e. satisfy a set of statements. In the current section, we will only focus on how to express the mappings from the statements to its model. We will not discuss alternative models.

The notion of alternative models may be compared with multiverses in cosmology. Maybe it is easier to compare with solutions of equations. Suppose all propositions are considered to be formulated in equational statements, eg. in statements like Loves(John, x)==True. Then the more equational statements you make, the more you constrain the solution space. If we allow for all solutions that are not contradicting the explicitly stated constraints, each solution space will be enormously large. For example, the one equation above will not be inconsistent with a solution which allows Inhabitant(Mars, x, Green)=True.

The mapping from the statements to its model in Model Theory is not direct, but is done in minimum two steps. For the first step, we will introduce the Type-token principle.

The Type-token principle

The letters as we write them on paper or on some other medium, we call tokens. The word “letter” contains one token l, two tokens e, two tokens t and one token r, altogether six tokens.

Several tokens may be considered to be of the same type. The two tokens e and e are of the same type e, and the two tokens t and t are of the same type t. The word “letter” uses four

types. If use of different fonts, sizes, colours, shapes, capital or small letters is not significant, we may consider them all to belong to the same type. A type may be assigned an ASCII code.

The types are considered to be abstractions, which are not written on the paper. This does not prohibit that you may list the types in some chosen style, like when listing the alphabet. But any chosen style is strictly not a characteristic of the type.

A word of tokens is called an inscription. The word “letter” is an inscription. Another occurrence of the same word is another inscription.

Identical looking inscriptions, like “letter” and “letter”, map to the same string, eg. to the string “letter”.

Strings are considered to be abstractions, which are not written on the paper. This does not prohibit that you may list the strings in some chosen style, like when listing the glossary of a language. But any chosen style is strictly not a characteristic of the string.

A subset of the strings are called terms. These are the terms that we introduced in the section on Predicate calculus.

Since constants are globally unique, all identical constant inscriptions map to the same constant string.

Variables are only unique within the scope of their quantifier. Hence, all inscriptions of a variable within a quantifier map to one variable string. If the same variable inscription appears within multiple dependent quantifiers, they all refer to the same string. If the same variable inscription appears within multiple independent quantifiers, there is one variable string for each independent quantifier.

We could have defined similar mappings from inscriptions to strings for predicate symbols, function symbols, logical operators and quantifiers, but we will not do so, because they are not relevant for the semantic interpretation to be presented in the next sub-section.

Semantics

Suppose the terms are constant strings. Then we say that each constant string denotes an entity, i.e. a set in Set theory. We can state this mapping in a meta-language, which contains the meta-predicate Denotes, eg.

Denotes(John, y)

The variable y is here the placeholder for the set that is denoted by the string John.

The next Figure illustrates how we for constants map from inscriptions via strings to entities/sets.

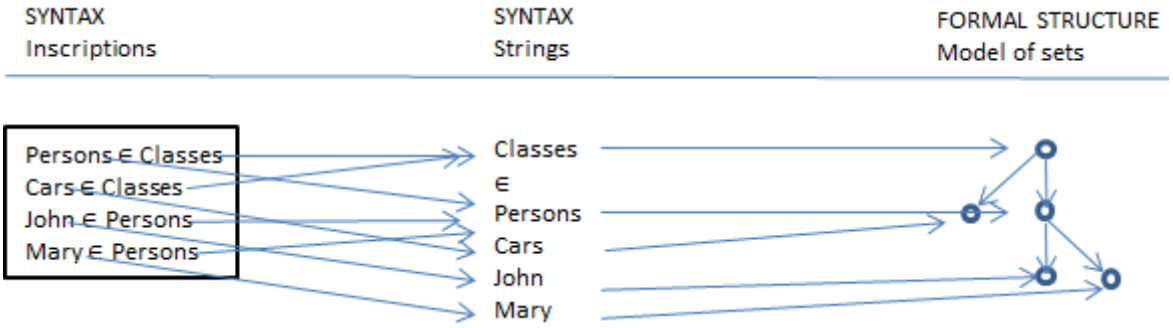


Figure 7.1. Mappings from inscriptions via strings to sets

In linguistics, the design of strings may be called morphology and not syntax. Except for the uniqueness requirement, we will not discuss morphology. Also, in this book, we will not deal with phonology. We only deal with written statements in some formal language.

Suppose we replace the constant John with a variable x, eg.

$\forall$x Denotes(x, y),

then, x may range over any constant string that has a Denotation mapping to some set y. In both these example expressions in this subsection, we should have quantified y, as well, but this we have omitted here. x may have other Denotations within other quantifier expressions.

In the previous subsection, we learned that strings are abstractions, which do not strictly appear on paper. A subset of the strings, called terms, is mapped to sets. The sets are imaginations in the mathematicians' head. They are concepts. These are idealizations, like points and infinities in mathematics.

These abstractions do not prohibit that strings may be given some representation, they become inscriptions. Also, sets may be given some representation, eg. through Set graphs.

Both strings and set structures are abstractions, which do not appear on paper. They are both eternal entities; strings appear up in heaven; sets appear in a still higher heaven. Only inscriptions exist here down on our earth. In Part 4 of this book, we will abandon the Type-token principle, as we will stick to inscriptions only, and we will allow similar inscriptions in lists to denote different phenomena.

The complete picture

A term x denotes a set/entity, which is an individual x $\in$ X in some domain X.

The set of actually denoted entities, by the given collection of statements, is called the Actual world. Relations and domains are subsets of the Actual world.

The given collection of statements is considered to include all statements that are derivable from the actually stated statements. This means that statements that are inconsistent with these statements are not included.

A quantified variable may take one of several possible constant values. What entity to be denoted is dependent on what constant value to be taken. A combination of actual constant values defines the Actual world.

There is a possible actual world for every constant value to be taken, and for every combination of such values. Such a possible actual world is called a Possible world.

The set of all Possible worlds is called the Universe of discourse.

The following Figure gives an overview of the syntactical worlds of terms and statements, and their respective denotation and assertion mappings to the worlds of sets/entities and propositions.

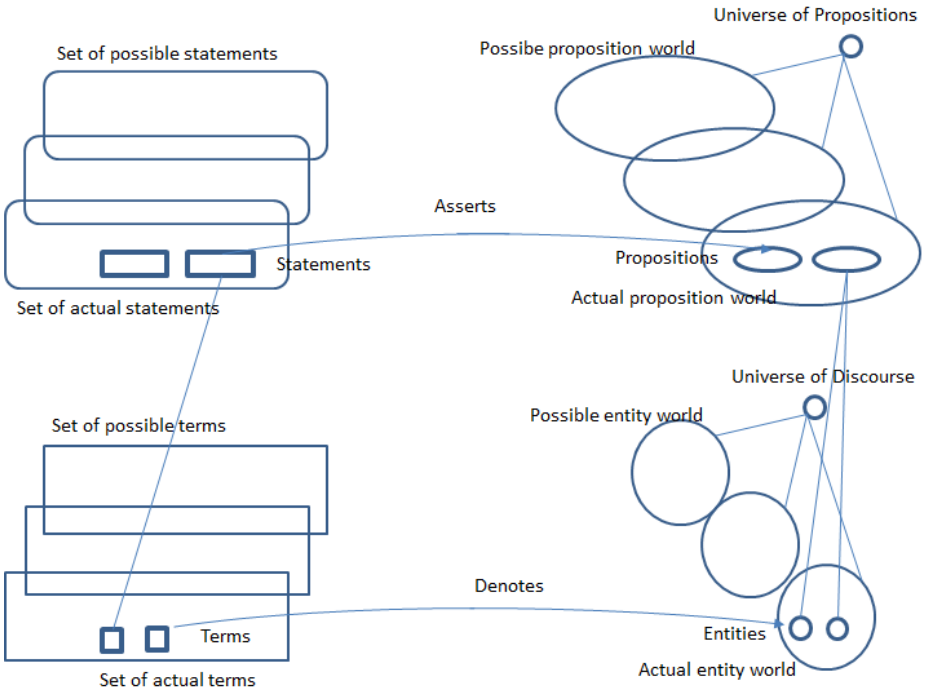


Figure 7.2. Illustration of semantic mappings

Elaborations

The use of the term model in Model Theory may differ from colloquial language, where you may create a model of something; the model may be a copy, may be with some simplifications. If you write a data model of a house, the data model the house. The data model is written in normalized statements within a formal language, and it is the syntax structure of these formalized statements that make up the model of the house.

When creating a data model of a house, we consider the data to make up the entire model, and we do not consider alternative interpretations of the data. We will later learn that most data do not model anything. Therefore, we should talk about data and data structures, not of data models.

Constants are according to the type-token principle considered to be globally unique. This is different from the approach taken in Part 1 section 3 on Nominalism. Each nametag of an entity was considered to be a root of a name space of tags of related entities. Hence, each constant term is the root of a name space of subordinate constant terms. There is no globally unique term as of Predicate calculus and Set theory.

Variables within Predicate calculus are only unique within the scope of their quantifier. The quantifier applies for a statement or set of statements combined by logical operators. In Predicate calculus there is no means to declare the variable to be local to an entity or term. However, you may state that the quantified statement is only True if the variable denotes an entity within a given set, eg. x ∈ Persons in

$$\forall x\ (x \in \text{Persons}) \wedge p(x),$$

This does not prohibit that x may be valid outside the scope of Persons, as elementary statements may be False, combined statements may be False, even the quantified statement may be False and still make sense. Use of bound quantifiers is therefore not a proper scoping mechanism.

Outlook

According to the type-token principle, inscriptions are mapped to strings. A subset of strings, called terms, are mapped to sets. This last mapping is done by a Denotes predicate, eg. Denotes(John, y).

The Denotation mappings may be questioned.

How can an inscription state a reference from an inscription to a string, which is not an inscription? Inscriptions and strings are claimed to belong to fundamentally different domains. One defending argument for this mapping is that it is many-to-one.

The mapping from strings to sets is harder to defend, as the sets are from a different domain from both strings and inscriptions. The mapping from strings to sets is many-to-many.

If strings and sets where written down as inscriptions in some normalized language, the mappings would be easier to defend. However, we see that the ideas behind Model Theory are very conceptual.

Finally, we may question if truth value logic and mappings from statements to propositions are needed in a universal language theory for stating facts and questions about the physical universe.

In subsequent sections, we will abandon all the mentioned notions from Model Theory, and we will introduce a Data transformation architecture as an alternative framework for interpretations. In the Data transformation architecture, we will map from inscription to inscriptions only.

8. Axioms of Set Theory

What is the foundation of Set theory?

Is this foundation useful for describing the physical universe?

This section shows how Predicate calculus is used to state axioms of Set theory [12-17]**. The discussion on each axiom is depressing, as the axioms do not fit with what notions we would like to see in a general language for describing anything in the physical universe, as outlined in Part 1 of this book. Due to this usage, the discussion herein is very different from the discussions found in most books on mathematical philosophy.**

The current axioms of Set theory are created to serve as a foundation of mathematics, in particular as a foundation of number theory, including ordering, infinity and continuity. This is not our application domain. We will abandon the very notion of unordered sets. Hence, the discussion of the other axioms is not essential for the conclusions of this book. Therefore, you may jump to the Discussion part of this section.

The current section ends with a longer discussion, where we abandon both truth-value logic and Set theory for describing the physical universe that we live in.

In this section we will introduce all axioms of the Zermelo-Frankel-Skolem version of Set theory, and we will refute them all.

The axioms will be formulated by using expressions stated in Predicate calculus.

Axioms

1. Axiom of Extension (Frege 1883)

   $\forall x\, \forall y\, \forall z\, (x \in y \leftrightarrow x \in z) \rightarrow y = z$

   Explanation
   If two sets y and z have the same members, then the two sets are the same. Strictly speaking: The two terms y and z denote the same set.
   A simpler formulation reads:
   Two sets are equal iff they contain the same elements:

   $\forall x: (x \in A \Leftrightarrow x \in B) \Leftrightarrow A=B$

   The order of the elements in the sets is immaterial.

   Assumptions
   This axiom assumes that
   a. That the set is unordered, that the position in a list of members is irrelevant
   b. That the set is extensional, ie. that members cannot be changed while the set remains the same

   Discussion
   We cannot accept these assumptions as a foundation of a universal language theory. If we start with a list, we may define a “power” list, eg. in alphabetic sequence, of every possible ordering of the list. This “power” list may have similarities to an unordered set, but it is doubtful that the “power” list would be useful.

Also, the list may keep the same identifier, when the members are changed.
In both formulations above, we have used the equality sign (=). It is possible to formulate the axiom without this sign. *x=y* may be defined as an abbreviation for the following formula (Hatcher 1982):

$$\forall z\,(z \in x \Leftrightarrow z \in y) \wedge \forall w\,(x \in w \Leftrightarrow y \in w)$$

In this case, the axiom of extensionality can be reformulated as

$$\forall x\,\forall y\,(\forall z\,(z \in x \Leftrightarrow z \in y) \Longrightarrow \forall w\,(x \in w \Leftrightarrow y \in w)),$$

which says that if *x* and *y* have the same elements, then they, x and y, belong to all the same sets w (Frankel *et al.* 1973).

2. Axiom of Empty Set (Skolem 1923, von Neuman 1925)

$$\exists\, y\,\forall x\,\neg\,(x \in y) \leftrightarrow y = \phi$$

Explanation
There exists a set $y = \phi$ which contains no member x.

Assumptions
From the extensional worldview of Set theory, there exists only one empty set in the entire set universe.

Discussion
We can imagine lists that have no element, but still we would consider the lists to be different, as it is their nametags and intensions that matter, as being different. Hence, we would consider the list of Persons and of Cars to be different, even if both contain no member.
Sets are defined by their memberships, i.e. by a function that takes the members as its arguments. However, for an empty set, this function has no argument. Hence, you may question if the empty set is a set.

3. Axiom of Schema Separation/Subset (Zermelo 1908, Cantor 1899)

$$\forall z\,\exists\, y\,\forall x\,(x \in y \leftrightarrow (x \in z \wedge p(x))$$

Explanation
For any set z, there exists a set y such that for any $x \in z$ such that p(x) holds, $x \in y$. This means that if you have a set, you can create a set that contains some of the elements of that set, where those elements are specified by stipulating that they satisfy some condition, ie. the predicate p(x).

Assumptions
This axiom (schema) states that predicates may be used to define subsets of z.

Discussion
There will be a need to state conditions, eg.

Grown-up=Person <> Person(Age) > 18,

which in a pseudo code states that a Grown-up is a Person that has Age higher than 18. For this, we will need conditions and functions, maybe not predicates. A mathematician trained in Predicate calculus would have written Age(Person); while we will write Person(Age). The shown notation will be explained in Part 4, Language

notions.

4. Axiom of pairing

$$\forall s\ \forall t\ \exists\ y\ \forall x\ (x \in y \leftrightarrow (x = s\ v\ x = t))$$

Explanation
For any sets s and t, there exists a unique set {s}and a unique unordered set {s, t}. Thus, there exists a set containing any two sets that are already created.
This axiom may be used as follows: If s is the empty set s=ϕ, then there must exist a y={s}. If t={s}, then there must exist y={s, {s}}, and so on. This axiom is used to create natural numbers including zero, 0=s, 1={s}, 2={s, {s}}, 3=={s, {s}, {s, {s}}} etc.

Assumptions
The axiom is about unordered pairs. I assume that this axiom could easily be modified to state that you may have directed mappings <s, t>, and even both way directed mappings, without this causing any deviation from the objective of the axiom.

Discussion
This axiom states that

- any set, s or t, is contained in another set y
- you have a mapping between any two sets.

I understand this as a capability. However, since the mapping is stated in the axioms, it will apply to any application of Set theory. Hence, the axiom states that for every set s or t – without limitation -, there exists another set y, ad infinitum. I cannot accept this endless recursion in any application.
Hence, this axiom states references y everywhere between any two sets s and t. These references may not be wanted in the actual applications.
The axiom seems to create a complete network of undirected mappings between any two nodes in a graph, and this may not be what we want in our applications.

5. Axiom of Union (Cantor 1899, Zermelo 1908)

$$\forall z\ \exists\ y\ \forall x\ (x \in y \leftrightarrow \exists\ u\ (x \in u \wedge u \in z)$$

Explanation
For any set z, there exist a y, such that if x ∈ y, then there exists a set u, such that x ∈ u and u ∈ z. For every collection of sets, there exists a set (the **sum** or union set) that contains all the elements that each belong to at least one of the sets in the collection.

Discussion
It is easy to understand the convenience of having an operator of union. However, what does it mean that the union exists? Whenever I create two sets, by default, the union of these sets also exists without me having asked for it? In a practical application I would expect any set only to contain those structures that I have defined, and not others by default. I would expect the union operator to allow me to create, not that there exists. If the sets are extremely large, I may not even be able to create a union. I cannot accept formulation like ∀z ∃ y, where z is unbound.
I cannot find all elements in an infinite set, and much less so create the union of two

infinite sets. I cannot accept that the Universe exists as an extensional set.

6. Axiom of Power-Set (Zermelo 1908)

$$\forall z \, \exists \, y \, \forall x \, (x \in y \leftrightarrow \forall w \, (w \in x \rightarrow w \in z)$$

Explanation
For any set z there exists a power set y containing all subsets of z.

Discussion
The axiom claims that the set of every subset of a set exists. This is similar to claiming that if the number of particles in the Universe is n, then there also exists a number n!. The axiom seems to make no consideration of energy, time and particles. It is insane.

7. Axiom of Infinity (Zermelo 1908)

$$\exists \, z \, (\phi \in z \wedge (\forall x \in z) \, (\forall y \in z) \, (\{x \cup \{y\}\} \in z)$$

Explanation
There exists a z such that $\phi \in z$ and for any $x \in z$ and $y \in z$, $\{x \cup \{y\}\} \in z$.
This axiom defines a set z of all natural numbers. z is an infinite set.

Discussion
You may apply recursion in computer science, but it applies in some context; and will stop when the computer runs out of power, or the Universe runs out of elementary particles. I cannot accept the use of unbound quantifiers in the formulation of axioms of Set theory. And, I do not accept any notion of infinity.
I may accept recursions without stop conditions, but there will be stop conditions within superior blocks of the recursion.

8. Axiom of Choice (Levi 1902, Schmidt 1904)

$$\forall y \, (\forall x \in y)(x \neq \phi \rightarrow \exists \, f(x) \, f(x) \, C)$$

Explanation
If y is a set of non-empty sets, there exists a function f such that $f(x) \in x$.
I find alternative formulation, which I will skip here.

Discussion
The function f(x) may have more arguments than x. x just tells in which set the member shall be picked. The function value is a member of x, ie. $t \in x$.
The idea in the axiom (schema) seems to be that you may pick any member of the set $x \in y$. This idea seems to be similar to Weierstrass' notion of variables, where you can pick any value within a domain. This notion may be needed for infinite sets. Before this, Francois Viete had a notion of variables running through a list. Viete's notion is closer to the needs of computer science. This does not prohibit that you by a search can choose a particular or arbitrary member of the list.
I do not think that this axiom should belong to the foundation. Also, textbooks tell that this axiom is consistent with the other axioms, but may not be needed.

9. Axiom Schema of Replacement (Skolem 1903, Frankel 1922, extension of axiom 3)

$\forall u\ \forall v\ \forall w\ (f(u, v) = f(u, w) \rightarrow v=w) \leftrightarrow \forall z\ \exists\ y\ \forall v\ (v \in y \leftrightarrow (\exists\ u \in z)$ $f(u, v))$

Another formulation reads:

$\forall x\ (\forall y\ (y \in x \rightarrow \exists z\ f(y, z)\ ) \wedge \forall w\ (f(y, w) \rightarrow w=z))) \rightarrow \exists v\ \forall u\ (u \in v \leftrightarrow \exists t\ (t \in x \wedge f(t,u))))$

Explanation

If z is the domain of the function f(u, v) where $u \in z$, then there exists a range of the function such that $v \in y$.

Discussion

This is an axiom (schema) for set formation by use of functions, while axiom (schema) 3 creates sets through use of subsets only.

10. Axiom of Restriction/Foundation (Skolem 1923, Neumannn 1925)

$\forall y\ (y \neq \phi\ \rightarrow (\exists\ x \in y)(x \cap y= \phi))$

An alternative formulation reads

$\forall y\ (\neg\ (y = \phi\ ) \rightarrow (\exists\ x\ (x \in y) \wedge \forall z\ ((z \in x) \rightarrow \neg\ (z \in y)))$

There are also formulations using functions and classes.

Explanation

Every non-empty set y contains an element x such that $x \cap y= \phi$.

This axiom states that a set cannot contain itself. This constraint is used avoid Russell's paradox. See Part 5 section 20 Paradoxes.

Note in axiom 4 that {s, {s}} contains s and {s}, while {s} does not contain {s}. Hence {s, {s}} and {s} do not contain all the same members. However, s is contained in both sets.

Discussion

If we had started out with lists, it would have been impossible to state that a list contains itself. Therefore, this axiom would not be needed. Two lists are different even if their name tags look identical.

Discussions

The Relational model for databases is an adaption of Set theory. In databases, we want to be able to manage information about anything in the entire physical universe. So, is then Set theory so general that it can define the structure of anything in the entire physical universe? I think that the Universe of sets is totally disjoint to the physical universe of phenomena, and that sets are not appropriate for describing the phenomena.

Any axiom defines a constraint on what is possible. The axioms of Set theory define constraints on what statements are permissible in Predicate calculus. Hence, Predicate calculus must be more general than Set theory. Does this mean that Predicate calculus allows for stating more than what is possible in the Universe of sets? The answer seems to be Yes.

Both Set theory and Predicate calculus are language theories existing within the physical universe. They are both contained in this universe. So how can they be capable of stating anything that is more general than what can be in the physical universe? I think that they are not more general than the physical universe.

May the notions of stepping outside the physical universe itself be found in the assumed mappings from inscriptions via strings to sets? I think so.

Are these conceptions buried in the type-token principle and/or of Model theory? Yes.

Are they buried in the notions of an idealistic and conceptual world of infinities, continuity, points and entities without any extension? Yes.

Are they buried in the notion of static and extensional sets? Yes, as well.

Have the logicians woven a curtain of illusions that they have raised before our eyes? They have.

We know that Georg Cantor, when inventing the notions of sets, was inspired by Thomas Aquinas' "proofs" about the existence of God, whom he called the Absolute. Religion may be understood as a substitute for a rational theory of behavior in this physical world. Is classical logic in the form of Set theory a similar substitute for a rational theory of language and reasoning, and of the physical world? I think so.

If we go back to Phenomenology in the Part 1 of this book, the phenomena are physical manifestations on the observer's input media. The phenomena are not idealistic entities like those of Set theory. The phenomena are finite and concrete.

A phenomenologist would require that his language expressions are isomorphic to the phenomena they describe. The nominalist will say that the phenomena are themselves data, and a description of the phenomena has the same structure as of the phenomena themselves. Only the syntactical sugar – which does not describe anything -, is allowed to deviate from the structure of the phenomena. The existence of the inscription John may describe the person John, but the letters in the inscription John may not describe anything. The letters are syntactical sugar, which may help identification, recognition and search.

In this book we will not discuss notions of consistency and completeness. The axioms of Set theory are provided to support these notions. Why have these notions come up in the formulation of the theory? I think that notions of consistency and completeness come from the belief that similar and not similar inscriptions may describe the same set. I abandon this belief.

Why has Predicate calculus allowed for stating more than what can exist in the Universe of sets? Because Set theory is a misconception from the very start. There are major flaws in its foundation and interpretation.

The axioms of Set theory do not make statements about individual sets or relationships, but state generic laws, which apply for the whole Universe of sets, using unbound universal quantifiers. If this Universe of sets is not the entire physical universe that can exist, what is it, and what is it for? Nobody knows, except that the axioms constrain the theory to something.

Why does Set theory use unbound quantifiers in all its axioms? The theory uses expressions like $x \in y$, but y can be any set, so the statements are unbound. If Set theory covers something less than the entire physical universe, the quantifiers should have been bound, and the scope should have been clarified. This is missing.

Sets do not look like physical objects of this world. Does Set theory then go beyond the boundary of the physical universe? I have the impression that Set theory tries to describe something idealistic which is entirely outside this physical universe.

Set theory seems to play a similar role as of religion. The text is about entities that are not of this world. Due to a lack of a rational foundation for the entities and their behavior of this world, the spiritual theory of Sets is used as a common language and reference for talking about this world.

The notion of sets seem to provide a mean to abstract beyond the syntax of the language. Sets are idealizations, like spirits. However, they are frozen spirits in a static world. I abandon this notion. I accept syntax only. Hence, I accept abstract syntax where some details of the concrete syntax are removed. The syntax is not static, and it has behavior. See more on this in Part 4 of the book.

Set theory is at least in mathematics used as a common language, or you may say “a common model”. The most prominent foundation work in mathematics is done by the pseudonym Niclas Bourbaki [30]. Under this name, a group of French logicians have founded most parts of logic and mathematics on Set theory.

For describing anything in the physical universe, I am convinced that we will need a different foundation. I believe that this new foundation will affect logics and mathematics, as well. I believe that the notion of extensional sets is a totally wrong foundation.

The Bourbaki’s themselves have a different opinion: “Despite the problems, the project is still alive and new editions are coming out. The founders of Bourbaki had a rule that members should retire at fifty. Cartier has suggested that perhaps it would be appropriate for Bourbaki to retire at fifty. This did not happen and Bourbaki is now seventy. Will there be a resurgence and a sudden return to the passion for change that gripped the founders? Probably not. There may not now be a need for this, and one reason that there is no need is of course the way that Bourbaki has shaken up the world of mathematics. Some love the approach, some hate it, but one cannot deny Bourbaki's influence.” However, I think there is a need for change.
Computer science may cause the need. Computer science may provide a return to finitism, constructivism and nominalism.

I know of extensions of classical Set theory, like Class theory and Category theory, but I do not think that these extensions and modifications will change my view.

Relational mathematics is based on Set theory and has been used as an inspiration to the SQL language of relational databases. However, SQL could just as well have been founded on list processing, and I think that this would have been a much better foundation.

I have used the metaphor “curtain of illusions” about classical logic. A colleague of mine, a physicist, had a similar saying: “The curtain of rationality is thin.” So my question about formal languages is then: Can we create a better foundation? In subsequent parts of the book we will try to do so.

Outlook

In the previous sub-section we are missing an alternative foundation to compare with Set theory and Predicate calculus.

In Part 4 section 13 on External terminology languages, we will indicate how to create this better foundation, and we will contrast the proposal with Predicate calculus. We will abandon truth-value logic.

In Part 4 section 16 on Denotations, we will treat concepts as an abstract syntax of the External terminology statements. We will abandon Set theory and contrast Predicate calculus with Existence logic.

In Part 5 section 20 on Paradoxes, we will make further comments on use of Set theory.

Part 3

Language Architecture

- What structure and capabilities should the observer possess? -

9. Language Architectures

Can we replace God, the Ant and human observers by automata? Can we extend the Cyborg in Part 1 of this book by an automaton that can do observations and interpretations?

In this section, we discuss the interpretation part. We consider any IT system to be an editor on data in a database. The editor is doing translation of data both ways between the human-computer interface and the database. We discuss the translation between two media only, but in principle, the translation may be multi-way.

We introduce an architecture of IT systems doing translation and editing of data at multiple media as an alternative to the language architecture of Model Theory. This alternative architecture is called the Data transformation architecture, to be presented in more detail in the subsequent section.

In section 11 we will compare this architecture with the camera metaphor from Part 1 of the book. In section 12 we will compare the architecture with Model Theory in Part 2 section 7.

Figure 7.1 of the section on Naïve Model Theory depicts a language architecture for classical logic. This is a three layer architecture:

- Inscriptions are mapped via
- Strings to
- Sets.

Figure 7.2 shows an extension of this first architecture, now covering statements and propositions, as well. Also, the Figure depicts actual and potential worlds, including the Universe of Discourse of these worlds.

In the previous sections, we have not addressed grammars, proof theory, automata theory and ways of executing the statements. These topics could also be considered being aspects of a language architecture.

In linguistics [11], i.e. the study of natural languages, you may use the architecture layers from logic, but add more layers, like

- Phonetics
- Surface structure
- Deep structure, which may correspond to the Logic layers
- Pragmatics, about causes and consequences of the utterances

Rather than considering processing of logic statements or natural languages, we in this book will study a different environment. We will consider an IT system [39], that

- Receives and sends data at one medium, eg. a data screen
- Stores and communicates data at other media, eg. a disk or a communication line

We will consider the arrangement of data on each medium to be expressions in some language. The expressions have a glossary and a grammar. The arrangements of pixels on a screen is considered to be expressions according to a grammar.

The IT system translates between expressions in the languages used at the different media, and the IT system enforces all the “business rules” of data in these languages. These rules comprise both constraints and derivations.

We consider the IT system to be

- A compiler/interpreter and generator for translation between expressions in these languages
- An editor and query processor for data in these languages

This translator and editor perspective on IT systems is different from the traditional perspective of IT systems performing business processes. We will come back to these differences in section 12. The function of performing translation and editing we call data transformation.

The data transformation perspective on IT systems is chosen because this provides a rich perspective on interpretation, translation and execution of statements in multiple languages, and it explains what is going on inside the observer in Part 1 of this book.

The next Figure depicts transformation of data expressed and organized in one language at the human-computer side, and expressed and organized in another language at the storage side.

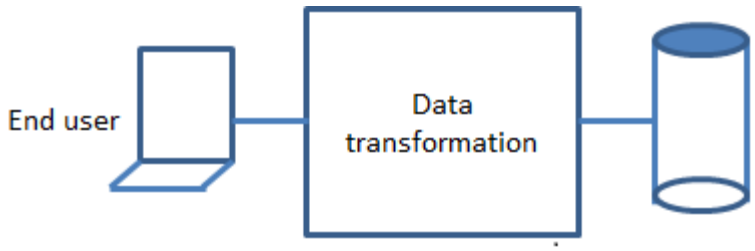


Figure 9.1. Data transformation

The Data transformation architecture, which will be presented in more detail in the next section, is based on the ISO three schema architecture, Annex A [8]. The data transformation is using a seven schema architecture, and it extends and tweaks the three schema architecture.

The three schema architecture is based on two principles:

- The hundred percent principle; all business rules are centralized to the data definitions, and are not dispersed out into processes outside the data; this principle we will apply
- The conceptualization principle; all core definitions are made at a conceptual and not at a syntactical level; this principle we will contradict, as we are nominalists and not conceptualists

More on these principles in the next section.

The data transformation between two media may be nested [45]. This is shown in the following Figure.

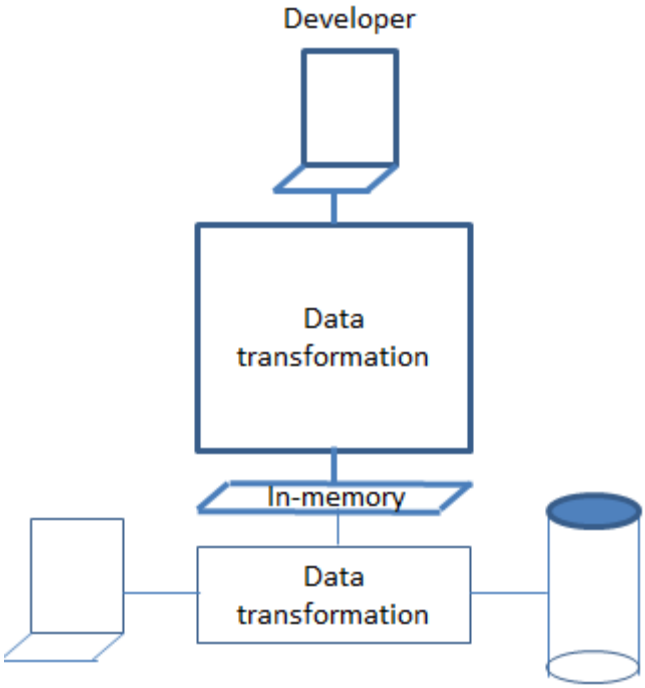


Figure 9.2 Nested data transformation

The application developer designs and documents the application in one language. These data are translated into executable code in an in-memory database. Both the end user in Figure 9.1 and the developer in Figure 9.2 may use the same data transformation tool. However, the languages and statements controlling their operations are different.

A data transformation tool may be compared with a spread sheet. You are using the same tool independently on what spread sheet application you are using. However, a spread sheet is only managing flat tables. A data transformation tool should be able to use and manage any structure of data.

Data transformation does not only bring a new perspective on design and implementation of IT systems; see section 12. It also gives a new perspective on design and use of user interfaces of IT systems. This perspective we call the data oriented approach.

Traditional design of human-computer interfaces is task oriented. The developer identifies the tasks to be performed, and tailor screens and their sequences to these tasks. Each screen may be small and provides only fragments of the data to be manipulated by the user.

When using the data oriented approach, you consider each screen to be one compound statement in one language. You are concerned with the completeness, clarity of and overview provided by this statement. You are designing statements that are so large that they may span pages, like in the German language. Still the organization and presentation shall be crystal clear.

The traditional designer designs tasks, and he is freezing the tasks into the software.

The new designer is a statement, language and data designer. Each screen may support multiple tasks, and the user may carry out any operation in any sequence in a modeless way, while the IT system ensures that no business rule is violated. The designer is, through the arrangement of data, concerned with presenting a rich and correct view of his Universe of Discourse.

As being explained above, the data oriented approach is not only a new way to perceive and understand IT systems. The approach also has normative implications on the ways to design and use IT systems. The impact on the end users' understanding of his application is profound.

This book will focus on the developers' perspective on statement, language and data design. But, the developer is designing the end user perspective. Therefore, we will show both.

Outlook

We have introduced an IT system between two media. The media may correspond to the photo in the camera metaphor in Part 1 of this book. Others may perceive the media to be means to look inside the camera, where a second observer observes the observer.

We have nested the architecture, such that a developer may control what the camera is doing.

This section serves as an introduction to the subsequent section.

10. Data Transformation Architecture

What will the IT architecture look like that can replace Model Theory?

In this section, we are opening up the boxes that were introduced in the previous section.

The Data transformation architecture [45] **is a layered architecture for translating data between multiple media and (formal) languages. Each layer is controlled by a schema containing data classes. The data instances are contained in populations.**

In Part 3 of the book, we in separate sections will introduce language notions for each layer of this architecture. The same language will apply for both classes and instances. Classes and instances in all the layers consist of inscriptions only. There is no string and no set as in Part 2 of this book. An instance inscription is a copy of its class inscription.

Part 1 of the book explains how the cyborg Ant can find the structure of its universe. Part 1 does not tell about classes of phenomena. The classes look identical to their instances, so we can get a class by taking a copy of an instance, or by combining several similar instances – having different values. See more on methodology in section 12.

The Concept layer in this architecture corresponds to the phenomena in Part 1 of this book. The combination of the External terminology and Contents layers corresponds to the data in Part 1. In the second last subsection, Elaborations, we compare the Data transformation architecture with the Naïve Model Theory from in Part 2 section 7.

Most physicists are talking about interactions between phenomena, where all phenomena are observed objects by an outside observer. However, some physicists are talking about phenomena observing each other. When a phenomenon is affected by another, it changes its state. The state of a primitive observer can be expressed by its position, spin, speed, direction etc. This notion of a primitive observer may be compared with Wittgenstein's language game, in Philosophische Untersuchungen: People are throwing words at each other, and the received words have effects on the observer. The phenomena are themselves observers.

In this section, we will consider a more intelligent observer: The observer creates a description of its Universe of Discourse that is isomorphic to its observed phenomena. This corresponds to Wittgenstein's picture theory, Annex A [63]: The normalized description becomes a picture of the UoD.

In this section, the observer will not observe the physical world of houses and stars, but will observe words at a user interface. All permissible words and their contexts of words are already defined in schemata. The data instances and their interrelations are created in the populations according to the rules expressed in the schemata. A comparison with the camera metaphor is found in a subsection of section 11.

The data transformation is

- translating data between multiple languages,
- transporting data between multiple media,
- enforcing constraints and derivations on the data, and
- doing editing of data

The data transformation architecture is shown in Figure 10.1.

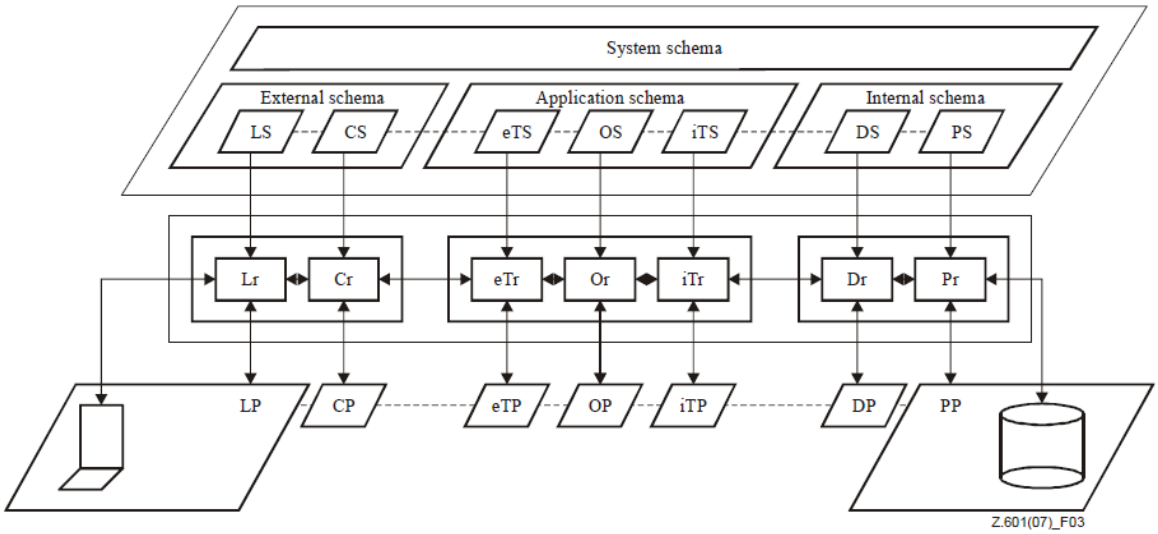


Figure 10.1. The data transformation architecture

The architecture consists of seven layers: Layout layer, Contents layer, external Terminology layer, Concept layer, internal Terminology layer, Distribution layer and Physical layer. These are indicated by the prefixes L, C, eT, O, iT, D and P respectively. See explanations of each in the bulleted list below.

Each layer contains schemata, processors, and populations. These are indicated by the suffixes S, r and P respectively. The schemata contain the classes that prescribe the permissible data instances. The populations contain the actual data instances. The processors are enforcing the data instances according to the rules expressed of the classes.

An example implementation of the architecture is indicated in Annex G. Some readers may prefer to look into this now, while others may prefer to proceed and learn about the languages before diving into a practical implementation.

Here follows a technical summary of the architecture.

The human interface part of the architecture contains the following:

1. The Layout Schema (LS) of one screen or report defines the layout of this user interface; LS prescribes the concrete syntax of the user interface
2. The Layout Population (LP) is the contents of one screen or report; its syntax is following the rules defined in its LS
3. The Layout Processor (Lr) communicates data to and from screens and reports according to the rules given in LS, and sends data to the Contents Processor (Cr) and receives data from the same.

4. The Contents Schema (CS) of a screen or report defines the abstract syntax of this user interface; in the CS, all syntactical sugar about fonts, colours, sizes, buttons, lines, shapes, location etc. are removed; there is a mapping between corresponding CSs and LSs
5. The Contents Population (CP) represents the abstract contents of one screen or report according to the rules defined in its CS; mappings between corresponding CPs and LPs are provided

6. The Contents Processor (Cr) communicates data to and from the Lr, and to and from the External Terminology Processor (eTr)
7. A combination of a LS and a CS is called an External Schema (ES); note that one CS may be combined with alternative LSs
8. Note that the LS prescribes the concrete syntax, while the CS prescribes the abstract syntax of the user interface; this is the concrete and abstract syntax of one compound statement; a compound statement is a combination of elementary statements found in the external Terminology Schema.

The application part of the architecture contains the following:

9. The External Terminology Schema (eTS), which
    a. defines all terms of the application in a given language
    b. defines the grammar of all normalized statements in that language
    c. defines all functions for constraints and derivations between these terms
    d. provides mappings to all its CSs and its Concept Schema (OS)
10. The External Terminology Population (eTP), which
    a. contains all statements and data within this language of this application, according to the terminology, grammatical structure and functions prescribed by the corresponding eTS
    b. contains all mappings to its CPs and its OP
11. The External Terminology Processor (eTr)
    a. communicates the eTP data to and from its Crs, and to and from its Or
    b. manages the eTP data according to the terminology, structure and rules stated in its eTS
    c. executes the rules on the eTP data
12. Note that there is one eTS for each language, eg. one for English, one for Chinese etc. if multiple languages are supported
13. Note that the statements in the eTS are normalized, while the statements in CS are not, and the CS statements may span several eTS statements
14. The eTS covers all conceivable normalized statements of the entire application in that language; while a CS only contains one compound statement per screen or report
15. The mapping from CSs to their eTS is many-to-one and homomorphic; no term in a CS can map to multiple terms in eTS

16. The Concept Schema (OS), which
    a. defines the abstract syntax common for all eTSs and iTSs
    b. defines all functions for constraints and derivations between these terms
17. The Concept Population (OP), which
    a. contains all statements and data of the application independently of in which language they appear according to the terminology, grammatical structure and functions prescribed by the corresponding OS
    b. contains all mappings to its eTPs and its iTPs
18. The Concept Processor (Or)
    a. communicates data to and from its eTRs, and to and from its iTRr
    b. manages OP data according to the terminology, structure and rules stated in its OS
    c. executes the rules on the OP data
19. Note that there is only one OS and one OP for each application; how to split OSs and OPs into multiple disjoint or overlapping applications is outside the scope of this book

20. eTSs and iTSs are considered to contain concrete syntaxes relative to the OS
21. The mappings from the contents of each eTS and each iTS to the OS is one-to-one and isomorphic
22. Most often, the contents of each eTS and each iTS covers the whole OS; if they are not, the designer may consider if the applications should be split
23. In order to avoid duplication of processing, and possible inconsistencies, common functions for all eTSs and iTSs are centralized to the OS, ref. both the 100 percent principle and the conceptualization principle; only the particular functions for each eTS and each iTS remain in these particular schemata

24. The Internal Terminology Schema (iTS), which
    a. defines all terms of the application in a language chosen for storage or communication of the data
    b. defines the grammar of all normalized statements in that language
    c. defines all functions for constraints and derivations between these terms
    d. provides mappings to all its DSs and its Concept Schema (OS)
25. The Internal Terminology Population (iTP), which
    a. contains all statements and data within this language of this application according to the terminology, grammatical structure and functions prescribed by the corresponding iTS
    b. contains all mappings to its DPs and its OP
26. The Internal Terminology Processor (iTr)
    a. communicates data to and from its Drs, and to and from its Or
    b. manages iTP data according to the terminology, structure and rules stated in its iTS
    c. executes the rules on the iTP data
27. Note that there is one iTS for each implementation language, eg. there may be one DBMS language for each internal database, eg. one database on a disk or one database In-memory, and there may be various languages for different communication protocols and middleware
28. Note that the statements in the iTS are normalized, while the statements in DS are not, and they may span several iTS statements
29. The iTS may cover all conceivable normalized statements of the entire application in that language; while a DS only contains one compound statement per database or message exchange
30. The mapping from DSs to their iTS is many-to-one and homomorphic; no term in a DS can map to multiple terms in iTS

31. The combination of the eTSs, OS and iTSs, we call the Application Schema (AS)

The internal implementation part of the architecture contains the following:

32. The Distribution Schema (DS) of a message class defines the abstract syntax of this message; in the DS, all syntactical sugar about storage or encoding technology are removed; there is a mapping between corresponding DSs and PSs
33. The Distribution Population (DP) represents the abstract contents of one message according to the rules defined in its DS; mappings between corresponding DPs and PPs are provided
34. The Distribution Processor (Dr) communicates data to and from the Pr, and to and from the Internal Terminology Processor (iTr)

35. The contents of a DS maps to the contents of one iTS only; one iTS may have several DSs
36. The mappings from the contents of a DS to the contents of its iTS is many-to-one and homomorphic

37. The Physical Schema (PS) of one database or message class defines the layout at this medium; PS prescribes the concrete syntax encoded on this medium
38. The Physical Population (PP) is the contents on one medium or in one message; its syntax is following the rules defined in its PS
39. The Physical Processor (Pr) communicates data to and from physical media according to the rules given in PS, and sends data to the Distribution Processor (Dr) and receives data from the same.
40. A combination of a PS and a DS is called an Internal Schema (IS); note that one DS may be combined with alternative PSs, and several DSs may support one PS
41. Note that the PS prescribes a concrete syntax, while the DS prescribes an abstract syntax

42. The External Schemata (ES) and Internal Schemata (IS) may collectively be called Presentation Schemata
43. The External Terminology Schemata (eTS), Concept Schema (OS) and Internal Terminology Schemata (iTS) may collectively be called the Application Schema

For coordination of the seven layers, we introduce the following:

44. The System Schema (SS), which
    a. Manage security data, including access control
    b. Manage directory data, including data for configuration control

The System Population and System Processor address implementation issues, which we will not discuss here.

Elaborations

Some people may think that the Data transformation architecture is too complex for ordinary database applications. I think that it is not, and I have much experience on designing large applications. The architecture provides a clear separation between the (external and internal) presentation layers and the application layer. The presentation layers deal with surface syntax and complex contents of non-normalized statements. The application layer deals with normalized statements, their execution across all data, and mappings to concepts, if needed.

If you consider compilation and interpretation of logic languages, you will find similar functionality as of the Data transformation architecture, but you may not find as clear a separation of layers as in the Data transformation architecture.

The Data transformation architecture defines mappings between data in different formats at different media. The data in all populations and all schemata appear as inscriptions. The mappings are made by inscriptions, as well. This is very different from the presentation of Model Theory within Part 2 of this book.

The architecture supports no notion of strings. However, the notion of a schema of a population provides a notion with some similarity to strings. The inscription contents of a schema is homomorphic to the inscription contents of its population. Hence, mappings

between two collections of inscriptions, in the population and schema respectively, are replacing the mappings from inscriptions to strings. Note that the schema may contain many inscriptions that look alike, but are different.

The architecture supports no notion of sets, as of Model Theory. However, the Concept schema of an application has some similarity to conceptual schemata and sets. The Concept schema contains inscriptions, where all the syntactical sugar within each eTS and each iTS is removed, such that only the commonalities of these remain. Therefore, the Concept schema is the common abstract syntax of all its eTSs and iTSs. This is a different use of the term concept from that of Set theory and of conceptual schemata, which allow for many-to-many mappings from the contents of the statements to the concepts/models. In the data architecture, this mapping is one-to-one and isomorphic from each eTS and each iTS.

The Data transformation architecture has introduced more layers and mappings than of Model Theory. The Data transformation architecture has a one-to-one mapping from one Layout schema to its Contents schema, and a many to one mapping from one Contents schema to its External terminology schema.

The architecture provides two levels of mappings from concrete to abstract syntaxes, and does this on both sides of the Concept schema. All mappings are made between inscriptions. Data is all there is; there is no non-syntactical notion.

Some implementations may not use all layers of the architecture. Many implementations may use only one External terminology schema. Then all function may be expressed in this schema, and both the Concept schema and the Internal terminology schemata may be discarded. If there are more than one External terminology schema, one of them may serve as the Concept schema. The separate Concept schema may be left out, because all the schemata are syntactical. The mappings to and from the Concept schema are just statements of synonymity, which may as well be expressed directly between External terminology schemata. This simplification of the architecture is depicted in the next Figure.

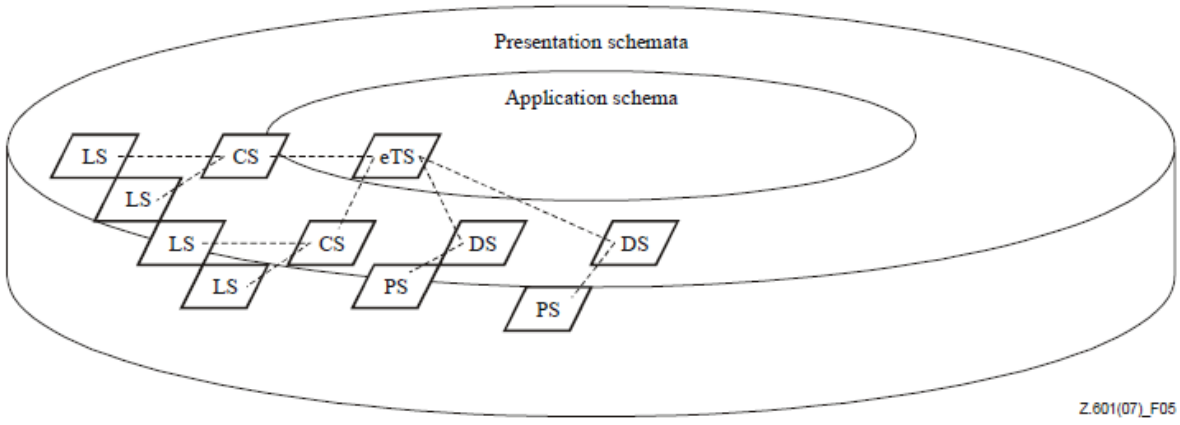


Figure 10.2 Simplified mapping between schemata

Already in 1983, this simplified architecture was put into operation on large and complex database applications [33], [38], [70]. This proves that the architecture is not only of theoretical interest, but may preferably well be used in practical implementations.

Systems implemented according to the Data transformation architecture may be interconnected into a set of communicating systems. Also, each system may be implemented in a multi-tier solution. When two systems or modules exchange data, they both have to share

the same data definitions, i.e. the same schema definitions, for this data exchange. Transformation of data can only appear inside a system or module. The next Figure shows how the implementation can be made multi-tier.

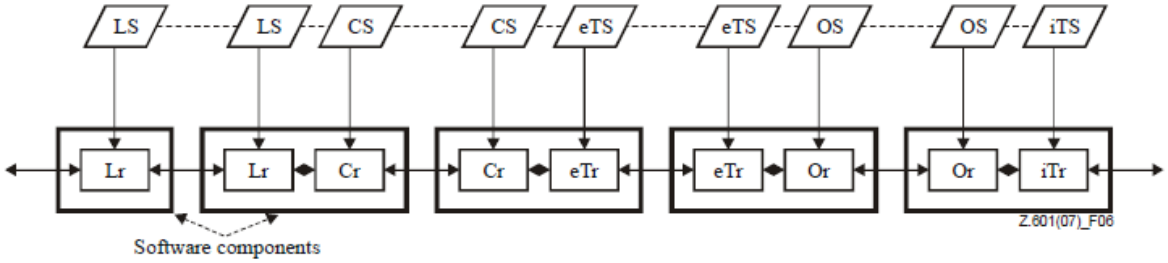


Figure 10.3. Multi-tier implementation of the data transformation architecture

The multi-tier implementation shows that all functionality need not be implemented in one box. However, if the total translation shall appear as one integrated whole, there needs to be a real-time transaction handling mechanism from end-to-end. Between different systems, the transaction handling may be long and not real-time.

The last Figure of this section shows how the implementation can be nested.

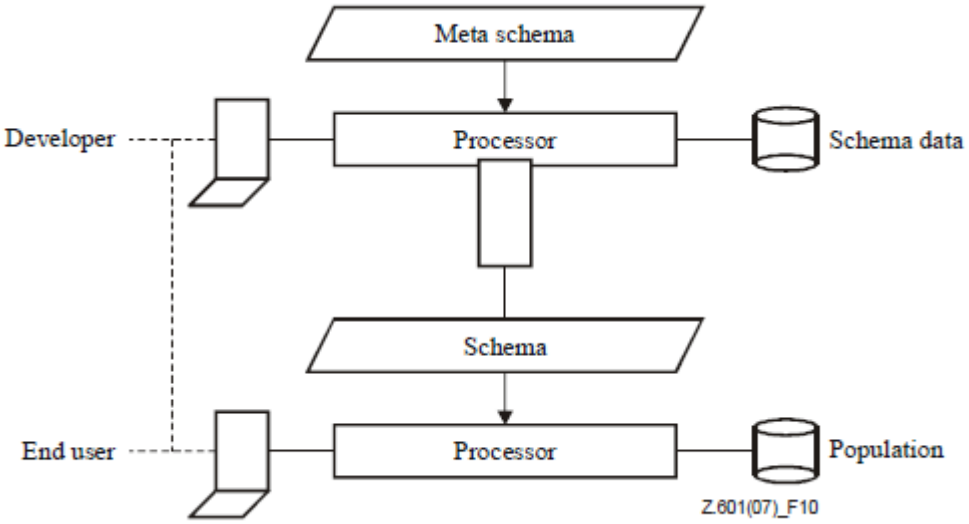


Figure 10.4. Nesting of the data transformation architecture

In the above Figure, the specification data appear in three forms

- At the user interface of the developer
- In the data dictionary database of the developer
- In the in-memory schema tables controlling the end users' application

These bullets indicate that the schema inscriptions, replacing strings and sets in classical logic, appear in multiple forms. In the next part of the book on the Languages themselves, we will learn that the schema notion used in the current section is too simplistic. We will nest schemata within schemata, and even in populations.

Outlook

In this section we have been constructing the architecture of a database application. This application may serve as an observer of phenomena and of data about these phenomena. We are not analyzing how humans may interpret statements; we are designing an observer that may understand its Universe of Discourse.

The Data transformation architecture is a framework for structuring specifications:

- The External terminology layer contains elementary statements and logic
- The Contents layer contains compound statements traversing the contents of the External terminology layer
- The Layout layer adds syntactical sugar

The architecture maps between inscriptions in these layers, and has no use of strings or sets.

Inscriptions are replicated from classes in schemata to instances in populations.

In the External terminology layer, each identical inscriptions denotes a different phenomenon. This is radically different from Model Theory.

The layered architecture allows for implementations in multiple tiers.

11. Requirements on data languages

What do we require from a language that may state anything?

Annex E lists the detailed requirements on the languages of each layer of the Data transformation architecture. In this section, we write a short technical summary.

In Part 1 of the book, we have learned how to find and state the structure of phenomena. We want to handle lists of lists recursively, and we want to support use of relative distinguished names. Additionally, we have strong requirements on mappings between the layers of the Data transformation architecture.

Data in all layers are inscriptions. We will require a homomorphic mapping from instance inscriptions to class inscriptions, ie. on their syntaxes. In the mapping from the layout to the contents, we remove syntactical sugar. In the mapping from the contents to the external terminology, we remove complex sentence structures. The basic understanding of the UoD, expressed in the External terminology layer, is preserved through these transformations.

In a second last subsection, we compare the Data transformation architecture with the camera metaphor in Part 1 of the book.

At the user interface of our IT applications, we want to support

- Use of local nametags both on headings and values
- Use of significant duplicates of nametags both on headings and values

We want the user to be able to read the specification of the data in the respective schemata, using these nametags only, and not having to learn additional constants and variables using a different notation. We want population data and schema data to look identical, and they should not need to have different languages.

We want to allow several Layout schemata to map to the same Contents schema, i.e. a many-to-one mapping. The mapping from the contents of a Layout schema to the contents of its Contents schema shall be isomorphic and onto, i.e. one-to-one. This means that when the Contents schema is given, a particular/default Layout schema may be generated automatically.

The designer may in the Contents schema have to state which data to be suppressed in the Layout, and will have to state which operations to be allowed.

Several Contents schemata may draw data from one and the same External terminology schema, i.e. a many-to-one mapping. The mapping from the contents of a Contents schema to the contents of its External terminology schema is homomorphic, i.e. a many-to-one mapping.

A Contents schema may start at any entity class in the External terminology schema, using a global distinguished name of the identifier attributes of this entity class. The contents of the Contents schema makes up a tree, which is traversing the data tree of the External terminology schema, using its references.

The nametags of the Contents schema appear as headings at the user interface. These nametags appear in the External terminology schema, as well. Hence, the External

terminology schema contains all the permissible words in the Contents layer and the Layout layer, and the External terminology schema defines the permissible grammar of these layers.

The External terminology schema contains a data tree, where each node is defined within a name space of its superior node. The External terminology schema allows for use of local nametags and significant duplicates. The data nodes in the External terminology schema appear as headings at the user interface.

Several External terminology populations may share the same External terminology schema. This correspondence is stated by a Schema reference from one data node to another. In principle, a population data node may have several Schema data nodes. Hence, the mappings from populations to schemata may be many-to-many, while most often the mapping is many-to-one or one-to-one. However, any data node within a schema or population may have additional schema references to other data nodes.

The data nodes recursively subordinate to a schema are called classes. The data nodes recursively subordinate to a population are called instances relative to their classes. One instance data node may be created from one class only in that schema. However, this class may be created from still higher classes. Also, the subordinate data to the data instance may be generated from a class in a different schema.

The mapping from data instances in a population to data classes in a referenced schema is homomorphic, i.e. a many-to-one mapping, However, the mapping from the instance to the class is only known at creation time, and may in some cases not be traceable at a later event. However, a specific reference from the instance to the class may be created, e.g. by tagging the instance (and the class) with a unique class label. Most often, this tagging is not needed, because all (not necessarily unique) class labels will appear among the instances.

For a larger scope and more details on requirements, see Annex E.

The camera metaphor

This subsection links the Data transformation architecture to the camera metaphor, which was introduced in Part 1 section 2 on Phenomenology.

The Data transformation architecture is about statements in some formal languages and translation of these. The camera metaphor is about images and interpretation of these. Hence, the two “architectures” may be considered being orthogonal to each other, only with a common concept layer.

However, the similarities are much greater. The Layout layer is managing the pixels in the image. The Contents layer contains the found structure of the image. This is the structure of phenomena that we discussed in Part 1 section 2.

The External terminology layer contains a normalized structure of several images, ie. of several paths through the graph. Hence, several overlapping paths in the Contents layer are replaced by one path plus conditions in the External terminology layer.

In the next part of the book, we will learn how to state the data structure in the External terminology layer before we can correctly state compound statements in the Contents layer. This is the opposite way of reasoning from the Ant’s reasoning in Part 1 of this book. Also, we will have to state the contents of a screen before we can design its layout.

Outlook

Most designers of human-computer interfaces start with their conceptions about their physical environment, eg. of tasks, and with conceptions of their own thinking about this environment. They may create architectures for this UoD.

In the data oriented approach, we start with designing the data to be used by the IT system, and design presentations of these data. The Data transformation architecture is about the IT system, and it is not about the human users and their environments. Due to the formal character of the data oriented approach, some will say that we do computer-human interface design. However, by using this approach, we avoid all superstitions about human thinking.

We are convinced that the data oriented approach is more efficient and flexible for the end-users than task oriented designs. See the subsequent section.

We have in this section been designing how a database application will process its data and what requirements this puts on the data.

Using the data oriented approach, the human developer of an IT system will do inside-out design of schemata, in order to get everything consistent and right. The end user may insert raw data, but these are processed according to the templates in the schemata defined by the developer. The schemata ensure that the end user's data, which move outside-in, will be correct.

The schemata correspond to the observer's pre-conceptions of what he observes. If he thinks there are Russian submarines, he may see them everywhere. If he thinks there are trolls, he imagines them everywhere in the forest, even if he hears only the wind.

Any instance may fit into several class templates, and it is a challenging task to find the right one. The inside-out design, outlined in the previous paragraph, ensures that the right class is already found when the end user inserts his data. However, you cannot hinder any misuse of the system; a user register a note as being a car, a car to be a person etc. The system will accept anything as long as the syntax is right.

If we only do image processing, it may be of no use to distinguish the External terminology layer from the Concept layer. However, if we use many different means of observation, like light, sound, touch etc., we may have an External terminology schema for each, and map to common concepts.

The mapping from the Layout layer to the Contents layer is a kind of filtering, where we remove not essential information, but which are used to find the contents. For example, colours and fonts may tell what kind of term is inserted or presented.

The mapping from the Contents layer to the External terminology layer corresponds to the analysis of commonalities of different paths through the contents data for creating a common normalized structure in the External terminology layer.

See also the last subsection of Annex E, Detailed Requirements.

12. Design of IT systems

How does the Data transformation architecture differ from ordinary designs of IT systems?

This section is a note on alternative approaches to design of IT systems, and indicates the impact and opportunities of using the Data transformation architecture.

On methodologies

There are many schools for development of IT systems. Most development methods belong to the System theoretical school of software development. According to these approaches, a part of the organization of an enterprise is analysed. Most often, the analysis is done through a hierarchical breakdown, and most often the breakdown is abstract, as it does not comply with the existing organization of the enterprise. Some even use standardized breakdowns for any enterprise, and do not describe the particularities of their own enterprise. This standardized breakdown may be used as a reminder of what aspects others have been studying.

Some methods make a breakdown all the way down to elementary processes, while others do not make such a complete breakdown. The processes are shaped to fit with the current environment, e.g. to access one or more external systems.

See the following Figure.

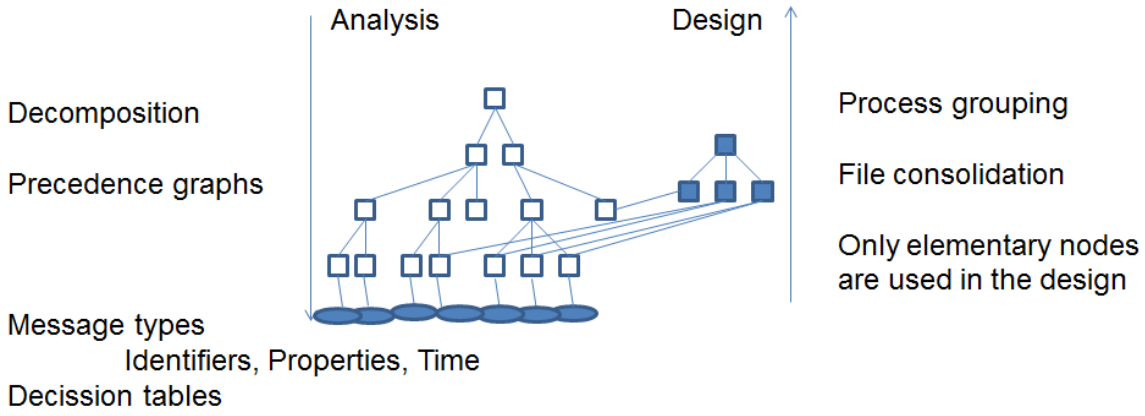


Figure 12.1. System theoretic breakdown and design

The analysis is done top-down, while the design is done bottom-up. In the design, elementary processes are grouped into modules and programs. See the right hand part of the above Figure.

In this text we will not explain or show examples of process decomposition, precedence graphs, message types, decision tables, file consolidation and process grouping. We only make some comments on the overall approach.

The implementation is done by creating workflows through the designed processes. Data and functionality are identified within each process, and are implemented within the designed processes.

This way, the developer freezes existing or imagined methods of work from the analysis into the new IT solution. Data are typically organized through identification of functional dependencies into a normalized relational database. This is not depicted in the above Figure.

There are many unfortunate aspects of using the approach outlined above:

1) Existing way of work is copied into the new solution
2) There is a strong belief in the existence of abstract processes, which exist only in the developers' head, until they are implemented as software modules
3) The implementation will normally only allow for one way of carrying out the work, and the flexibility for updating data in different sequences will be very limited
4) The developers are programming the users to work in a certain way
5) The developers are not helping the users to understand what they are managing
6) Business rules are implemented in work processes outside the data, which may lead to inconsistencies if work is carried out in a different way or sequence
7) Data definitions are taken for granted from the old solution, or from existing natural language dictionaries, and no serious redesign of data is considered
8) Data are not designed for achieving overview, efficiency and flexibility
9) The developers think that it is processes and not data that make the solution efficient
10) Or much of the above is copied from an existing COTS solution, without serious analysis or redesign

The described approach seems to take the existing enterprise as its Universe of Discourse (UoD). However, the enterprise is not primarily managing data about itself. It is managing data about customers, services, telecommunication network, cars, etc., which belong to a UoD outside the enterprise. This other UoD is most often not properly studied.

The developers should study this other UoD and ensure that they are studying the right UoD. Many developers are missing the right UoD, and we realize that it is no trivial task to identify it correctly.

The developers should observe the entities and their relationships within this right UoD. It is not sufficient to study written texts or existing data. To understand the UoD and to design data for representing it are not two different tasks. It is the data and their data structure that represent the understanding. Ref. Part 1 of this book.

Based on an understanding of the entities and relationships, the developers should design a data structure. In doing so, they should be focusing on the existential dependencies between entities, as these will be implemented as name bindings and functional dependencies between the data.

Is study of the entities and their relationships in the right UoD sufficient? It is not. You are going to design data that are efficient and flexible to manage.

We will not here present guidelines for good data design, but we will give an example of challenges.

> Suppose you are going to manage cables between sites. Then you may create two entity classes, Cable and Site. You add two relations for termination of each Cable in two Sites. However, there may be many cables between the same two sites. Hence, you may add an entity, Site relationship, that represents two Sites in alphabetic sequence. Note that the sequence is about the data, and is not about any property of the Sites. Then you add a relation showing which Cables belong to which Site relationship. You could define a function, which generated a Site relationship whenever the first Cable between two Sites is created. However, maybe you rather want to identify the Cables within its Site relationship. Therefore, you define the Cables as being contained in their Site relationship. And so on.

The example illustrates how we can define and massage the data structure for a UoD. The entity Site relationship may not exist among the phenomena of the UoD, as the Site relationship is defined by the identifiers of the Sites in alphabetic sequence. These data and their sequence do not exist out in the UoD. Hence, we have created a data structure that is not a data model of the UoD. We create data that improve overview, efficiency and flexibility. We do not create data models.

Having said all this, it is not forbidden to look at existing data designs, to see what you need to accommodate, what you can reuse, and what you can improve.

How do we do the data design work? Can we write requirements? If you ask a stone age person to identify requirements for a vehicle, you may get enough to invent a sledge, or for using an animal or a human to ride on. I see that most expert users write only requirements according to what they have today. Foolish developers freeze their words into the new design.

To invent a good data structure is like inventing a new vehicle, a car or airplane. You cannot take the users requirements literarily for this kind of design. The developer needs to have a good understanding of what kind of work the user is doing, of his UoD, see challenges and opportunities, and create a good design that the users or other developers could not imagine. He must be a good designer and a good architect. Some call this Design of artefacts. I could have written guidelines on how to do this work, but this is not the purpose of this book.

In the previous section and Annex E, we have written requirements for Language design. Haven't we then contradicted what is here said about requirements? Well, we have had the new design in mind when writing the requirements. Without knowing the answer, it would have been impossible to write the requirements this way. Without having a design, you cannot write complete and consistent requirements; rather you would write a lot of bewildering requirements.

Human-computer interface design

The process-oriented approach is clearly linked to task oriented design of human-computer interfaces. The tasks may be leaf nodes of the process decomposition, some are manual and some are automatic. The tasks of the human user are identified and linked together, using a workflow approach. This is very similar to process design according to Friedrich Taylor's Scientific Management. The focus is on what tasks to perform in which sequence. Activity theory provides a combination of process and task design.

The task-oriented approach typically leaves out study and design of the language used at the screens and how the arrangements of fields describe the Universe of Discourse - and convey the understanding of the UoD. In the data oriented approach, the design of terms and their grammar are put into focus. The designer is concerned with how accurately and efficiently they convey information about the Universe of Discourse.

In the data oriented approach, the end-user data are edited in an editor, and the developer may altogether skip design of tasks. The users edit data about their Universe of Discourse, which they understand. To edit data in a database application is in principle no different from editing data in a text processor. Hence, the data oriented approach offers a different design, implementation, use and understanding of database applications. A spreadsheet is a good example use of a data oriented approach. However, the spreadsheet has a very simple data structure.

See example screen layout designs in Annex G and in Part 4 section 15, using the data oriented approach.

Long transactions

In sections 9 to 11 we have outlined the architecture of an IT system. Often an IT system does not exist in isolation. It often exchanges data with other IT systems and humans. We may therefore extend the IT system to manage its own interaction with its environment, to know where it has sent data, from where to receive data etc. For this purpose, this subsection is a short note on order handling.

In the previous subsection, we have indicated how we may go along to create the data structure of the right UoD. The population data prescribed by this data structure we may call the Main register.

The enterprise itself can be considered to be a UoD, as well. The data about the enterprise we call an Order register, also called a transaction register.

How do we create a data structure for the Order register? You may not need to, because you can use a framework for any enterprise, and add the particular classes from the Main register of the particular enterprise. However, all population data will be different for each enterprise. The point made here is that the orders can only send data that the IT system already has in its Main register, and receive data that will go into the Main register. See the mapping between Part of Main register and Part of Order register in the Figure below.

In addition to what is said in the previous paragraph, we need to design the workflow. The IT systems, organization units, and partners may be the Actor entities of the Order register. We want to manage communication between these. Hence, we create Queues, which have In-relations to and Out-relations from Actors. The Queues may contain Orders. Orders are long transactions between Actors. Short transactions are only carried out within each IT system, and each system guaranties consistency of all its population data in real time. Long transactions across systems do not guarantee consistency in real time, but may give consistency after some time.

The indicated schema notions for Actors and Queues may be the same for all enterprises, but the instances in the population will be specific for each enterprise.

Orders may only contain data of the same kind as of the Main register. Hence, their data structures should be similar. However, in the Main register, the data are defined locally to the Main register. In the Order register, the corresponding data should be defined locally to each Order. Additionally, there should be relations between the contents of the Main register and the Order register, showing what different Orders are saying about a particular entity. This mapping is indicated in the next Figure. Actors and Queues are not illustrated.

Both the Main register and the Order register may present and edit data according to the Data transformation architecture.

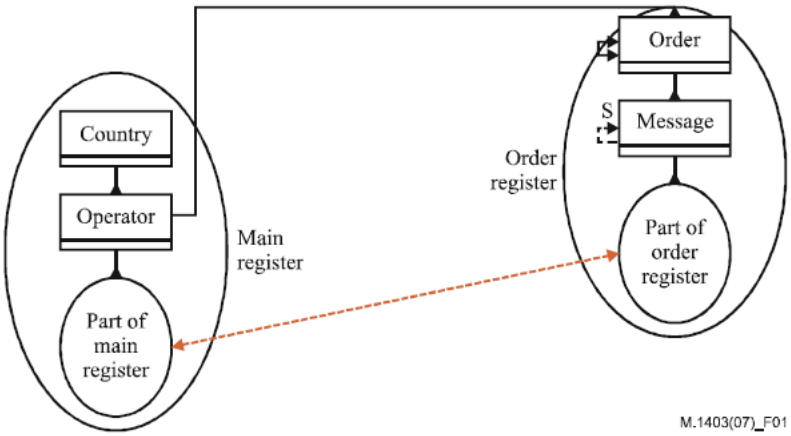


Figure 12.2 Illustration of Order register and the mapping to the Main register (illustrated by a dotted two-way arrow)

If you look up an entity in the Main register, and list all the mentioning of this same entity in maybe different Orders in the Order register, you will see the history of all actions on this entity. Messages within the Message class in Figure 12.2, have a similar role as of exchanged normalized Statements in classical logic. Contents populations have a similar role as of compound non-normalized Statements in classical logic.

Much more may be said about design of both the Main and the Order register; this we will not do. When this design is done, other schemata of the layered architecture must be designed, or they may be generated automatically. Here we will not tell how. The final design may be an editor/translation solution as described in section 11 on Language architectures.

In the next part of the book, we will present the proposed languages.

Outlook

The Data transformation architecture provides a new framework for design of database applications.

Process oriented analysis and design is the traditional approach for development of IT systems. In this approach, data are typically taken as-is, and are not seriously analyzed and redesigned.

Process oriented analysis and design tend study existing way of work and freeze this into the new solution. The complexity of the IT solution tends to become high.

When doing data oriented analysis and design, we study the UoD being managed by existing way of work. Hence, process oriented analysis and design and data oriented analysis and design tend to study different UoDs.

The data oriented design should focus on designing data for the main register. The objective is to design data that are efficient and flexible to use by end users. Efficiency and flexibility are achieved by data design, not by process design. This is contrary to what most people think.

All aspects of the solution, including orders and work flows, may be designed by using the data oriented analysis and design.

Through use of data oriented analysis and design, we may achieve very low complexity of the IT solution.

Through use of the Data transformation architecture, we enable the database application to understand both its data and their behavior.

The Data transformation architecture shows the design of a sophisticated observer. In the process oriented design, the business logic is dispersed into particular programs per user function. The process oriented design may perform or support the users' tasks, but its software is no intelligent observer.

An observer may be simple or complex. The Ant may observe its environment with or without the camera. When using the camera, the camera becomes an integral part of the observer. When a person looks at a screen from a distance, the screen is a part of his environment. When the person comes close to the screen, he becomes submerged into it, and only the contents of the screen belong to the environment. When a person is driving a car, he can feel the road shoulder under its tires; the car is an integral part of the observer. The body of the car transmits the signals, and no specialized medium is needed for the communication. Nerve cells are specialized means for detection and transmission, but a blind person's stick may do, as well. These examples show that what counts as an observer may depend on the situation. Any phenomenon may be an observer of some other phenomena. The observing phenomenon must somehow become affected by the observation; if not, it is not an observer of the other phenomena.

Part 4

Language notions

- What language is needed to express the observer's view? -

13. External Terminology Language

What will a language for stating anything look like?

The External terminology layer contains the observer's understanding of his Universe of Discourse.

In this section, we introduce a three-dimensional data tree for normalized statements in the External terminology layer. Every node is defined subordinate to its superior node. We may by navigation state references between any nodes in the data tree. The nodes represent the phenomena.

Each node in the data tree is represented by a colon (:), corresponding to the pixels in Part 1 of this book. The colons may or may not have nametags. The colons are organized in three directions. The three directions tell if the phenomenon is observed from the superior one, if they are observed in parallel, or if a condition is stated. Only the directions distinguish terms and operators.

This new kind of logic is called Existence logic, where the operators do not map to truth-values, but impact creation, execution or deletion of nodes. Hence, operators are treated similar to functions. Schema-population references are introduced as particular kinds of functions, and these references may be used recursively.

Execution of the data tree goes from the root towards the leaves, and there is no separate control flow as of traditional programming languages. Functions are general program notions, and they execute throughout the data tree in an algorithmic way. The data tree may grow and shrink during the execution.

In this part of the book, we will use examples from every day life, such as persons and cars. The examples could just as well have been about galaxies and particles, but the examples are not of relevance. The understanding of the universe is provided by the structure of colons.

The data tree

Data are arranged in lists of lists. The lists may have three dimensions:

- Subordinate
- Next
- Condition

The opposite directions in the lists are

- Superior
- Previous
- Instruction,

respectively.

We may refer to the list structure as a three-dimensional data tree, as often we disregard the Next-Previous ordering, but strictly speaking, the data make up lists of lists, recursively.

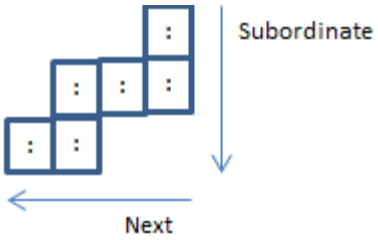


Figure 13.1. Example two-dimensional lists of lists

The same list structure may be written in a two-dimensional alphanumeric notation. Here, subordination is indicated by indentations, while Next is indicated by line shifts. The size of the indentations has no meaning, but the positioning below each other is essential.

```
System
        Car 1
        Car 2
        Person 1
                Owned car A
                Owned car B
```

Figure 13.2. Example alphanumeric notation

The same expression may also be written in a one-dimensional notation. Here commas indicate Next, and parenthesis indicate the Subordinate.

```
System (Car 1, Car 2, Person 1 (Owned car A, Owned car B))
```

Figure 13.3. Example one-dimensional notation

Right hand parenthesis may often be left out, but not always.

```
System (Car 1, Car 2, Person 1 (Owned car A, Owned car B
```

Figure 13.4. Example one-dimensional notation without right hand parentheses

The two-dimensional notation may distinguish the data node from the nametag of this node. Here colons (:) are used to indicate the data nodes, and nametags are written in a separate column.

```
:               System
        :       Car 1
        :       Car 2
        :       Person 1
                :       Owned car A
                :       Owned car B
```

Figure 13.5. Example alphanumeric notation

The nametag and its subordinate letters may even be considered being an element of the subordinate lists of lists. The nametags are not mandatory. Therefore, any structure without nametags is permissible. Also, there is no rule that nametags need to be different; nametags anywhere in the structure may be identical.

The colons in Figure 13.5 represent the phenomena; they are not the phenomena. In order to distinguish these nodes from the nametags, we may use two levels of colons. The rightmost colon represents the nametag, and under this comes the letters in a sequence.

```
:     :                 System
      :     :           Car 1
      :     :           Car 2
      :     :           Person 1
            :     :     Owned car A
            :     :     Owned car B
```

Figure 13.6. Separate indication of nametags

A node which can be found by going to the Subordinate node, and then recursively to Next nodes, is called a Contained node. A Contained node is identified within the scope of the superior node. Hence, the superior node is the root of a name space for Contained nodes.

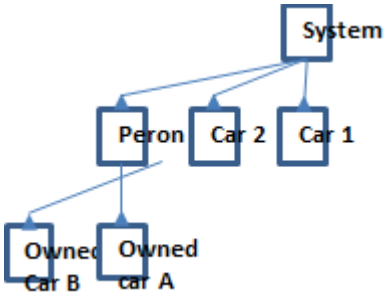


Figure 13.7. Depiction of Data tree with Contained nodes

In Figure 13.7, containment is indicated by reversed arrowheads.

We may transcribe the contents of Figure 13.7 into the terminology of Part 1 section 2 on Phenomenology: The Figure states that a System observes a Person, Car 2 and Car 1. The Person observes Owned car B and Owned car A.

The lines indicate lines of observation/instrumentation.

Conditions and references

Some nodes in the data tree may contain Conditions, which state references to other nodes in the data tree. Each data node can have only one Condition. In the next Figure, we see a Condition on Car 1, that it has the colour Red.

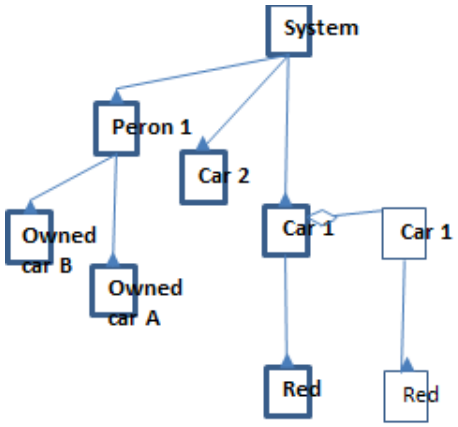


Figure 13.8. Simple condition

In a two-dimensional notation, we use a condition symbol (<>) to indicate the third dimension.

```
:                   System
      :             Car 1
            :       Red
            <>      Car 1 (Red
      :             Car 2
      :             Person 1
            :       Owned car A
            :       Owned car B
```

Figure 13.9. Alphanumeric notation for a simple condition

The Condition states that if the colour Red is not found under Car 1, then Car 1 is deleted, and the contained Red item and the Condition goes with it. We are later going to present a full If-Then-Else statement, but so far, we have exhibited aspects of the If-Then-Else statement only. We will present a reinterpretation of the truth based If-Then-Else statement to be about the existence of data nodes, their creation and deletion. This logic we call an Existence logic.

We will also need a notion about something that does not exist. For this, we introduce the negation symbol (!). See Figure 13.10. Note the use of post-fix notation for negation.

```
:                   System
      :             Car 1
            <>      Car 1 (Red !
```

Figure 13.10. Alphanumeric notation for negation

The above example may be read in a pseudo English language as “System has Car 1 on the condition Car 1 exists and has Red not exists”. The operations are always on the existence of data nodes, and not on truth values. The term “has” in the quotation may be replaced by the term ‘observes’.

In the above example, the Condition (<> Car 1 (Red !) is satisfied, i.e. Car 1 (Red is not found. Hence, the Condition is satisfied, so Car 1 will not be deleted.

Conditions may state references to any node in the data tree. For doing so, you may not only navigate to contained nodes, but have to navigate to superior nodes and then to their contained nodes.

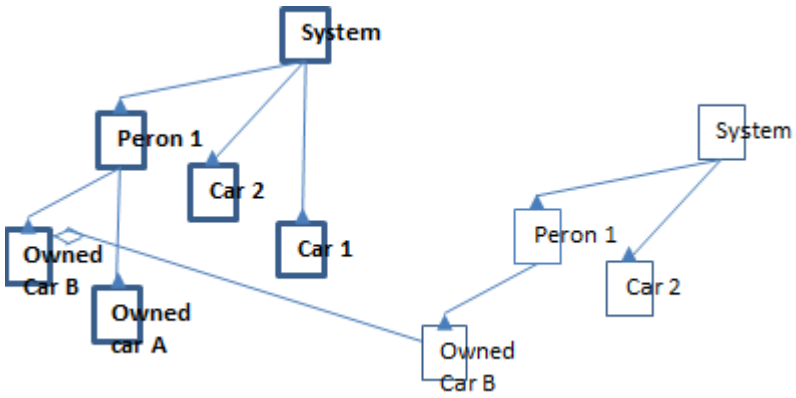


Figure 13.11. Simple reference

For navigation to superior nodes, we will use an apostrophe (‘) in the alphanumeric notations.

```
:                       System
    :                   Car 1
    :                   Car 2
    :                   Person 1
        :               Owned car A
        :               Owned car B
            <>          Owned car B ‘Person 1 ‘System (Car 2
```

Figure 13.12. Alphanumeric notation for a simple reference

Note that we in the previous Figure have used a one-dimensional notation to state the reference, as a two-dimensional notation for this would have taken up too many lines.

The next Figure illustrates how execution of the simple reference is performed. We illustrate execution of the condition, but the instruction part is left out, so far. We see that a read head of the processing unit (♦) is following the condition branch, while the control head is controlling the main data structure, ensuring that it is satisfying the condition.

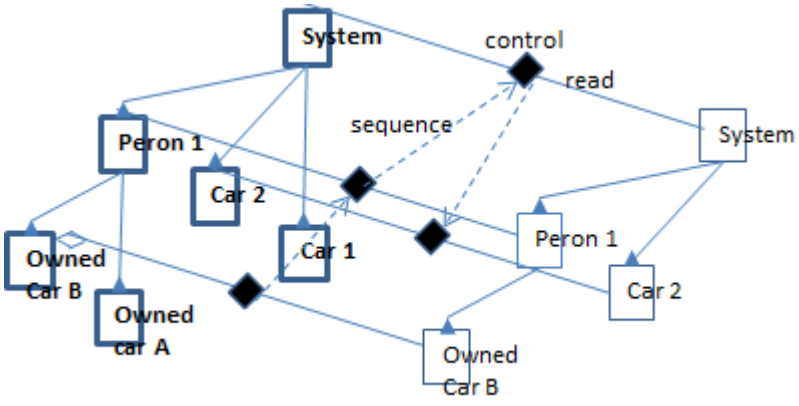


Figure 13.13. Execution of a Condition

In Figure 13.13, the reader should note the similarity to RNA processing of DNAs. The dashed arrows depict the processing sequence.

Stating all the logic, required with a reference between two entities, may become much more complex than shown in this example. If the data architecture is used, this logic can be deferred to the Physical database layer. Here Cars may be stored in records with a globally unique record number, and Persons likewise. The reference Owned car may be implemented as a physical pointer. Hence, for a simple application, simple references may be all we need, and all we need to know. We do not need to express all the logic in the External terminology layer. Also, much of the shown navigation may be skipped.

From the previous Figures, we realize that any data structure may be stated under a Condition. All of it must be found in the main data structure for the Condition to be satisfied.

From the previous Figures, we realize that a Condition may contain both Contained and superior nodes. In Figure 13.8, 13.9 and 13.10, the Condition is the root of the condition data tree. In Figures 13.11, 13.12 and 13.13 it is not, as we have to navigate all the way up to System to find the root. Also, several conditions may be combined, as shown in Figure 13.14 by branching of the navigation path.

```
:                                   System
        :                           Car 1
        :                           Car 2
        :                           Person 1
                :                   Owned car A
                :                   Owned car B
                        <>          Owned car B
                                /:  Red
                                /‘  Person 1 ‘System (Car 1
                        :           Red
```

Figure 13.14. Alphanumeric notation for a complex condition

In Figure 13.14 line 8 and 9, the slash (/) indicates Red is subordinate to Owned car B (in the Condition line 7), and Person 1 is up (‘) from this Owned car B.

In Figure 13.14, first the Condition that the Owned car B is Red is checked, then that there exists a Car 1. The Condition is satisfied when both are satisfied. Hence, this expression is similar to a conjunction.

There may be two-way references between nodes in the data tree. This is illustrated in Figure 13.15.

```
:                           System
        :                   Car 1
                :           Owner
                        <>  Owner ‘Car 1 ‘System (Person 1
        :                   Car 2
        :                   Person 1
                :           Owned car A
                :           Owned car B
                        <>  Owned car B ‘Person 1 ‘System (Car 1
```

Figure 13.15. Alphanumeric notation for a two-way reference

Figure 13.15 does not express mutual dependences between the two references. The simplest way to express this is to make references to the common implementation at the Physical

layer. The reader may be surprised by these comments on implementation, but when using the data architecture, you will consider what pre-made tools are available at each layer, and you will consider where to find the best place for implementation. This comment does not imply that the logic cannot be expressed in the External terminology layer. It means that the full expression would take much space and clutter the essential expressions in this layer.

We could have introduced specialized functions to indicate deference to the Physical layer, but will not do so. See implementation in Annex G. Often, the logic is only expressed as informal comments at the External terminology layer.

Instructions

For expressing dependencies, we will need to introduce write instructions. Write instructions are performed if some Condition is satisfied. The write instruction is stated in the same dimension as Conditions, but in the opposite direction. In a two-dimensional notation, we use the write symbol (><) to indicate this instruction direction.

```
:                       System
    :                   Car 1
        :               Owner
            <>          Owner ‘Car 1 ‘System (Person 1
            ><          Owner ‘Car 1 ‘System (Person 1 (Owned car B
    :                   Car 2
    :                   Person 1
        :               Owned car A
        :               Owned car B
            <>          Owned car B ‘Person 1 ‘System (Car 1
            ><          Owner ‘Car 1 ‘System (Car 1 (Owner
```

Figure 13.16. Alphanumeric notation of mutually dependent references

The above example states in line 3 that Owner may be created under Car 1 in line 2, if the condition in line 4, that the referenced Person 1 exists, holds. In line 5, the reference Owned car B is created under Person 1. However, it is inefficient to state the navigation from Owner to Person 1 both in the Condition and the Instruction. Therefore, we move the Instruction under Person 1 in line 4. This is shown in the next Figure.

```
:                       System
    :                   Car 1
        :               Owner
            <>          Owner ‘Car 1 ‘System (Person 1>< Person 1 (Owned car B
    :                   Car 2
    :                   Person 1
        :               Owned car A
        :               Owned car B
            <>          Owned car B ‘Person 1 ‘System (Car 1 >< Car 1 (Owner
```

Figure 13.17. Improved notation of mutually dependent references

In line 4 we have stated that if Person 1 is found, then we create the reference Owned car B from this Person 1. If this Owned car B already exists, then there is nothing to create. Note that we could have replaced Person 1 after the instruction by a colon (:), as there is no need to test the nametag here. The same applies after the Condition, where we could have replaced

Owner by a colon. Use of the nametags may increase readability, but requires some more processing.

Note that in Figure 13.17, the Instruction part is linked to the Condition. Hence, the Instruction takes place on the main data structure, i.e. on the conditioned side of the Condition. The Instruction is not testing if Person 1 contains an Instruction to create an Owned car B.

Note in line 4, from Owned car B we should have stated the navigation back again to Car 1, as <> Owned car B ‘Person 1 ‘System (Car 1. We have skipped this part of the expression, as the example is already sufficiently complex. Similarly, for the last line of Figure 13.17.

The above additions makes it clear that the reference one way will result in creation of a reference the other way, as two functions are calling each other recursively. We do not accept infinite recursions, and it is therefore the responsibility of the designer to design functions such that all recursions become finite and efficient. The simplest way to do so, is to defer this logic to the implementations, e.g. in a commercial database management tool of the Physical layer. The External Terminology Schema needs only to contain the data elements Owner and Owned car, to which we may attach the access paths to the common implementation of both.

Note that we in Figure 13.17 treat “Car 1 has Owner A who is Person 1” as a very different fact from “Person 1 has Owned car B which is Car 1”, even if the two facts are mutually dependent. When interpreting natural language into logic statements, people – including Frege - tend to believe that “Car 1 is owned by Person 1” and “Person 1 owns Car 1” state the same fact. We do not think so.

Classical logic is about what is the case, independently of when they are created. Classical logic is about a static universe. In Existence logic, we execute the conditions only when Owner or Owned car or some other entity is created.

The logic

We may now summarize the working of the revised If-Then-Else statement:

- If a Condition (<>) is satisfied,
  - i.e. if a main data structure is found that is identical to that of the Condition
- Then execute the subordinate data tree of the conditioned node and
- Execute the Instruction (><) of the conditioned node
  - If the Instruction part contains nodes that do not exist in the main part, then create these by copying from the Instruction part
- Else, delete the conditioned data node,
  - With all its Conditions, contained data, and Instructions

Together with the negation symbol (!), this is all the operations we need. These operations have similarities to truth-value logic, but they do not give truth-values as outputs. They

- test existence of terms,
- create terms,
- execute terms
- or delete terms

The terms have no global existence, but exist subordinate to other terms, and you have to navigate between terms.

In classical logic you do not create or delete terms, and you do not navigate between terms.

We will come back to the details of execution, and we will add the notion of functions. However, the Condition and Instruction notions provide the needed basic functionality.

Schema and type notions

A node may contain a node with the nametag S, which has a reference to some other node. This other node is called the schema of the first node. The first node is called the population of this other node. The other node may contain a node with the nametag P, which makes a reference to the first node.

The nodes that are recursively contained in the schema are called classes relative to the nodes that are recursively contained in the population. The nodes recursively contained in the population are called instances relative to the classes.

```
:                                   System
        :                           Schema
                :                   Car
                        :           License number
                        :           Owner
                                <>  Owner ‘Car ‘: (Person
                :                   Person
                        :           Name

        :                           Population
                S
                        <>          S ‘Population ‘System (Schema
                :                   Car
                        :           License number
                                :   1
                        :           Owner
                                <>  Owner ‘Car ‘: (Person (Name (JOHN
                :                   Car
                        :           License number
                                :   2
                        :           Owner
                                <>  Owner ‘Car ‘: (Person (Name (JOHN
                :                   Person
                        :           Name
                                :   JOHN
```

Figure 13.19. Example Schema and Schema reference (S)

Note in Figure 13.19, there are three Car inscriptions, one in the schema and two in the population, in addition to references in the Condition-s. None of the three inscriptions refers to the same phenomenon. Hence, we allow for identical inscriptions in lists to denote different phenomena. This violates the Type-token principle of classical logic. Note also that subordinate data to the two Car instances may not appear if there is no constraint that prescribes them. Hence, a population containing two Car inscriptions only is permissible.

Note in Figure 13.19, that in the Schema we often state references between entity classes only, while in the Population, references have to be made to identifiers of entity classes, eg. to the value JOHN of the Name attribute of a Person entity. This addition to the references can be made by pre-compilation, or in real time during execution. In the schema, the references to entity classes are sufficiently unique. The identifier attributes are only needed among the instances.

It is possible to make references between instances without using identifiers, as well. Then navigation has to be used, eg. navigate to the recursively Next person until the right one is found. A simpler solution is to map the entities to records with physical addresses in the Physical layer. This way, we allow for having significant duplicates in the External terminology layer, while the burden of keeping them apart is handled in the Physical layer.

Furthermore, note that all class labels are copied from the Schema into every entity instance. The instances make up lists. This structure of the instances is very different from the notion of sets.

We have not yet shown where the values, like 1, 2 and JOHN come from.

The mapping from instances to classes is homomorphic. This means that

- There is a many-to-one mapping from instances to classes
- Each instance belongs to one class only, i.e. strong typing is used
- If there is a containment association between two instances, there shall be a containment association between the corresponding classes, as well
- If there is a reference between two instances, there shall be a reference between the corresponding classes, as well
- Operations and functions are copied from the classes into instances, and are executed at the instance level

The schema-population references may be applied recursively, as shown in the next Figure.

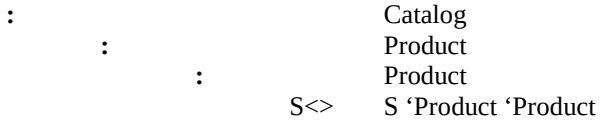


Figure 13.20. A schema reference within a schema

In the previous Figure, a Product may contain Product-s. The contained Product has its superior Product as its schema. Hence, the schema permits that the contained Product may contain Product-s recursively to form a tree of Product-s.

The next example shows how the Contract class may inherit properties from the Catalog class. This allows references between Product instances within Catalog instances, and allows references between Product instances in Contract instances.

```
:                                                   Schema
        :                                           Catalog
                :                                   Product
                        :                           Identifier
                        :                           Product
                                <>                  Product ‘Product ‘: (Product
        :                                           Customer
                :                                   Name
                :                                   Contract
                        :                           Identifier
                        S<>                         S ‘Contract ‘Customer ‘Schema (Catalog

:                                                   Population
        S<>                                         S ‘Population ‘: (Schema
        :                                           Catalog
                :                                   Product
                        :                           Identifier
                                :                   A
                :                                   Product
                        :                           Identifier
                                :                   B
                        :                           Product
                                <>                  Product ‘Product ‘Catalog (Product
                                                    (Identifier (A
        :                                           Customer
                :                                   Name
                        :                           A
                :                                   Contract
                        :                           Identifier
                                :                   123
                        :                           Product
                                :                   Identifier
                                        :           1
                        :                           Product
                                :                   Identifier
                                        :           2
                                :                   Product
                                        <>          Product ‘Product ‘Contract ‘(Product
                                                    (Identifier (1
```

Figure 13.21. Recursive use of schema references

Any Product (third line) may contain a reference (fifth and sixth line) to another Product (third line) within the same Catalog. The Catalog is indicated inside the reference by a colon (:) only. The reason is that this reference will be reused inside Contract-s and not only within Catalog-s. A Customer class (seventh line) contains a Contract class (ninth line). The Contract class contains a schema reference to the Catalog class. This means that the Contract class may have the same contents as a Catalog class. Note that the schema reference is only inside the Schema and will not itself be instantiated in this example.

Catalog-s containing Product-s with references may be created in the Population. Contract-s containing Product-s with references may also be created in the Population. In the previous Figure, the references between Product-s can only take place within one and the same Catalog, or within one and the same Contract. This constraint is removed in the next Figure, where a reference is stated from a Product under Contract to a Product under Catalog.

In the next Figure, in the Condition in the Schema, line 6, we have used recursion (&) when ascending (‘) and when descending (()), in order to allow for the arbitrary long references from Product (instances) under Contract to Product (types) under Catalog. This way of specifying allows Product (instances in install base) to inherit and refer to the definitions used for Product (types) in the Catalog instances. Much more could be said and explained about this example, but we abstain.

```
:                                                   Schema
        :                                           Catalog
                :                                   Product
                        :                           Identifier
                        :                           Product
                                <>                  Product ‘& (& Product
        :                                           Customer
                :                                   Name
                :                                   Contract
                        :                           Identifier
                        S<>                         S ‘Contract ‘Customer ‘Schema (Catalog

:                                                   Population
        S<>                                         S ‘Population ‘: (Schema
        :                                           Catalog
                :                                   Product
                        :                           Identifier
                                :                   A
                :                                   Product
                        :                           Identifier
                                :                   B
        :                                           Customer
                :                                   Name
                        :                           A
                :                                   Contract
                        :                           Identifier
                                :                   123
                        :                           Product
                                :                   Identifier
                                        :           1
                                :                   Product
                                        <>          Product ‘Product ‘Contract ‘Customer
                                                    ‘Population (Catalog (Product (Identifier
                                                    (A
                        :                           Product
                                :                   Identifier
                                        :           2
                                :                   Product
                                        <>          Product ‘Product ‘Contract ‘Customer
                                                    ‘Population (Catalog (Product (Identifier
                                                    (A
```

Figure 13.22. Recursive instantiation

In the previous Figure, the schema reference (line 11) to Catalog appears only inside the Schema, and will not be populated into the Population. This reference tells that a Contract class may have the same contents as a Catalog class.

The Condition-s (on Product-s within within Contract-s within Customer) in the Population state references according to the prescription in the Catalog class. Hence, they may state

references from Product-s in Contract-s to a Product in a Catalog instance, as shown. These references are all within the Population.

The schema reference may be used to define value types.

```
:                       Schema
    :                   Letter
        :               a, b, c, d, …
    :                   Name
        S<>             S ‘Name ‘: (Letter
```

Figure 13.23. Definition of Name

Figure 13.23 allows a Name to be any sequence of Letter-s, like aaab, baa etc.

If we want to reuse this definition for a Customer’s Name, we cannot use the schema reference, because this would allow for multiple instances of the value within the Customer’s Name. If we want a one-to-one mapping, ie. isomorphism, from Customer (Name to Name, we could have achieved this by adding a cardinality constraint, but we will rather add a particular Type function (T) as a shorthand for the schema reference plus the cardinality constraint.

```
:                       Schema
    :                   Name
    :                   Customer
        :               Name
            T<>         T ‘Name ‘Customer ‘: (Name
```

Figure 13.24. Definition of attribute type of Customer Name

Type declarations may be used recursively to define complex entity, attribute and value structures.

Categorization of nodes

Any node in the data tree is called a term.

The root of the data tree is categorized as being an entity.

Contained nodes of an entity may themselves be entities. One entity may contain several contained entities. One entity can only have one superior entity.

Contained nodes of an entity may alternatively be attributes or attribute groups. An entity may have several attribute instances from an attribute class, and several attribute group instances from an attribute group class. An attribute can have only one superior attribute group or entity. An attribute group can have only one superior attribute group or entity.

Attribute groups may contain attribute groups or attributes. An attribute group may contain several attributes, and contain several attribute groups.

Each attribute shall contain one value or be empty. Multivalued attributes should be avoided, as this is replaced by allowing multiple attributes or attribute groups of the same class. However, the single value of an attribute can have arbitrary complexity. Note that entities and attribute groups shall not contain values.

Some entities may provide a reference to another entity. Entities providing references are called roles of the other entity. A role is itself an entity. Role entities may themselves contain

attribute groups and attributes. These attributes and attribute groups may be different from those of the referenced entity. Example: The role entity Owned-car may contain a local identifier, eg. A, B, C of the Owned-car-s of this Person; this identifier may be different from the identifier of the referenced Car. Also, Owned-car may have attributes about Date of purchase, which the Car may not have.

Note that in the Relational model, it is attributes of relations that take up the role notion. In Existence logic, it is entities – and not attributes or values – that take up the role notion.

In the schema, references may be stated between entities, and need not be stated between attributes, as of within the population.

Functions

Any node having a Condition is a Function. Every node in the data tree may have a Condition, or extra nodes may be added with a Condition for the Function. Note also that a value of an attribute can be a Function. Also, a value may contain a Function.

Whenever a Function data node is created, changed or deleted, the Function is executed, and the Function must be defined to work properly for these state changes. Note that the creation, change or deletion happen in the Population according to its prescription in the Schema.

The condition side, i.e. the right hand side, of a Condition symbol makes references to all the arguments of the Function. The arms from the Condition may branch off at any place and stretch out to the arguments found in the data tree. The Function is defined to process what it finds at the tips of the branches. A branch may point at a particular individual node, eg. the Colour of a Car, or the branch may point at a kind of individual node, eg to all Owned-car-s of a particular Person. See Figure 13.25.

```
:          Person
    :      Owned-car
    #<>    # ‘Person (Owned car
```

Figure 13.25. A Function, #, for counting the number of Owned cars of a Person

The Function definition is only dependent on what arguments it finds at the tip of its branches. The navigation to these tips may vary for every use of this function nametag.

Figure 13.25 does not tell where the function value is to be put. The Function will contain one or more Instructions (><), each with a branch pointing to where to put the function value.

The Condition branches may be compared to dendrites of a nerve cell. The Instruction branches may be compared to axons. Figure 13.26 shows how a status is set in all Owned-car-s of a Person.

```
:                Person
    :            Status
        :
            <>   : ‘Status (New
            ><   : ‘Status ‘Person (Owned-car (Status (Free
    :            Owned-car
        :        Status
```

Figure 13.26. A Function setting the Status to Free of all Owned cars of a Person whose Status is set to New by some user

In Figure 13.26, all details of the Function are spelled out, and hence, the Function has got no separate nametag different from the colon (:) itself. The colon indicates the value of the Person's Status. Note that the Function is contained inside the value.

Functions may get their definition, except for the navigation, from a library of Function types. This is similar to the library of value types, as indicated in Figure 13.24. Also, the reference to the library is done through the type reference, T.

The Function may contain one or more Instructions. Each Instruction may branch off from the condition branches or branch off from the recursively contained nodes of the Function node. Except for the navigation, an Instruction is a write Instruction.

A Function may have many Instructions, but not more than one per data node. The reason is that the Function is defined as a program of If-Then-Else statements, which can have many inputs and many outputs. In classical logic, a function is a many-to-one mapping. We allow a Function to be a many-to-many mapping.

Functions may call each other recursively, but not indefinitely. It is the responsibility of the designer to define functions and function calls such that the execution terminates and become efficient.

A Function contains instructions to a processing unit. Inputs, outputs and intermediate results are held in the processing unit until the control is left from the Function node to the superior node.

A Function may contain other Functions, which may be placed anywhere in the Condition branch of the Function.

Cardinality constraints are particular kinds of Functions. Figures 13.27 and 13.28 show two examples.

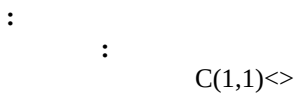


Figure 13.27 Min and max cardinality on a value node expressed as a Condition on the value

Figure 13.28. Min and max cardinality on a value node expressed as a Condition on the attribute

In Figure 13.28, the first Condition, to the left in line 2, states that the Colour shall have a value, Colour (: . Then the value contains a Function (indicated by the second Condition symbol, (C(1,1)<>, to the right) stating that there shall be only one value.

In the first Figure, 13.27, the cardinality Constraint is executed whenever the value is changed. In the second Figure, 13.28, the cardinality Constraint is evaluated only when the Colour attribute is created or deleted.

In Figures 13.27 and 13.28, the Condition is empty, i.e. nothing is stated on its right hand side. This means that all arguments are built into the condition function, eg. C(1,1), itself.

Also, the type of the function may be built into it. Therefore, we do not write T<> … under the Cardinality function. We could, as well, have removed the dummy Condition symbol, <>, but it is helpful to tell that this is a function.

Note that all the above examples of Functions are stated in a schema. They will be copied into a population before execution. Hence, the formulations are made on classes, while the execution takes place on instances.

Identifiers

The Identifier function, Id, is introduced to support use of unique relative distinguished names of data instances.

Suppose we have the following schema:

(Garage (Car)).

We may then create the following populations:

(Garage (Car, Car, Car)), (Garage (Car, Car)) etc.

We see that instantiation is made by creating significant duplicates. We use copy semantics.

Suppose we have the following schema:

(Garage (Name (Id<>), Car (Name (Id<>))).

Id is the Identifier function, which is placed subordinate to the Name attributes Garage (Name and Car (Name. Note that Garage contains Name and Car.

We may then create the following populations:

(Garage (Name (Id<>), Car (Name (Id<>), Car (Name (Id<>), Car (Name (Id<>))), (Garage ((Name (Id<>), Car (Name (Id<>), Car (Name (Id<>))) etc.

Instantiation is made by duplication. Note that the Id function is instantiated, as well.

If we put in values, we may get the following population:

(Garage (Name (Id<>, A), Car (Name (Id<>, 1), Car (Name (Id<>, 2), Car (Name (Id<>, 3))), (Garage ((Name (Id<>, B), Car (Name (Id<>, 4), Car (Name (Id<>, 5))) etc.

Note that the values, A, 1, 2,…, come next to the Id functions.

Note that here the values are not generated by the system. The values are inserted by the human user, or by another software system, or by a Function that is not shown. Note that validation of the uniqueness within the value space is done by the Id function only, and does not follow from the data tree itself.

In this sub-section, we have used the linear notation, which requires use of a lot of parentheses. Use of the two-dimensional notation, maybe combined with the one-dimensional notation, gives better readability, and less use of parentheses.

Variables

In classical logic and mathematics, we may use variables that apply to several alternative values. The variables are used as intermediaries in the calculations. In the final calculation of a specific answer, the variables are replaced by values, i.e. constant terms.

In Existence logic, we do not use variables. Each value comes with its entire superior data tree. This tree may contain attributes, attribute groups and entity classes. It may even contain

values of attributes of superior entities. The superior data tree serves as a container that holds the value. The superior data tree is never discarded, as it tells the entire context of the value.

In classical logic, when variables are replaced by values, we lose information on where the values came from. We can only go back in the calculations, if they are stored, to find their origin. In Existence logic, the entire context remains in the final production.

However, an attribute may be empty, i.e. it may hold no value, eg. Car (Colour. We may refer to any value as Car (Colour (:. A specific value is written as Car (Colour (Red. The symbol for any data item, :, has a similar role as that of a variable, but we do not use variable names.

However, a Condition may refer to an attribute that holds a value, and the attribute may have a nametag. The function that is conditioned may be defined such that it takes what it finds subordinate to the referenced node to be its input.

:		Person
	:	Owned-car
		: Weight
	SUM<>	SUM ‘Person (Owned car (Weight

Figure 13.29. A Function SUM that adds up the Weight-s of all Owned car-s of a Person

Note in Figure 13.29, that the SUM function is defined to take the contained values of the Weight-s as input. Where to put the output is not defined in this example. The example shows that the attribute name may serve a similar purpose to that of a variable.

The condition part of a function may refer to any node in a data tree. This node may have a complex contained structure. This contained structure is processed by the control head in the same way as when processing any data tree with a read head until it comes back to the root data node. On the tour through the contained data tree, the function will pick up whatever data it can recognize and perform its calculations, and the result is delivered under the data node being referenced in the instruction part. This extends what is previously said about the functioning of Conditions and Instructions.

Existence logic has the class notion within schemata. Generic rules are stated in the schemata. These rules are stated as functions, which are instantiated into the populations.

Values are not only contained in the populations. All permissible values also appear in schemata, together with functions for how to combine them into final inscriptions. Due to use of recursion, the schemata may be much more condensed than their populations, where every actuality is written out.

Since variables are not used in Existence logic, quantifiers go away with them. Where are they hidden? How do we do without them?

Any class definition applies for every instance of the class. Hence, a class definition is in a way a universal quantification. The universal quantifier symbol is replaced by the schema reference, i.e. S<>.

Where is the existential quantifier? Every condition, i.e. <>, requests the existence of the paths indicated at its right hand side. This is in a way an existential quantification.

The hidden quantifiers in Existence logic are not unbound. Any expression is bound within the scope of its superior data node, and the navigation paths state the scope of the navigation.

Scope

In classical logic, bound quantifiers are used to state scopes. Existence logic is using two mechanisms for stating scopes:

- Contained terms, as a contained term only can exist inside the scope of its superior term
- Navigation within a Condition or Instruction expression

Most often we navigate up to the first common superior of the conditioned term and the referenced term. The navigation puts a constraint on the reference. In Figure 13.22 we have navigated to a still superior node in order to allow for the wanted navigation path in the population.

Outlook

We have outlined a new formal language, called Existence logic, which we will compare with other approaches.

Control flow with function calls is the organizing principle of traditional programming languages. Data structures are typically defined separately, and are only referred to from the main part of the programs.

We have taken a different approach. The data structure is our organizing principle. The execution starts at the root of the data tree, proceeds along a branch to a leaf, then to a next leaf, a next branch etc. until it is all covered. The execution goes depth-first. In the next section on Contents language, we will come back to how the execution will be delimited, but it will be delimited by data.

Traditional programming and specification languages use a rewriting grammar. A syntax tree of intermediate and leaf nodes is produced from the grammar. Only the leaf nodes are used in the final production.

We have taken a different approach. The data structure is the grammar. Hence, there is no distinction between a grammar and the data structure, and there is no distinction between the data structure and the syntax tree. Also, we use an attachment grammar. This means that the entire syntax tree appears in the final production. Figure 13.22 is a good example of this. We may very well leave out leaves and branches in the final presentation at the Layout layer, but the superior nodes must always appear. They do not only give the context, but they tell what we are talking about. In the next section on Contents language, we will learn that we for compactness reasons may suppress some of the superior nodes from being presented in the Layout populations, if the presentation is still unambiguous.

We note that in principle, all superior nodes must remain in the final production.

We observe that similar inscriptions, like Car, Car and Car, as of Figure 13.19, may refer to very different phenomenon instances.

In traditional database schema languages, we have one notation for the schema, while the instances in the population are represented and stored in a different way.

Classical logic is different from this, as there is no different notation for stating generic facts and stating actual facts. The actual facts may use more constants, and the generic facts may use more variables, but there is no distinction between these statements in principle. Classical logic has no class notion. The term “Persons” is in classical logic interpreted as denoting the set of all person-s. The term “person” is taken as a variable over the individuals in the set Persons.

We have taken a different approach. We consider the schema to contain templates or prototypes for the instances in the population. “Persons” and “Person” appear as different entity classes in the schema, and they will appear as different entity instances in the population. We may have many entity instances of each entity class.

We copy syntax structures from the schema into the population. Hence, the expressions in the schema are identical to expressions in the population.

Operations and functions are instantiated into the population before being executed. The instantiated operations and functions may not be stored permanently.

The copying semantics from classes to instances is the reason for having a post-fix notation. If we had used a pre-fix notation, like “managed object class Person”, then we could not copy all of this into the population, as an individual Person is not supposed to be a managed object class. In order to allow for the homomorphic mapping from instances to classes, we just write Person for an instance and Person for the corresponding class. These two Person inscriptions will denote different things - a particular instance phenomenon and its class phenomenon. See about denotation of classes in Part 5 section 19, More on existence.

The schema states what is permissible, while the population states the actualities. Hence, the schema states a disjunction of possibilities, while the population states a conjunction of what actually is. The list notion of Next items states the disjunctions and conjunctions.

The Condition states a conjunction of what must be the case, or deletion if it is not satisfied.

The Instruction states consequences, through insertions.

Containment states a strong conjunction.

In our new logic, the logic is always on the existence of data nodes, not on truth-values. Therefore, negation is about Exists Not. We call the proposed language Existence logic.

During execution, the data tree may grow, shrink and change. Existence logic works on a dynamic tree of data. Existence logic is not constrained to a static set of data or conceptual structures, like Classical logic.

The programming language APL uses prefix notation for operators and functions. This allows for use of more general operators and functions than when using the traditional infix notation. In Existence logic, we may have even more generality, as every program can be a Function. The data can simultaneously be Functions when being data, or Functions are added as particular contained Function nodes to the data. Therefore, we use a post-fix notation.

Any data node having a Condition, i.e, any term having a Condition, is treated both as data and a Function. Hence, we need no separate name space for Functions. The Function notation has the following form:

```
F       <> …   x …
           …   >< … y …
```

Here, the dots informally indicate navigation. x and y indicate arguments and function values, respectively. <> refers to the arguments, and >< refers to the function values.

In classical logic and mathematics, the inputs to a function are found by names. We replace naming by navigation.

We use naming conventions that are different from classical logic and mathematics. Rather than using globally unique names of constants and variables, we have to navigate to the place where we find the parameter values, and where to deliver the function value. We navigate through inscriptions and not by use of strings.

We may simplify navigation to appropriate value types and function types by declaring these as unique within a data tree or within a branch of a tree.

Note that the data tree has a similar role as a syntax tree. The Functions are attached to the nodes of this tree, the same way as functions are attached to nodes in an attribute grammar.

A Function is only executed when the Function data node is created, modified or deleted. This means that Functions are executed when incremental changes to the data are made.

Let us compare with Predicate calculus. Predicate calculus has much resemblance to natural language. When using Predicate calculus, we may use predicates like Owns and Is-Owned-By. We may also find truth-values, like Yes/True and No/False. Also, maybe we find quantifiers, like For-all and There-exist. The statements are laid out in a flat space, and their order is not essential. Predicate calculus supports a linguistic and inference perspective.

Existence logic is more like a construction of a building or of a motor. The entire specification is a carefully designed arrangement of syntactical constructs. Conditions and instructions provide interactions between the parts, like the gears and wheels of a clock work. Existence logic supports an engineering and design perspective.

In Existence logic we do not need to use nametags, but they are convenient to have. The nametags are used for search, for simplifying reference statements, and they ease human readability.

In Existence logic every phenomenon is represented by a colon (:), and they are arranged in three dimensions. In Existence logic there is no logical operator, no quantifier, no variable, no predicate, no truth-value, and no interpretation toward sets.

When using classical logic, we typically will use global name spaces. We may achieve this by prefixing the terms. However, this may create duplicate processing.

When executing a data tree in Existence logic, the “prefix” will appear only once – as a superior data node -, and it will define a natural scope of the execution. We avoid the mapping to the flat world of logic and conceptual structures. Therefore, execution of Existence logic can be more efficient. We retrieve only what is strictly needed. Hence, Existence logic is efficient for traversing from instance to instance, and we need not do set operations.

As you will see throughout this book, I have not much trust in natural languages, and thinking based on natural languages. In order to do valid thinking, we have to make a careful design of a model of our UoD, and reason on this. Existence logic is this modelling and design language. Natural language may only be used to explain and report on this model. You cannot and should not extract the model from the natural language, or reason on natural language expressions.

Classical logic has a similar deficiency to natural languages when trying to give overview of and insight into a problem space. Use of schemata, as of Existence logic, gives a very compact and precise understanding of the problem space. See the documentation of an example large application in Annex D.

The External terminology schema and its External terminology population define the observer’s understanding of its Universe of Discourse. Both data and their behavior are expressed in Existence logic.

The External terminology schema gives a good overview of the notions in an application. The External terminology population gives the insight into each fact and their relations.

We have not explored the use of Existence logic for analytic purposes, such as stating logical and arithmetic constraints, and finding optimal solutions. Classical formulations have a couple of hundred years lead on this. Maybe, the simplest approach is then to transform Existence logic formulations into classical logic and use their tools.

14. Contents Language

How do we make compound statements?

This section introduces compound statements in the Contents layer. These statements combine normalized statements from the External terminology layer by adding reflexive pronouns, like “which” and “of”.

The Contents layer is used to state selections and prepare presentations, while all logic and execution are expressed as normalized statements in the External terminology layer.

The Contents population corresponds to the Ant’s navigation paths in Part 1 section 2 of this book. Section 2 shows the paths before the Ant realizes that there are several phenomena of the same node.

In the Contents layer, there are overlapping paths. These overlaps are removed in the External terminology layer, as they are replaced by references between phenomena through use of Conditions.

The Contents layer defines routes through the External terminology layer network.

Which

The next Figure illustrates a Contents schema for the following statement in a pseudo English language, and reads: “A Contents population has a Car-s which has an Identifier, has Colour, has Owner, “which” each is a Person has Name, has Age, has Child-s “which” each is a Person has Name.” The Contents schema prescribes the Contents population. The Contents schema does not distinguish between singular and plural forms, like in the pseudo English. The cardinalities are stated in the External terminology schema only.

```
:                                                   Car
    :                                               Identifier
    :                                               Colour
    :                                               Owner
        <>                                          Owner ‘Car ‘:
            /:                                      Person
                :                                   Name
                :                                   Age
                :                                   Child
                    <>                              Child ‘Person ‘:
                        /:                          Person
                            :                       Name
```

Figure 14.1. Example Contents schema

It is important to note that the navigation in Figure 14.1 is done in the External terminology schema, and not in the Contents schema itself. This may be understood as if each Contents schema is stated as a condition on the root node, Car, in the External terminology schema.

In the above example, in the Conditions, eg. line 5, we are navigating up to the External terminology schema, but when instantiating, this term, External terminology schema, is

replaced by the term External terminology population or some other nametag. Therefore, we indicate this node by a colon (:) only.

The tip (Person) of the navigation path, under the External terminology schema, is moved to the next line, preceded by the slash (/), which indicates that the preceding path is found in the line above. The entire expression below and indented to the Condition expresses the condition path.

In a pseudo English language we may skip the Condition line, <> Owner ‘Car ‘:, and read the slash line, /: Person, as “which is Person”, or as “which is a role of Person”. The slash is a continuation symbol from the previous line.

This Contents schema defines a query where the user may give the Identifier of a Car, and reads its Colour, Owner, the Owner’s Name and Age, and his Child-s.

We observe that the Contents schema is a tree, which navigates across the External terminology schema. In this example, the Car, and the Person-s have globally unique identifiers, so they form independent branches in the External terminology layer. The Contents schema forms one single branch that refers across these branches of the External terminology schema.

The end user will see a screen with headings taken from the Contents schema, and an empty field for a value of each attribute. Note that the headings are identical to the nametags of the Contents schema and of the External terminology schema. The Contents schema and the External terminology schema define the grammar and glossary of the end user screens.

The user fills in a value, eg. 123, under the Car (Identifier, and enters a filled in screen, as shown in the next Figure. As the Contents schema is already a condition on the root node in the External terminology schema, the addition of 123, ie. Car (Identifier (123, will identify this instance in the External terminology population, and the search will state a condition on this instance.

The resulting Contents population will form a branch starting with the single Car, list its (single) Owner, and list (all) his Child-s.

If you create a table of the attributes from Figure 14.1, the columns will become

Car		Owner/Person		Child/Person
Identifier	Colour	Name	Age	Name
123	RED	JOHN	52	BILL
				MARY

Figure 14.2. Example screen shot from Figure 14.1

That the Owner is a Person and Child is a Person are indicated by slashes (/) in the headings.

In the next Figure, we have made the example somewhat more complex by defining Person Name-s within County-s only. We assume that the Person and his Child-s live in the same County. However, we need to navigate from the Person back up to his County in the second condition to be able to refer to his Child-s who are Person-s, as well, but we do not need to list the County-s’ Name in this case, as we assume that they are identical to that of the Owner/Person. We assume that the Car Identifier-s are globally unique.

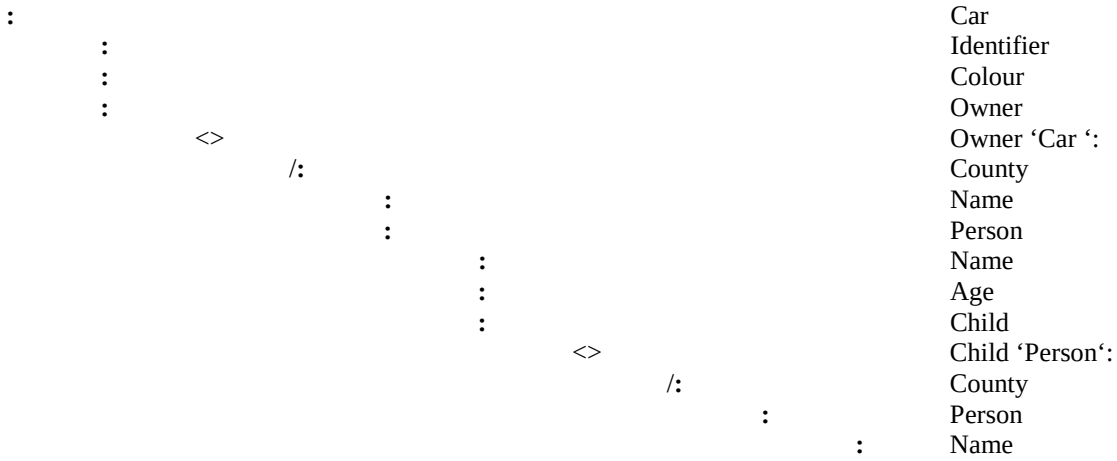


Figure 14.3. Example Contents schema with local Person Names

If you create a table of the attributes from Figure 14.2, the columns will become

Car		Owner	‘County	Person		Child/Person
Identifier	Colour		Name	Name	Age	Name
123	RED	*	XXX	JOHN	52	BILL
						MARY

Figure 14.4. Example screen shot from Figure 14.3

In the above screen shot, we have underlined the entity class headings. We have indicated the Owner role by a star (*) and the superior County by a hyphen (‘). Both the Child role and the referenced Person entity are listed under Child/Person. Note that in his search, the user may fill in any of the above fields, eg. Car (Identifier (123 and Person (Name (MARY. Then the search will return only the paths/lines containing MARY in the Person (Name. The values extend the condition of the search. For a classical logician, this may be understood as a logical AND. If the user fills in both BILL and MARY, this will be understood as two independent conditions, ie. a logical OR on the Child/Person entity. For negation and more complex operations, a separate Operator field is needed. This will not be shown in this book.

There is a problem with the presentation in Figure 14.4, as you may think that the County XXX is the Owner of the Car, while County is only a part of the global distinguished name of the Person. We want the County to be a related entity to the Person who is the Owner. Therefore, we will suppress the first listing of County during the navigation, and only list the Person Name for Owner, and thereafter navigate up to County again after having visited the Person. There is a need for a separate column for stating the suppression, but this is not shown, as we will not here go into a full tool design. We get Figure 14.5.

```
:                                                   Car
    :                                               Identifier
    :                                               Colour
    :                                               Owner
        <>                                          Owner ‘Car ‘:
            /:                                      County
                :                                   Name
                :                                   Person
                    :                               Name
                    :                               Age
                    ‘                               County
                        :                           Name
                    :                               Child
                        <>                          Child ‘Person‘:
                            /:                      County
                                :                   Person
                                    :               Name
```

Figure 14.5. Example Contents schema with navigation to superior

Note in Figure 14.5 the ‘-symbol in line 11 for navigating up to County from Person.

From the above specification, we may get the following screen:

Car		Owner/Person		‘County
Identifier	Colour	Name	Age	Name
123	RED	JOHN	52	XXX
				Child/Person
				BILL
				MARY

Figure 14.6. Example screen shot from Figure 14.5

In Figure 14.6, we have placed the Child-s below the County-s to avoid ambiguity. We will here skip further discussion of the layout. More on this in the next section.

The important addition in Figures 14.5 and 14.6 is the navigation to superiors, indicated by the apostrophe (‘). The apostrophe may in pseudo English read as “of”: “A population has a Car which has an Identifier, has Colour, has Owner, “which” is a Person has Name-s, has Age, “of” County has Name, and (the Person) has Child-s “which” each is a Person has Name.”

Outlook

We have shown a way of formulating compound statements that comply with the elementary statements in the External terminology schema.

Each Contents schema defines a composed sentence structure. Each Contents schema is a template for one sentence. Each Contents population is such a compound sentence.

The schema and its population allow addition of commands for retrievals, insertion, deletion and modification. The Contents schema states what is permissible; the Contents population states operations on each individual data item.

The Conditions in the Contents schema do not refer to a data tree in the Contents schema. They refer to Conditions in the External terminology schema, and indicate navigation across the Condition symbols there in. This is the normal reading of Conditions within Conditions.

The Contents schema may be understood as a Condition on a node in the External terminology schema. The validation of this Condition takes place when the Contents schema is created.

The Contents population may be seen as a Condition on a node in the External terminology population, and is evaluated every time a search or other operation is activated.

From a given Contents schema, various Layout schemata may be generated automatically by use of defaults. There may not be any need for manual design of the layouts. See the subsequent section.

The examples shown in this section are somewhat simple, as the example Car (Identifier is globally unique. If the identifier is local, we will have to use the global distinguished name of this root entity, and create an appropriate layout for this. This layout is not a topic for this book.

A Contents schema is a full specification of a compound sentence structure. Its grammar is stated by navigation through the normalized statements in the External terminology schema.

A Contents population states an actual sentence. The sentence can act both as a query and as a statement of facts. The sentence is according to the grammar found in its Contents schema, and the sentence may be very large and complex, and may cover a large screen or many screens.

When designing Contents schemata, the designer has to foresee the implied layout on screens. All layout details may be stated through the Contents schema, and there is no need for a separate layout design or to instruct movement of data to or from various fields on the screen.

When data instances are inserted by human users at the screens, the Contents schema already knows the classes of the data. The data are by default moved through the External terminology layer where the appropriate constraints and derivations are executed.

A Contents population defines paths of data received or presented by an observer. The Contents schema defines the permissible paths.

15. Layout

How do we create a concrete layout of the statements at the user interface?

This section outlines the layout of screens in a form-filling human-computer interface, and indicates how the layout may be designed from a given Contents schema.

This section will not propose a language for defining Layouts, as this is the application domain of many existing languages and tools. The application developer will not need to see these tools.

We start with the Contents schema defined in Figure 14.5, which is repeated in Figure 15.1.

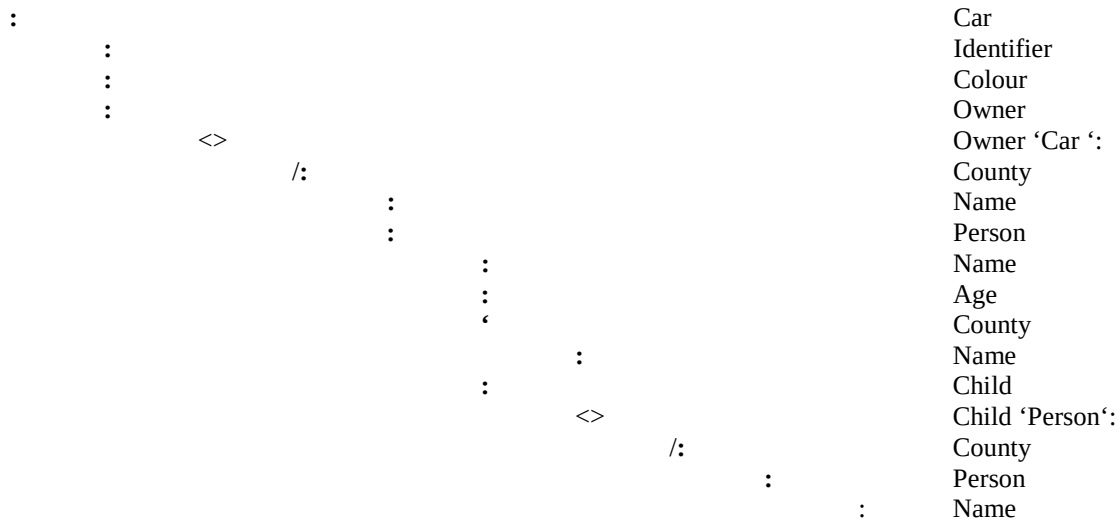


Figure 15.1. Example Contents schema with navigation to superiors

In typical implementations, we have experience with defining Contents schemata which are ten times as large and complex as of Figure 15.1.

As each schema and each population is a tree, there will be only one root node, eg. Car, in each screen. Note that if you would like to list all Car-s in a population, you would have to start with the identifier of the population and then list its Car-s. The Contents schema language supports no notion of unbound universal quantifiers. As there is only one Car in each screen, the Car data should not take up a whole column of the screen, but may be placed above the table shown in Figure 14.6. Also, it is not necessary to state the domain, Person, of each role, so the headings Owner/Person and Child/Person may be replaced by Owner and Child, respectively. It is inconvenient to use extra lines for the Child-s, so we may indicate that they are Child-s of the entity to the left of the County-s by adding a slash, /Child.

The resulting screen may look like Figure 15.2.

Car			
Identifier	Colour		
123	RED		
Owner		‘County	/Child
Name	Age	Name	Name
JOHN	52	XXX	BILL
			MARY

Figure 15.2. Example screen

Note that in Figure 15.2 we have created a tree of headings, and a tree of values. The presentation is not a flat table. The presentation principle is clear, and should be interpretable for an untrained eye. We will here not discuss operations for retrieval, search and write.

We have not repeated the class labels for every data instance, but in other layout examples, we will do so, eg. when using graphical icons to represent the entities.

In the next and last Figure, we have indicated references between entity instances by branching lines, and we see the branching from left to right.

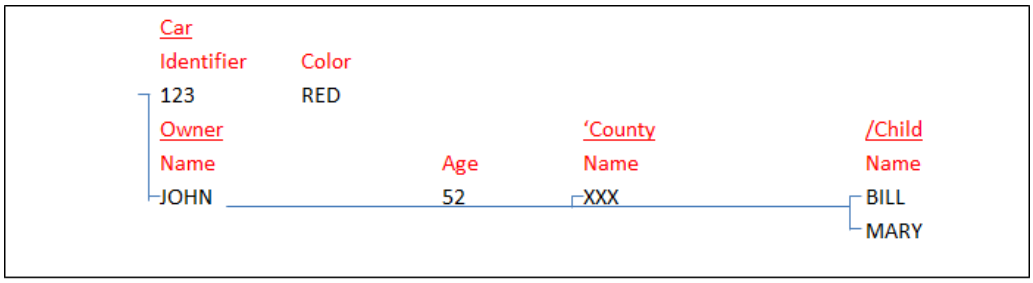


Figure 15.3. Graphical indication of references between instances

The reader should note that the example presentation allows the user at his human-computer interface to understand the structure of the data as defined in the External terminology schema and its population. The screens present data, not tasks. The presentation sequence and special characters (‘, /) indicate the functional dependencies. The special characters may be omitted, but they help the understanding.

Note that we in the alphanumeric presentation indicate functional dependencies to entity/role classes only, and not between attributes/fields. In a graphical presentation, lines would be used to indicate functional dependencies between entity/role instances, not between classes.

The Layout presentation in Figure 15.3 shows the functional dependences between data. If we had used flat tables only, as of the Relational model for databases, we would not see these functional dependences. The user would therefore not be able to provide a unique interpretation of the tables, ie. he would not be able to tell what data relate to which data. The presentation in Figure 15.3 allows only for one unique interpretation.

In this section, we have shown user interfaces for end users only. In Part 3 section 9, Language architecture, we have shown that the architecture can be nested. Hence, we can use similar presentations as of Figure 15.3 to create the user interfaces for developers. Not only so, the class labels appear in the user interface for end users; the end user may point and click at them, and this way access his data dictionary to read the class definitions being population data. The entire specification of the application will be available to the end user. See an

example specification in Annex G. We summarize that we do not start out with tabular presentations at the screens that need a programmer or analyst to interpret them. We start with the normalized grammar and glossary in the External terminology schema, add compound statements in Contents schemata, and create a presentation layout that is uniquely interpretable to the end user. We are creating the clarity that is often missing in other user interfaces. This clarity allows the user to read, use and understand both his population data and their specifications in the data dictionary.

Outlook

We have shown how to create statements that are uniquely interpretable to the end user. This is a very different approach from presenting task oriented screens, flat tables or natural language statements to the end user.

We have proposed a way to fold out class data from the Contents schema to become headings on a rectangular screen.

We have additionally proposed a way to fold out instance data from the Contents population to become values under the headings on a rectangular screen.

Both the schema data and the population data form trees. The root of the class tree together with the root of the instance tree are placed on the top of the screen. The roots use global distinguished names.

We have shown how to present local identifiers and reversed local identifiers of nodes other than the root.

The screens may allow for all kinds of queries and editing in a modeless dialogue.

The Layout population defines the concreate appearance of an observer's image before the analysis of the image. The Layout schema defines permissible variations within this image.

16. Denotations

What can we denote with our statements, and how do we do this?

This section introduces Ogden's triangle and three kinds of denotations, corresponding to the edges of the triangle. Hence, this section revisits and extends Part 1 section 3 on Nominalism, but now with the new tools, which we have learned to know in section 13 on the External terminology language. We will learn how to state the mappings between the External terminology layer and the Concept layer of the Data transformation architecture.

Figure 16.1 depicts Ogden's triangle. The corners may be denoted data, phenomena and concepts, respectively.

The arrows between the corners we interpret as follows:

- Data may denote Phenomena; this mapping we call data-phenomenon semantics
- Data may denote Concepts; this mapping we call data-concept semantics
- Concepts may denote Phenomena; this mapping we call concept-phenomena semantics

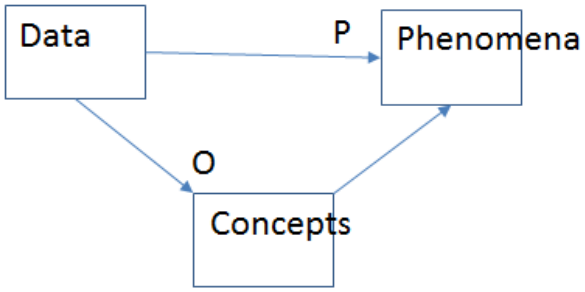


Figure 16.1. Ogden's triangle

There are other kinds of semantics, which we will not discuss.

Our three kinds of semantics would need three different predicates if we were using Predicate calculus to define these. We need only two of them to distinguish references from data to phenomena and from data to concepts. Our two references we will call:

- H - Phenomenon
- O - Concept

These are role labels as seen from any data node.

Note that data-phenomenon semantics is about the existence of Phenomena. Data-concept semantics is about the existence of Concepts.

Note that any Data node may denote a Phenomenon or Concept. If it does, then according to the requirement of having isomorphic mappings, the Data item's superior data node needs to denote the superior Phenomenon or Concept. This is illustrated in Figure 16.2.

```
:                                   System
    :                               ETS
        :                           O <> O ‘ETS ‘System (CS
        :                           Car
            :                       O <> O ‘Car ‘ETS ‘System (CS (Car
            :                       License number
            :                       Colour
                :                   O <> O ‘Colour ‘Car ‘ETS ‘System (CS (Car (Colour
            :                       Owner
                <>                  Owner ‘Car ‘ETS (Person
                :                   O <> O ‘Owner ‘Car ‘ETS ‘System (CS (Car (Owner
        :                           Person
            :                       O <> O ‘Person ‘ETS ‘System (CS (Person
            :                       Name

    :                               CS
        :                           Car
            :                       Colour
            :                       Owner
                <>                  Owner ‘Car ‘CS (Person
    :                               Person
```

Figure 16.2. Example denotations

In Figure 16.2, ETS is an External terminology schema, which has an O (Concept) reference to a Concept schema (CS). This needs to be so, because every data node, like Car and Person, having a denotation mapping to a concept, must have a superior node that maps to the superior Concept. Note that both ETS and CS are contained in the same System. Hence, System denotes nothing.

Note in the above Figure, that we have shown ETS and CS. In order to get the instantiation right, we in the references would have to replace these by colon-s.

Car Identifier and Person Name are just introduced as nametags to ease search, identification and references between data. They do not denote any concept. However, Colour denotes a concept.

It may be convenient to introduce Identifier and Name among the concepts, as well, because this may simplify mappings between several terminologies.

Note then, when we instantiate the schemata in Figure 16.2, it will be a Person instance that denotes a concept instance. Different Person instances denote different concept instances. The Person Name will denote nothing. This is different from classical logic, where it is assumed that it is a Name string that denotes. In our External terminology language, each Person inscription denotes a different individual Person concept.

We may have several External terminology schemata, one for each natural language to be supported. In ETS-English we may have Car. In ETS-Norwegian we may have Bil. Both may have a Concept reference (O) to a common concept. The only common thing about the two inscriptions is that they both are words. We mark this commonality in the Concept schema with a colon (:). The colon is a symbol for a pixel. A pixel is the only commonality between the two words. Any concept is a pixel.

This pixel may map to internal implementations in the Internal terminology schema (iTS) and the Internal schema (IS).

If every term in an ETS denotes a Concept, we may get the CS by removing the nametags in the ETS.

Classical logic revisited

In Predicate calculus we may declare Owns(Person 1, Car 1)=Owned-by(Car 1, Person 1).

In our External terminology language, we may formulate the two sides as

- Person 1 (Owned car <> Owned-car ‘Person 1 ‘System (Car 1, and
- Car 1 (Owner <> Owner ‘Car 1 ‘System (Person 1, respectively.

We see that the observations are made from two different sides, Person 1 and Car 1, of different phenomena, Owned car and Owner. Therefore, they are treated as different facts, and they cannot be declared being synonyms, like in classical logic.

Classical logic does not have any notion on where phenomena are observed from, what phenomena are observed, and in which sequence/direction they are observed. In the External terminology language we state that from Person 1 you observe an Owned-car, and from Owned-car you can refer to the Car 1.

The External terminology language allows for recording from where, of which, in which direction data are recorded or uttered.

The External terminology statement

Person (Name (1), Owned-car (1))

may in Predicate calculus be stated as

Person (1) ∧ Name (1) ∧ Owned-car (1) ∧ Person-Owned (1, 1)

We see that the Predicate calculus statements are split into fragments connected by logical connectives. Each fragment is a Predicate that maps to a truth-value. The truth values are shown in the following expression:

1 ∧ 1 ∧ 1 ∧ 1

Existence logic provides no such mapping.

We may extend Predicate calculus by set membership statements:

(1 ∈ Persons) ∧ (1 ∈ Names) ∧ (1 ∈ Owned-cars) ∧ ({1, {1,1}} ∈ Ownerships)

Note that here the individual person and the individual car are the same entity, as they have the same individual identifier, 1.

We might have extended Set theory with prefixes, getting Person.Name.1 and Owned-car.1, but then much of the rationale of using Set theory would disappear. If we additionally remove the logical connectives, we may come close to Existence logic.

Existence logic statements are branching observation paths having no connective and no mapping to truth-values. The left hand parentheses in Existence logic show containment, while in Predicate calculus the parentheses enclose parameters. In Predicate calculus, only variable names are used to link roles in different Predicates.

∀x ((x ∈ Persons) ∧ (x ∈ Names) ∧ ∀y ((y ∈ Owned-cars) ∧ ({x,{x,y}} ∈ Ownerships)))

Classical logic maps statements to truth values, and connects statements by logical connectives. As Existence logic does not use truth values; it has replaced the connectives by

lists. Lists in a schema give options, ie. a disjunction of what can be instantiated. A list in a population shows what actually exists, ie. a conjunction of actualities. The S reference points out a schema, and the P reference points out a population. A list can be both a population and a schema simultaneously. It is the usage of the list that tells how it is to be used - as a disjunction of possibilities or conjunction of actualities.

Classical logic states propositions, ie. what is stated. Classical logic has no notion of observations and phenomena. Existence logic identifies what is observed from where. Existence logic states observations, and nothing but observations. It has no need of stating that the statements are true.

In relational databases, we have the notion of a universal relation. All information is put into one large table. Likewise, we may do in Predicate calculus. We may create one universal predicate, which is true only when all the logical expressions of primitive predicates are true. The universal predicate will contain all the terms of the primitive predicates. Like in the universal relation case, the universal predicate will not show how the terms relate to each other. Existence logic shows the structure of terms.

We see that Set theory and Predicate calculus create fragmented statements and mappings. In Existence logic, we neither map to idealistic concepts nor to truth values. Also, we do not state axioms, as of Set theory. We replace the notion of axioms by stating templates – in schemata - that act as prototypes for the instances – in populations. The templates are contained in schemata.

As an afterthought, it is hard to understand how the classical logicians could have created their classical logic notions and languages. They seem to have been too much influenced by natural human languages. What would the aliens think, if they saw our classical logic languages? They might not want to communicate with us.

Discussions

The External terminology language allows for using local name spaces. The language uses noun logic, eg. Owned-car, rather than predicates, eg. Owns. Classical logic uses global name spaces and has an awkward and not meaningful word order.

The claim that Owns(Person 1, Car 1) and Owned-by(Car 1, Person 1) state the same fact, i.e. state the same notion, is a basic assumption in Begriffsschrift, Annex A [31], by Gotlob Frege, and has been followed by most logicians after him. We, however, consider the two expressions to state very different facts. Not only are the facts different, but their impact, i.e. the functions from them, are different. To Own something is a different experience from being Owned-by. We do not accept that an observation in one direction is causing an observation in the other direction in all cases.

Frege seems to consider the graph of Person-s, Car-s, and Ownership-s to be observed from the outside. Therefore, he has come to the notion of concepts. We are observing the graph from the inside, i.e. from each Person and each Car, and observe the phenomena Owned-car and Owner, respectively. We are phenomenologists, and the phenomena have the same structure as of the data, while the mapping may not be onto either way.

Frege is using a two-dimensional tree structure in his notation. This is used to state conditions and do reasoning on truth-values. His notation only superficially resembles Existence logic. Frege may in his notation, cover higher order logic, such as lambda calculus. Russell and Whitehead have in their Principia Mathematica, Annex A [157], developed a notation that resembles current Predicate calculus, and is a predecessor of it.

In this section, we have seen that both phenomena and concepts appear as data inside some observer. Concepts are similar to expressions in the External terminology language, where some concrete details are removed. These expressions, in both layers, are normalized elementary statements, different from the non-normalized compound statements of the Contents layer.

Our concepts are pixels, which may or may not contain nametags. This brings us back to Part 1 section 2 on Phenomenology and section 3 on Nominalism.

In Model Theory, the concepts are sets in a Universe of sets. Sets are made up of pure thought, and are not made of data. The Universe of sets is static and infinite, while data are dynamic and finite. The Universe of sets is a graph, where the nodes are sets, and edges are set memberships. The nodes are point like entities, with no characteristic, except their contained sets. In mathematics, all sets are built up from the empty set. This set is the only set that is not built up from other sets. Absolute realists or absolute conceptualists consider this Universe of sets to exist independently of anyone mentioning or describing it. This Universe of sets is considered to be separate from the physical world of phenomena. We may consider the Universe of sets to be a Universe of ideals, which deals with nodes, like points and infinities and the relationships between these.

The Universe of sets has a similar role to that of the Spiritual world, known from religion. The Spiritual world is thought to be teeming with life. It is dynamic, as spirits have behavior. However, all physical laws may be suspended, and spirits may not need to consume energy to operate. The spirits may be at any time and place simultaneously, and operate differently at each place. Hence, it is far beyond my comprehension to understand how spirits can exist or be identified. The Spiritual world is considered to exist separately from this physical world of phenomena. The spirit God is considered to be the first mover, by whom all physical phenomena are created.

The Universe of sets is the closest we can come to a mathematicians Heaven. The empty set has a similar role as of God, being the creator. All statements are interpreted against sets. The Universe of sets explains all the data. In religion, the Spiritual world explains the existence of the physical world, and of good and evil. In our new theory of language and phenomena, we have abandoned both the Universe of sets and the Spiritual world. We describe the physical world, and the description itself is inside this physical world.

Truth-value logic is a calculus on meta-statements about statements. Hence, a predicate maps from terms to truth-values, which we calculate on. The statements and their denoted sets have no behavior. In contrast to this, Existence logic is about the behavior of phenomena and of data. The creation or disappearance of some data has impact on other data.

Note that we distinguish three-dimensions of expressions, not to be confused with the tree-dimensional data tree. The three dimensions are:

- In-scription
- Pre-scription
- De-scription

All writings are inscriptions. When I write my name in a population, I write an inscription. This name may neither describe nor pre-scribe anything.

What is written in a schema is a prescription of what may be written in a population.

An inscription in a schema or population is a description if it contains a denotation mapping to a phenomenon or concept. In most applications, we do not manage descriptions.

If you are not describing something, you are not modelling, as to model means to describe something.

Software developers are often claiming that they do modelling, and that they are not doing specification or design. What are they modelling? Their enterprise, the UoD of their enterprise, or their implementations? Often, they do not know, and they do not write down the denotation mapping. Since the developers are only defining classes, while the end users insert the instances, I will claim that the end users are doing the modelling (in the informal sense). The developers define templates and constraints for the end users' modelling. Hence, the developers are prescribing what the end users can write. The developers are seldom doing modelling.

I understand that the conception of doing modelling affects the developers' designs. It may also lead them to believe that their task is just to describe, and not to design. Alternative designs have different impact, and put much more freedom and responsibility on the developer. Different developers may claim that they are doing modelling, but they are still creating different designs. The danger is that they are not understanding that they are designing. I want developers to stop using the term modelling. They are designers.

The logicians have a different notion of models in Model Theory. If the set structure is the model, the sets play the role of being concepts. Are the sets then modelling the phenomena? If so, why do they not spend a word on the mapping between sets and phenomena? And, why do they not write down the denotation mapping from statements (in predicate calculus) to the individual sets? Because, sets are not inscriptions? We have learned in this book that set structures do not resemble our notions of the physical universe. Sets are not appropriate models of the phenomena.

We want to get rid of the notion of alternative interpretations. Therefore, we want to fix the External language nametags to the phenomena and concepts. To do so, we have to turn the phenomena and concepts into becoming data themselves.

Classical logic does not support the distinction between prescriptions and inscriptions.

The Concept schema defines the permissible concepts or phenomena within the observer's Universe of Discourse. The Concept population defines the actual concept instances within this UoD. Note that we both have concept classes and concept instances. In most applications we do not need concepts, and do not need denotational semantics, as we can do with External terminology data only.

Outlook

We have indicated how to make denotation statements, and have discussed how Existence logic deviates from Classical logic.

In Part 5 section 20 you will find discussions on how to be misled by Classical logic and Set theory.

17. Execution

How do we execute statements in Existence logic?

In this section, we provide an informal presentation of execution according to the Data transformation architecture.

End user dialog

The end user at his screen calls a user function that corresponds to a Contents schema identifier.

The Contents schema forms a tree, with no reference between nodes in this tree. The Layout of an empty screen can be created by default and automatically, based on the contents of this schema tree. Some nodes may be tagged as to be suppressed. One empty field may be created for each attribute in the data tree.

The Layout shall fit into rectangular windows on the screen, while the Layout may be much larger than this window, both horizontally and vertically. Hence, paging and scrolling mechanisms must be implemented, which we will not discuss in this text.

The user may customize the generated form by pointing at and removing columns and headings; thus creating a subset to be presented. This reduction of the tree corresponds to projection in the Relational model.

The end user fills in the global distinguished identifier of the root entity of the Contents population. He may fill in additional constraints on other data, but we skip discussions on Query-By-Example features in this text. Basically, whatever values he fills into other fields will be taken as constraints on these fields. These constraints specify a selection.

The IT system returns a filled in screen with data from the Contents population. When doing this, the IT system must execute the mappings from the External terminology schema via the Distribution schema to the Physical schema. This mapping is basically made up of a list of record and field names in the database, where the sequence in the list states the implementation of the relative access path between two items in the Contents population. Also, the list may include program calls to routines which unpack fields and traverse alternative access paths.

The Contents population forms a tree, and normally, this population has no reference between nodes in the tree. During the read operation, no constraint is executed, except the Query-By-Example constraints in the read request.

The user is editing the filled in form. He may point at and delete data, modify and insert data. For insertion, he may need to expand the form with more empty lines.

The user press Enter, and sends the filled in and revised form to the IT system. Now all the constraints of the External terminology schema are executed. All Functions are called, as if they were database routines. Any non-compliance to the constraints are returned to the user for further editing. Normally, derived data are generated as a consequence of this editing. However, there is also an option to generate derived data during reading, as indicated with the program calls mentioned above.

After successful updates, the IT system makes a new read in the database of the entire Contents population to be shown at the screen.

The user may now do more editing, or he may, as in any empty or filled in screen, point at a field of an identifier, and get a new form, which is taking the corresponding entity as the root. If the field is empty, he gets only an empty form. This way, the user can click and jump around in the database, without having to remember names of forms and user functions. Also, the very first form and function may be default.

The end user is traversing his database pretty much like the Ant in Part 1 of this book. The Data transformation architecture defines the architecture of its camera. The updates correspond to the Ant's reasoning and annotations as it travels through its phenomena. Now the Ant is deriving and creating new data and new phenomena and bridges between these. The schemata define the Ant's capabilities.

Note that the user may be doing any reading and editing in any sequence, as long as he does not violate the constraints of the External terminology schema. The user may even access the data definitions in his External terminology schema for help, using the same dialog mechanisms as of his application. We observe that the proposed end user dialog is mode-less.

Implementation

During execution, both the Contents schema and the Contents population are kept in tables in an in-memory database. These tables are supported with information about Functions and program calls, and access paths to the database(s).

The execution of the data trees is by depth first, i.e. first out to the tip of each branch, conditions before the main branch, and then sequentially through all the branches. The execution must be limited to the size of the screen window, such that no extra data are retrieved or executed. Note that some Function calls, conditions and programs may traverse far outside what is seen within the actual screen. The IT system ensures that the data are consistent throughout the entire database (application) in real time for every full update from a Contents population. Hence, from an end user point of view at his screen, the system acts as one consistent whole.

As population data and schema data are being accessed simultaneously during execution, which translates between multiple layers, we will present the execution in some more detail. For update of one more new item in the Contents population, the system will create the corresponding item in the External terminology population by instantiation from the External terminology schema. If schema references or value types are used in the schema, the instantiation may need to be done recursively. Conditions or Functions associated with the External terminology items are executed.

If the Condition is not satisfied, then the conditioned item will be discarded, the Contents population will be returned to the user with a tag, e.g. giving a red colour to the fields with faults and a warning/explanation. If the Condition is satisfied, then all Instructions and derivation Functions of this item will be executed.

Note that if recursion is used for navigation in the External terminology schema, the number of levels up or down may be stated in the Contents schema, and will be stated in the Contents population. Hence, there should be no need for complex searches in the External terminology population. All execution is limited to what is strictly necessary. There is no endless search.

Note that the External terminology schema may contain Conditions, Instructions and Functions, which may not appear in the Contents schema. These Conditions, Instructions and Functions are copied into the External terminology population of this particular item. The execution may be done with the same read-control/write head as of the Contents population.

The External terminology population is mapped via the Distribution population to the Physical population. This mapping may activate other programs, which are being listed in the access paths within the Distribution schema.

So far, we have skipped discussions about Access control. The System management schema will have to include an advanced feature for stating what user groups from which terminal may access which Contents schema during which time period. It will also state which data area, i.e. horizontal partition of data, the user is allowed to operate on. Within the Contents schema, each item of the data tree is tagged with the permissible set of operations, eg. read, insert, delete or modify. In principle, you may do all these operations in the same screen on any data if they are not explicitly prohibited.

The Copy function

The arguments of a Function are referenced by the tips of its Condition. The control head will traverse everything it finds under the data items being referenced by the Condition, and the Function will take what it finds therein as argument values, and calculate the function values. These values are delivered by the control head under the data items being referenced by the Instruction.

If there are just Conditions and Instructions, and no Function symbol, then a pure copying takes place from the data item referenced by the Condition to the data item referenced by the Instruction.

```
:                                   System
        :                           Car
                :                   Name
                :                   Owner
                        :           <> : ‘Owner <> Owner ‘Car ‘System (Person (Child (Name
                                    >< : ‘Owner ‘Car (OwnerChild (Name
                        :           Name
                        <>          Owner ‘Car ‘System (Person
                :                   OwnerChild
                        :           Name
        :                           Person
                :                   Name
                :                   Child
                        :           Name
                        <>          Child ‘Person ‘System (Person
```

Figure 17.1. A Function which takes the value under Person (Child (Name as input and copy this to becoming a value under Car (OwnerChild (Name whenever Owner is updated

Line 8 states the navigation from Owner in line 4 to Person in line 11.

Note that the second Condition symbol in line 5 is a part of the navigation path via line 8, and is not another Condition.

Line 5 points at the argument Child (Name in the second last line. Whatever is contained in the referenced Name is taken as the argument value. Here we show the specification at the

class level, and not at the instance level, at which it will be executed. The attribute Name has a similar role as of an argument variable, and the contained structure will be an argument constant.

In line 6 we see the instruction part. It refers to the Car's OwnerChild (Name in line 10. The contents of Child (Name is copied to the contents of OwnerChild (Name. OwnerChild (Name is the function value variable, while its contents is the function value constant.

In order to avoid duplicate navigation, the instruction part can be placed inside the condition path.

```
:                               System
        :                       Car
                :               Name
                :               Owner
                        :       <> : 'Owner <> Owner 'Car
                                / 'System (Person (Child (Name
                                / >< (OwnerChild (Name
                        :       Name
                        <>      Owner 'Car 'System (Person
                :               OwnerChild
                        :       Name
        :                       Person
                :               Name
                :               Child
                        :       Name
                        <>      Child 'Person 'System (Person
```

Figure 17.2. A function with common navigation up to Car

The slashes in Figure 17.2 indicate that line 5 is branching into lines 6 and 7. All Instructions within the Condition of line 5 are affected by this Condition.

I have been uncertain on if a separate Copy function symbol should be used in Figures 17.1 and 17.2, but has so far concluded not to do so. I consider the Copy function to be a basic function of the language. Care must be taken if we do this, because it will result in copying via the Instruction part whenever its Condition refers to a non-terminal. If the Condition has no Instruction part, then there is no difficulty with referencing non-terminals. In this case, the structure referenced at the tip of the Condition may not need to be traversed.

More on execution

In Figure 13.17, we saw Conditions and Instructions referencing non-terminal nodes. The example is modified in Figure 17.3, where only class labels are used, and Car (Name and Person (Name are added, and they are both tagged with an Id Function..

```
:                               System
        :                       Car
                :               Name (Id<>)
                :               Owner
                        <>      Owner 'Car 'System (Person>< Person (Owned car
        :                       Car
        :                       Person
                :               Name (Id<>)
                :               Owned car
                        <>      Owned car 'Person 'System (Car >< Car (Owner
```

Figure 17.3. References to non-terminal nodes

Figure 17.3 shows what the user will see in his dictionary. This will be precompiled and appear as shown in Figure 17.4 in the in-memory database.

```
:                       System
      :                 Car
            :           Name (Id<>)
            :           Owner
                  :     Name
                  <>    Owner ‘Car ‘System (Person
                        / (Name
                        />< Person (Owned car (Name
      :                 Car
      :                 Person
            :           Name (Id<>)
            :           Owned car
                  :     Name
                  <>    Owned car ‘Person ‘System (Car
                        / (Name
                        / >< Car (Owner (Name
```

Figure 17.4. Precompiled references

If a Condition or Instruction refers to a non-terminal node, then the controlled head shall execute all the subordinate data items of that node. This means that when the first Condition refers to Person, then the control head shall execute whatever data it can find under Person. This may comprise his Name, the syntax of the Name, Age, value interval of Age etc. Also, the Person’s Owned car-s and their Conditions will be executed.

If we do not want to execute all of this, then the reference cannot be stated to Person only, but has to be stated to a subordinate item, eg. to Person (Name, which has to be added both to the main data tree and to the Condition. This fix of the Condition statement may be done through a pre-compilation.

When executing the branch under the non-terminal node, the control head is no longer controlled by the read head. The control head becomes an independent read head, and it may control another controlled head.

The impact on Figure 17.4 is that the Instruction >< Person (Owned car in the eighth line will be read by the read head, and control an Insertion of Owned car under Person by the control head.

Under Owned car B, the control head will find another Condition, in the third last line of Figure 17.4, which instructs a third head to execute under Car, and so on indefinitely.

We see that the procedure being applied on Figure 17.4, will result in an unbound number of read-write heads trying to insert Owned car and Owner. As they are already inserted in the first round, their later insertion attempts may have no effect, except resulting in a never ending execution loop. We may avoid this loop either by adding extra Conditions, or we may identify the loop through pre-compilation or at run-time, and stop it when the insertion has no effect.

Each control head will be discarded when it comes back to the node where it was created.

In Figure 17.4, we have seen how an Instruction (in line 8) may cause a creation of a new data item somewhere else (in the third last line). If we under this last data item find another Instruction that refers back to the first data item, we may also retrieve data from the other place, and bring them to the first place. Hence, we may do copying both ways.

In practical applications, it will be a waste of time to write the Conditions, Instructions and additional logic for two-way mutually dependent references. The roles, Owner and Owned-car, will be mapped directly to lists of database notions, like database sets, records and pointers, and all the logic will be managed by the database management system.

Outlook

This entire section outlines how execution may be carried out in a database application using the Data transformation architecture. We tell what is executed at the screen side, in the application and at the database side. We define and execute the logic in layers that are best suited for the purpose.

Talk about logic being implemented in the database or in the mapping to the database may be confusing to some people. They would like to see all the logic being formulated and implemented at one place, in the External terminology schema. However, in most cases, only an informal specification of the behavior is needed in the External terminology schema, and will be understood by most users.

The Condition statements are very informative in the External terminology schema, as they indicate what is referenced. However, they need not be executed in this layer.

Navigation inside the Condition defines the scope of the reference. Suppose we are defining an electrical Box, and we want to define an electrical coupling between Components. Is the scope of the coupling between two Components within the Box? Is the scope a coupling to another Component in another Box in the same House, the same County, the same Country or other. The navigation path within the Condition will tell what is referenced and how. Hence, the navigation within a Condition is a scoping mechanism.

In most graphical notations you may only draw a line to indicate a reference between two Components, but you are not able to indicate the scope of the navigation. The alphanumeric notation of Existence logic is explicit about the navigation.

Note that a Condition may take many inputs, and its Instruction may give many outputs. This is illustrated in the next Figure.

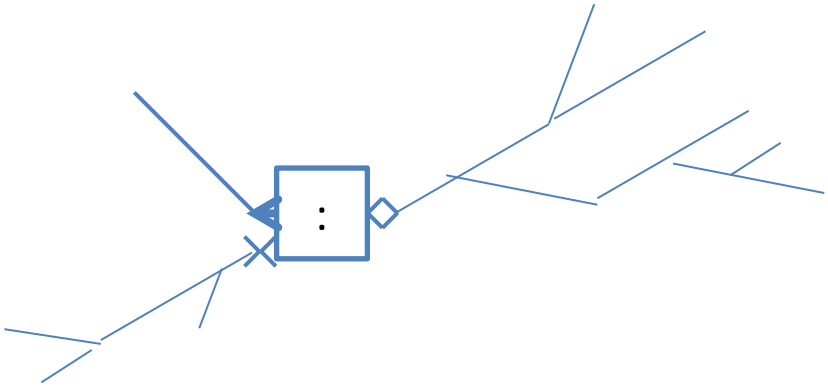


Figure 17.5 Branching of Conditions and Instructions

Both Conditions and Instructions may execute data subordinate to their referenced data items. Data referenced by the Condition may be copied to the place being referenced by the Instruction.

In the section on External terminology language, we have illustrated a processor with one read head and one control/write head.

In the section on the Data transformation architecture, we have depicted one processor per layer. Each processor addresses its schema and its population, and takes inputs and gives outputs. We could imagine that each processor puts its outputs into the population of the next layer. Hence, the input and output arrows may not be needed in the Figure.

At each layer, schema data will have to be instantiated into population data. Both schema processing and population processing may conveniently have their own pair of read-control/write heads. Therefore, we may have two processors per layer, each with their own pair of read-/control/write heads.

We may also separate translation of data in one direction from translation in the other direction. Hence, we may get four processors for each processor in the Data transformation architecture. There will be a need for proper interactions between these processors.

If we create a multi-tier implementation, each tier will handle two or more layers. Hence, we may need eight or more processors per tier.

In Annex G, you will find a simple example, and will see how this example is executed line by line.

Part 5

Post Lude

- What is the impact of using the proposed observer framework and language? -

18. Language considerations

Do we trust the proposals outlined in this book?

In Part 4 of this book, Language notions, we have introduced what could be the foundation of a language that will satisfy the Language architecture in Part 3, respond to the concerns made in Part 2, Classical logic, and how to support the worldview from Part 1, Perspective.

It is not the objective of this book to present a manual for the language. Also, there are many aspects of language design that need further discussions.

This section shall therefore not be understood as my views, but be considered as attempts to challenge my own views. The questions in this section shall not be understood as proposals. They indicate explorations outside my current knowledge and stance.

We may envy Bertrand Russell for the clarity and simplicity of his writings. However, Ludvig Wittgenstein found him to be shallow. Wittgenstein himself could question the ground, on which he was standing. Russell said that Wittgenstein had to "Understand or die". In this section, we will ask questions about our own understanding, about trivialities and fundamentals.

An issue about terminology: We are talking about the root of the data tree. However, when going towards the root, we are talking about Superior nodes. Closer to the root of a normal tree on the ground should be down, not Superior. From a naming perspective this is up. Maybe, our "data tree" is under the ground, and we should not be talking about a tree, but talk about a data root with branches, and up is towards the trunk?

If we are talking about a tree; a botanist may call its leaf elements for leaves or buds, not pixels. However, leaves and buds are complex entities. They are not the elementary means of creating anything. The pixel notion indicate non-divisible entities inside the observer; we are not out in the forest. Pixels are the primitive elements of information; they are not the complex creatures within the jungle. The trunk and branches of the data tree are made of pixels.

When standing on the first item in a list, we are talking about its Superior node. Likewise we do, when standing on an item that is not the first one. When looking down, we are distinguishing between Subordinate and Contained. Should we have separate names for up from the Contained node, different from up from the First node?

In a Condition, we may navigate either down to the Contained nodes or up to the Superior node. Therefore, this is a kind of vertical list, where the Condition is in the middle.

For the main data tree, we have a list to the Next from the first item in the list. We do not have a Previous of the first item. Could there be any use of this?

In a Condition, we may have other Conditions and other Instructions. However, in the main data tree, we can only have lists with elements that may Contain other elements. The elements that are not first cannot have a direct Superior. Could there be any use of this? Could we link lists with different Superiors?

The two previous paragraphs ask if we could use the three dimensions more extensively than what is proposed so far.

Could the negation (Exists Not) be defined as a Previous or Superior as mentioned above? Should negation be defined as a Condition that always fails?

We may have Conditions and Instructions within Conditions. Could these be linked back to the main data tree, creating loops, and what would be the impact?

Could we link Conditions and Instructions to Conditions that are executed separately? These questions seem to contradict the strict argumentation in this book to manage observed phenomena only.

Our Universe of Discourse is not closed; it will always be affected by entities that we cannot observe. How do we deal with such effects? By creating a still larger data tree? Or, creating a network of entities of which we can only observe a tree? Will such a network be relevant when creating the processors?

Connectives in classical logic are two-to-one mappings, eg. a conjunction maps from two inputs to one output. Fredkin automata use three inputs and three outputs. All inputs come out as outputs, but what comes out at which output channel is dependent on the logical operation. This way, the Fredkin automata unify data and energy, and they consume nothing. They also store their entire history as all calculations are reversible. However, the output will be very long, and you will spend all the energy on searching for the right answer among the outputs. Are Fredkin automata relevant when designing automata for executing Existence logic?

We have been talking about Conditions and Instructions as a third dimension, but why is this dimension so short? May data trees be linked, such that a condition branch in one tree become a main branch in another tree? May this be used to describe waves and movements?

Could the data tree be rotated, such that the dimensions could switch roles?

Our data tree is left-handed. A right-handed three would do, as well. What will happen if we combined left-handed and right-handed branches, such that they can execute each other? I think that one of them will take control and change the other to its own handiness.

The proposed language seems to be executable by one single von-Neuman machine. Could there be a room for a multitude of machines executing different parts of the data tree separately?

What is the program for a machine that can execute the data tree? Can this program be formulated by using the proposed Existence logic itself?

Conditions and Instructions are executing the main data tree as if the tree is DNA and they are RNAs. Can branches be decoupled and executed separately? In a way, yes. The main data tree may contain many conditions under each other, and they may be executed separately. We have allowed additional Conditions and Instructions to be contained in a Condition. They all operate on the main data tree. We may define Contents schemata and Contents populations to be such Conditions on the main data tree in the External terminology layer. Would it be possible, and would there be any use for having Conditions that controls the contents of its own Condition branch only?

In Existence logic, we use Conditions to state references. The Conditions may or may not refer to nametags. To refer to and search for nametags may be convenient, but is not necessary.

In most other formal languages, the names are essential. Where ever you write a particular name, it is assumed to refer to the same entity. The same entity is assumed to appear in every

syntactical context where this name appears. This is a notion of non-locality. The non-locality applies for constants, variables, predicates, functions, logical connectives, quantifiers, arithmetical operators etc.

In Existence logic, different entities appear at all the places using the same nametag, and they do not refer to each other. Even when stating references, using Conditions, they are not denoting the same entity. A role entity is always different from its domain entity.

In Existence logic, the Conditions contain explicit navigation threads that follow the branches of the main data tree. The navigation threads are most often very short and state the scope of the references. This way, Existence logic is based on locality, and does not allow for non-locality.

The non-locality in classical logic follows from the Type-token principle. The locality in Existence logic follows from the use of explicit navigation threads.

We have introduced particular symbols, like colon, comma, apostrophe, backslash, etc. Are these convenient, or should some changes be made?

I am happy with the two-dimensional notation with indentations, as it allows for removing most of the parentheses found in the programming language Lisp. A manual for the language will have to include a one-dimensional, two-dimensional and graphic notation, and variants of these. We see in the examples, that we have combined one- and two-dimensional notations.

The data tree has three dimensions plus time. This sounds like a three dimensional geometry plus time. If the data tree shall describe a three-dimensional space, I expect this data tree to be isomorphic to the space it is describing. Does this mean that the basic language necessarily needs to have the same dimensions as the space it is describing? Since it itself is inside this space, it cannot be more general than that space. Is then design of the basic language the same undertaking as of understanding the geometry of the physical universe? I think so.

Can a language inside the physical universe be more general than the universe it is inside? Can the physical universe be more general than the language used to describe it? I think not. Have we in our language proposal used all the features that the geometry permits? I think not. Are there language features that we are missing? I think so.

What is the impact of the previous paragraphs on the understanding of physics, quantum mechanics, string theory, cosmological models etc. Have physicists been using the Weierstrass notion of variables too freely? Yes, I think that it is too simplistic to use the common notions of classical logic and mathematics when trying to develop a Grand Unified Theory of everything. I think that we have to question the fundamentals of our formal languages. This book is a contribution to doing this.

In our Existence logic language, we are executing lists sequentially; this is more in line with the early thinking about variables by Francois Viete.

Even if free variables and multidimensional spaces have some similarity to geometry, there is a need to ask if they are different topics. I think they are, and I think that it is important to distinguish these.

Traditionally, rewriting grammars are used to analyze and generate natural languages. What are the drawbacks and benefits of using the attachment grammars? Note that we use suppression to remove some items in the Layout layer.

I will end this section with a comment on truth-value logic. Some natural languages frequently use the words Yes and No. These words mean Yes-it-is-True and No-it-is-False.

Some natural languages do without Yes and No. We have in Existence logic replaced questions about truth by questions about Existence of entities, and if they do not Exist, we discard the conditioned entity. We have seen that our proposed language works without truth-value logic.

Our language provides an Existence logic. This does not prohibit that we may add attributes to the data structure, where we tag the data with truth-values. This is permissible, but not required. Since we have not developed a truth-value logic, we cannot calculate truths. However, we can do very well without truth.

Truth-value logic provides meta-statements about the statements. It states that a statement is true or false. We believe that our Existence logic is more fundamental than truth-value logic. Existence logic provides the statements without truth or falsity.

We could add truth functions to our data structure. Each truth function would have a similar role as of predicates, map from values/terms to Truth values of Truth variables. In a separate branch of the data tree, we could define all the Truth variables. The logical operations are defined as functions/functors between these. We may calculate/derive the truth of a combined or derived statement. Why should we do this? The only purpose may be to explain the mapping to classical truth-value logic. However, truth calculations may not serve any practical need.

Finally, the Data transformation architecture may be defined using the Existence logic. This will be needed for defining the tool environment and do nesting of the architecture.

We may compare Existence logic with a Turing automaton. A Turing automaton operates on a one-dimensional tape, while the Existence logic operates on three-dimensional lists. The Turing automaton over-writes data depending on data elsewhere on the tape. Existence logic states conditions at one place in the data tree on data at other places in the data tree, reads instruction parts of the condition and carries them out elsewhere.

Turing automata operate on one data cell at a time, while Existence logic may copy whole branches of data. Like Turing automata, Existence logic inserts and delete data, and do not calculate Truth values as of classical logic. In a way, Existence logic is closer to the Turing automaton than classical logic. At the same time, Existence logic is closer to the applications' needs than classical logic.

Existence logic deals with finites only, but has no boundary in any direction. Existence logic will in principle not address data on a medium, as the data need no medium. However, the pixels of the data are made up of matter, and therefore you need matter to create the data.

The colon symbols in Existence logic form threads in three dimensions. There is no need of any alphabet, like 0 and 1, but the relative direction of the threads is significant. This thread language gives association to the Inca Quipu knot language, but this connection may be superficial. The conditions may be understood as knots; if so, the nametags are full of knots. The nametags typically appear at the ends of the threads. The nametags appear as leaves on the data trees. Some conditions appear closer to the root of the tree. Most threads are smooth, but may have a lot of branches.

The branches seem to form a fractal geometry. This seems to be a valid characterization of the phenomena at surface of the earth on which we live, but other parts of the physical universe – eg. vacuum - may be smoother.

The threads of phenomena may be observed out in nature. The branches of a plant express how the plant is creating edges between phenomena. It observes the sun, and moves its branches towards this most interesting phenomenon. The growth of the branches can be

compared to the movement of the Ant in Part 1 of this book. The structure of the tree tells how the tree has observed its phenomena.

Likewise, the root of the tree expresses its learning history about water in the soil. Threads of fungus express learning about electrical charges in the soil. Arms of a snow crystal tell what it has learned about temperature gradients in the atmosphere.

We note that any entity can be an observer. What it can observe depends both on its properties and its environment. The Data transformation architecture in Part 3 section 10 outlines the structure of a complex observer. When doing recursive observations – creating threads of observations -, it takes the roles of the entities which it observes.

The threads of inscriptions in three dimensions in Existence logic provide a very different worldview from the binary truth-value perspective of a one-dimensional tape. The use of context independent labels on the one-dimensional tape in classical logic seems to break up the threads, which we see when using the three dimensions in Existence logic. Without the three dimensions, we could not see the threads.

The treads apply both for phenomena and their descriptions. Both phenomena and their descriptions are inscriptions of data. Descriptions are made by taking copies of the threads of phenomena.

The denotation mapping between descriptions and phenomena is made up of data, and it forms a ladder between the descriptions and the phenomena. This is comparable to a DNA strand.

The colons may be placed in a three-dimensional grid. The colons will then form strings among blank grid elements. Hence, a two character alphabet, blank and :, is used.

If the colons are placed in a two-dimensional grid, special characters - like <> and >< - are needed to indicate the third dimension. In two dimensions, a condition branch is hanging down from its root. However, in three dimensions, this branch goes parallel to the main data tree in any of its directions. This is illustrated in Figure 13.13, which also illustrates the execution. We have illustrated the execution dimension in no other part of this book.

If the colons are placed in one dimension only, parentheses - (and) - are used to indicate the first dimension, and commas - , - are used to indicate the second dimension.

The two- and one-dimensional notations are used in this book.

The definition of Existence logic by using Existence logic is not included in this book. The definition may state nothing, but it can give a compact documentation of the language and provide a condensed understanding of it.

In Existence logic, we have as of yet not proposed any means to insert an element in a list. We do this by mapping to the Physical layer, or by deletion and copying.

A physicist will ask how we represent motion. We may change a reference between the moving body and observed positions on its path, or we may copy all constituents of the body to the next observed position on the path.

Outlook

This entire section is a discussion of the language notions that are introduced in this book. We recommend that you browse through the entire section.

19. More on Existence

Do words, phenomena, concepts and classes exist?

This section discusses existence of derived phenomena, derived data and existence of classes. The last subsection compares our language notions with features of other programming and specification languages.

Phenomena

The road authority may see a Vehicle, the registration authority may see a Car, and a person may see his Owned car. A relational database designer may claim that these are three roles of the same entity. However, what is this entity? Is it a Thing, which belongs to no specific class? Most database designers pick one of the roles, eg. Car, and elevate this role to become the domain, which the others are roles of.

We have learned that all three roles are phenomena, and none of them has priority over the other. The three phenomena Vehicle, Car and Owned car are seen from three different phenomena Road authority, Registration authority and Person, respectively.

There may be dependencies between the three phenomena Vehicle, Car and Owned car, such that some properties of one phenomenon are derived from properties of other phenomena, and these derivations have to be expressed. Also, one phenomenon Owned car may exist only if Car exists. If so, this condition must be stated, or we may generate a Car whenever an Owned car is inserted, which has to be stated, as well.

In an entity based approach, if we delete a relationship, we also delete its roles. If we delete an entity, we also delete all its relationships and roles. In Existence logic, the phenomena, ie. the roles, have their own existence. Hence, we may delete the Car, while the Owned car remains unchanged. we may also associate another Car to the existing Owned car. The Person keeps his Owned car while the Car is replaced.

We observe that the role based worldview has impact on how we define and manage our data. It impacts the amount of work to be done, and how to do it. This role based worldview we call phenomenology.

Data

In the previous sub-section, we have discussed existence of phenomena only. We will now discuss data about the phenomena. We have learned in this book that the data will be isomorphic to the phenomena. We will use the same labels on the data as of the phenomena. From each data node Car we will have a Denotation reference to a corresponding phenomenon node Car, etc.

The data node Car will contain an attribute Car registration number, which may or may not appear among the phenomena. Often it may be convenient to add a corresponding identifier among the phenomena, just to identify the individual Cars; this corresponds to fixing the plate with the identifier onto the physical Car. If so, we may also add a Denotation reference from the data node Car (Registration number to a corresponding phenomenon node Car (Identifier, etc. The data node Car (Registration number and the phenomenon node Car (Identifier may or may not contain the same or similar values.

If the data node Owned car has an identical identifier attribute to that of the data node Car, database designers use to define Owned car as a role of Car. We see that this role and domain notion is then made dependent on this particular design of the identifier of Owned car. This needs not be so in Existence logic.

The relational database structure tells about organization of data, and not about the phenomena. The relational database structure is not isomorphic to the structure of phenomena. Therefore, relational databases are not proper candidates for Concept schemata or External terminology schemata.

We have in this book learned that the structure of the External layer shall be isomorphic to the structure of the Concept layer and of the Phenomena. Despite our teaching, no practitioner is satisfying this requirement as of yet. There exists a great confusion on what is a data structure and what are the roles of various kinds of data structures. Many practitioners define data as they are stored in their database. Some claim that they create conceptual graphs; they write what they think, but they do not know what they should think. Additionally, they do not know how to ensure that the implementation complies with the specification, and have only informal mappings between the two. Very few use the data structure to define the terminology and grammar of the user interface. We provide the Data transformation architecture to define the role of each data structure.

Derived data

In the previous subsections, we have seen that phenomena may be derived, and data may be derived. The mappings between the two structures shall be isomorphic, but the mappings may not be onto either way.

Among the phenomena, you may observe wheels, windshields, doors, windows, etc., and derive a Car. The constituents, from which the Car is derived, may not have terms denoting them among the data.

Among the data, you may have derivations, and derived data, that may not correspond to anything among your phenomena. Among the phenomena, you may only observe Owned cars, but among the data, you create a Car for each. If this is so, you must have a means to create the Car (Identifier automatically for each Owned car. If you haven't, you may have to create the Car data manually with its identifier, before you create the Owned car data.

The above text explains how constraints and derivations may be different among the data in the External terminology layer from the phenomena, despite the isomorphism requirement.

Populations

In the literature, there are many texts discussing the distinction between plurals and singulars. See eg. the War of Universals. In this book, we have considered the class Person to be a prototype/template for each instance Person. During the theological battles in the middle ages, they did not have notions like class, variable or set. Some therefore considered Persons to be an ideal prototype of each imperfect Person.

In logic interpretations as of today, eg. in Interpreted Predicate Logic, the term Persons is taken as denoting the set of all Person-s.

In this book, we do not interpret natural language. We design External terminology data for our own use, i.e. we are language designers. We may then create a class Persons, which may relate to another class Person. The class Persons may have one instance for each County, and

Person may have an instance for each Inhabitant. We may relate instances of these two classes.

We treat both plurals and singulars as class labels, and we instantiate them the same way. A plural does not denote a set of individuals, as we allow the collection of individuals to change while the plural remain unchanged.

The old nominalists thought that plurals/universals were just flat words, and did not denote anything, like what the singulars did. We treat plurals and singulars on an equal footing.

Classes

We have in this book learned that the class Person may be instantiated into as many Person instances as we have a need of: Person, Person, Person, Person etc. This is achieved by allowing for use of significant duplicates. The class inscription Person plays a different role from each instance inscription Person, and each of these is different, even if they look the same.

Each Person inscription may contain an Identifier attribute, and its value may be different for each Person.

Each Person instance may denote a different phenomenon instance. For making it possible to state this reference from data to phenomena, there has to be an isomorphic mapping from the data class Person to the phenomenon class Person. Hence, we have a situation where a data instance cannot denote something if the corresponding data class does not denote the corresponding phenomenon class.

We see that the old nominalists were wrong when claiming that classes cannot denote. Their mistake is found in their conception of the real world, to be outside their body. We have learned in this book that phenomena are inside the observer, and phenomena are data with classes, instances, derivations and constraints, like any other data.

Outlook

In this section we have seen that

- Phenomena correspond to roles
- Roles may exist without entities and relationships
- Data may exist without them denoting phenomena
- Data should be defined as they appear to the end user
- Data may be derived from other data, and phenomena may be derived from other phenomena
- We are designing data; we are not taking natural language terms and expressions at face value
- If instance data denote phenomena, then their class data have to denote the corresponding class of phenomena, as well
- Both Platonists and Nominalists were wrong during the War of universals, as they did not have the right framework and notation to sort out the issues

In this book we have seen that the External terminology language resembles object oriented languages, but there are some differences, as well.

- Object oriented languages typically use global class names; we do not.
- Object oriented languages typically use reserved words for the class level only, and

may not have a notation for the instances. Therefore, they may not be able to use any instance as a class for still other instances.

- Object oriented languages use encapsulation; we do not.
- We have constraints and functions subordinate to the data; this is a similarity. However, we do not encapsulate the method, ie. the behavior definition.
- We move all aspects of implementation to other layers of the Data transformation architecture. We move all notions of visibility to the access control part of the System management layer and to identification of permissible or not-permissible operations in the Contents schema.
- We do not support the notion of subclasses and generalization as of object oriented languages. For example, we do not consider a Car to be a subclass of Vehicle. We consider the two classes to be two different phenomena classes of phenomenon instances being seen from different observers. Object oriented languages have typically provided a poor understanding of relations and roles.
- We have replaced generalization by the more flexible schema and value type notions. The class reference – from instances to classes - needs seldom to be stated, but applies everywhere. The schema reference may be used recursively within a schema, and it may be used among the instances.

The proposed language has much similarity to functional languages, like Lisp, but the details about name spaces and navigation are different. The Data transformation architecture is different, and the role of the language relative to this architecture and interpretation against the world of phenomena are different. Due to the similarity to Lisp, the proposed language has similarities to Lambda calculus. The language does not make any distinction between lower and higher order features, and does not apply truth-value logic.

Existence logic is not about the truth-value of propositions. Existence logic shows the existence of phenomena and their data, states requirements on these, and does derivation, such that phenomena and data behave.

20. Paradoxes

Are they paradoxes, or are they formulations in defunct languages?

I argue that the paradoxes are formulations in defunct languages.

In this section, we discuss three paradoxes. We do not solve the paradoxes; we refute the problem formulations in each of them.

The Liar's paradox has the greatest philosophical impact, and our presentation on this paradox gives a good understanding of how Existence logic deviates from Classical logic. Russell's paradox is easiest to refute. Zeno's paradox illustrates what exists, and what does not. I believe that these three examples will give the reader tools to address other paradoxes.

Finally, I have added a text on the Pythagorean theorem, as this has been used as the proof of the existence of infinities and continuity. I refute this claim. From this discussion, I infer that numbers denote the size of an area, and not the length of a line.

The Liar's paradox

The Liar's paradox reads "This sentence is false." Natural language is a good means to fool oneself. Predicate calculus is good for that, as well. We will start with discussing the term "This". We have already in Part 1 section 4 on Existence learned that God does not exist, the Universe does not exist, and I, i.e. the process that does the thinking, does not exist when I am thinking it. However, if I record my own thinking at a higher frequency than my own thinking, and then play the recording afterwards, I can see that I did exist when observed from now, but I cannot exist in real time.

What then about "This sentence"? It cannot exist when I am uttering it. But, it does exist when I observe it after it is being uttered or written. Is it then the Contents of the sentence which is False, or is it the Truth-value of the sentence which is labelled wrong? Classical logic does not distinguish these, because classical logic treats subjects as if they are objects.

If we state "That sentence is false", then there is no paradox, because now we refer to another sentence. "This sentence" does not exist when we are uttering it, because it is a part of the subject doing the utterance.

In Predicate calculus, the statement may read False=Truth-value(This statement). We could simplify the claim by writing False=This statement, but we have no means to refer from the term "This statement" to the expression containing it. Model theory only refers to sets and propositions, and does not refer to the statement itself or to other inscriptions.

Rather than analyzing the paradox in natural language or predicate calculus, we will reformulate the statement in our Existence logic in the External terminology layer. The Statement has two parts, its Contents and its Truth-value. The Truth-Value can take values True or False, and is a meta-statement about the Contents. This means that the Truth-value cannot be set before the Contents is set; it can only be calculated when the Contents has been given. The Contents is declared as

Contents (This statement (Truth-value (False

Figure 20.1 Contents of the Liar's statement

This means that the Contents is about the Truth-value of itself, ie. about this statement that contains the Contents. It is not about This statement in the Contents.

The Truth-vale of a statement is dependent on the existence of mappings to phenomena, and cannot be decided from the Contents of the statement. The statement is within an External terminology layer population (ETP) and the phenomena are in a Phenomenon layer population (PP):

```
:                                   System
    :                               ETP
        :                           This statement
            :                       Contents
                :                   This statement (Truth-value (False
                H<>                 H ‘Contents ‘This statement ‘ETP ‘System
                        /           (PP (This statement (Contents
                :                   Truth-value (True
    :                               PP
        :                           This statement
            :                       Contents
                :                   This statement (Truth-value (False
```

Figure 20.2 Truth value of the Contents of This statement

In the PP, the Contents is This statement (Truth-value (False. See last line. In the statement in the ETP, the Contents is the same. See the fourth and fifth line. There is a Phenomenon denotation (H<>) from the Contents in the ETP to the Contents in the PP. See sixth line with a continuation in the seventh line. As this denotation refers to a phenomenon that exists in the second last line, the Truth-value of the Contents in the ETP is set to True. See fifth last line. The Function that sets this Truth value is not depicted in the Figure.

Now we see that there are many instances of “This statement” in the above Figure. There are six such inscriptions. Additionally, there are three instances of “Truth-value”. In classical logic they are all treated as if they are from the same string and refer to one and the same entity. In Existence logic we treat them all as being different data, they may describe different entities, and in this formulation the paradox disappears.

The above example may be simplified, but this will not change the message. The Type-token principle has been creating the confusion in the Liar’s so called paradox.

We could in Existence logic have reformulated the use case, by deleting the PP and its subordinate structure. We may let the Contents contain a reference to itself. This is depicted in the next Figure.

```
:                                   System
    :                               ETP
        :                           This statement
            :                       Contents
                :                   This statement (Truth-value (False
                H<>                 H ‘Contents ‘This statement (Contents
                :                   Truth-value (True
```

Figure 20.3 Truth value of the Contents referring to itself

The Truth-value will still be True, because the H denotation refers to a phenomenon that exists and is isomorphic to itself. The Contents is This statement (Truth-value (False. Even if the Contents contains False and the Truth-value contains True, there is still no paradox, because False in the Contents has no bearing on True in the Truth-value.

Suppose the Contents is not describing itself, but describe its Truth-value. In order to get an isomorphic structure, we will introduce an extra level, PP, above the Truth-value. Now it is This statement that contains PP (Truth-value.

```
:                                   System
    :                               ETP
        :                           This statement
            :                       Contents
                :                   Truth-value (False
                H<>                 H 'Contents : (PP
            :                       PP
                :                   Truth-value
```

Figure 20.4 Truth value of the Contents of Truth-value (False

This formulation is more in line with the intention of the paradox. There is an H reference from the Contents to the PP (Truth-value. The Contents must be isomorphic to the PP (Truth-value. Hence, the Contents must be of the form Truth-value (True or Truth-value (False, if the Truth-value can have only these two values.

A descriptive statement can only be created after the phenomenon has been created, ie. after the Truth-value has come into existence, while the Truth-value of a statement can only be calculated after the statement, ie. its Contents, is written.

In the above example, we have left the PP (Truth-value blank before the Contents is created. In this example, the Contents is about the PP (Truth-value, which does not yet have a value. The PP (Truth-value is about the isomorphic mapping H<> to the PP (Truth-value phenomenon, which does not yet exist.

Before the value of the Truth-value is created, we will calculate the PP (Truth-value (False, because the PP (Truth-value did not exist before this calculation. Hence, it is False that the Contents describe a False phenomenon, as the False phenomenon does not exist. This is not a paradox as being claimed in the Liar's paradox. Note that in this reasoning we have applied a three-valued logic, True, False and Does-not-exist. Classical logic has no notion of Does-not-exist, because it is reasoning on static structures only.

If we claim Contents (Truth-value (True and the PP (Truth-value is still missing, then we will calculate PP (Truth-value (False, which means that the claim is False. It is False that the statement is True about its phenomena. There is still no paradox. We still apply three-value logic.

The claimer may now observe the PP (Truth-value again, and sees that now it has been set to False, and the H reference is to a PP (Truth-value, which does exist. Then he will have to change his claim to Contents (Truth-value (False. The Truth-value will be calculated to True. Now the Contents False is True.

The claimer may make a third observation, see that the PP (Truth-value is True, and changes his claim to Contents (Truth-value (True. A re-calculation of the Truth-value will give True. Hence, the Contents and the Truth-value have become aligned.

We will summarize our analysis in a table:

Original PP (Truth-value	Contents	Re-calculated PP (Truth-value
(empty)	False	True
(empty)	True	False
False	False	True
True	True	True

The re-calculated PP (Truth-value is the conclusion about the mapping between the statement and its phenomenon. The Original PP (Truth-value is on what the claimer observes before he creates the Contents of his statement.

We see that we do not get an infinite recursion of that it is true-that-it-is-false or that it is false-that-it-is-true as of classical logic. In three steps, we get it is true-that-it-is-true.

Classical logic assumes that the statements are static declarations about a static world, and that they are not the result of reclaims and recalculations. This is a third reason for the unfortunate interpretations of the Liar's paradox when using natural language and classical logic.

We recapture the three misconceptions of classical logic:

1. Subjects are treated as if they are objects; we claim that only objects exist
2. Similar inscriptions are taken to refer to the same string and entity; we consider them all to be different
3. Statements and the Universe of sets are considered to be static worlds in a flat name space; we use data and phenomena that are affecting each other dynamically in lists of lists, ie. we have dynamics in a non-flat worldview

We observe that it is quite foolish to talk about your own Truth-value, which may even not exist. Therefore, we may re-label the paradox as the Fool's paradox.

Russell's paradox

According to naive set theory, any definable collection is a set. Let *R* be the set of all sets that are not members of themselves. If *R* is not a member of itself, then its definition dictates that it must contain itself, and if it contains itself, then it contradicts its own definition as the set of all sets that are not members of themselves. This contradiction is Russell's paradox. Symbolically:

$$\text{Let } R = \{x \mid x \notin x\}, \text{ then } R \in R \iff R \notin R\text{“}$$

Note that this paradox is resolved by introducing the Axiom of foundation in Part 2 section 8. We will here undertake a discussion on its cause rather than use of an axiom to fix it.

This paradox has a relationship to the Kalam argument about the existence of God:

1. "Everything that has a beginning of its existence has a cause of its existence;
2. The universe has a beginning of its existence; Therefore:
3. The universe has a cause of its existence" Voila, God.

Both paradoxes try to reason outside the scope of their Universe. Without saying it, Kalam assumes that the Universe is an extensional set, which can be observed from the outside, just like in Russell's paradox. This is not surprising when we know that Cantor's Set theory was inspired by Aquinas' "proof" of the existence of God.

Aquinas called God for the Absolute. This term appears in Cantor's study of Absolute transfinities.

There is another notion than infinity in Set theory that has a relation to God. This is the empty set. All other sets are defined as sets of sets. They are created by the mathematician using his recursive functions. All sets are defined by their extension. The empty set is different. The empty set is created by none, as it has no extension. The empty set is the God in the Heaven of sets. The empty set must exist before the beginning of time; before anything is uttered. All other sets can be created from the empty set.

If the Universe were intensional, there may always be more to see outside the current Universe of Discourse, and there is no need for a God.

So what is wrong with Russell's and Kalam's reasoning? They assume that the Universe is an extensional set, and that they can use unbound quantifiers to talk about everything in the Universe, everything that they are not able to see, not able to find and not able to list.

Note that the Kalam argument does not lead to that there being only one thing, God, outside the Universe. Russell's paradox does not lead to that there being only one type/class notion outside the Universe of sets.

Finally, we observe that Russell writes about some set not being a member of itself, as this is not prohibited by the axioms of Set theory. If membership has some similarity to the physical world of phenomena, then it is an awkward notion that something can be a member of itself. In Existence logic, something similar is not possible: Nothing observes itself, and nothing can contain itself.

In Existence logic, we have abandoned the notion of sets. We start with you, the observer, your properties and your time, and you observe a finite list of related phenomena, their related phenomena etc. We do not start with ur-elements, like the empty set, and we cannot build up a Universe of sets recursively. We start at the root of the data tree, and the data tree is finite. Our Universe of Discourse is intensional, and not extensional. Existence logic does not allow us even to formulate Russell's paradox.

We refute Russel's paradox by refuting

1. The existence of sets
2. The notion of building sets bottom-up, by aggregating elements into sets
3. The notion of doing recursion infinitely
4. The use of unbound quantifiers
5. The notion of observing the Universe from the outside; the set of all sets is not observable, and hence does not exist
6. The notion of a static Universe, where you do not observe a before and after state of a creation (of the set of all sets)
7. The notion of stepping outside the Universe of sets, by creating a set of all sets
8. The notion of mapping statements to sets outside the physical universe
9. Using the statements and word order of Predicate calculus that do not comply to observation sequences; Predicate calculus consists of fragmented statements that do not observe observation sequences

Note that Existence logic has no notion of in-consistency. The External terminology schema states what is permitted in its population. Existence logic allows for having lists of phenomena without identifiers, and they may be over-written. Hence, consistency makes no sense.

In retrospect, it is hard to imagine that they could create Set theory within classical logic, and that the foundation of classical logic has not been revised. Classical logic seems not to be adapted to the real world phenomena.

Zeno's paradox

In this paradox of Achilles and the Tortoise, Achilles is in a footrace with the tortoise. Achilles allows the tortoise a head start of 100 meters, for example. If we suppose that each racer starts running at some constant speed (one very fast and one very slow), then after some finite time, Achilles will have run 100 meters, bringing him to the tortoise's starting point. During this time, the tortoise has run a much shorter distance, say 10 meters. It will then take Achilles some further time to run that distance, by which time the tortoise will have advanced farther; and then more time still to reach this third point, while the tortoise moves ahead. Thus, whenever Achilles reaches somewhere the tortoise has been, he still has farther to go. Therefore, because there are an infinite number of points that Achilles must reach where the tortoise has already been, he can never overtake the tortoise.

This paradox is typically resolved by using convergent infinite series. The problem formulation is accepted, and the problem is resolved.

In this book we have argued that infinities do not exist. Hence, we must reject the problem formulation. So what is wrong? Whenever Achilles has run a distance, you have to make an observation. This observation needs to take a finite amount of time and use a finite amount of energy. If you add this to the time Achilles needs to run, you will realize that infinite amounts of time and energy are needed for observing that Achilles is approaching the tortoise through the infinite number of steps. Achilles will never make it! He will never make it if we are following the procedure devised by Zeno. When the distances become short, the time needed for the observations become dominant. Hence, Achilles will have to stop, accelerate and decelerate, between each measurement. Therefore, he cannot keep a constant speed. He will become infinitely slow when running between two close measurement points.

Zeno has in his thought experiment assumed that Achilles and the tortoise can exist without being observed, and even do so infinitely many times. This leads to the absurdity, and current classical mathematics resolves the problem by introducing the notion of convergent infinite series in order to come back to something sensible.

We have learned in the section on Existence that for something to exist, it must be observed. Things do not exist when they are not observed. To observe is a physical process, which requires time and energy. This way, we do not solve Zeno's paradox. We refute its problem formulation.

The Pythagorean theorem

Already in Part 1 of this book, we objected to the existence of points without extensions. Therefore, we will have to object to the notion of a line without width, as well. Hence, in this subsection, we will explore how the Pythagorean theorem may affect our finistic worldview.

The Pytagorean theorem, $a^2 + b^2 = c^2$, where a and b denote the short sides, and c denotes the hypotenuse in a right angle triangle, has been used as the final proof of the existence of numbers, c, which have an infinite decimal expansion. This may not be called a paradox, but this continuous worldview seems to contradict the notion of a finite world.

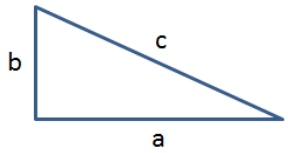


Figure 20.5 Right-angled triangle

Case 1: If we stay with the continuous worldview, then if a=b, then $c=2^{1/2}a=2^{1/2}b$.

We may assume that the triangle is put into a coordinate system with a along the positive x axes and b along the positive y axes, and the right angle is in the origo.

Case 2: If the space is a grid of square tiles, and we can only move up and down along the a and b axes, if a=b, then the shortest route between the end points (0,b) and (a,0) is a+b=2a. We take the shortest route to mean the distance, and the number of tile sides being passed to be the measure of the distance. When counting tile sides along the shortest route, we conclude that c=2a=2b.

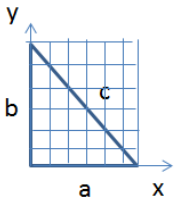


Figure 20.6 a=b

Case 3: Suppose we keep the length of c fixed, c, and rotate the triangle around origo 45 degrees against the clock. c will then become parallel to the x axes. The distance from origo along the y axes to the midpoint of c will be c/2. The shortest length from origo to the endpoints of c will then become a=b=2*c/2= c. When counting tile sides along the shortest route from origo (0,0) to (-c/2,c/2), we conclude that c=a=b.

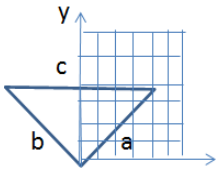


Figure 20.7 Rotated 45 degrees

In case 3 we have rotated the triangle from case 1 and 2 as if the space were still continuous. If the space were a grid of tiles, this rotation may not be possible. Also, it would be hard to define what a right angle means, except from a few special cases. We assume that the area of the triangle, A=(c*c/2)/2, will remain the same in the cases 1, 2 and 3.

So, why does counting of tiles along a broken line not give the right answer in cases 2 and 3?

We go back to case 1, and put tubes with the same and constant width along all three edges of the triangle. We put randomly spread particles with the same particle density in all three tubes. The particles should be small compared to the width of the tubes. We may take the

number of particles in each tube to be a measure of the length of the tube. We will then find $c=2^{1/2}a=2^{1/2}b$.

When counting the number of particles in the tubes, we are calculating the size of an area. We only estimate the length of the tubes by dividing the area on the size of the cross section of the tubes.

The number of particles in the tubes is a finite number, as there is a finite number of particles per meter of the tubes. Hence, if the particle density is constant within the resolution of our measurements, then the Pythagorean theorem appears to be valid. This constant density we may also achieve by putting a gas with constant constituents, pressure and temperature into the tubes. To obtain a measure of the length of the tubes, we may count the number of gas molecules in each tube.

May then the space between the gas molecules need to be continuous? It may not, but it must be dense below the resolution of the measurements.

Let us go back to the deduction of the Pythagorean theorem.

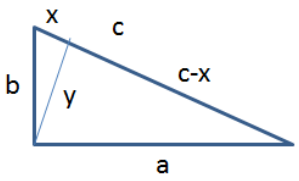


Figure 20.8 Calculation of area

The area, A, of the triangle is given by A=a*b/2, i.e. the half of a rectangle.

The length of the hypotenuse is calculated from A=c*y/2, where y is the height up to origo.

The height splits the triangle into two triangles, which each has the same form as the original triangle. See the angles.

Hence, we get x/y=a/b=y/(c-x). x is the distance from where y meets c to an end point of c. We have here two equations to calculate x and y. We put the resulting y into the formula for calculating A from c. We then have two formulas for A, and since we assume that both areas are equal, we get an equation for calculating c from a and b. This gives the Pythagorean theorem.

We note that we are not defining c by counting tile sides along a line. We get c by assuming that the area stay fixed during rotation. The area can be defined by calculating the number of tiles within the border lines of the triangle. The smaller the tiles are, the better will the resolution of the calculation be.

There are two assumptions behind these calculations:

- The area stay fixed, and is not dependent on way of calculation
- The angles/lines may be rotated, and stay fixed during rotation

In cases 2 and 3 above, we may keep the area constant, but the lengths of the lines have been distorted, and it is hard to define what an angle means, as we only move horizontally or vertically along the tiles when counting their numbers. In case 2, c=a+b, and in case 3, c=a=b.

The above second bullet point is about rotation of lines. Suppose we keep one end point of a line fixed, and rotate the other end point around the fixed point. What kind of operation is this?

During rotation, you integrate over the area covered by the line. You create a sector, which has the size of a triangle. The height is the length of the line, r. The base of the triangle is the length being traversed by the free endpoint. You may calculate the number of tiles within the sector, and estimate the length of the base line of the sector, l, from A=r*l/2.

The angle of the sector is defined in radians by u=l/r. Hence, A=r*r*u/2. u for a full circle is given by u=2*pi.

We conclude that rotation is about calculating areas.

We understand what a straight line or stiff stick is when they are aligned with the axes of the grid. But what are they during rotation? The tiles cannot move during the rotation. There must be some inhomogenities/particles spread over the tiles that move during the rotation.

We use the inhomogenities/particles to calculate lengths, and they may be used to calculate areas, as well. The fundamental notion is an area, not a line. However, the inhomogenities/particles must have a constant density during rotation. This allows for defining the notion of a distance.

The resolution of our measurements is at the size of a particle, or the distance between particles.

The diameter of an atom is typically $150*10^{-12}$ meters. This may be our resolution. The Planck length is $1{,}6*10^{-35}$ meters. This means that the physical space may or may not be a quadratic grid at the Planck level, while we only measure distances at the atomic level. At the Planck level, the notion of a distance may lose relevance and the notion of a grid may make no sense. Maybe we can define areas at the Planck level? Maybe the Heisenberg uncertainty principle, which involves Planck's law can be interpreted as a relation over an area.

The mistake made in the cases 2 and 3 seems to be that we've been trying to count the number of tiles along an ill-defined line. We should have counted the number of inhomogenities/particles. This may mean that the distance between the inhomogenities/particles may not be relevant. It is the number of inhomogenities/particles that counts.

Note though that in a quadratic grid, the number of inhomogenities/particles is the half of the number of connections between these. Hence, we may have counted the number of connections within the area, and obtained the same result as when counting tiles/nodes.

From the above discussion, we conclude that the Pythagorean theorem may remain in a finite world, if the spacing within the finite grid is far below the resolution of our measurements and calculations. We can therefore do without infinities.

The reader should note that the dispute about infinitesimals has been going on since before Newton and Leibniz. I go back to the old finitist position, which has a long time ago been overthrown by the continuous worldview.

We conclude that only areas exist, while lines and points only define the boundaries between areas. We may paint a line as a long and thin area.

The discussion herein has been about lines and areas. In this text we will not discuss volumes, energy and time. For an observer to see a pixel, he will need to be affected by an energy quantum. The wave function of light indicates that light is not following a line, but follows a volume.

Numbers

Having refuted the notions of infinities, we are ready to have a look at number theory. We have abandoned points and lines. How can numbers represent points on number lines?

We will study the following expressions:

A	B	C	D	E
$x^{1/2}$	$2^{1/2}$	$2^{1/2}{}_{\text{Stop condition}}$	$2^{1/2}{}_{\text{2 decimals}}$	1,41
1/x	1/3	$1/3\ {}_{\text{Stop condition}}$	$1/3\ {}_{\text{3 decimals}}$	0,333

We recognize the elements of column A are functions.

Column B shows functions, as well, but have a hidden variable, x, for the number of decimals or iterations in the expansion of the number. If we assume that $x \rightarrow \infty$, then the elements may denote a number with an infinite decimal expansion. However, we have abandoned infinities as well, so I will claim that column B does not show numbers.

Column C shows functions with Stop conditions, eg. number of decimals. The function value will be different for every Stop condition.

The function applications in column D will produce a specific number as their function value.

In column E we show actual numbers being produced by the function applications. However, the decimal form of a number is also a function application – using multiplications or divisions by ten - to position the number on a number line.

We observe that number theory is mainly a part of function theory.

When mapping numbers onto real world phenomena, we position the numbers on a line. However, we have learned that lines do not exist. Hence, the number line does not exist. A number will have to denote the size of an area. This area may form a rectangle. The number 1,41 has the resolution 0,01. Given this resolution, the number represents a rectangular area consisting of a series of 141 square tiles; each tile has the area 0.01*0,01.

This area is a unit area, ie. has the size 1. Its sides are measured as areas, as well. Each side has the size 1. And, the series of 141 tiles has an area of 141 units.

Outlook

We have shown that the paradoxes are formulations in defunct languages. Using the notions proposed in this book, the paradoxes are dissolved. Therefore, this entire section can be read as arguments for use of Existence logic.

Additionally, we have shown how to do away with infinities, points and lines. We classify most of number theory to belong to function theory. Numbers denote the size of areas, not lines.

21. Post Script

In this book, we have learned that observations create threads of phenomena inside an observer. These threads may be combined and played like a movie or dream. See on the Virtual Travels technology in Part 1 section 2. In nature, these threads are expressed as nerve cells, branches of trees, threads of fungus etc.

Descriptions are created by making copies of the threads of phenomena.

Both phenomena and their descriptions are made of data inscriptions inside the observer. The data may or may not contain nametags, which themselves may be created by threads. Nametags may be convenient to have, but are not needed. This is very different from traditional formal languages, which rely on unique names of constants, variables, predicates, functions and operators.

Denotations make up a ladder of data between the threads of descriptions and the threads of phenomena. In most cases, these denotation mappings are not needed, but we need to understand how data may or may not reflect phenomena.

In Existence logic we need only one symbol (:) to state any thread of data and operations on the data. We need no alphabet, proposition, operator or truth-value. Use of Existence logic results in very compact, comprehensive and efficient implementations of large database applications. See Annex G.

Conditions contain separate threads that navigate like creepers along the branches of the main data tree. This is very different from references by names in traditional languages. The conditions may result in deletions or copying. Copying is the main execution mechanism in Existence logic.

Classical logic is about propositions, ie. about what is said. Existence logic is about what phenomena are observed from where. It is about phenomena observing each other. This is a very different starting point from that of Frege, Russell and Wittgenstein.

Classical logic uses logical connectives. In Existence logic, lists in the schema state disjunctions of alternatives, lists in the population state conjunctions of actualities. One and the same list may serve both the role of being a schema and a population. If a schema reference is replaced by a type reference, the population node may only contain one instance. This means that the type reference expresses an exclusive or. This way, we replace all notions of Classical logic. To map between Classical logic and Existence logic is no trivial undertaking, as the constructions of the naming trees are different, and Existence logic is not about truth.

We have provided a new foundation for stating and interpreting statements and for designing IT architectures. We believe that this foundation is efficient in use.

We have used rather few references in the main part of the book, but the Bibliography shows that we have undertaken deep studies prior to the writing.

The book focuses on the foundation of formal languages. However, the book crosses the boundaries between several professions, such as philosophy of knowledge, philosophy of mathematics, philosophy of nature, foundation of computer science, logic, linguistics, cosmology, theology, engineering and design.

I am aware that some of the material is provocative, and I will be happy if I can provoke some logicians to argue with or against the book.

My core expertise is in computer science, but my approaches to methods, architecture, languages and designs are different from those of most others; so I will be happy to provoke some of my colleagues in computer science, as well.

Despite the possible controversies, my approaches to methods, architectures, languages and design are practical. I have myself developed large IT systems along these lines, and I believe that many more attempts should be made.

After having drafted the book, I have read papers by David Mermin and others on interpretation of quantum mechanics, using Bayesian statistics. Here, the statistics tell how much information the observer holds about the object at another site. Bayesian statistics is not about the stochastic behavior of the object itself. Bayesian statistics tell what the observer knows, and the uncertainty is in him, is not in the behavior of the object, and is not about a particle in a superposition. Bayes' and Shannon's views on information fit nicely with the phenomenological view on Existence presented in the Part 1 of this book, and which we have adhered to in the development of Existence logic. We abandon a realistic interpretation of the world. However, we adhere to rationality. This means that we are using Wittgenstein's picture theory and measurement theory for interpreting observations, and we are using algorithmic calculations for reasoning.

In this book, we started out with example everyday phenomena like persons and cars, and have reasoned on what features are needed for describing anything. These features will be needed for describing phenomena at the Plank level up to towards describing the Universe itself.

For representing the three dimensions of Existence logic at the Plank level, we will need three properties, like gravitation, charge and spin. These properties may be in the same particle or spread on different particles. Ref. Yuji Hasegawa's Quantum Cheshire cat experiment, which indicates that the particle properties and the spin of a neutron may take different paths. The Plank particles will have to be randomly oriented in vacuum, but may be coupled differently and more strongly inside elementary particles or within a force field. Physicists will have to identify these realizations, as this goes far beyond my competence. I have been speculating if the relative Planck distances could vary to create the gravity of galaxies and the large structure of the physical universe without having to introduce dark matter and dark energy. Could strong curvatures and rifts of space create matter, rather than the other way around? This would fit well with the geometric view of the universe and of languages being presented in this book.

I am aware that Planck particles are considered by many to be hypothetical only. However, uncertainty and duality of light indicate that photons are moving in a cloud of Planck particles.

I am aware of Eric Verlinde's theory on entropic gravity. I am also aware of the critique of his theory, and of its relation to the analysis of black holes. Verlinde projects information about the arrangement of bodies and their temperatures onto a holographic closed surface that surrounds the bodies. The entropy is a measure of the information density, ie. the number of bits per area unit, on this surface. He uses this to calculate the gravitational force at this point. The calculation is illustrated by trying to draw an elastic thread out through the surface, while the thread tries to contract behind the surface, due to vibrations coming from the increased temperature within the closed surface.

In this book, we define a closed surface around the observer, ie. around the subject, and not around the objects, as of Verlinde. The observations are made on the inside of the surface, and not on its outside. Maybe, Verlinde is assuming that the observer is the entire universe outside the surface. Then the outside of the surface is the inside of this God given perspective. I argue against use of such context independent perspectives.

In this book, the information is an organization of pixels, and are not called bits. The pixels are organized as threads inside the observer, and they are not evenly dispersed bits on the surface. There seems though to be several parallels between this book and Verlinde's theory, so they might take ideas from each other.

In this book, I have indicated a way of thinking on the universe from a language point of view. I do not claim to have all the answers. However, I hope that others can take inspiration from my approach. In many sections throughout the book, I have indicated areas that could benefit from further explorations. See, for example, Part 5 section 18 on Language considerations.

I have insisted on an isomorphic mapping between descriptions in the language and phenomena of the universe, as we cannot step outside our language. Hence, when having defined the language, we have defined what phenomena can be described by using this language. The definition of the language notions can only be made in an informal way, as there is no formal language that is more general than our universe, in which the language exists.

Outlook

I hope that you will read the entire Post Script.
As stated in Part 4 section 13: "We have not explored the use of Existence logic for analytic purposes, such as stating logical and arithmetic constraints, and finding optimal solutions." Existence logic is a design language, which gives a precise understanding of what exists and how it is constructed.

22. References

What is my background? As stated earlier, I have not been doing serious studies as an integrated part of writing this book. I have been doing studies of relevant topics earlier in my career. Most of the material is taken from my compendium on Specification Languages and their Environment version 3.0 January 1991 by Arve Meisingset at University Studies at Kjeller Norway. See the Annexes.

Here are internet links that I have been looking up during the drafting of this book:

[1] ***Philosophical realism***. http://en.wikipedia.org/wiki/Philosophical_realism

[2] ***Scientific realism***. http://en.wikipedia.org/wiki/Scientific_realism

[3] ***Realism***. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/realism/

[4] ***Structural realism***. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/structural-realism/

[5] ***Conceptualism***. http://en.wikipedia.org/wiki/Conceptualism

[6] ***Nominalism***. http://en.wikipedia.org/wiki/Nominalism

[7] ***Phenomenology*** (philosophy). http://en.wikipedia.org/wiki/Phenomenology_(philosophy)

[8] ***Type-token distinction***. http://en.wikipedia.org/wiki/Type%E2%80%93token_distinction

[9] ***Pierce's type-token distinction***. http://en.wikipedia.org/wiki/Pierce%27s_type-token_distinction

[10] ***Types and Tokens***. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/index.html

[11] ***Linguistics***. http://en.wikipedia.org/wiki/Linguistics

[12] ***Set theory***. http://en.wikipedia.org/wiki/Set_theory

[13] ***Naive set theory***. http://en.wikipedia.org/wiki/Naive_set_theory

[14] ***Zermelo–Fraenkel set theory***. http://en.wikipedia.org/wiki/Zermelo%E2%80%93Fraenkel_set_theory

[15] ***Definition:Zermelo-Fraenkel Axioms***. http://www.proofwiki.org/wiki/Definition:Zermelo-Fraenkel_Axioms

[16] ***Paradoxes of set theory***. http://en.wikipedia.org/wiki/Paradoxes_of_set_theory

[17] ***A history of set theory***. http://www-groups.dcs.st-and.ac.uk/history/HistTopics/Beginnings_of_set_theory.html

[18] ***Philosophy and the infinite***. http://www.logicmuseum.com/cantor/Phil-Infinity.htm

[19] ***Propositional calculus***. http://en.wikipedia.org/wiki/Propositional_logic

[20] ***First-order logic***. http://en.wikipedia.org/wiki/First-order_logic

[21] ***Second-order logic***. http://en.wikipedia.org/wiki/Second-order_logic

[22] ***Model theory***. http://en.wikipedia.org/wiki/Model_theory

[23] ***Model Theory***. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/model-theory/

[24] ***Semantics***. http://en.wikipedia.org/wiki/Semantics

[25] ***Lambda calculus***. http://en.wikipedia.org/wiki/Lambda_calculus

[26] ***Intensional logic***. http://en.wikipedia.org/wiki/Intensional_logic

[27] ***Montague grammar***. http://en.wikipedia.org/wiki/Montague_grammar

[28] ***Free logic***. http://en.wikipedia.org/wiki/Free_logic

[29] ***Free Logic***. Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/logic-free/

[30] ***Nicolas Bourbaki***. http://en.wikipedia.org/wiki/Nicolas_Bourbaki

[31] ***Automata theory***. http://en.wikipedia.org/wiki/Automata_theory

[32] ***Automata Theory***. http://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html

[33] ***Church–Turing thesis***. http://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

[34] ***Relational algebra***. http://en.wikipedia.org/wiki/Relational_algebra

[35] ***Multiset***. http://en.wikipedia.org/wiki/Multiset

[36] ***Euler diagram***. http://en.wikipedia.org/wiki/Euler_diagram

[37] ***Finite model theory***. http://en.wikipedia.org/wiki/Finite_model_theory

Here are ITU-T Recommendations of which I have been the editor and main contributor. The Recommendations are aligned with the ideas presented in this book.

[38] ***Z.351*** (03/93). ***Data Oriented Human-Machine Interface Specification Technique – Introduction***. http://www.itu.int/rec/T-REC-Z.351-199303-I

[39] ***Z.352*** (03/93). ***Data Oriented Human-Machine Interface Specification Technique – Scope, Approach and Reference Model.*** http://www.itu.int/rec/T-REC-Z.352-199303-I

[40] ***M.1401*** (05/2005). ***Formalization of interconnection designations among operators' telecommunication networks***. http://www.itu.int/rec/T-REC-M.1401-200505-S

[41] ***M.1402*** (05/2012). ***Formalization of data for service management.*** http://www.itu.int/rec/T-REC-M.1402-201205-I

[42] ***M.1403*** (08/2007). ***Formalization of generic orders.*** http://www.itu.int/rec/T-REC-M.1403-200708-I

[43] ***M.1404*** (08/2007). ***Formalization of orders for interconnections among operators' networks***. http://www.itu.int/rec/T-REC-M.1404-200708-I

[44] ***M.1405*** (08/2007). ***Formalization of orders for service management among operators.*** http://www.itu.int/rec/T-REC-M.1405-200708-I

[45] ***MSTR-NREG*** (09/2016). ***Telecom Network Registration.*** Technical report, http://www.itu.int/pub/T-TUT-TLCMGT-2016.

[46] ***Z.601*** (02/2007). ***Data architecture of one software system*****.** http://www.itu.int/rec/T-REC-Z.601-200702-I

A Bibliography is found in Annex A. Here you find many of my relevant publications, but not all of them. The Telenor archives may contain a lot of working papers that are not listed. Some of my publications are presented at international academic conferences. During the thirty years between writing the compendium and writing this book, I have read much of which I have not taken any notes.

Annexes

Annex A Bibliography

[1] ANSI DAFTG. ***Reference Model for DBMS Standardization***. Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group, May 1985. SIGMOD RECORD Vol. 15 No. 1 March 1986

[2] Amble T. ***Datamodellspråk – Et stadium i symbolske formalismer***. SINTEF STF 14 A86053. Technical University of Trondheim, NTH

[3] Amble T. ***Logic programming – An introduction to NTH-PROLOG***. Technical University of Trondheim, Norway, 1984

[4] Amble T. ***Logic Programming and Knowledge Engineering***. Addison-Wesley Publishing Company, Inc. 1987

[5] Asimov I. ***Asimov's Biographical Encyclopedia of Science and Technology***. Pan Reference Books. 1975

[6] Barker S F ***Realism as a Philosophy of Mathematics***. Prentice-Hall Inc. 1964

[7] Barwise J ed. ***Handbook of Mathematical Logic***. Introduction to first order logic, Fundamentals of model theory, Axioms of set theory, Aspects of constructive mathematics, The type free lambda calculus. North-Holland Publishing Company, 1977

[8] Bennet C H. ***Notes on the History of reversible Computation***. IBM Journal of Research and Development, Vol. 32, No. 1, January 1988

[9] Bennet C H, Landauer R. ***The Fundamental Physical Limits of Computation***. Scientific American, July 1985

[10] Bjørner D and Jones C B ed. ***The Vienna Development Method: The Meta-language***. Springer Verlag, 1978

[11] Blikle A, Tarski A. ***Naïve denotational semantics***. Information Processing 83 – IFIP congress series. Edited by R. E. Mason. North-Holland Publishing Company, 1983

[12] Brachman R J. ***What's in a concept: structural foundation of semantic networks***. Harvard University, Int. J. Man-Machine Studies, (1977) 9, pages 127-152

[13] Bræk R, Meisingset A. ***The ITU-T Languages in a Nutshell***. Telektronikk 4/2000 p4-19

[14] Brookshear J G. ***Formal Languages, Automata and Complexity***. The Benjamin/Cummings Publishing Company, Inc. 1989

[15] Carrol J, Long D. ***Theory of Finite Automata***. Prentice-Hall International, Inc. 1989

[16] Chang C C, Keisler H J. ***Model Theory***. North-Holland Publishing Company, 1990

[17] Chen P. ***The Entity-relationship Model – Towards a unified view of data***. ACM Transactions on database systems, Vol. 1, pages 9-38, March 1976

[18] Childs D L. ***Extended Set Theory. A general model for very large, distributed, backend information systems***. Proceedings VLDB, 1977

[19] Church A. ***The Calculi of Lambda-conversion***. Annals of Mathematics studies Number 6. Princeton University Press, 1941

[20] Clifford J, Warren D S. ***Formal Semantics for Time in Databases***. ACM Transactions on Database Systems, June 1983

[21] CODASYL. ***An information algebra***. Phase 1 report – Language Structure Group of the CODASYL Development Committee. Communications of the ACM, April 1962

[22] Codd E F. ***A relational model for large shared data bases***. CACM (13, 6), pages 377-387, 1970

[23] Collette J-P. ***Histoire des Mathematique***. Editions du Renouveau Pédagogique, Inc. Canada 1973, 1979

[24] Dahl O-J. ***Can Program Proving be made Practical?*** No 33 1978, ISBN 82-90230-26

[25] Dahl O-J. ***Logic Program Programming and Specifications***. No 84 1984, ISBN 82-90230-83-4

[26] Dahl O-J, Myrhaug B, Nygaard K. ***Simula information Common base language***. Norwegian Computing Centre, Publication No. S-22, October 1970

[27] Davis P J, Hersh R. ***The mathematical experience***. Penguin books, 1980

[28] ***Encyclopedia Britannica.*** Applied logic, Formal logic, History of logic, Philosophy of logic, Set theory, Analytic and linguistic philosophy, Epistemology, History of Western Philosophy, Philosophy of science, The branches of knowledge, Linguistics, Automata theory, etc. Encyclopædia Britannica, Inc

[29] Fenstad J E, Normann D. ***Algorithms and Logic***. Institute of Mathematics. University of Oslo, 1984.

[30] Føllesdal D. ***Semantics***. University of Oslo. Detailed reference missing.

[31] Frege G. ***Begriffsschrift***. Georg Olms Verlagsbuchhandlung. Hildesheim 1879

[32] Galitz W O. ***Handbook of Screen Format Design***. Second edition (1981, 1985), North Holland

[33] Girard J-Y. ***Linear Logic***. Theoretical Computer Science, 50, 1987

[34] Gireault C and Reising W ed. ***Application and Theory of Petri Nets***. Springer Verlag, 1982

[35] Goody J ed. ***Litteracy in Traditional Societies***. Cambridge at the University Press, 1968.

[36] Grietheusen van J J ed. ***Concepts and Terminology for the Conceptual Schema and the Information Base***. ISO TC97 SC5 WG3 N695. ANSI 1982.

[37] Grietheusen van J J and King M H ed. ***Assessment guidelines for Conceptual Schema Language Proposals***. ISO TC97 SC5-3 N.991, ANSI August 1985

[38] Grisman R. ***Computational linguistics***. Cambridge University Press,1986

[39] Gundersen O. ***Språket som teori og som empiri – m.m***. No. 3, Årg. 6. Institutt for filosofi. Universitetet i Oslo

[40] Guttag J V, Horning J J. ***The Algebraic Specification of Abstract Data Types***. Acta Informatica, 10.27-52, 1978

[41] Henkin L. ***Formal Systems and Models of Formal Systems***. Detailed reference missing.

[42] Holland J H et al. ***Induction***. The MIT Press, 1987

[43] Huges R I G. ***Quantum Logic***. Scientific American, October 1981

[44] ISO. ***Several Working documents on Naming and Addressing and on Abstract data types***. ISO TC 97 / SC21 / WG1

[45] ISO 7498. ***Information processing system – Open Systems Interconnection – Basic Reference Model***. ISO International Standard 7498, first edition – 1984 – 10 15

[46] ISO DP8807. ***LOTOS – A formal description technique based on the temporal ordering of observational behavior***. ISO TC97 / SC21 / WG1 N423, DP8807, Information Processing Systems – Open Systems Interconnection - 1985

[47] ITU. ***Directory Convergence Documents***. ISO Draft Proposal 9594, CCITT X.500 series, February 1988. International Telecommunication Union

[48] ITU. ***Document Architecture, Transfer and Manipulation***. CCITT Draft Recommendation T.400-T.500 series, February 1988. International Telecommunication Union

[49] ITU-T Rec. Z.100. ***SDL – Specification and Description Language***. International Telecommunication Union

[50] ITU-T Rec. Z.301-Z.341. ***CCITT Red book Volume VI – Fascicle VI.13 Man-Machine Language (MML)***. International Telecommunication Union

[51] Jardine D A ed. ***The ANSI / SPARC DBMS Model***. North-Holland Publishing Company, 1977

[52] Jeffery R. ***Formal Logic: Its Scope and Limits***. McGraw-Hill Book Company, 1967, 1981

[53] Johansen H. ***Automater, Petri-nett og elementære regneskjema***. TF report no 82/86. Telenor Research

[54] Kangassalo H ed. ***Fourth Scandinavian Seminar on information modelling and database management***, Names and denotations in conceptual schemas, Conceptual schema abstraction mechanisms, Multimedia documents in office systems. University of Tampere, 1985

[55] Kent W. ***Consequences of Assuming a Universal Relation***. ACM Transactions on Database Systems, December 1981

[56] Kloster G V, Cochran D R, Philips M. ***Techniques and Methods for Defining Man-Machine Interface Requirements.*** Computer Technology Associates

[57] Koestler A. ***The Sleepwalkers***. The Penguin Group, 1959 etc.

[58] Lee R M. ***A Denotational Semantics for Administrative Databases***. Database Semantics (DS-1). Elsevier Science Publishers B. V. (North-Holland) IFIP 1986

[59] Levey A. ***Basic Set Theory***. Springer-Verlag, 1979

[60] Manna Z, Waldinger R. ***The Logical Basis for Computer Programming***, Volume 1, deductive reasoning. Addison-Wesley Publishing Company, Inc. 1985

[61] Martin J. ***Design of Man-Computer Dialogue***. Prentice-Hall, 1973

[62] Meisingset A. ***A data flow approach to interoperability***. Telektronikk, 89(2/3), 52-59, 1993

[63] Meisingset A. ***A Methodology for the Classification of Human-Machine Interface Constructs***. TF report no 23/87, Telenor Research

[64] Meisingset A. ***A Minimal Language for Everything***. TF report no 27/89. Telenor Research. NIK'89, Stavanger. ICCI, Toronto, Canada, 1989.

[65] Meisingset A. ***A status report on system development methods***. TF lecture note no 26/91, Telenor Research. NOCUS, Oslo 1991.

[66] Meisingset A. ***Alternative Positions to current Mathematical Philosophy***. TF report no 28/87, Telenor Research

[67] Meisingset A. ***Automation approach to software quality***. Telektronikk 1.99, p17-22, 1999

[68] Meisingset A. ***CCITT data oriented human-machine interface specification technique***. TF lecture note no 24/92, Telenor Research. SETSS, Florence, Italy, 1992.

[69] Meisingset A. ***Data design for access control administration***. Telektronikk, 89(2/3), 121-128, 1993

[70] Meisingset A. ***Datamodellering – en kort introduksjon***. DND, Oslo, 1978

[71] Meisingset A. ***Datamodellering – et brukerorientert systemunderlag***. Systemutvikling, Oslo, 1980

[72] Meisingset A. ***Datamodellering – fremtidens programmeringsspråk?*** NordData, Bergen, 1979. Programmering, Oslo, 1979

[73] Meisingset A. ***Datamodellering – hva er feil med eksisterende metoder***? NordData, Stockholm, Sweeden, 1978

[74] Meisingset A. ***Datamodels in the context of the 3 schema architecture***. TF report no 25/89, Telenor Research. SIBUG, Bergen, 1989.

[75] Meisingset A. ***Datamodellspråk – praktisk anvendelse av Enhetsmatematikk***. TF report no 21/87, Telenor Research

[76] Meisingset A. ***Datastrukturalgebra***. Telenor, 1978

[77] Meisingset A. ***DATRAN – guide for technical management***. TF report no 31/87, Telenor Research

[78] Meisingset A. ***DATRAN – brukerdokumentasjon***. Telenor, 1984

[79] Meisingset A. ***DATRAN i et kunnskapsteknologisk perspektiv***. TF report no 30/87, Telenor Research

[80] Meisingset A. ***DATRAN – sluttbrukerdokumentasjon***. TF report no 32/87, Telenor Research

[81] Meisingset A. ***DATRAN og DIMAN***. DND, Oslo, 1987

[82] Meisingset A. ***Development of Large IT Solutions***. A003, UTAR Kampar Campus, Perak, Malaysia. 11 September 2012

[83] Meisingset A. ***Difficulties with Mapping OMT Specifications into GDMO***. ECOOP Workshops 1997: 12-16, Jyvaskyla, Finland, 9-13 June 1997

[84] Meisingset A. ***Diskusjon av prinsipper for sammenkopling av EDB-systemer i Televerket***. TF report no 54/89, Telenor Research

[85] Meisingset A. ***Draft CCITT Human-Machine Interface Specification Technique***. TF report no 1/91, Telenor Research

[86] Meisingset A. ***Enhetsmatematikk – en kort orientering***. Telenor, 1985

[87] Meisingset A. ***Enhetsmatematikk – en kort orientering om bakgrunnen for denne***. TF report no 13/86, Telenor Research

[88] Meisingset A. ***Entity migration and its consequences for the choice of fundamental mathematical constructs***. Second Scandinavian Seminar on information modelling and database management p221-244, Kangassalo H ed. University of Tampere, Finland, 1983

[89] Meisingset A. ***Et ekspertsystem som en triviell databaseanvendelse***. TF report no 09/87, Telenor Research. DND, Oslo, 1987

[90] Meisingset A. ***Formal Description of the Universe – a popular summary***. Institute for Informatics (IFI) doctor colloquium at the University in Oslo (UiO), 2014

[91] Meisingset A. ***Graphic GDMO***. Telektronikk 2.97 p94-96, 1997.

[92] Meisingset A. ***Hiding Middleware within a development environment***. TF note no 50/98. Telenor Research. EURESCOM DOT workshop, Heidelberg, Germany, 1998

[93] Meisingset A. ***Human-machine interface design for large systems***. Telektronikk, 89(2/3), 11-20, 1993.

[94] Meisingset A. ***Human-Machine Interface Specification Technique***. TF report no 17/90, Telenor Research

[95] Meisingset A. ***Information Systems Architecture***. Second Scandinavian Seminar on information modelling and database management p115-148, Kangassalo H ed. University of Tampere, Finland, 1983

[96] Meisingset A. ***Information systems planning in a competing telecommunication environment***. TF note no 34/98, Telenor Research. ZNIIS, Moscow, Russia, 1998.

[97] Meisingset A. ***Innføring i enkel enhetsmatematikk***. Telenor, 1985

[98] Meisingset A. ***Introduction to Information Systems Architecture***. Information Systems Architecture. Telektronikk 1.98 p3-11. Volume 94 No. 1 – 1998 ISSN 0085-7130

[99] Meisingset A. ***Interoperability Reference Model***. TF note no 95/95, Telenor Research. ECOOP workshop, Aarhus, Denmark, 1995

[100] Meisingset A. ***Middleware***. ITU workshop on software for telecommunication. Moscow, 2001. ITU-T and ITU-D joint workshop, Bangalore India, 2001

[101] Meisingset A. ***Objectorientering i DATRAN***. TF note no 10/90, Telenor Research

[102] Meisingset A. ***Omdefinering og omregistrering av nettdata***. TF note no 54/99, Telenor Research

[103] Meisingset A. ***Perspectives on the CCITT data oriented human-machine interface specification technique***. TF lecture note no 10/91. Telenor Research. SDL Forum, Glasgow Scotland, 1991.

[104] Meisingset A. ***Problemer med mengdelære***. Telenor, 1980

[105] Meisingset A. ***Retningslinjer for Menneske-Maskin-Kommunikasjon***. TF report no 20/87, Telenor Research

[106] Meisingset A ed. ***Report on HCI standardization***. http://www.itu.int/itudoc/itu-t/com10. ITU-T, Geneva, Switzerland, 1999

[107] Meisingset A ed. ***Report on Middleware standardization***. http://www.itu.int/itudoc/itu-t/com10. ITU-T, Geneva, Switzerland, 1999

[108] Meisingset A. ***Sakshåndtering – elektronisk post for komplekse data***. TF report no 22/87, Telenor Research

[109] Meisingset A. ***Software dependence on company organization***. TF note no 22/2000, Telenor Research

[110] Meisingset A. ***Specification Languages and their Environment.*** Compendium version 3.0, January 1991. University Studies at Kjeller, Norway.

[111] Meisingset A. ***Strategic Software Standardization.*** TF note no 74/2000, Telenor Research

[112] Meisingset A. ***System Independent Human-Machine Interface***. TF report no 24/87, Telenor Research

[113] Meisingset A. ***Systems Planning Using ODP***. EDOCW 2010: 357-360, Vitoria, Brazil, 25-29 Oct. 2010

[114] Meisingset A. ***Systemarkitekturer for nytt nettinformasjonssystem***. TF note no 38/99, Telenor Research

[115] Meisingset A. ***Systemteorier i filosofi***. Telenor, 1981

[116] Meisingset A. ***Systemutvikling – flerløps systemutviklingsmodell og arbeidsmetodikk***. TF report no 29/87, Telenor Research

[117] Meisingset A. ***The draft CCITT formalism for specifying Human-Machine Interfaces.*** Telektronikk, 89(2/3), 60-66, 1993

[118] Meisingset A. ***The Three Schema Architecture – a practical implementation***. Institute for Informatics (IFI) doctor colloquium at the University in Oslo (UiO), 2014

[119] Meisingset A. ***The Urd Systems Planning Method***. Information Systems Architecture. Telektronikk 1.98 p12-21.

[120] Meisingset A. ***Theoretical Foundation of Conceptual Modelling***. Third Scandinavian Seminar on information modelling and database management p7-10, Kangassalo H ed. University of Tampere, Finland, 1984

[121] Meisingset A. ***Theoretical Foundation of Modelling Languages***. TF report no 54/86, Telenor Research

[122] Meisingset A. ***Theory of science***. ITU-T workshop on philosophy and applicability of formal languages. http://slideplayer.com/slide/2914361/. Geneva, Switzerland, 2001

[123] Meisingset A. ***Three dimensions of formal languages***. ITU-T workshop on philosophy and applicability of formal languages. http://slideplayer.com/slide/2914361/. Geneva, Switzerland, 2001

[124] Meisingset A. ***Three Perspectives on Information Systems***. Information Systems Architecture. Telektronikk 1.98 p32-38.

[125] Meisingset A. ***Unity mathematics – a proposal for a general specification language***. TF report no 19/87, Telenor Research

[126] Meisingset A. ***Utredninger om nytt nettinformasjonssystem.***. TF note no 39/2000, Telenor Research

[127] Meisingset A. ***What Language Features for which Aspect of a Reference Model***. TF note no 92/95, Telenor Research. ECOOP workshop. Aarhus, Denmark, 1995

[128] Microsoft. ***Microsoft Pascal***. Reference manual, (1981-1985)

[129] Nijssen G M. ***A Conceptual Framework for Information Analysis***. Control Data and University of Brussels. October 1978

[130] Nijssen G M. ***A Gross Architecture for the Next Generation Database Management***. Modelling in Data Base Management Systems. North-Holland Publishing Company, 1976

[131] Nilsson B. ***On Models and Mappings in a Database Environment***. Statistiska sentralbyrån, Stockholm, Sweden, 1980

[132] NTH. ***The NU Algol Programming System for UNIVAC 1107/1108 Programmers guide and reference manual***. Computing Centre NTH Trondheim Norway, Tapir December 1969

[133] O'Donnell M J. ***A Critique of the Foundation of Hoar Style Programming Logics***. Communications of the ACM, December 1982

[134] Ramsay A. ***Formal Methods in Artificial Intelligence***. Cambridge University Press, 1988

[135] Robinson A. ***Introduction to Model Theory and the Mathematics of Algebra***. North-Holland Publishing Company, 1974

[136] Rucker R. ***Infinity and the Mind***. Paladin Books, 1984

[137] Russell B. ***Mathematical logic as based on the Theory of Types***. American Journal of Mathematics, 30, pages 222-262

[138] Sammet J E. ***Programming languages: History and Fundamentals***. Prentice-Hall, Inc. 1969

[139] Shoenfield J R. ***Mathematical logic***. Addison-Wesley Publishing Company, Inc. 1967

[140] Skjeldrup H K, Wisnes A H. ***Den europeiske filosofi***. Gyldendal Norsk Forlag, Oslo, 1970

[141] Skolem Th. ***Selected Works in Logic***. Universitetsforlaget Oslo, 1970

[142] Smith J M, Smith D C P. ***Database Abstraction: Aggregation and generalization***. University of Utah. ACM Transactions on Database Systems, Vol. 2, June 1977, pages 105-133

[143] Sowa J P. ***Conceptual Structures***. Addison-Wesley Publishing Company, Inc. 1984

[144] Spurkland S, Salvesen A, Hauksson B. ***Form og mening***. NR 791. Norwegian Computing Centre.

[145] Steel T B Jr. ***A minimal Conceptual Schema Language for Life, Universe and Everything***. Database Semantics (DS-1). Elsevier Science Publishers B. V. (North-Holland) IFIP 1986

[146] Steel T B Jr. ***The Interpreted Predicate Logic Approach***. ISO TC97 SC5 WG3 N695. Edited by J. J. van Grietheusen ANSI 1982.

[147] Stone I F. ***Retsagen mod Sokrates***. Spektrum, 1989, 1990

[148] Tarski A. ***Der Wahrheitsbegriff in formalisierten Sprachen***. Studia Philosophica, 1, pages 261-405, 1936. Logic, Mathematics and Metamathematics. Oxford University Press, 1956

[149] Tarski A. ***The semantic conception of Truth***. Reprinted from Philosophy and Phenomenological Research, 4 (1944)

[150] Tiles M. ***The Philosophy of Set Theory***. Basil Blackwell Ltd, 11989

[151] Torkildsen T. ***Looking «Inside» Computations***. NIK'90. Department of Informatics, University of Bergen, Norway, 1990

[152] Tourenstzky D S. ***Lisp. A gentle Introduction to Symbolic Computation***. Harper & Row. Publishers. 1984

[153] Wall R. ***Introduction to Mathematical Linguistics***. Prentice-Hall, Inc. 1972

[154] Weigeland H. ***Conceptual Models in Prolog.*** Database Semantics (DS-1). Elsevier Science Publishers B. V. (North-Holland) IFIP 1986

[155] Wette E. ***Definition eines (relative vollstendigen) formalen System konstruktiver Arithmetik***. Springer, 1969.

[156] Wheeler J A. ***World as a System self-contained by Quantum Networking***. IBM Journal of Research and Development, Vol. 32, No. 1, January 1988

[157] Whitehead A N, Russell B. ***Principia Mathematica***. Cambridge at the University Press, 1910-1913 pages 1-93

[158] Wittgenstein L. ***Tractatus Logico-Philosophicus***. 1922

[159] Wittgenstein L. ***Philosophical Investigations***. 1953

[160] Wos L. ***Automated Reasoning***. Prentice Hall, 1984

[161] Zermelo E. ***Untersuchungen über die Grundlagen der Mengenlehre***. Mathematische Analen, I, 65, pages 261-281, 1908

[162] Zernadas A. ***Temporal aspects of logical procedure definition***. Information Systems, Vol. 5, No. 3 1980

[163] Zurek W J. ***Workshop on Complexity, Entropy and the Physics of Information***. Addison-Wesley Publishing Company, 1990

Annex B Popular Summary

May we have the contents of the book at a glance?

If you want to have an overview of the subjects covered by the book, you may start here.

This Annex gives an overview of main topics, section by section, but does not give the explanations and motivations that are found in the main part of the book.

Motivation

Our descriptions of the Universe exist within the physical universe itself. Hence, the statements and their interpretations cannot contain magic that goes outside the scope of this Universe.

We write all our knowledge of the Universe in our descriptions of it. If you cannot write it, you do not have knowledge of it. Hence, the Universe cannot contain aspects that cannot be described. If such aspects did exist, we would not have language to state their existence.

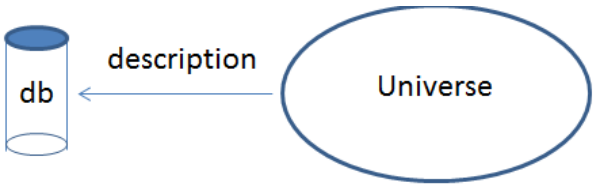


Figure B.1. One-to-one mapping between a description and what it describes

Perspectives

In traditional thinking, the observer exists outside his Universe and creates a description that is outside the Universe, as well. We call this observer God.

However, we claim that the observer must be inside the Universe, and so must his description. We call this observer the Ant. It describes its Universe of Discourse, which is limited by its position, capabilities and interests. Its view is intensional and not extensional. The intension may be about people, cars or anything else. Don't confuse the word intension with intent, which is about purposes like wealth, happiness, conquests etc.

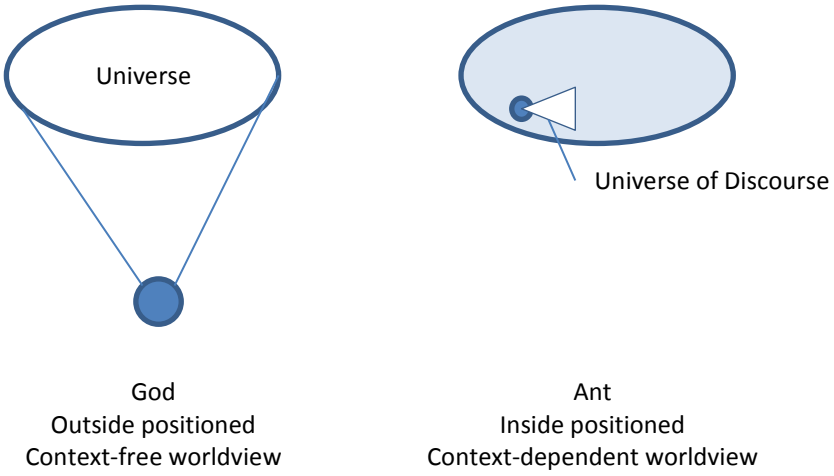


Figure B.2. Advocates for a context-dependent worldview

Phenomenology

We equip the Ant with a camera, and consider the integration of the Ant and its camera to be a Cyborg. Everything that exists for the Ant, exist as phenomena on the image inside its camera. The Ant creates an integrated image of all its phenomena.

The Ant is a phenomenologist: The phenomena are all there is. What phenomenon to be observed is dependent on the Ant's position, capabilities, instrumentation, timing etc.

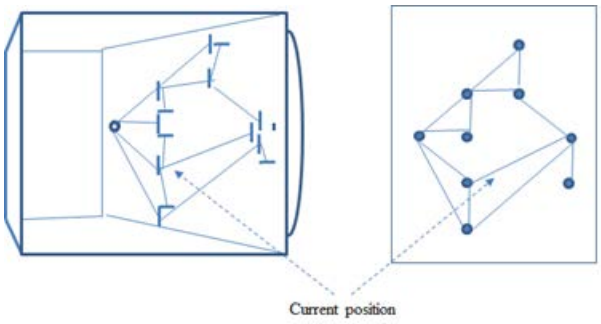


Figure B.3. The Ant's integrated image of its surroundings as seen from God

The Ant's graph tells what phenomena are observed from which phenomenon. To analyze which phenomena are observations of one and the same node/entity is no simple task.

The Ant's phenomena make up lists of lists, recursively. The phenomena correspond to roles, and the Ant's graph does not look like God's graph of nodes and edges.

Nominalism

The phenomena are data inside the observer. The Ant may create nametags for each phenomenon.

An extreme nominalist claims that data are all there is. Data like Person A and Car 1 may exist, but they may denote nothing. Our Ant considers both the phenomena to be data and the nametags to be data. Also, the mappings between phenomena and data are data. Nominalism specializes and extends the phenomenological viewpoint.

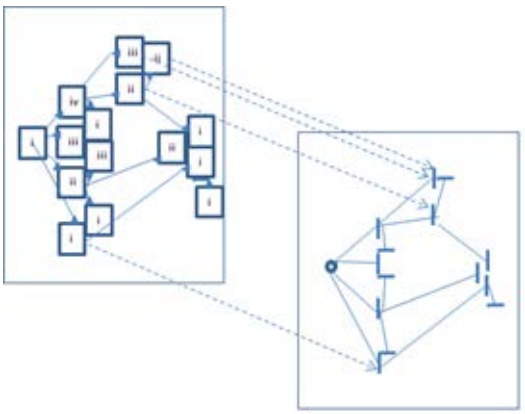


Figure B.4. Isomorphic mapping between data and phenomena

The Ant lets each nametag be a root of a name space of contained nametags. The structure of nametags is mapped one-to-one to the structure of the phenomena, but the mapping needs not be onto each phenomenon. There may be nametags without phenomena, and phenomena without nametags.

Existence

The phenomenologist

1. May claim: Phenomenon A exists
2. He points – e.g. at a screen - at the phenomenon image
3. He says: This is A
4. He records the place and time of the observation – at the screen
5. He delimits the phenomenon from its surroundings, e.g. by pointing
6. He has an instrumentation chain from the image at the screen to the entity being observed
7. He has validated the instrumentation chain
8. He records the place and time for the measurement at the entity side, or he calculates this from the known time for transfer from the entity side to the phenomenon side
9. Then he may add: The statement “Phenomenon A exists” is True.
10. If anything of the above is missing, then the statement “Phenomenon A exists” is False

The nominalist will add:

1. The observer puts in place a mechanism that ensures that the nametag A is unique within a name space having entity B as its root
2. The observer puts in place a mechanism that ensures that the entity A does not change its identity before a second observation
3. The observer puts in place a mechanism that ensures that nobody else can copy or use the nametag A before the second observation
4. The observer reads in a second observation the nametag A within the context B
5. The observer states: Phenomenon A (still) exists
6. Then he may add: The statement “Phenomenon A exists” is True in the second observation
7. If anything of the above is missing, then the statement “Phenomenon A exists” is False in the second observation

Hence, we may prove that

- The Universe does not exist – as an extensional set
- I do not exist – as a subject; only objects do
- God does not exist – as an observed and tagged phenomenon
- Infinities do not exist – since they cannot be observed
- Ants exist – if they are observed
- A particular Ant exists – if it is tagged

See reasoning about each bullet in section 4 of the book.

Naïve Set theory

We introduce Set Graphs to describe finite sets, as a replacement of Venn diagrams, because each Venn diagram can only depict two levels of sets.

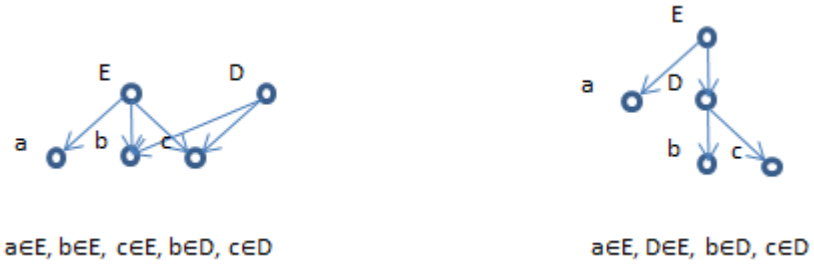


Figure B.5. Example Set Graphs

We show that ordered pairs break down in case of recursive relationships, and that you may experience entity migration, when new data are added. Entity migration is known from database theory, as a situation where the data structure has to be redefined when data are added.

Predicate Calculus

The calculus is about the truth-values of the statements, not about their contents.

The truth-values of statements are meta-statements about the statements, i.e. meta-statement about the contents.

We may not need truth-value logic in a general language about the contents of the physical universe.

Naïve Model Theory

The Type-token principle may be formulated as:

- Constants are globally unique
- Variables are unique within the scope of the quantifier

In the next Figure, tokens are mapped to types, inscriptions are mapped to strings. A subset of strings is called terms, and these are mapped to sets.

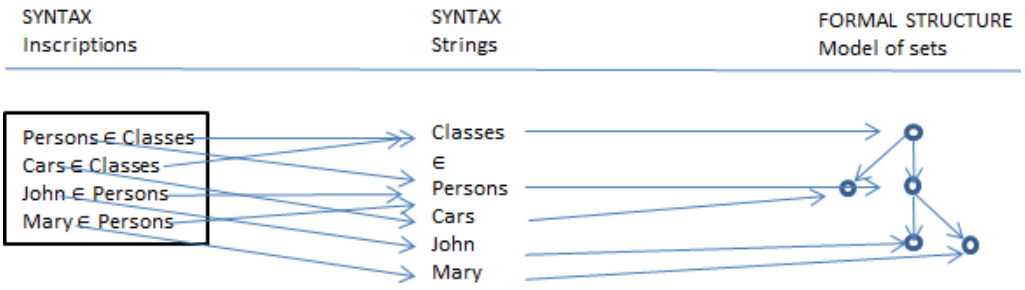


Figure B.6. Mappings from inscriptions via strings to sets

Understanding of the above principles is essential for the understanding classical logic.

The next Figure gives a more complete picture. Note that the Universe of Discourse is defined to be the set of all Possible Worlds of Sets. Note that all these worlds are extensional sets, and we do not discuss Modal logic.

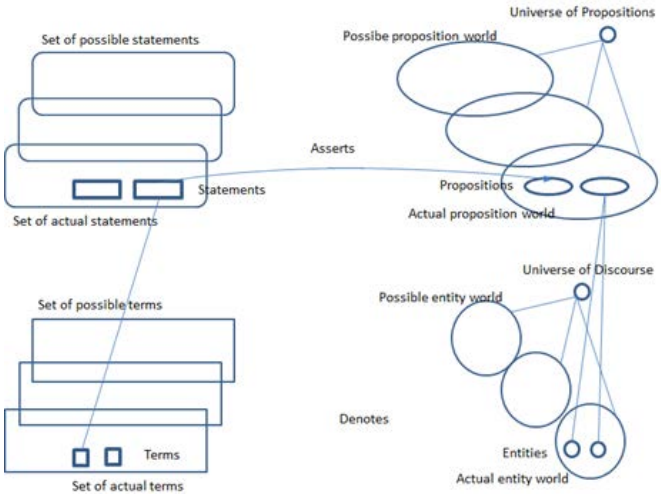


Figure B.7.Advanced Model Theory

Axioms of Set Theory

We abandon them all. The logicians seem to have raised a curtain of illusions before our eyes.

The Universe of sets seems not to belong to this world. It is a static structure of pure thought, from eternity to eternity. Somebody needs to create a rift through this static structure, creating a Big Bang of dynamic structures. This book is trying to make this rift and indicate how to create descriptions of this world.

Language architecture

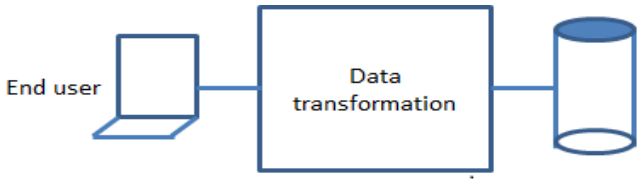


Figure B.8. An IT system is considered being a translator and editor

Data Transformation Architecture

The Data transformation architecture of an IT system is used as a replacement of Model Theory.

In the Data Transformation Architecture, the content of each screen is represented with a concrete and an abstract syntax. Also, each External Terminology is represented with a concrete and an abstract syntax. The notions of the abstract syntax of the External

terminology are called concepts; this definition of concepts deviate from most other approaches. Similar mappings are made through the internal layers towards eg. databases.

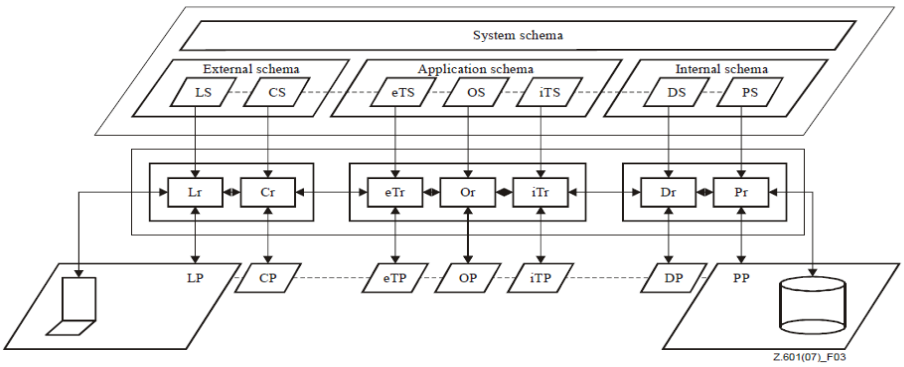


Figure B.9. Data Transformation Architecture

Requirements

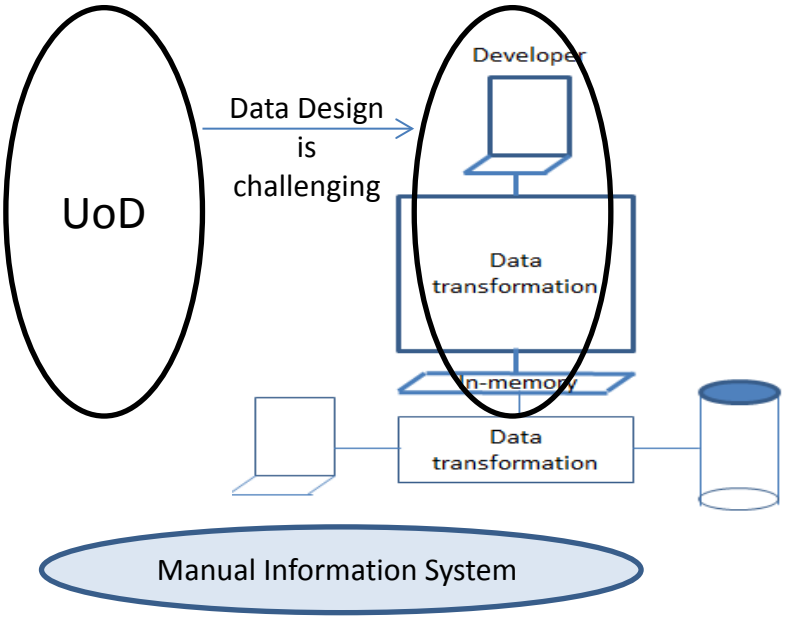
The mapping from concrete to abstract is isomorphic, i.e. one-to-one, and onto. The mapping from abstract Contents to concrete External Terminology is homomorphic, i.e. many-to-one. The mapping from instances in populations to classes in schemata is homomorphic. This also deviates much from other approaches. The schema contains templates for every instance, and defines the permissible structure of the instances.

Design of IT system

We discourage process analysis and design, as this leads to conservation of way of work, and implements business logic outside the data.

We recommend data design with the aim of providing efficient and flexible management of data. The Universe of Discourse is what the data are about, not how data are managed.

Business logic is implemented subordinate to data.



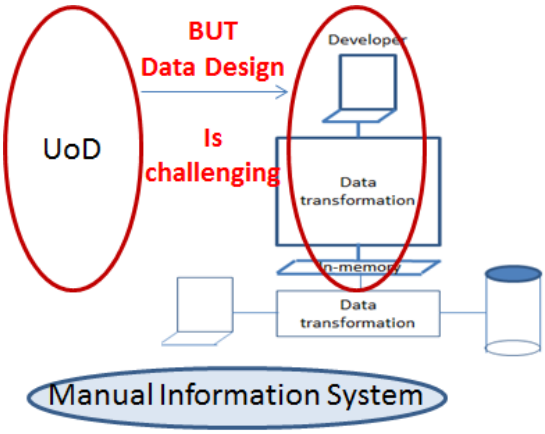


Figure B.10 Data Design for data oriented solutions

External Terminology Language

Elementary and normalized statements are stated in the External terminology layer.

The External Terminology Language consists of a three-dimensional list of lists. In a two-dimensional notations, condition (<>) and instruction (><) symbols are used to indicate the third dimension. The data tree represents name bindings, as subordinate nodes are identified within their superior node.

The language uses an attachment grammar, and not a rewriting grammar. This means that any branch of data is a valid production as long as all superior nodes remain. Any subordinate data node may be discarded. The superior nodes are used to state context.

The language uses Existence logic, and not Truth-value logic. This means that conditions state requirement on existence of other nodes, and may result in creation or deletion of other nodes. The conditions are not about truth.

```
:                                   System
        :                           Car 1
        :                           Car 2
        :                           Person 1
                :                   Owned car A
                :                   Owned car B
                        <>          Owned car B ‘Person 1 ‘System (Car 2
```

Figure B.11. Example expression

The next Figure shows how a read head (of a processor ♦) is following the condition (<>), shown to the right side, while the control head is checking its instructions against the structure to the left.

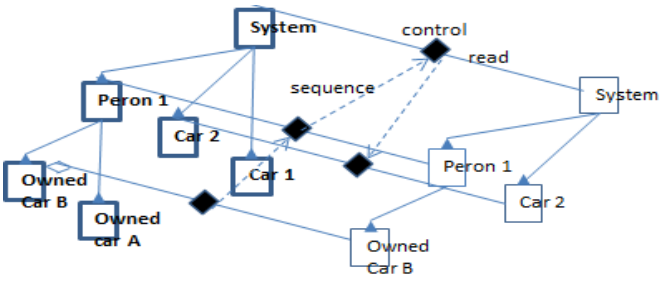


Figure B.12. Execution

Existence logic supports schema references between any two data nodes, not illustrated here. The subordinate data branch to the population node contains instances relative to the classes in the subordinate branch to the schema node.

Contents Language

Compound statements are stated in the Contents layer.

The next Figure shows the contents of an example Contents Schema. The Contents Schema traverses the data tree of the External Terminology Schema, and creates a cut through it.

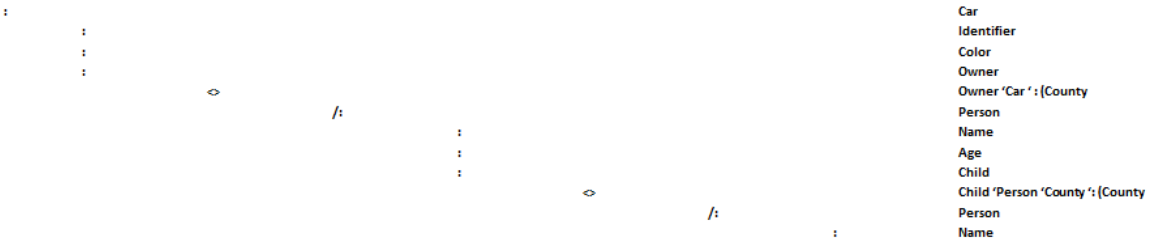


Figure B.13. Example Contents Schema

The Contents Schema may start at any node in the External Terminology Schema, and makes up a tree, as well. Each Contents Population has only one root node.

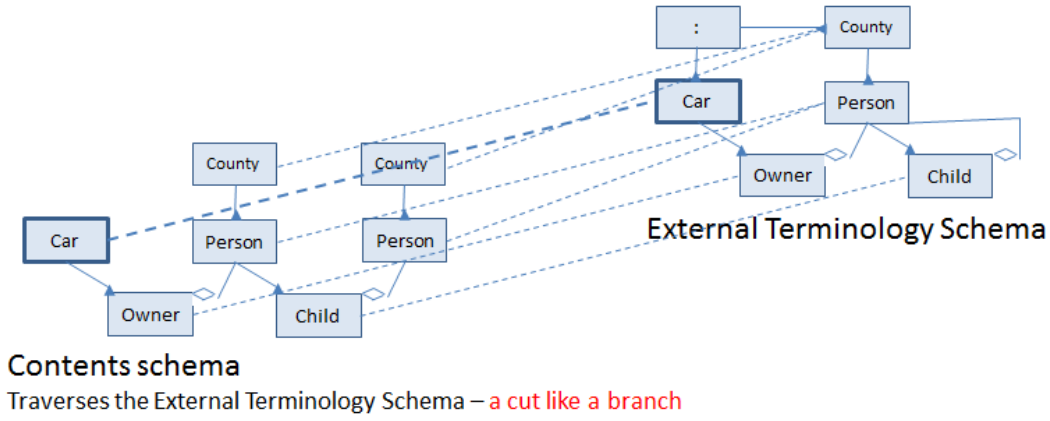


Figure B.14. Correspondences (dashed lines) between a Contents Schema and its External Terminology Schema

Layout

Layouts at screens and reports may be generated by defaults and automatically from the Contents layer.

The next Figure shows a possible layout of a screen using the Contents Schema in Figures B.13 and B.14.

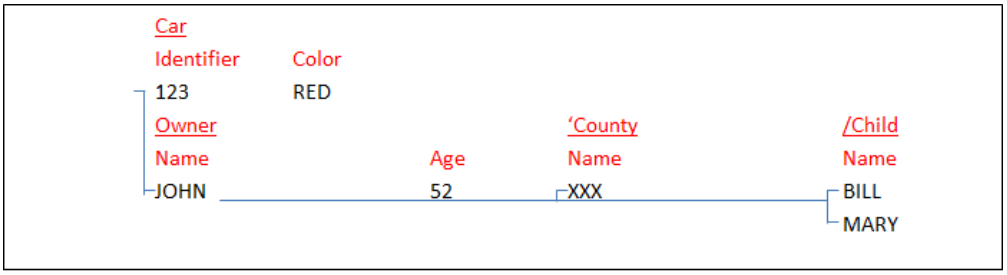


Figure B.15. Example screen layout

Note that the screen layout is a tree, and is not a flat file.

Language Considerations

- Is the foundation right?
- Are the language constructs right?
- What about execution?

The language is list oriented.

The language resembles object-oriented languages, as it is treating functions to be subordinate to data. However, the language does not support subclasses, as this presumes a realistic worldview. The role notion of Existence logic provides a phenomenological worldview.

The language supports schema references between data nodes, recursively, and has no strict distinction between classes and instances.

The Data transformation architecture separates implementations and presentations from applications into distinct layers.

The language does not support encapsulation, as access control is separated into the system management layer and Contents schemata.

Both queries and responses are tree-structured, and not flat files.

More on Existence

We observe that when using the schema references, classes have to exist, for instances to exist.

We revisit the War of Universals, and conclude that both sides were wrong.

Paradoxes

We revisit

- The Liar's paradox
- Russell's paradox
- Zenon's paradox

Using our new language notions, we conclude that the problem formulations of them all were wrong. We revisit the Pythagorean theorem to find argument against infinities.

Post Script

The purpose of the book is to indicate the need for a new foundation. I have indicated solutions, but I have not herein developed a manual for a new language.

I hope to challenge people.

Also, I hope that people can contribute and improve.

Annex C Graphic Notation

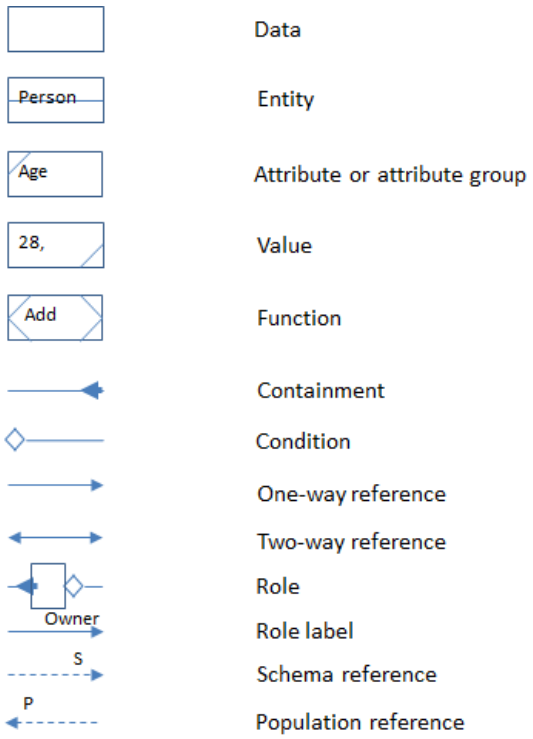


Figure C.1 Graphic symbols

Also, attribute groups and attributes may be placed inside the boxes, as shown in the next Figure. Indentations are used to show the structure.

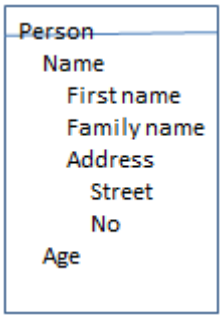


Figure C.2 Attribute groups and attributes within entities

Cardinality constraints, C(n, m), may be added inside and outside boxes, with or without the C. Also, the Identifier constraint Id may be added.

Often the root node of the External terminology schema graph is not depicted. Hence, the schema may have the appearance of a network, even if it is a tree, with references between the nodes.

Annex D Documentation

This Annex shows an example documentation of an application using the notions presented in this book.

Recommendations in the International Telecommunication Uninon - Standardization Organization (ITU-T) using notions described in this book, typically start with a Scope section.

“Scope
Service management is the universe of discourse (UoD) of this Recommendation. This UoD comprises products, customers, accounts, contracts, deals, addresses, prices and various segments and relationships between these.
…”

Then follows an External Terminology Schema graph.

Not that the root node is left out in the graph. Hence, the graph looks like a network, while it is a tree with references between the nodes in the tree.

“**External terminology schema**

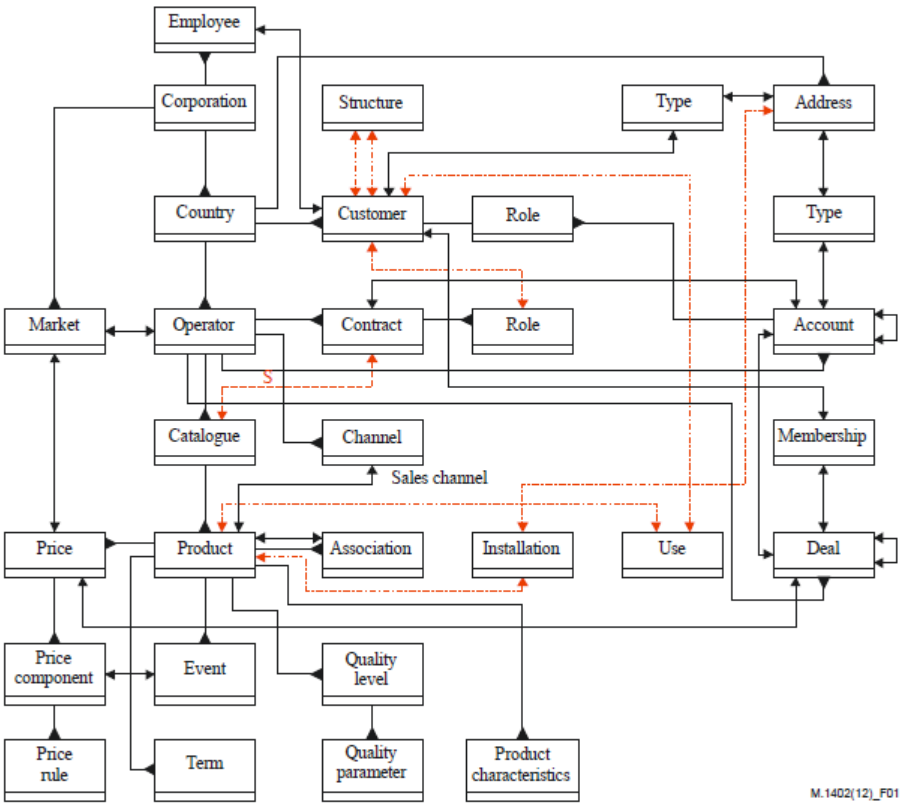


Figure 1 – External terminology schema graph for service management”

The red lines may here be read as black solid lines.

Thereafter follows a textual documentation, where indentations are used to show containments.

“Corporation

This Recommendation defines a schema of a system of a Corporation. The schema is called Corporation.

The schema contains data classes of the system. Each class acts as an original/prototype that can be copied into (several) instances subordinate to each of the populations of the schema. The S reference from a population to another data item makes the other item act as a schema relative to the population item.

Overlapping subsets of a schema may be implemented as separate systems.

- Country

  A Country may be an independent state or a state within a union.

  Operators are identified with a unique International Carrier Code within a Country. This means that a corporation must be defined with (minimum) one Operator within each Country.

  - CC

    Country Code is a three-character value according to ISO 3166-1.”

Annex E Detailed Requirements

Can we see a full requirement specification of the proposed language?

This Annex lists requirements on every schema and schema language of the Data transformation architecture. The focus is set on the core schemata of the architecture, and we have not elaborated here on the external concrete syntaxes.

The last subsection outlines how these requirements differ from classical logic.

Introduction

The External terminology schemata define the terminology and grammar of one application system.

Therefore, one such schema should be clearly defined before development of any other part of the application system, i.e., before defining the other schemata.

Hence, the requirements on the notation for the External terminology schemata come first in this section and should be understood before requirements on other schema notations are discussed.

Note that the requirements on the physical schemata, i.e., the external Layout schema and the internal Physical schema are not explored in detail in this section.

The requirements are presented in the following sequence:

- External terminology schema;
- Concept schema;
- Contents schema;
- Layout schema;
- Internal terminology schema;
- Distribution schema;
- Physical schema;
- System (management) schema.

Requirements on notations for the External terminology schemata

Introduction
The External terminology schema should be capable of expressing the syntactical richness of elementary statements and their constituents as encountered at the human-computer interface of one system. The following requirements may all be considered means to this end. However, the full capability to the human user is only provided by the addition of the features of the Contents and Layout schemata, as indicated in the text. Also, features of the System management schemata are needed for management purposes, and the Internal schemata are needed for implementation.

Requirements

1) The contents of an External terminology schema are recursively made up of lists of elementary statements and their constituents, which may contain references to other statements or constituents.

a) The External terminology schema is itself a node in a data tree, which also contains its elementary statements and other constituents.
b) The purpose of the External terminology schema is to prescribe permissible contents of its External terminology populations.
c) The purpose of restricting the External terminology schema to elementary statements and their constituents is that the creation of compound statements is deferred to the Contents schemata.
d) The expression 'elementary statements' is meant to refer to data structuring notions, like objects and attributes, and is not meant to require use of sentential logic; the expression 'and their constituents' is meant to indicate value syntax and behavior expressions.
e) Lists within the External terminology schema are used to express alternatives, i.e., disjunctions, of what is permissible in the External terminology population.
f) Lists within lists are used to express context.
g) As superior nodes of lists always will have to exist, they express a kind of conjunction.

2) The content of any External terminology schema is homomorphic to its External terminology population.
    a) For each node in the External terminology population, there is an identical node in the External terminology schema.
    b) For each node in an External terminology schema, there may be zero or more nodes in its External terminology population. This instantiation is made by copying class labels and references between classes into the External terminology population.
    c) Any node in the External terminology schema is called a class relative to its corresponding nodes in the External terminology population, which are called instances relative to their class. The nodes are classes and instances relative to each other, and are not marked as such by any reserved words.
    d) If there is a S(chema) reference from a node to another node, then this other node is called a schema relative to the first node. The schema node contains the classes of the instances contained in the first node.
    e) If there is a P(opulation) reference from a node to another node, then this other node is called a population relative to the first node. The population node contains instances of the classes contained in the first node.
    f) Given an instance and its corresponding class; then the homomorphy requirement implies that the superior node of the class is itself a class of the superior node of the instance.
    g) If a reference is made between two instances, then the homomorphy requirement implies that a corresponding reference is made between the corresponding classes.
    h) The purpose of this requirement is to allow end users to foresee permissible instance structures (just by making copies of the schemata) when they see the class structures, and vice versa.
    i) One implication of this requirement is that the user can see no difference between a class and an instance label, when they are observed in isolation. Only their recursively superior structures and

references to schemata or populations can show the different roles.

j) In predicate calculus, there is no distinction made between instances and classes; the statements are just checked for consistency or inconsistency. However, we make a distinction between instances and classes, because we need to know what data are prescriptions for which data. We do not make the distinction in an absolute way. Data are just instances and classes relative to each other.

3) The name label of any node in the External terminology schema can be alphanumerical or graphical.
    a) The purpose of this requirement is to allow use of any character set, and the designer should be allowed to design his own characters.
    b) Graphical name labels may be pixel or vector based. This is used to prescribe graphics of the instances, as the class labels are copied into each one of its instances.
4) The notation of the External terminology schema shall be postfix and have no superficial block structure.
    a) The purpose of disallowing superficial block structures is to satisfy the homomorphy requirement; i.e., the structure of the classes shall be homomorphic to the structure of their instances. Hence, extra nodes or levels among the classes are disallowed.
    b) The purpose of requiring the postfix notation is due to the homomorphism requirement, as well. If (logical) operators will not appear above or between the nodes among the instances, they cannot appear above or between the corresponding classes either. Hence, all constraints and derivations are expressed subordinate to some class node.
    c) The previous bullet explains why the External terminology schemata contains elementary statements only, and do not contain expressions that are consistent with or derivable from the elementary statements.
    d) Derived statements, that are themselves elementary, are contained in the External terminology schemata. The derived statements may be derived by a compound function that must itself be contained in the External terminology schema.
5) Class labels are only unique within the scope of the superior class label, if a larger scope is not explicitly stated at a superior node. See c.
    a) This means that each node acts as a block for its subordinate nodes.
    b) Identical class labels can be reused for different purposes within different classes.
    c) Some classes may contain further constraints (on naming), e.g., that all recursively subordinate name labels to a constrained class must be unique.
    d) The implication of this requirement of using local class labels is that references to a class must contain the path from the current node to the referenced node. Note that use of a globally unique path of class labels may not work, as the exact path from node to node is needed to express the exact scope of the reference.
    e) Note that the path of the reference between class labels prescribes the permissible paths of references between instances. Hence, the navigation paths provide a scoping mechanism of the references.
    f) Another implication of using local class labels is that no class label

has a unique identification without providing the entire globally distinguished class label, which is made up of the path of its recursively superior class labels. Hence, to state a class label without providing its context makes no sense.

g) The superior class of a local class label is used to express the context of the local class label. Use of local class labels allows users to use their own terminology for classes and not to invent artificial labels to satisfy the naming convention of a particular specification language.

h) As class labels are copied into becoming instance labels, use of local class labels has implications for the naming tree of the corresponding instances. Use of local class labels prescribes name bindings between the corresponding instances.

6) Class labels need not be unique within the scope of their superior class label, as use of duplicates is permissible.
   a) Duplicate class labels are distinguishable from each other, due to their position in the list subordinate to their superior class label. Also, the subordinate structure of or references from or to duplicate class labels may be different, thus helping to distinguish the class labels.
   b) Use of duplicate class labels allows users to use their own terminology of classes and not to invent artificial labels to satisfy the naming convention of a particular specification language.
   c) The handling of lists of instances, where the instances are only distinguishable by their position in the list, is another reason for allowing use of duplicate class labels. As there is no fundamental difference between instances and classes, they are treated equally. Nodes are only classes and instances relative to each other.
   d) The position of a duplicate may change during execution, and duplicates may be created and deleted, while references to and from the duplicates are maintained.
   e) Within the context of a certain class, duplicate class labels may be disallowed by an explicit constraint. This certain class may be the schema itself.
   f) Data designers may avoid use of duplicate labels without this being explicitly stated as a constraint.
7) The External terminology schema notation should allow for recursion.
   a) Any node in the population or schema may have one or more S(chema) references to any other node in a population or schema.
   b) This means that there is no strict distinction between a population and a schema, and the notations of both are identical.
   c) This also means that a schema may have a meta-schema, etc.; or a node within a schema may have a S(chema) reference to another node within this schema, or of some other schema, or of some other populations or to a node within its own population.
   d) A S(chema) reference to a superior node is used to state recursion. The contents of the referenced superior node may act as a schema of what can be contained in the current node; hence, some contents of the superior node may be copied into the current node – maybe repeatedly.
   e) Note that any reference – not only a S(chema) reference – may refer to any node, including a superior node of the current node.

f) Any class of a schema may be instantiated into zero or more instances. If there is no explicit cardinality constraint on the class (relative to its superior class), then any number of instances may be generated relative to their superior instance.
g) Note that if an instance is created, also all its recursively superior instances must already exist or be created.
h) However, if an instance is created, no subordinate instances may be created, if there is no explicit constraint requesting this creation of subordinate instances.
i) This way, the schema prescribes the existence of all recursively superior nodes of an instance, while subordinate nodes may not exist. Therefore, the schema is considered to provide an attachment grammar as opposed to a rewriting grammar. In a rewriting grammar the superior nodes are discarded, and only leaf nodes are left in the final production. In the attachment grammar the recursively superior nodes provide the required context and must exist in the final production, while the leaf nodes may not be needed.
j) Note that while all the recursively superior nodes within the External terminology layer are required in the Contents layer, some of these nodes may be suppressed in the Layout layer.

8) The External terminology schema should be capable of expressing any logical, arithmetical and quantification statements.
    a) This is called the 100% principle.
    b) The purpose of this requirement is to be able to express all constraints and derivations within the External terminology schema, and not have to rely on behavior specifications outside this schema.
9) An External terminology schema should be capable of prescribing the abstract grammar of the elementary statements and abstract syntax of the terms of an end user terminology.
    a) Each External terminology schema prescribes different end user terminologies, e.g., English, French, Chinese and graphics.
    b) An External terminology schema does not define the common concepts of different end user terminologies, but one External terminology schema may act as the source of other end user terminologies defined in other External terminology schemata.
    c) An External terminology schema may have source references to a Concept schema of the current External terminology schema.

Part 4 section 13 presents the External terminology language.

Requirements on notations for the Concept schemata

Introduction

A copy of an External terminology schema may play the role of being a Concept schema of other External terminology schemata.

This means that the notation for defining any External terminology schema satisfies the notational requirements for defining any concept schema.

The Concept schema (and its population) may contain more concepts than supported in some of its External terminology schemata (and their populations).

Also, the External terminology schemata (and their populations) may contain data details that are not supported by the corresponding Concept schema (and its population).

The notational requirements for the Concept schemata and the External terminology schemata are the same.

If both External terminology schemata and Concept schema are used, then all common constraints and derivations for every External terminology schema may be expressed in the Concept schema only. Only, specific additional constraints are expressed in each External terminology schema.

However, if some constraints and derivations are only expressed in the Concept schema, they are not easily accessible and readable in the External terminology schemata.

In addition to requirements on the definition of the concepts and their associations, mappings between nodes in the External terminology schemata and concepts are needed. This mapping is called a denotation mapping. In this text we address the mapping between data and concepts, not between data and phenomena, and not between concepts and phenomena.

Requirements

1. If a node in an External terminology schema denotes a concept in its Concept schema, then all its recursively superior nodes shall denote a concept as well, and a path of references between concepts shall represent the node structure.
    b) The node structure should be mirrored by an identical concept structure, such that the concept structure becomes isomorphic to the node structure;
2) If a node in an External terminology schema does not denote a concept in its concept schema, then no recursively subordinate node shall denote a concept either.
    a) This allows subordinate nodes in the External terminology schema to provide 'syntactical sugar', e.g., be helpful for identification, but without denoting any concept.
3) An attribute in the External terminology schema may denote a role (of an object class) in its Concept schema.
    a) This allows value types to be represented as object classes in the Concept schema.
4) Any association in the External terminology schema that denotes something, must be represented by a path of (one or more) references in the Concept schema.
5) In one External terminology schema there may be several synonymous terms, even if each synonym normally should be defined in a separate External terminology schema.
6) Every node in the External terminology schema must denote a concept or path of concepts in the Concept schema; exceptions are made for 2.
7) Except for 2, 3, 4 and 5, the graph of the Concept schema should be isomorphic to the graph of any of its External terminology schemata.

Requirements on notations for the Contents schemata

Introduction

The Contents schemata define the abstract syntax of the statements presented to the human end user.

Each Contents schema refers to one External terminology schema only.

Each Contents schema defines a compound statement of some of the statements contained in its External terminology schema.

Requirements

1. The elementary statements in the External terminology schema are combined into compound statements in the Contents schemata by the use of reflexive pronouns with the proper references.
    b) The references in the Contents schemata shall be able to express both parenthesis structures and alternative sequences of presentation.
    c) The previous sub-bullet means that the compound statement should be able to express the following example compound statement: "Site relationship has subordinate trail (which) has subordinate trail section (which) has subordinate physical link connection which is (physical link (which) has subordinate physical link connection)". Here the globally distinguished name of the referenced physical link connection of the physical (both in parenthesis) is given by providing the identifier of the superior physical link first. "Which" is the reflexive pronoun; note that this is used to state references only, and not to state subordination, but is here added in parentheses to ease the reading.
    d) The first sub-bullet also means that the compound statement should be able to express the following compound statement: "Site relationship has subordinate trail has subordinate trail section has subordinate physical link connection which is (physical link connection has superior physical link)". Here the globally distinguished name of the referenced physical link connection will be given last.
    e) The parenthesis structures shall be capable of expressing branching, like in the following statement: "Site relationship (has subordinate trail has subordinate trail section has subordinate physical link connection which is (physical link has subordinate physical link connection)) has subordinate identifier". Here Identifier is an attribute of site relationship.
    f) Also the parenthesis structures should be capable of expressing delimitation of execution, e.g., traversal of a reference one way, may not automatically lead to a traversal of the opposite reference and control of consistency, even if this is required by the External terminology schema.
    g) Note that the above informal examples use just one kind of parentheses, but that different kinds may be needed in the Contents schema notation.
    h) Note that the compound statements in the Contents schemata are traversing the elementary statements in the External terminology schema in various permissible ways.
2) The paths expressed in the Contents schema shall never deviate from the paths expressed in the corresponding External terminology schema.
3) The Contents schema shall not define derived data that are not already defined in the External terminology schemata.
4) The notation of the Contents schema should be capable of expressing recursion that does not appear in the external terminology schema, e.g., if there is a loop of

associations in the External terminology schema, the contents schema may state that this loop shall be traversed recursively until no more data instance is found.

5) The notation of the Contents schema should be capable of expressing logical and arithmetical conditions on the data defined in the External terminology schemata, e.g., the External terminology schema defines that site relationship has subordinate trail has subordinate length (that is given in meters). A corresponding Contents schema may restrict the selected trails to be within the range 1000-2000 meters.
6) The notation of the Contents schema should be capable of expressing what data will be presented in the Layout layer and what will be suppressed.
7) The notation of the Contents schema should be capable of expressing permissible operations, e.g., insert, delete, modify and read, on data instances prescribed by this Contents schema.

Part 4 section 14 presents the Contents language.

Requirements on notations for the Layout schemata

Introduction

The Layout schemata define the concrete syntax of the data as they are presented to the human end user.

Each Layout schema refers to one Contents schema only, but there may be several alternative Layout schemata for each Contents schema.

The layout may define separate fields for operators that may not appear in the Contents schema. Also, the layout may define advertisements etc. that may not appear in the Contents schema.

Requirements

1) The notations for the Layout schemata should provide multimedia support.

Part 4 section 15 presents the Layout language.

Requirements on notations for Internal terminology schemata

Introduction

The Internal layer may adapt to the technology used for communication over a telecommunication line or in a database management system. Therefore, the Internal layer may not convey or support the terminology used to communicate with a human user. However, the Internal layer may have to convey information needed to translate between different terminologies of different users.

One such example can be translation between codes according to international, regional and company standards. In these cases, different object structures are used, and there is no one-to-one mapping between fields. Hence, several fields in one standard have to be communicated

to provide the contents of one field in another, and only parts of the fields are needed to generate the one field.

From this explanation follows that more information may be needed in the internal terminology schemata than conveyed in the used External terminology schema, and the mappings between the schemata (via the Concept schema or not) may be complex, using detailed navigation and selection of information from unpacking the value syntax of the fields.

Requirements

1. The notations used for the Internal terminology schemata must be capable of expressing the data structures used in implementations of peer communicating systems.
2. The notations used for the Internal terminology schemata may not be capable of expressing the logic and arithmetic of the application system, i.e., the notation may not support all capabilities of the External terminology schemata.
3. The notation used for expressing the mappings between the Internal terminology schemata and the External terminology schemata (via the Concept schema or not) must be capable of expressing detailed navigation and selection down to the value syntax of fields.

Requirements on notations for the Distribution schemata

Introduction

A Distribution schema defines one message class for communication between systems or with internal media of the application system.

The message class may be elementary, as in object-oriented communication, or the message class may be complex, as in file transfer.

The Distribution schema states a view of the corresponding Internal terminology schema.

Requirements

1) The notation for the Distribution schema must be capable of expressing any view of the corresponding Internal terminology schema.
2) The notation for the Distribution schema must comply both with the Internal terminology schema and the Physical schema being used, but need not express a strict subset of either.
3) The Distribution schema must be capable of defining any vertical or horizontal partitioning of data.
4) The Distribution schema must be able to address its peer systems and media and define communication with these.

Requirements on notations for the Physical schemata

Introduction

A Physical schema defines the structure of the data as represented on an internal medium, e.g., on a hard-disk or a telecommunication line.

Requirements

1) A notation for communication over a telecommunication line must be capable of defining the encoding of signals on this line.
2) There may not be a one-to-one mapping between signals defined in the encoding and messages defined in the Distribution schema.
3) The notation for communication over the telecommunication line must be capable of defining the mapping between message classes defined in the Distribution schema and signal classes in the Physical schema, and define the protocol for this implementation.
4) A notation for a database must be capable of defining the storage organization and storage details as perceived by the database designer.
5) The database views defined in the Distribution schemata may deviate much from the physical organization of the database, defined in the Physical schema.
6) The notation for defining the physical organization of the database must have means for stating proper references and access paths for its Distribution schemata.

Requirements on notations for the System management schemata

Introduction

The System management schemata contain security and directory data for one or more systems.

Requirements

1) A System management schema may contain data for more than one system, but must contain each system identifier and clearly indicate which resources, e.g., schemata and populations, relate to which system.
2) Security data may address Application layer, Contents layer and Distribution layer data; they will rarely address Layout and Physical layers.
3) Directory data will provide overview of all layers and how they relate to each other.
4) Separate recommendations will apply to security and directory data.

Discussions

The schema notion contains templates for the data that can be copied into the populations. This way, in a population you can only state something that is explicitly permitted within the schema.

Since both the population and the schema are themselves inscriptions, you state the references between them and thereby provide the permissions for the instantiations.

To accomplish the same is somewhat more cluttered in classical logic, because here you cannot refer to a particular inscription. You can only refer to a set denoted by the inscriptions. Therefore, you have to define a set denoted by the schema name (S); and a set denoted by the population name (P).

For every constant and variable (x) in the schema, you have to write x ∈ S. For every constant and variable (x) in the population, you have to write x ∈ P. If the schema data and the population data are put into separate files, you may create these additional statements automatically.

Additionally you will have to state the following constraint: If (x ∈ S) then ¬ (x ∈ P), and If (x ∈ P) then ¬ (x ∈ S). However, this only states what is right, but classical logic does not prohibit that you state false statements.

In Existence logic it is impossible to make statements in the population which are not permissible according to its schema. This is a fundamental difference between classical logic and Existence logic. After Henrik Ibsen: 'Yes, to think it, and even wish it, but to do it?' Classical logic can say it, but not do it. Note that nothing prescribes that you must have a schema in the external terminology statements, but the schema notion may be very useful.

The reader should note that explicit disjunctions and conjunctions are not normally needed in the external terminology schema language, as the lists in the schema indicate what are the permissible alternatives to be copied into the population. This means that the external terminology schema is a normalized organization of statements, where the lists are replacing disjunctions in classical logic. The hierarchical structure of data, of lists within lists, expresses that the subordinate data have to be together with their superior data, i.e. a kind of conjunction.

Lists within the population correspond to conjunctions in classical logic.

The external schema language does not have a clear distinction between constants and variables, as in classical logic. All the terms in the population appear in the schema, as well.

During execution, all needed data and functions from the schema are copied into the population.

In classical logic and mathematics, a function application is, as a result of the execution or resolution, replaced by the function value term. In Existence logic, functions are treated differently. We will let any term with an attached condition symbol (<>) act as a function. This symbol indicates a third dimension of the list of lists. During execution, the function may access its arguments anywhere in the data structure, and it may place its function value anywhere else. Arguments are found through references by navigation, and instructions tell by navigation where to put the function values.

In Existence logic we do not use predicates.

Annex F Entity Migration

Entity migration has been mentioned several times in this book; so tell us what it is?

This Annex exemplifies how definitions of entities and their associations have to be changed when adding new information to definitions using Set theory. The difficulty stems from associating the information with entities, and not with the roles of these entities. Hence, we need a language that focuses on roles. I recommend that we discard entities and their associations.

If we want to use Set theory to interpret information in a database, we may want to treat the entities as being sets. We will associate information with these entities/sets and relationships. However, when we want to add information, we may want to distinguish the roles of the entities, and associate different information with each role. Let us use an example:

We have an entity Person-John and want to associate this entity with the entity Contract-date-01.01.2000. See Figure F.1.

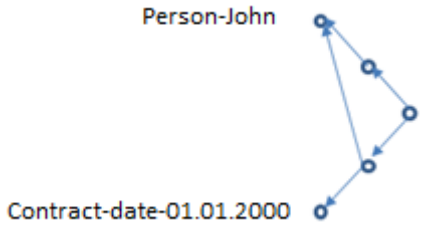


Figure F.1 An entity with its associated information

Then we want to extend this database with Company-A, and to insert a Contract relationship to Person-John. If so, we may want to associate the Contract-date-01.01.2000 with the Contract relationship, and not with the Person-John. Hence, we need to change the Set structure.

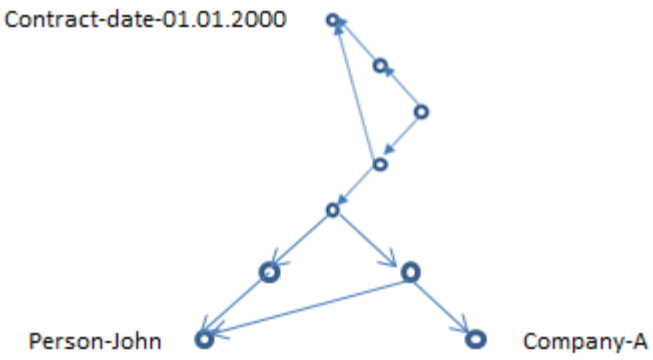


Figure F.2 A relationship with associated information

We may also want to treat Send-date-01.01.2000 and Receive-date-01.01.2000 as two different pieces of information. We may want to associate these two dates to the Customer and Vendor roles of the relation. Hence, we will have to define each role as an entity.

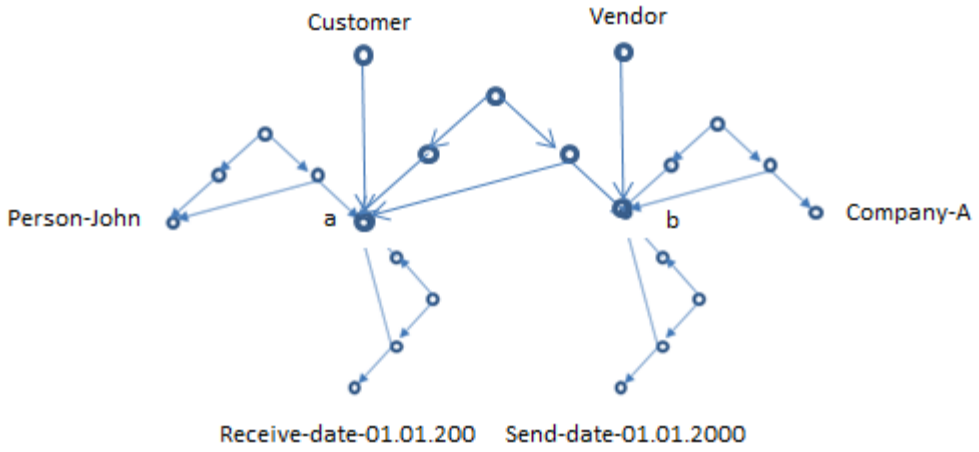


Figure F.3 Two roles with associated information

And, so it goes. We add other roles, like Employer-John, Employed-John, Friend-John, Father-John etc. We realize that we have to identify roles and not entities, and we want to associate different information with each role.

The same kind of thinking applies to relationships. We do not want to associate information with entities and relationships in a realistic or conceptualistic worldview. We want to associate information with the roles, which act as phenomena.

The roles are not seen from other entities. They are seen from other roles. The Company may have several roles, like Vendor-A, Tax-payer-A, Corporation-A, Employer-A, Partner-A etc. The Customer-John role is seen from the Vendor-A role only.

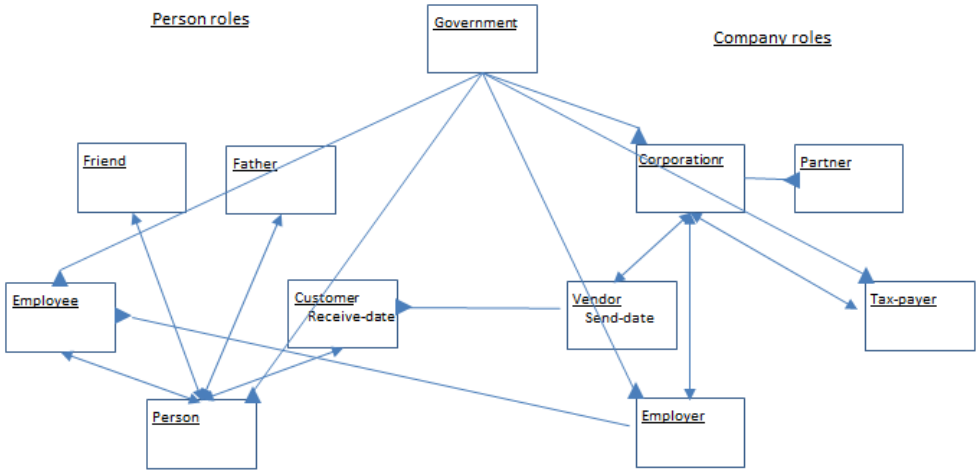


Figure F.4 The role representation

What entity is seen from which entity, eg. Employee seen from Employer, is indicated by reversed arrowheads representing containments in the above Figure. The Employee is not employed by the Vendor, Corporation, Partner or Tax-payer as we may be inclined to say in natural language.

We realize that we need notions of roles being observed from other roles. Whenever we identify roles, we have to be very conscious about which role they are observed from, and each role is only observed from one other role.

We have learned that the entity-relationship thinking of Set theory has been too naïve, as it has not identified from where the entities and relationships are being observed, and have not clearly distinguished the various roles, which may have different attributes. When the role notion is clearly understood, we can do away with the relationship and set notions.

Note that we also do away with subclasses or sub-sets, as they presuppose a global context independent worldview.

Annex G Example Implementation

Can you show a realistic example?

In this book we have been discussing the foundation of a language for inscribing, prescribing and describing anything in the entire physical universe. Some sections and some paragraphs make statements about down to earth implementations, eg. in databases. Some readers will prefer to see such implementations before going into the theoretical discussions.

We have learned that data are organized in a three dimensional structure of lists of lists, and data in databases are normally not organized in predicates combined by logical connectives. Rather, a database can be considered to be a building of terms, where containments make up the pillars, next associations in lists make up the beams, conditions are the connectors, and instructions allow events to take place.

In this Annex, we will sketch how a database application using the Data transformation architecture can be designed, and how Existence logic may be used to define this application.

The Architecture

We will use the Data transformation architecture, as depicted in Figure 10.1. We will omit the Concept layer and the Internal terminology layer. This means that the Distribution schema will map directly from the External terminology schema to the Physical schema, i.e. to the database schema.

We will use the nesting architecture depicted in Figure 9.2. In this Annex, we will use the schema depicted in Figure G.1 for the contents of the data dictionary.

Data Dictionary Schema

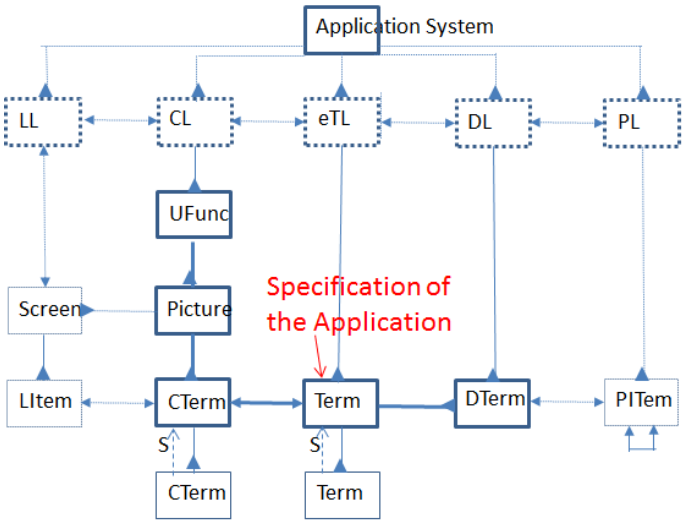


Figure G.1 Schema of data dictionary

In Figure G.1, LL means Layout layer, CL means Contents layer, eTL means External terminology layer, DL means Distribution layer and PL means Physical layer. An Application System will contain all these layers.

Note that we do not split the Application System into schemata and populations, as these will appear inside each layer, and they are indicated as terms containing S(chema) references in these layers. We have no absolute distinction between schemata and populations, but the data dictionary will focus on schema data.

The external Terminology layer contains Term-s. These Term-s may contain Term-s recursively, indicated by the S(chema) reference from subordinate Term to superior Term in the above Figure. The tree of Term-s may state the contents of the entire Application, including all its logic.

The Physical layer contains Physical layer terms, PlTerm, which are connected in a network, indicated by the recursive reference from PlTem to PlTerm in the above Figure. The PlTerm-s may be records, sets, fields etc. See below on network databases.

The Distribution layer contains references from Term-s in the External terminology layer to PlTerm-s in the Physical layer. These references are called DTerm-s. See the Figure. The DTerm-s state an incremental access path among the PlTerm-s in the Physical layer from the superior Term to the current Term in the External terminology layer. This access path is a strict sequence of Distribution terms, DTerm-s. The DTerm-s may additionally contain identifiers of procedures, which can do constraint checking, derivations, parsing of syntax etc.

The Layout layer may refer to a set of Screen-s. The Screen-s and their contents may be created dynamically, and may not be pre-defined. Each Screen contains Layout layer items, called LlItem in the Figure. The LlItem-s of an alphanumerical screen may consist of fields in a strict sequence from left to right continuing in the next line. Also, more complex structures may be used. The LlItem-s may have font, size, colour and other attributes.

The Contents layer may contain a list of User functions, UFunc, where the first UFunc may be default. See the Figure. Each UFunc may contain a list of Picture-s, where the first Picture may be default. Each Picture may contain Screen-s of the Layout Layer. If the Screen-s are created dynamically, i.e. only when they are needed, a Picture may have one single Screen at any time. Each Picture contains Contents terms, called CTerm in the Figure. The CTerm-s map from Term-s in the External terminology layer to LlItem-s in the Layout layer. Most often, one CTerm maps from one Term to one LlTerm. The CTerm-s form a tree, which may span across many branches of the Term tree. See the S(chema) reference.

The Application

We will use a simple example application, being depicted in Figure G.2.

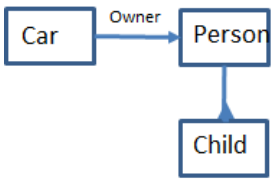


Figure G.2 Example External terminology schema

The Database

We will store the data in a CODASYL network database. Figure G.3 shows the database schema.

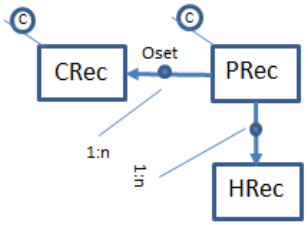


Figure G.3 Example CODASYL database schema

Figure G.4 shows an example instance database. Note that CODASYL is using Calc sets to link records with the same hash key. CODASYL sets are used to link child records. The CODASYL sets give references by physical addresses and record numbers. The references form a ring, may be two-way and refer to the parent record. CODASYL sets are efficient for traversing from record instance to record instance, like when retrieving a Car, its Person-s and their Child-s. Using CODASYL databases, we avoid the handling of functional dependencies and set operations. However, we will benefit from having a layer above the database layer, and this is what the external Terminology Layer (eTL) does.

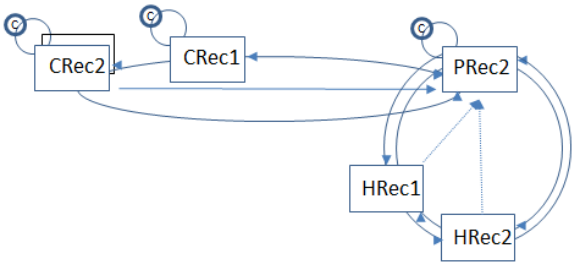


Figure G.4 Example CODASYL instance diagram

Distribution schema

The Distribution schema from the External terminology schema to the CODASYL database schema may look as stated in the three rightmost columns in Figure G.5.

:			Car	CRec	R	
	:		License number	Lfield	F	
	:		Owner	OSet	S	R
		<>	Owner ‘Car ‘: /(Person			
:			Person	PRec	R	
	:		Name	Nfield	F	
	:		Child	HSet	S	
				HRec	R	
		:	Name	Hfield	F	

Figure G.5 Example Distribution Schema

The colon (:) to the very right in the Condition in line 4 is a place holder for the external Terminology Schema, which contains Car, Person and their Child-s.

The rightmost column, containing the R, tells that the OSet is going in the reverse (R) direction, i.e. from PRec to CRec. The second last column indicates the kind of the database terms, i.e. record (R), set (S) and field (S). The DTerm-s indicate incremental access paths, eg. <HSet, HRec> from a Person to its Child-s.

Contents schema

The Contents schema tells what data to be shown in a screen Picture and how these data are presented. The Distribution schema tells how the data are accessed.

Figure G.6 shows an example Contents schema combined with its Distribution schema.

```
:                       IMDR  Car       E     CRec    R
    :                   IMDR  L.No      I     Lfield  F
    :                   IMDR  Owner     R     OSet    S      R
      <>                      Owner 'Car ':
          /:            IMDR  Pers      E     PRec    R
            :           IMDR  Name      I     Nfield  F
            :           IMDR  Child     E     HSet    S
                                              HRec    R
                  :     IMDR  Name      I     Nfield  F
```

Figure G.6 Example contents of Data Dictionary

In the example, we look up a Car with a License number (L.No), which is its identifier (I) attribute. Then we list its Owner-s, who are Person-s (Pers), each with their identifier Name. Then we list each Person's Child-s with their Name. Note that we have not defined that a Child is a Person. Hence, Pers (Name is not a domain of Child (Name.

The column with IDMR states that the user may insert (I), modify (M), delete (D) and read (R) this CTerm.

The column with E, I and R indicates entity, identifier attribute and role, respectively.

In-Memory Database

The contents of Figure G.6 may be loaded into an in-memory database and made more compact and suitable for execution. The execution is line by line. See the in-memory database in Figure 9.2.

Layout

The layout of the screen and the end user dialog features may be generated from the Contents schema by using defaults.

The resulting empty screen picture may look as shown in Figure G.7.

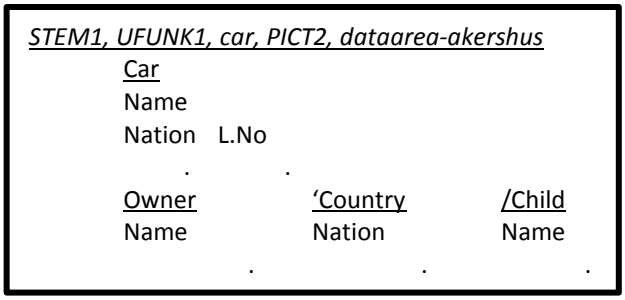
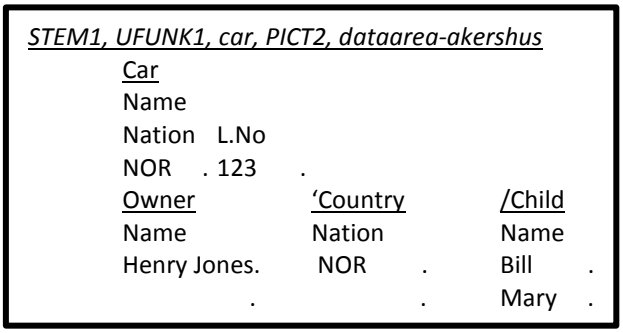


Figure G.7 Example empty screen picture

The user may fill in the Car's L.No, and gets the screen in Figure G.8.



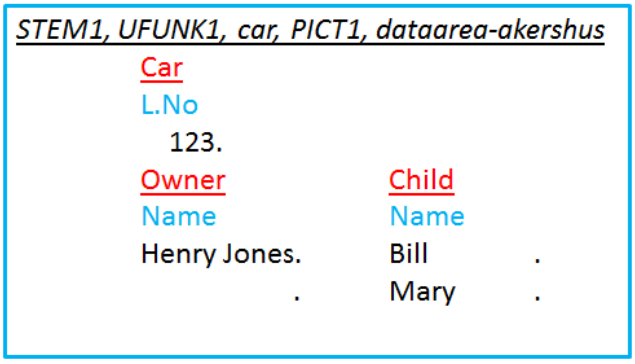


Figure G.8 Example filled in screen picture

Summary

Using the Data transformation architecture, we may create an editor that allows end users to do all kinds of editing in any of the screen pictures. He may do projections and selections. He may do paging and scrolling. He may point at any identifier attribute, and make its entity

become the root of a new screen picture. He may point at a heading and get information from the data dictionary. As these user functions are available in every screen, they need not be specified for any screen. Here we will abstain from discussing the access control.

You may consider Figure G.6 to be an almost complete program, and for each screen picture, you may write only the leftmost Contents part of it. The Distribution part is common for all screen pictures of these External terminology data classes.

Note that the specification in Figure G.6 may be executed line-by-line from top to bottom. There is no loop and no go-to in the Contents population.

Conclusion

We have indicated how to define a database application, using the Data transformation architecture and Existence logic. There is much more to say on how to do this, and what you may get out of it. However, the compactness and clean structure of the specification should be convincing. With a minimal specification, you get maximum functionality.

Our first implementation of these ideas was put into operation on a large and complex database application in 1983. Time is rape for others to do likewise.