Functional object-types as a foundation of complex knowledge-based systems
Lucardie, G.L.

DOI:
10.6100/IR431412

Published: 01/01/1994

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):
Eindhoven: Technische Universiteit Eindhoven DOI: 10.6100/IR431412

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. Jan. 2019
Functional Object-Types as a Foundation of Complex Knowledge-Based Systems

Larry Lucardie
Functional Object-Types as a Foundation of Complex Knowledge-Based Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven.
op gezag van de Rector Magnificus, prof.dr. J.H. van Lint,
voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op donderdag 22 december 1994 om 16.00 uur

door

GEORGE LARRY LUCARDIE

geboren te Rheden
This thesis gives an account of the conceptual and mathematical foundations of knowledge-based systems. As such, it is of interest for computer scientists who work in the field of Artificial Intelligence or Database Technology. Since many results of the research have practical implications, it is also of interest for organisations that already use knowledge-based systems to manage and control their valuable knowledge resources or for organisations that intend to do so in the near future.

I have attempted to make the study self-contained by including elementary definitions when necessary. The only prerequisites are some familiarity with software engineering, especially conceptual modelling, and a certain mathematical maturity.

Many people contributed to this thesis. I would like to thank my supervisors Harrie Timmermans (Technical University of Eindhoven) and Henk Koppelaar (Technical University of Delft) for their stimulating and fruitful comments. Furthermore, I am grateful to the members of the Knowledge-Based Systems Group of the Netherlands Organization for Applied Scientific Research (Building and Construction Research): to Adriaan Huijsing, Cuno Duursma, Hans van Keulen and Johan de Gelder. In many ways they contributed to the results of the thesis. I would also like to acknowledge Carla and Bert Mathlener for preparing the drawings.

Then, I would like to thank Frank Dignum (Technical University of Eindhoven), Paul Hendriks (Catholic University of Nijmegen), Theo Arentze (Technical University of Eindhoven), Ton van der Smagt (Catholic University of Nijmegen) for their valuable comments and sharpening the ideas. Furthermore, I am grateful to Marcel Groosman (Netherlands Organization for Applied Scientific Research) and Jan Stark (Netherlands Organization for Applied Scientific Research) who provided me with the opportunity to spend some time on writing the thesis. I would also like to acknowledge Mandy van Kasteren (Technical University of Eindhoven) and Peter van der Waerden (Technical University of Eindhoven) for their help.

Finally, I am grateful to Jan Baltussen for significantly improving the English. Most of all, I am grateful to his daughter Miriam Baltussen for her unconditional support.
CHAPTER 4: FUNCTIONAL OBJECT-TYPES FOR RECONSTRUCTING KNOWLEDGE UNIVERSA

4.1 THE CONCEPTUAL BASIS OF RECONSTRUCTING AND REPRESENTING KNOWLEDGE UNIVERSA

4.2 BASIC TERMINOLOGY OF CONCEPTS: TERMS, OBJECT-TYPES AND OBJECTS

4.3 VIEWS ON THE NATURE OF OBJECT-TYPES

4.4 FUNCTIONAL OBJECT-TYPES
  4.4.1 The Nominal Definition
  4.4.2 The Real Definition: Context-dependencies
  4.4.3 The Real Definition: Objects
  4.4.4 The Real Definition: Interaction between Object-types and Objects
  4.4.5 Functional Equivalence

4.5 THE CURRENT PRACTICE OF RECONSTRUCTING OBJECT-TYPES

4.6 A FUNCTIONAL EVALUATION OF RECORD-BASED REPRESENTATION FORMALISMS

4.7 CONCLUSION: IMPLICATIONS AND PERSPECTIVES

CHAPTER 5: DECISION TABLES AND PROLOG AS A CONCEPTUAL MODELLING LANGUAGE FOR FUNCTIONAL-OBJECT-TYPES

5.1 INTRODUCTION

5.2 DECISION TABLES
  5.2.1 Formal Background
  5.2.2 Functional Object-types and Decision Tables
  5.2.3 Conclusion

5.3 PROLOG
  5.3.1 Formal Background
  5.3.2 Functional Object-types and Prolog
  5.3.3 Conclusion

5.4 CONCLUSION

CHAPTER 6: THE ADVANCED KNOWLEDGE TRANSFER SYSTEM

6.1 INTRODUCTION

6.2 MAIN FUNCTIONS OF AKTS

6.3 RECONSTRUCTING A KNOWLEDGE UNIVERSE
  6.3.1 Reconstruction Facilities: The Graphical Decision Table Editor
  6.3.2 Validation Facilities: The Integrity Control Sub-System
  6.3.3 Reduction Mechanisms
1.1 BACKGROUND

Knowledge-based systems entered the computing scene in the seventies as the first practical products of the Artificial Intelligence laboratories. Since then numerous knowledge-based systems have been developed in various areas of science, space travel, engineering, business, planning, architecture, banking and medicine. Today, they have attained a permanent and secure role in industrial and commercial settings. In comparison to other technologies such as robotics or vision systems, knowledge-based systems experience a rapid diffusion and uptake in world economy (Hayes-Roth & Jacobstein, 1994).

What is the nature of these systems? Knowledge-based systems are computer programs that embody knowledge to solve problems ordinarily addressed by humans. They derive this problem-solving competence from the knowledge they contain about a domain of discourse. This attribution of the power of knowledge-based systems to knowledge is laid down in the knowledge principle:

'Knowledge Principle: A system exhibits intelligent understanding and action at a high level of competence primarily because of the specific knowledge that it contains about its domain of endeavour.' (Feigenbaum, 1989, p.5)

The contents of the knowledge principle may seem obvious to the reader. Yet, in the sixties the view prevailed that logical structures together with a general purpose reasoning engine form the cornerstone of intelligent action. In contrast, a corollary of the knowledge principle is that reasoning processes are too weak and are not the source power that leads to high level competence. The knowledge principle simply states that if a knowledge-based system is to perform well, it should know a great deal about the world in which it operates. Without knowledge, reasoning will not help. The knowledge principle further indicates that knowledge-based systems are knowledge processing machines that generate solutions to problems which on the one hand require significant human expertise and on the other hand require computer based reasoning with knowledge.

To give a further impression of the significance and nature of knowledge-based systems, we revert to an example of space travel. At the Kennedy Space Centre, NASA engineers developed a knowledge-based system, named LES, to monitor the behaviour of a small but complex control system. The function of this control system is to follow the loading of liquid oxygen onto a space shuttle 6 to 8 hours before takeoff. The space shuttle's fuel tank carries 140,000 gallons of liquid oxygen which are
loaded into the shuttle from a storage tank several miles away through a system of pipes. A battery of sensors along the path keeps engineers informed about the progress of the loading operation. At any time, one of these sensors may indicate an alarm suggesting that some part of the process has failed. As long as nothing is going wrong, LES just 'watches' the dials. At the first sign of trouble LES alerts the operator and creates pictures and schematics including plumbing and electronic circuits from its knowledge (Figure 1). LES also has diagnostic capabilities. For a given sensor discrepancy, for example, it can classify each object involved as being the cause, or incapable of being the cause, or as being suspect.

(Source: Lafferty, 1988, p.19)

(A) Shuttle Display with Rectangular Box around Malfunctioning Area at Centre Bottom (B) Expanded View of Malfunctioning Area

The main motivation to develop LES has been the growing awareness of NASA that knowledge about loading liquid oxygen onto a space shuttle is an essential commodity that should be carefully managed. A failure in this process could lead to the abortion of a shuttle flight with significant loss of money and time. LES is considered beneficial because it improves the availability and the speed of accessibility of knowledge:

'By virtue of their extensive knowledge of the 'wiring diagram' of the loading system, NASA scientists can check other data and can usually make a determination within a few minutes. There are some 200 pages of schematics of the system, which contains some 300 sensors and has thousands of replaceable components. The scientists must quickly isolate the particular segment of the system in question, look over the schematic, check the readings for consistency, and then make their determination. The first problem is speed. However, another problem begins to arise as systems of this kind grow older. The experts who built these mechanisms and who understand
The quotation tells us that LES has been developed to cope with the volume and complexity of knowledge involved in process monitoring and fault location: extensive knowledge is needed about the wiring diagrams, the number of schematics, sensors and the number of replaceable components. Complex knowledge is needed to understand the intricate mechanisms involved. By putting this knowledge at the disposal of operators, LES enables them to accelerate the decision-making process. Between 10 and 40 seconds are currently required for the man-knowledge-based system combination to make a diagnosis which would otherwise take at least a few minutes.

The story of LES is not unique. In general, the prediction is that knowledge-based systems will become vital applications of the workplace in many organisations to improve the availability and accessibility of knowledge (Cohen, 1989, p.22; Smith, 1986, p.3). This expectation is based on the growing recognition that knowledge is an essential commodity for effective and efficient functioning of trade and industry (Beerel, 1987; Wiig, 1988). As a growing number of companies see knowledge, from cost and quality considerations, as an important, but difficult to manage economic factor, we think that knowledge-based systems will become essential tools for improving knowledge management. In this respect, knowledge processing machines are expected to shape the pattern of computer-applications in the nineties and beyond.

1.2 PROBLEM STATEMENT

The most important challenge in the construction of knowledge-based systems that can indeed come up to these high expectations, is to provide these systems with capabilities to process and communicate knowledge in close and transparent interaction with humans. Unfortunately, the technology underlying knowledge-based systems is far from adequate to meet this challenge. Despite growing investments, operational knowledge-based systems are still few and far between. Many projects yield a prototype system and more than once the extension to an operational system appears to be such an extremely difficult step that many systems simply do not come into use.

Why is the development of operational knowledge-based systems so difficult? One cause is that, initially, it always appears so easy that time and other resources are routinely underestimated. Other possible causes are unrealistic expectations, inadequate use of tools and techniques for modelling knowledge, lack of involvement of experts, imperfect collaboration between experts and knowledge engineers, lack of embedding knowledge-based systems in organisational structures and problems with the technical infrastructure of these systems.

This thesis deals with a more fundamental problem: the process of modelling knowledge for implementation purposes. Modelling knowledge is a fundamental phase in the development of knowledge-based systems. In this phase, knowledge
about the application domain is collected, structured and documented. Modelling knowledge is frequently cited as the major bottleneck in building a knowledge-based system (see for instance Clancey, 1985; Neale, 1988; Welbank, 1983). Why is modelling knowledge so difficult? Attributing the difficulties of modelling knowledge to the use of inadequate modelling methodologies and techniques is a step in the right direction. The observation that the difficulties are due to the complex nature of knowledge also contains elements of truth. However, these considerations are rather superficial and do not provide guidelines to deal with the complexities of modelling knowledge.

The goal of this thesis is to identify the main dimensions that underlie the problem of modelling knowledge and to seek solutions to each of them. The thesis reveals the following dimensions:

1. Confusion of knowledge and knowledge representation formalisms
2. Lack of an adequate theory of the nature of knowledge
3. Lack of an adequate formal language

An important claim is that finding the right way of dealing with these dimensions significantly advances our approach towards modelling knowledge and provides a substantial basis for systematically developing knowledge-based systems.

1.2.1 Confusion of Knowledge and Knowledge Representation Formalisms

Confusion of knowledge and knowledge representation formalisms manifests itself by focusing almost exclusively on the pros and cons of a particular representation formalism. One representation formalism offers multiple inheritance and cancellation, another provides demons and rules, another again is considered versatile because of its efficiency in computational processes. Even the process of modelling knowledge is often performed by using a system's representation formalism. This approach leads to many problems: mapping knowledge onto the representation formalism is difficult or impossible, poor validation facilities, bad maintenance and poor explanation facilities. In brief, it can be said that focusing exclusively on knowledge representation formalisms and applying these formalisms to modelling knowledge obstructs our conception of knowledge and unnecessarily complicates its modelling.

Special attention ought to be paid to mathematical logic. The appropriateness of mathematical logic as a representation formalism has been debated by computer scientists for a long time. Opinion on this topic varies. As an example of this Israel (1983) mentions McCarthy's and Minsky's disagreement. McCarthy (1980) believes that a knowledge-based system should reason according to the well-worked-out languages of mathematical logic, whether or not this is the way people actually think, whereas Minsky (1982) is convinced that we should try to get a knowledge-based system imitating the way the human mind works, which is, he thinks, almost certainly not with mathematical logic. McCarthy's and Minsky's conflicting views are not about the usefulness of mathematical logic -on this point they are of one mind- but revolve around the degree of difference between mathematical logic and common-sense
knowledge. Considering the pervasiveness of mathematical logic in many fields of computer science it is obvious that its role should be clearly defined.

Judging from the previous, what we seem to need is a disentanglement of knowledge and knowledge representation formalisms and a clear description of the role of mathematical logic plays. A cohesive proposal to disentangle knowledge from knowledge representation and to allocate a clear role to mathematical logic is Newell's introduction of the knowledge level (Newell, 1981). The knowledge level is a separate computer systems level to define the nature of knowledge. Newell claims that a sharp distinction of the knowledge level and the symbol level (the level of representation formalisms) leads to a comprehensive and consistent view of knowledge and knowledge representation and helps in assigning mathematical logic its proper role.

Newell introduces the knowledge level as a solution coming from the practice of Artificial Intelligence. Before Newell proposed the knowledge level, Woods (1975), for instance, had already written an article entitled What's in a Link?. Woods found that many techniques used in semantic networks are not appropriate for representing knowledge. To expose the confusion of knowledge and knowledge representation formalisms, several other scientists posed the 'What's in a representation formalism' question. Clancey's (1983) question was 'What's in a rule?'. Pointing out the weaknesses of production rules, Clancey observed that a lot of knowledge lies outside the production rule formalism and that appealing to this knowledge enhances the capability for understanding and modifying a knowledge-based system. Brachman's analysis of the definitional capabilities of frames serves similar purposes concerning frames (Brachman, 1985).

From a knowledge level perspective a special role is reserved for mathematical logic. It has been suggested as a basis for a knowledge level analysis to specify what a knowledge-based system does and knows (Clancey, 1985; Kowalski, 1979; Kowalski, 1985; Newell, 1981). Mathematical logic is just viewed as another representation formalism with the extra twist that it is appropriate for the analysis of knowledge. It provides a convenient formalism for studying classical knowledge modelling problems associated with a wide range of knowledge-based systems such as knowledge systems (Walker, 1987), expert systems (Lucas & Van Der Gaag, 1991) expert database systems (Smith, 1986), semantic databases (De Brock, 1989), deductive or logic databases (Das, 1992; Gallaire, Minker, & Nicolas, 1984) and decision support systems. A system of mathematical functions normally specifies or underlies the knowledge of these systems. We call such a formal description of knowledge a knowledge universe.

Distinguishing the knowledge level clarifies a great deal of the controversy between McCarthy and Minsky. McCarthy is right when he stresses the importance of mathematical logic for representing knowledge, but mistakenly thinks that common-sense knowledge can be adequately captured by running a sound theorem prover over mathematical logic. Common-sense knowledge is a knowledge level issue that requires much more than just mathematical logic. Minsky is right when he states that mathematical logic has major weaknesses because it is not the knowledge itself. Minsky, however, is not right, when he tries to clarify an important weakness of mathematical logic by referring to Joe the well-known bird that cannot fly because he is dead, an ostrich, a penguin or because he has his feet set in concrete. The problem
that Joe is viewed as an exception because of being a bird that cannot fly is a knowledge level issue that cannot be solved by mathematical logic, but by a theory of the nature of knowledge. This observation takes us to the second dimension.

1.2.2 Lack of an Adequate Theory of the Nature of Knowledge

Though the distinction of the knowledge level may be helpful, it does not suffice. Newell's description of the knowledge level is a theory describing the knowledge level, but it is not a theory of the nature of knowledge. Newell's theory gives such a theory its proper place in computer science, for the knowledge level is a separate level to define the nature of knowledge for implementation purposes.

A theory of the nature of knowledge should help us to get a clear and consistent view of knowledge. Such a view advances the choice of techniques, methods and methodologies to model knowledge. At this moment, however, there are divergent and often conflicting views of knowledge. An example of such conflicting views can be obtained by comparing the view of Wiederhold (1984) and that of Brachman & Levesque (1986). While Wiederhold makes a sharp distinction between knowledge and data, Brachman and Levesque consider both knowledge and data as types of knowledge. A satisfactory theory of the nature of knowledge that provides us with a clear and consistent view of knowledge and that helps to deal with cases like Joe, the exceptional bird, seems to be missing.

In many theories of the nature of knowledge concepts play an important role as classificatory and storage mechanisms for mathematical functions that define a knowledge universe. A concept has an intension and an extension. The intension of a concept is a set of constraints (or conditions) that should be satisfied by an object to belong to the class covered by the concept. The intension refers to an object-type. The extension of a concept consists of the set of objects complying with the object-type. Object-types and objects are the two sides of the same coin. The notion of concept includes both.

This distinction between object-types and objects helps to characterise what a knowledge-based system does and helps to analyse the problem of modelling knowledge. From LES and other knowledge-based systems, we can abstract that these knowledge processing programs perform a matching task: they all apply constraints (the object-type) to a set of real-world referents (the objects) to obtain a match. An object matches if it can be classified as an object-type.

LES, for instance, contains an object-type which we may call adequate monitoring and fault detection system. The object-type consists of a set of constraints that defines what an adequate monitoring system is and does. The set also includes constraints that enable LES to process an alarm and to propose solutions to the operators. To apply these constraints to the monitoring and fault detection system, which is its only object, LES also needs knowledge about the attributes of the monitoring and fault detection system. An alarm indicates that the actual state of the monitoring and fault detection system is changed. A match means that the monitoring system is indeed a monitoring system that functions well or that an alarm is adequately dealt with.
Another illustration can be taken from robotics. A satisfactory use of a robot in a production environment requires an object-type *adequate robot* consisting of constraints that keep a robot from producing faulty products. To see whether the robot, the object, matches these constraints, we have to know a number of attributes of the robot such as type, velocity and the positioning movements it is capable of executing. A match means that the robot shows the desired behaviour or that deviating behaviour is dealt with.

![Diagram](image)

*Figure 1.2: Matching Object-types and Objects*

Matching is a complex activity (see also Goel, Soundararajan, & Chandrasekaran, 1987). Object-types usually have a complex nature. The same is true for objects (Siebes, 1990). In this thesis, we view the problem of modelling knowledge as the problem of describing or *reconstructing* the complex nature of object-types and objects. We intentionally use the word reconstruction, because a computer scientist has to reconstruct the object-types and objects as they occur in reality. Just as a traffic accident or a crime that happened in reality, are reconstructed, we reconstruct object-types and objects to enable a knowledge-based system to perform matches. Matching requires the use of abstraction mechanisms to adequately describe the complexities of object-types and objects (Figure 2).

Characterising the competence of knowledge-based systems as a capacity to perform matches, explains the recognition that *integrating Artificial Intelligence and Database Technology* constitutes an essential step toward developing knowledge-based systems (Jarke & Vassiliou, 1984; Murdoch & Johnson, 1990; Smith, 1986). This recognition is mainly fuelled by the idea of complementarity, i.e. the idea that Artificial Intelligence concentrates on complex object-types, whereas the focus of Database Technology lies with complex objects. For some time now, this recognition has been inducing major research activities exploring the relationships between both computer technologies.

These activities, however, suffer from the same confusion mentioned before. The main thrust of research concentrates on transferring data structures and processing techniques from Artificial Intelligence to Database Technology or vice versa. The use of records to store production rules (Herwijnen, Houten, Houtsma, & Romkema, 1990), the use of frames to store relational data (Chow, 1987), or the addition of rule processing algorithms to databases (Stonebraker, 1984) are all examples of research conducted at the symbol level.
Recently, many researchers from both computer sciences apply Newell's distinction and emphasise the appropriateness of studying the relationships between Artificial Intelligence and Database Technology at the knowledge level (Brachman & Levesque, 1986; Kent, 1979; Twine, 1989). A knowledge level integration explicitly focuses on the semantic structure of a knowledge-based system and does not bother about representation formalisms or systems that are well-known in Artificial Intelligence and Database Technology. The concern is explicitly with the knowledge of a knowledge-based system.

Even when one follows a knowledge level strategy, the reconstruction (modelling) of object-types and objects presupposes a well-founded theory. Since the existence of objects closely depends on object-types (Martin & Odell, 1992), these theories concentrate on the complexities of object-types. Several basic theories on how to reconstruct the object-type of a concept can be distinguished. Characteristic for the classical theory is the presupposition that an object-type consists of a univocal set of necessary and sufficient conditions. Because classification of objects as object-types is less univocal and more complex than the classical theory accounts for, alternative approaches have been developed. One of them, the probabilistic theory, subscribes to the classical idea that an object-type is a set of sufficient and necessary conditions, but exclusively on a theoretical level. The probabilistic approach assumes that all sorts of random disturbances at the empirical level cause problems in the delineation of the extension (fuzzy sets). By utilising mathematical measures of similarity between objects defined over an essentially apriori given set of attributes, the probabilist tries to eliminate the random disturbances, so that univocal criteria can be proved to underlie the fuzzy extension at the theoretical level (Stepp & Michalski, 1986, p.4).

In the prototypical or stereotypical theory, object-types are described by means of a prototype. A prototype shares many attributes of objects so it reflects a central tendency category of objects. The description of a prototype consists of so-called necessary conditions. Since no object will satisfy all the necessary conditions, the question whether an object belongs to the extension of an object-type depends on the degree of resemblance with the prototype. Probabilistic and prototype conceptualisation methods have much in common and prevail in Artificial Intelligence (Brachman, 1985) and Database Technology.

In the theory of functional classifications, the reconstruction of an object-type takes place through a goal- or function-oriented process in which functional equivalence constitutes the basis for classification (Hendriks, 1986; Lucardie, 1992; Reitsma, 1990; Van Der Smagt, 1985; Van Der Smagt & Lucardie, 1991). The central notion is that at a theoretical level we cannot univocally define an object-type. In this respect the functional theory corresponds with the probabilistic and prototype theory. What is different is that the functional theory offers a totally different explanation of fuzziness. In contrast to the probabilistic and prototype theory, the functional theory emphasises that fuzziness has a systematic character. The solution of fuzziness is sought neither in the elimination of random disturbances (such as measuring errors), nor in the comparison of objects with a prototypical object-type. The functional solution is typified by the systematic identification of several object-types. These object-types originate through functional equivalence: the phenomenon that objects, possibly differing in many respects, are equivalent in achieving a nominally specified
function in a certain context. This leads to the rejection of abstracting object-types on the basis of extensions and of the possibility of describing an object-type through stereotypes.

We argue that the theory of functional classifications offers promising perspectives to improve the development of knowledge-based systems, because it accounts for fundamental concerns of reconstructing object-types and objects. This theory of the nature of knowledge requires a formal language that supports the functional reconstruction of object-types and objects. This takes us to the third dimension.

1.2.3 Lack of an Adequate Formal Language

The specification of the knowledge of a knowledge-based system in a knowledge universe or a conceptual model requires a formal language capable of modelling the complexities of application-domains according to the functional theory. The question is what language to choose.

Mathematical logic is an obvious candidate language. It permits unambiguous description of knowledge. It has the disadvantage, however, that for many people it is too complex as a result of which only restricted modelling and validation is possible. Furthermore, mathematical logic does not tell us how to model functional object-types nor does it offer facilities to simulate their behaviour.

So, the language to be selected should display the strong points of mathematical logic without displaying its weak points. Such a modelling language should further provide appropriate facilities for validating a conceptual model as well as for simulating its behaviour (Loucopoulos & Karakostas, 1989; Twine, 1989).

Some authors claim that the joint application of Decision Tables and Prolog meets these requirements, because both have a firm basis in mathematical logic and together offer a range of powerful formalisms and techniques that allow a formal unambiguous description of real-world phenomena that is close to natural understanding (Reilly, Salah, & Yang, 1987, p.30).

1.3 RESEARCH ISSUES

The characterisation of modelling knowledge as an activity of reconstructing complex object-types and complex objects, explains the need to integrate Artificial Intelligence (to deal with complex object-types) and Database Technology (to deal with complex objects). The three previously described dimensions that interfere with the reconstruction of object-types and objects lead to the following research issues:

1. What are the advantages of integrating Artificial Intelligence and Database Technology at the knowledge level compared to symbol level integration? Does a knowledge level integration contribute to the process of modelling knowledge?
II. Does the theory of functional classifications constitute a conceptual advance for realising a knowledge level integration, and, if so, in what respects? Does the theory help to define object-types and objects to enable knowledge-based systems to perform matches?

III. Can the joint application of Prolog and Decision Tables be considered as an adequate knowledge level language for describing functional object-types? That is, does it comply with the general requirements applied to modelling languages and the particular requirements for functional object-types?

Perhaps the most difficult part of this thesis is to show that the analytical framework offered by functional object-types provides a substantial basis for developing knowledge-based systems and making them suitable for application communities from physicians to geneticists and civil engineers.

1.4 OUTLINE

Each research issue is tackled in several chapters.

The first research issue concerns the value of a knowledge level integration of Artificial Intelligence and Database Technology relative to a symbol level integration. It requires reviewing the notion of computer systems levels with special emphasis on the implications of distinguishing the knowledge level from the symbol level (Chapter 2). Subsequently, we characterise symbol level strategies and discuss associated disadvantages of following them. To clarify knowledge level strategies, we introduce a system of mathematical functions that assesses knowledge in a knowledge universe. The research issue is rounded off with an exemplification of a knowledge level integration: a view of databases from the knowledge level (Chapter 3).

The second research issue concerns the value of the theory of functional classifications as a pivot for a knowledge level integration of Artificial Intelligence and Database Technology. It requires explaining the role of concepts as organisation principles for a knowledge universe and describing their main components: terms, object-types and objects. The evaluation of the theory of functional object-types takes place by comparison with the prevailing prototypical and probabilistic theories. We argue that, compared to these competing theories, the theory of functional object-types offers promising perspectives, because it accounts for fundamental concerns of reconstructing functional object-types (and objects) and constitutes a knowledge level integration of Artificial Intelligence and Database Technology (Chapter 4).

The third research issue concerns the value of the integrated application of Decision Tables and Prolog as a formal language for functional object-types. It requires examining the potentials of Decision Tables and Prolog as two related and complementary conceptual modelling languages for formalising functional object-types. This examination is based on the formal background of Decision Tables and Prolog (Chapter 5). Next, we report on the Advanced Knowledge Transfer System.
(AKTS) that offers extensive facilities for working with Decision Tables and Prolog (Chapter 6).

A case-study in the domain of the chemical degradation of brick masonry walls serves as an example of developing a knowledge-based system according to the theory of functional object-types using AKTS (Chapter 7). An epilogue ties up the thesis. It contains the principal conclusions, the relation between functional object-types and quantitative object-types and the potential implications of (reconstructing) functional object-types including the role of AKTS (Chapter 8).
CHAPTER 2

THE KNOWLEDGE LEVEL

2.1 INTRODUCTION

Before ascertaining the advantages of a knowledge level integration of Artificial Intelligence (AI) and Database Technology (DBT), the first research issue of the thesis, we must describe the knowledge level and ascertain its utility. Newell introduced the knowledge level in his presidential address to the American Association of Artificial Intelligence (AAAI) as a new and deviating computer systems level (Newell, 1981).1

The impetus to propose the existence of the knowledge level was threefold. First, Newell points to the emphasis on knowledge representation issues in AI as if they were the real locus of intelligence. Second, Newell mentions the stereotype that logic is not useful for AI. This stereotype, Newell explains, is a consequence of the disappointment caused by a machine-oriented formulation of first-order logic called the resolution principle, which proved not powerful enough to display the expected intelligent behaviour. Third, Newell observes that there is a web of conflicting opinions about knowledge representation. He refers to a Special Issue on Knowledge Representation by Ron Brachman & Brian Smith (Brachman & Smith, 1980) in which the answers of a large questionnaire on knowledge representation issues were analysed. The main result published in the Special Issue displayed an overwhelming diversity of opinions in which no consensus on any substantial knowledge representation issue could be found.

According to Newell, these three items -too much attention on knowledge representation, the minor role of logic and the lack of consensus on any question of substance in knowledge representation- sufficiently indicate that the prevailing views in AI of knowledge and knowledge representation are inadequate. Newell claims that the distinction of the knowledge level leads to a comprehensive and consistent view of knowledge and knowledge representation and helps to assign to mathematical logic the role it deserves.

The main objective of this chapter is to investigate this claim. As the knowledge level is a computer systems level, the first thing to do is to review the notion of computer systems levels (Section 2.2). Then, we characterise the knowledge level as a computer systems level with specific, deviating properties (Section 2.3). Next, we build on this characterisation and ascertain the advantages and disadvantages by discussing research done at the knowledge level (Section 2.4). Finally, we articulate a

---

1 Also published in Artificial Intelligence 18 (1982), pp.87-127.
number of conclusions regarding the utility of the knowledge level for integrating AI and DBT (Section 2.5).

2.2 THE NOTION OF COMPUTER SYSTEMS LEVELS

The notion of computer systems levels occurs through computer science with varying degrees of utility and precision (see for instance: Tanenbaum, 1976). A useful and precise stratification of computer systems levels is the one introduced by Bell & Newell (1971). A computer systems level in their stratification can be described as follows:

'A level consists of a medium to be processed, components that provide primitive processing, laws of composition that permit components to be assembled into systems and laws of behaviour that determine how system behaviour depends on the component behaviour and the structure of the system.'

(Newell, 1981, p.5)

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Register-Transfer Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td>Digital Systems</td>
</tr>
<tr>
<td>Medium</td>
<td>Bit Vectors</td>
</tr>
<tr>
<td>Components</td>
<td>Registers</td>
</tr>
<tr>
<td>Composition Laws</td>
<td>Transfer Path</td>
</tr>
<tr>
<td>Behaviour Laws</td>
<td>Logical Operations</td>
</tr>
</tbody>
</table>

(Source: Newell, 1981, p.5)

*Figure 2.1: Defining Aspects of the Register-Transfer Level*

Examples of computer systems levels in Bells & Newell's stratification are the *device level*, the *circuit level*, the *register-transfer level* and the *symbol level*. Figure 2.1 shows the aspects of the register-transfer level. The medium to be processed is bit vectors, the components consist of registers. The relations between the registers come into existence by transfer paths and the logical operations that operate upon the bit vectors in the registers (for instance: send register 4 bits 14 to 18). In this way, the aspects of the register-transfer level form a digital system.

Another computer systems level that lies above the register-transfer level is the symbol level. Figure 2.2 shows the aspects of the symbol level. Now, the medium to be processed is not bit vectors but symbolic expressions and processes. The components are memories which are connected by associative laws. The behaviour of this symbol system can be characterised as a problem solving process.

There are many instantiations of each level, e.g. many circuit systems at the circuit level, many digital systems at the register-transfer level and many symbol systems at
the symbol level. Examples of symbol systems are rule-based systems, record-based systems, frame-based systems and so on.

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Symbol Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td>Symbol System</td>
</tr>
<tr>
<td>Medium</td>
<td>Symbolic Expressions plus Processes</td>
</tr>
<tr>
<td>Components</td>
<td>Memories: Operations</td>
</tr>
<tr>
<td>Composition Laws</td>
<td>Designation; Association</td>
</tr>
<tr>
<td>Behaviour Laws</td>
<td>Problem Solving Process</td>
</tr>
</tbody>
</table>

(Source: Newell, 1981)

Figure 2.2: Defining Aspects of the Symbol Level

The definition of each level can take place in two ways. First, each level can be described autonomously without any reference to any other level. Secondly, we can reduce each level to the level below it. Every aspect of a level, medium, components, laws of composition and laws of behaviour, can be defined in terms of systems below them. In Section 2.5 such a reduction is described.

Computer systems levels vary in each of the aspects, but share common features. Newell mentions four common features (Newell, 1981, p.5). First, the specification of a system at a level always completely determines a definite behaviour for the system at that level (given initial and boundary conditions). Second, the behaviour of the total system results from the local effects of each component of the system processing medium as its input to produce its output. Third, the immense variety of behaviour is obtained by system structure, by the variety of ways of assembling a small number of component types (though perhaps a large number of instances of each type). Fourth, the medium is realised by state-like properties of matter, which remain passive until changed by the components. These common features are necessary to understand the special character of the knowledge level.

2.3 THE KNOWLEDGE LEVEL

Bell & Newell’s stratification is extended by Newell (1981). Newell proposed a new and distinct computer systems level lying just above the symbol level which he called the knowledge level (Newell, 1981). The knowledge level is a true computer systems level, but it has a number of deviating properties.

The introduction of the knowledge level is primarily intended to have a separate level to define the nature of knowledge. The abstract definition of knowledge that is often employed views knowledge as a competence to select actions for realising goals (Newell, 1981; Schreiber, 1992; Stepp & Michalski, 1986). This competence is accomplished by an intelligent system residing at the knowledge level. This system,
called the agent, is composed of knowledge. The structure of an agent is extremely simple. First, the agent has a set of goals. The set of goals is just another form of knowledge with the distinction that an agent explicitly strives to realise these goal components. Second, the agent has a set of actions out of which it will choose according to the Principle of Rationality. Third, the agent has knowledge relating goals to actions.

From this abstract definition of knowledge, it follows that relations between goals and actions are the basic components of knowledge. The Principle of Rationality structures the relations between goals and actions and controls the agent's behaviour. More precisely, it is formulated as follows:

'Principle of Rationality: If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.' (Newell, 1981, p.8)

The law is a simple form of rationality referring to an action leading to a goal. It will only be effective under certain simple conditions. In many situations the Principle of Rationality is not suited for determining behaviour. Auxiliary principles can cover some of these situations. For instance, in case multiple actions are involved the Principle of Rationality is extended with the auxiliary Principle of Equipotence of Acceptable Actions asserting that every action leading to a goal is equally acceptable from the viewpoint of the goal itself. A second auxiliary principle, covering situations in which actions are connected to multiple goals, is the Preference of Joint Goal Satisfaction: 'For given knowledge, if goal G1 has the set of selected actions \{A1.i\} and goal G2 has the set of selected actions \{A2.j\}, then the effective set of selected actions is the intersection of \{A1.i\} and \{A2.j\}.' (Newell, 1981, p.9).

Notwithstanding the utility of these extensions, on many occasions, Newell states that they fail to provide effective guidelines for predicting the agent's behaviour. Even after the introduction of other auxiliary principles accounting for goal preferences, risk and uncertainty, the elementary extensions of the central Principle of Rationality are not sufficient to cover all situations. Knowledge level models describing the environment, do not contain encompassing principles which state that multiple goals need to be compatible or which solve incompatibility. The failure to determine behaviour uniquely, the probabilistic elements and the incapability to describe the entire range of behaviour all indicate that knowledge level models are approximations of reality. In this respect the behaviour of the knowledge level deviates from other computer systems levels which display a deterministic behaviour. The usefulness and theoretical status of these knowledge models regarded as approximations are discussed in Chapter 4.

Figure 2.3 shows the defining aspects of the knowledge level. As might be expected, the agent is the system and knowledge is the medium at the knowledge level. The components are goals, actions and bodies. Bodies constitute the relations between goals and actions. The law of behaviour is the Principle of Rationality.

The knowledge level is reducible to the symbol level. In this sense, it is a true systems level. But we can single out characteristics of the knowledge level that deviate significantly from the common features of computer systems levels. First, the
agent is not a deterministic machine. Many situations are not describable at the knowledge level. Thus, as emphasised before, it is an essential aspect of this level to accept the presence of probabilistic elements. Second, the behaviour of the agent contrasts with the behaviour of systems at other levels. At the knowledge level, there is no composition law that connects the components. The behaviour of the agent is governed by one central principle whereas the behaviour of systems at other levels is determined by the local processing of components. Third, at the knowledge level there is a complete absence of significant structure in the agent. Fourth, knowledge is not realised by state-like properties of matter, but remains an abstract competence-like notion.

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Knowledge Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td>Agents</td>
</tr>
<tr>
<td>Medium</td>
<td>Knowledge</td>
</tr>
<tr>
<td>Components</td>
<td>Goals; Actions; Bodies</td>
</tr>
<tr>
<td>Composition Laws</td>
<td>None</td>
</tr>
<tr>
<td>Behaviour Laws</td>
<td>Principle of Rationality</td>
</tr>
</tbody>
</table>

(Source: Newell, 1981)

**Figure 2.3: Defining Aspects of the Knowledge Level**

The third point needs additional explanation. It appears that the absence of significant structure in the agent causes confusion among researchers as it seems to conflict with the reconstruction of knowledge level models. For a good understanding, please note carefully that the third point does not state that there is no structure at the knowledge level, but that the agent's components, goal, actions and bodies, are structurally identical to each other! The agent may lack a significant structure, but the knowledge contained by the agent indeed has structure and is positively open to modelling activities: knowledge level descriptions are not about structures in the agent but about structures of knowledge!

An indispensable part of Newell's proposal is the existence of a symbol level. Knowledge, viewed from a knowledge level perspective as a competence to select goal-related actions, is reduced at the symbol level to structures and processes. At the symbol level, data structures with particular properties and associated processes carry out problem solving to realise a goal-oriented functionality. The data structures contain knowledge and the processes provide access to the captured knowledge. The following well-known equation illustrates the idea of a representation formalism:

\[
\text{Representation} = \text{Knowledge} + \text{Access}
\]

Newell (1981, p.14) explains this equation as follows:
'The representation consists of a system for providing access to a body of knowledge, i.e. to the knowledge in a form that can be used to make selections of actions in the service of goals.'

The symbol level describes how knowledge processing systems arrive at actions by searching problem spaces and databases. There are a lot of symbol level systems, such as frame-based systems, rule-based systems, record-based systems and graph systems. Each system has its own data structures and operations acting upon them.

Figure 2.4 shows the situation that involves an observer and an agent. The observer treats the agent at the knowledge level, i.e. ascribes knowledge of goals, actions and of the relation between goals and actions to it. What the agent really has is a symbol system for reasoning of what actions it will take. The picture is not as trivial as it may seem at first sight. It points out the importance of understanding the interaction between the capabilities of humans and knowledge-based systems and helps to configure the dyad of the observer and the agent in such a way that an optimal functionality of the man-knowledge-based system interaction can be attained. An explicit decision on the division of tasks is helpful to effectively exploit the specific capabilities of humans and knowledge-based systems. Chapters 6 and 7 exemplify these types of decisions.

2.4 DISCUSSION

So far, we have described the knowledge level as a computer systems level that allows us to deal with knowledge independent of particular symbol level systems. The knowledge level brings some clarity in the relation between knowledge and the symbol level systems representing that knowledge. It disentangles what a knowledge-based system knows from how it knows this. Our initial description of the knowledge
level seems to confirm Newell's claim that the identification of the knowledge level leads to consistent and comprehensive views of knowledge and knowledge representation and helps to assign to mathematical logic its proper place.

For further investigation of this claim, we have to look at how other computer scientists view Newell's proposal and deal with it. Newell's theory has, on the one hand, stimulated many scientists to turn the knowledge level to technical use (see for instance: Berg-Cross & Price, 1989; Brachman & Levesque, 1986; Bylander & Chandrasekaran, 1988; Clancey. 1983; Clancey, 1985; Clancey, 1992; Levesque, 1984; Schreiber, 1992; Steels, 1990; Steels, 1992), the knowledge level has, on the other hand, also been criticised, for instance, by Fox (1986) and Sticklen (1989).

2.4.1 Claimed Advantages of the Knowledge Level

Computer scientists advocating the knowledge level point to a number of advantages referring to the (a) reconstruction of knowledge level models, (b) assessment of the role of representation formalisms, (c) evaluation and comparison of representation formalisms and finally to the (d) specification of the role of mathematical logic.

A. Reconstruction of Knowledge Level Models

Modelling knowledge refers to a fundamental phase in the development of knowledge-based systems in which knowledge about the application domain is collected, structured and documented. A knowledge model is of extreme importance. Its purpose is to contain a complete description of what a knowledge-based system does and serves as a basis for the design and implementation of a knowledge-based system. Levesque (1984) formulates the advantage of reconstructing knowledge level models as follows:

"In terms of system design, the main reason for distinguishing between the knowledge level and the symbol level, is to allow the functionality of a system to be treated independently of its symbolic implementation. In particular, it allows us to consider new operations on a knowledge base (that can be explained in terms of existing ones) without necessarily committing ourselves to any particular implementation style.'

(Levesque, 1984, p.206)

The citation underlines the advantage related to modelling knowledge without being biased by symbol level issues which can confuse the views of knowledge. In general, having a clear view of knowledge is experienced as a distinct advantage of the knowledge level. Brachman & Levesque (1986, p.77) and Twine (1989, p.125) also emphasise that a knowledge level analysis leads to knowledge level models, free of inadvertent implementation biases.

Other scientists draw attention to the advantages related to the availability of such implementation-free models. Mesequer (1992) uses knowledge level models for
validation and design purposes. According to Steels (1992) and Clancey (1983; 1985) a knowledge level perspective permits the development of deep models enabling the realisation of robust systems. These systems are capable of providing -when compared with explanation facilities in traditional systems which are simple replays of used data structures- more subtle and sophisticated justifications of conclusions. David & Krivine (1990) have shown this by improving reasoning facilities through the construction of knowledge level models. Another example of enhanced explanations by using knowledge level models is provided by Neches, Swartout, & Moore (1985).

What to think of these advantages? We can agree on the advantage of implementation-free knowledge level models. We also commit ourselves to the advantages related to the availability of such models. However, we miss the observation mentioned in research at the knowledge level that the knowledge level is a separate level that allows us to define the nature of knowledge. We miss serious attempts to define this nature of knowledge.

B. Role of Representation Formalisms
Smith (1982) made a contribution to assessing the role of knowledge representation components in his knowledge representation hypothesis:

'Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge the overall process exhibits, and b) independent as such semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.' (Smith, 1982, p.33)

Smith's hypothesis indicates that knowledge representation components of a knowledge-based system have to comply with two constraints: (a) it must be possible to interpret them as propositions representing the knowledge of a knowledge-based system so that we are able to assess their knowledge level import and (b) the components must play a causal role in the intelligent behaviour of a knowledge-based system. This causal role should agree with our understanding of these components as propositions representing knowledge. The behaviour of the knowledge-based system should be understandable as if the knowledge-based system believes these propositions. A critical component of a knowledge-based system is a set of representation structures that we can interpret as sentences representing what the system knows.

It is worth noticing in Smith's hypothesis that the two constraints imposed are in potential conflict with each other. The constraint that a representation formalism should have a causal effect in the behaviour of a knowledge-based system seems reasonable. However, it may conflict with the constraint that it must be possible to assess the knowledge level import of a representation formalism. Production rules, for instance, have as causal effect in the behaviour of a rule-based system, but are not
easy to interpret. A good understanding of what knowledge is present in the set of production rules, requires a well-reconstructed knowledge level model. This model should be described by means of a formalism that facilitates the assessment of its knowledge level import. In Chapter 5 we extensively discuss a formalism to understand the knowledge level import.

C. Evaluation and Comparison of Representation Formalisms
Furthermore, the knowledge level provides a vantage point to compare and critically examine the properties of representation formalisms. These comparisons and examinations should relieve developers of computer systems of the burden of having to mould each problem to suit their implementation tools. They continue to hold no matter what symbol level decisions are made. Another advantage of the knowledge level is its use as a vantage point to evaluate and compare representation formalisms (Brachman & Levesque, 1986). Chapters 3, 4 and 7 give examples of evaluations and examinations of representation formalisms from a knowledge level perspective.

D. Role of Mathematical Logic
There has been much arguing about the role of mathematical logic in problem solving. McCarthy (1980) believes that a knowledge-based system should reason according to the well-worked-out languages of mathematical logic whereas Minsky is convinced that we should try to get a knowledge-based system imitating the way human minds work which, he thinks, is certainly not with mathematical logic.

The distinction of the knowledge level brings some clarity into this area. From a knowledge level perspective mathematical logic is a representation formalism similar to many other formalisms with the twist that it is especially appropriate for the analysis of knowledge. We can not only apply mathematical logic directly in the reconstruction of knowledge level models, but we can also use mathematical logic to analyse the knowledge level content when using other representation formalisms. The latter is possible, because mathematical logic can also be used to model other representation formalisms (see Chapter 7). In this sense, mathematical logic is not just a representation formalism.

Whether knowledge represented through mathematical logic is really comparable to the way the brain works is beside the point. In many respects knowledge imparted to a computer differs from knowledge in human brains. Working at the knowledge level we refer to knowledge in computers. Thus we do not have to pursue psychological realness, but are satisfied with the artificial knowledge present in our knowledge-based system.

2.4.2 Claimed Disadvantages of the Knowledge Level

Newell's claim has not gone unchallenged. It has attracted Fox's (1986) and Sticklen's (1989) criticism. Fox's main objection is that the knowledge level does not provide an adequate foundation to study the competence of distributed or multi-agent systems,
because it focuses on a single agent. Fox grounds his observation on an examination of three knowledge-based systems to identify their database requirements: Rl for computer configuration, ISIS for job-shop scheduling and Callisto for engineering project management. These systems were found in need of (1) access to one or more databases by a single knowledge-based system (2) one or more knowledge-based systems accessing the same database, and (3) multiple knowledge-based systems co-operating to solve a single problem.

(Source: Fox, 1986, p.460)

Figure 2.5: Defining Aspects of the Organisational Level

Fox is convinced that problems requiring distribution of problem solving, for instance when multiple, conflicting goals are involved, cannot be solved at the knowledge level. Therefore, he introduces the organisational level. The defining aspects are displayed in Figure 2.5.

The system at the organisational level is an organisation. The components are agents or other organisations. In Fox's view, agents can be split up, for instance, into decision-making agents and agents that only retrieve information. The medium to be processed consists of transactions and the law of composition is formed by contracts. There are many behaviour laws. Finally, there is another aspect that measures the costs involved in realising goals.

What to think of Fox's proposal and his criticism of the knowledge level?

According to Fox, the aspects defining the organisational level for problem-solving competence of multi-agent systems are needed to deal with issues concerning the:

- Impact of knowledge partitioning
- Impact of incomplete and inconsistent knowledge
- Relationship between the number of agents and problem solving competence
- Impact of conflicting goals
- Impact of resource contention

If we view Fox's approach as a signal that a smooth communication between agents in a multi-agent system requires knowledge that should also be modelled, we agree. We note, however, that modelling knowledge to arrange, for instance, communication paths is a regular knowledge level issue. In our view, the distinction of a separate
level on top of the knowledge level adds nothing. There is no reason to deal with the partitioning of knowledge or the impact of multiple goals at a new level. Rather, the occurrence of multiple goals and other issues related to multi-agent systems, indicates that the Principle of Rationality and Newell's other first principles have to be extended. These extensions should form a theory of the nature of knowledge. One should not forget that the knowledge level is explicitly meant to provide a separate level to define the nature of knowledge. Newell's first principles only give the initial impetus to the formulation of such a theory. Thus, we prefer working on a theory of the nature of knowledge in stead of distinguishing additional levels.

Sticklen's criticism (1989) is more fundamental. A central objection is that the identification of the knowledge level supports the retrospective analysis of the behaviour of a problem-solving agent, but does not help to yield predictive and empirically verifiable statements about that behaviour. Sticklen exemplifies his objection by referring to Clancey's knowledge level description of heuristic classification (Figure 2.6). Sticklen concludes that heuristic classification is unable to yield verifiable predictions of the problem-solving behaviour of an agent.

![Figure 2.6: (A) Heuristic Classification (B) Example of Heuristic Classification](image)

The second component of Sticklen's argument is the observation that each well-established scientific theory must make verifiable predictions. This view is, as Sticklen states, broadly accepted within the scientific community. Subsequently, Sticklen indicates two ways of making such predictions and exemplifies them with examples from physics. First, predictions can be obtained by working out a closed form solution of some mathematical expression. Sticklen refers to Newton's gravitation theory which helps to predict the forces that two point masses will exert on each other and the trajectories the masses will follow. Second, predictions can be made by using mathematical equations which have no closed form solutions, as a basis for a numerical simulation. The results of a numerical simulation are predictions. Sticklen points out there are two types of knowledge necessary to obtain this second kind of prediction: knowledge of the mathematical equations and knowledge of how to perform a numerical simulation based on these equations.
On the basis of these two components -Clancey's knowledge level description and the view that a theory must make verifiable predictions- Sticklen draws the conclusion that the knowledge level as it currently stands is incomplete.

![Figure 2.7: Multiple Heuristic Matches](image)

How to react to Sticklen's criticism in the light of our description of the knowledge level? When we go through Sticklen's attack on a heuristic classification in the MYCIN system, it is rather surprising that the scheme (Figure 2.7) showing that immunosuppressed could be indicative for the disease-category gram-negative infection but also for leukaemia without giving a decisive answer on which inference we should expect in a certain problem-solving situation, brings Sticklen to the conclusion that Newell's knowledge level cannot make predictions and therefore is incomplete. The conclusion surprises us for three reasons.

Firstly, the heuristic classification scheme is no more than an abstract display of knowledge present in the MYCIN-system. It is imaginable that if the details of the knowledge were displayed instead of the abstract scheme, it would be quite possible to see under what circumstances immunosuppressed leads to gram-negative infection or to leukaemia. Criticising the knowledge level, because it does not engender predictions of the behaviour of an agent, seems like looking at a graphical display of a database scheme (for an example see Chapter 7) and concluding that database models are incomplete, because many types of relations cannot be expressed. Just as a graphical display of a database scheme should omit relations, the abstract display of the heuristic classification example should omit certain details, too.

Secondly, Sticklen considers Clancey's approach as a typical knowledge level analysis that 'is pervasive across knowledge-based systems' (Sticklen, 1989, p.234). From the perspective of many knowledge level adepts this may be true, but from our perspective it is disputable whether Clancey's heuristic classification is indeed representative of research at the knowledge level. Newell is very clear when he states that the knowledge level is a separate computer systems level to define the nature of knowledge. Newell himself formulated the first principles that can lead to such a theory: the Principle of Rationality and extensions to it such as the Principle of Joint Goal Satisfaction and the Principle of the Equipotence of Acceptable Actions. The question is whether Clancey's heuristic classification can be understood as a theory of the nature of knowledge. Evidently, heuristic classification provides a useful
methodology for modelling knowledge, but does it, for instance, explain the complexities of knowledge, does it thoroughly deal with the fundamental nature of abstractions such as generalisations, specialisations and aggregations and associations, does it add important principles which are in accordance with the Principle of Rationality? Chapter 4 presents a theory of the nature of knowledge and deals with these questions.

Thirdly, the knowledge level is a framework for developing theories of the nature of knowledge and for exclusively dealing with knowledge and it is not meant to make predictions. A theory of the nature of knowledge helps to develop a concrete knowledge level model of a certain domain (Lucardie, 1992). A knowledge level model can be constructed to realise predictions in accordance with the first principles formulated by Newell. If the model is valid and reliable, the agent will be capable of making predictions. If a knowledge level model cannot predict reality, it is just a reflection of human capabilities. Just like us, a knowledge-based system cannot predict which team will win a football game. This is not a critical note addressed to the knowledge level, but rather it is a fact of reality that we have to accept. It seems that Sticklen does not distinguish between the knowledge level, a theory of the nature of knowledge and a concrete knowledge level model constructed according to a theory. The question now is: what is the object of Sticklen's attack: (1) the mere conception of a separate computer systems level that exclusively deals with knowledge, or (2) the first principles described by Newell that form an initial impetus to a theory of the nature of knowledge or (3) knowledge level descriptions he knows about such as the Clancey's heuristic classification or some combination of these three items?

Sticklen's remark that a scientific theory is commonly accepted to make empirically testable predictions contains elements of truth. However, Sticklen does not inform the reader what empirically verifiable precisely means. It is rather surprising that Sticklen, while unfolding his argumentation, does not refer to the scientific research of the Logical Positivists and simply passes by essentially epistemological discussions. The Logical Positivists (1925-1936) were empiricists who built their scientific theories on the Verification Principle: to understand a non-logical and non-mathematical statement that $p$, means in principle to be able to specify under what empirical circumstances $p$ is true, that is to be able to specify how $p$ can be empirically verified (the Logical Positivists recognised that logical and mathematical statements are analytical, have an a priori character and do not have any empirical pretensions; therefore, these statements are not subject to the Verification Principle). Sticklen has no eye for the problems the logical positivists encountered because of the Verification Principle. Even they could not maintain the Verification Principle and had to adjust it into the Confirmation Principle which is less constraining.

Another point Sticklen skips and that, all the same, is important for his argumentation is that explaining and predicting have the same logical structure for the logical positivist. The formal structure of the Hempel-Oppenheim explanation model (Hempel & Oppenheim, 1948) can be used to illustrate this. The Hempel-Oppenheim model consists of explanans sentences and explanandum sentences. The explanans sentences contain at least one lawlike statement and a number of preconditions. The
explanandum sentence describes an individual event. The explanandum sentence can be inferred from the explanans sentences. For example:

1. heat causes iron to expand (explanans sentence 1: lawlike statement)
2. this wire is made of iron (explanans sentence 2: empirical precondition 1)
3. this wire is heated (explanans sentence 3: empirical precondition 2)
4. this wire is expanded (explanandum sentence)

The structural identity of this explanation model with a prediction model is remarkable: if we can logically infer the explanandum sentence from the explanans sentences, then we can, if we know in advance the lawlike statements and the empirical preconditions, predict the explanandum, on the same deductive basis. The same remarks can be made for other explanation models such as the inductive-statistical model (Hempel, 1962). Note that accepting the Hempel-Oppenheim model and the inductive-statistical model does not automatically imply a logic positivist position. Karl Popper described and accepted the model, though he was one of the greatest opponents of the logic positivists.

If Sticklen does have reasons to distinguish prediction from explanation, he should make them explicit. If he does not, his own remarks would also refer to retrospective explanations at the knowledge level. Then, Sticklen might observe that the heuristic match step from immunosuppressed to leukaemia lacks explanatory power because it has little utility for assessing under what circumstances this is true.

The great interest Sticklen attaches to verifiable predictions seems inversely proportional to the profoundness with which he handles the notions of verification and prediction. Furthermore, Sticklen easily skips essential scientific epistemological issues such as verification, confirmation, corroboration, falsification and the occurrence of several types of facts that are related to verification and prediction.

Apart from these epistemological issues, if Sticklen intends to test the validity of Newell's proposal as a scientific theory, he should not test the possibility of making predictions, but test the validity of Newell's claim: he should examine whether we are indeed able to distinguish knowledge from its representation formalisms and whether the role of representation formalisms and mathematical logic is defined more clearly by using a knowledge level perspective. Furthermore, he should test the tenability of the first principles of Newell. This would be a very fruitful way of working. Unfortunately, Sticklen hangs his entire criticism on the capacity to make predictions.

Sticklen phrases his criticism by proposing an extension of the knowledge level: the knowledge level architecture:

'If a problem solving agent may be decomposed into the cooperative efforts of a number of sub-agents, the larger agent can be understood at the knowledge level by giving a knowledge level description of the sub-agents and specifying the architecture the composition follows.' (Sticklen, 1989, p.243)
Just like in Fox's proposal, the knowledge level architecture explicitly accounts for multiple agents. In the view of Sticklen, decomposing an agent into sub-agents and setting up communication paths and message protocols between these sub-agents, enables us to build simulators for making predictions. Sticklen has in mind the kind of predictions that needs two types of knowledge. Therefore, Sticklen needs multiple agents. We think, however, that the mere fact that we can differentiate knowledge and that we can model dependencies between several knowledge modules suffices to attain such a functionality. Decomposing agents therefore is unnecessarily complicating.

Part of Sticklen's criticism focuses on representing control of knowledge level models. We note, however, that the distinction of the knowledge level enables us to concentrate explicitly on knowledge independent of types of knowledge. Thus pointing at a supposed omission of a type of knowledge is somewhat odd. In addition, in Clancey's definition of the knowledge level competence of heuristic systems this type of knowledge (level control) is important and 'knowledgeable' (Clancey, 1985).

Finally, in Sticklen's view knowledge level models do not provide direct guidelines for the development of knowledge-based systems. Therefore they are computationally inadequate and have a non-operational character. Typical of this part of Sticklen's criticism is that for the greater part it fulminates against what is generally viewed as the greatest advantage of the knowledge level: the avoidance of implementation biases by describing the functionality of a knowledge-based system without any commitment to a particular implementation style. Also in traditional ways of developing computer systems, numerous arguments are mentioned that implementation-free descriptions of a system, often called a functional specification or requirements specifications, are essential for a system to be successful (Davis, 1988).

However, the knowledge level is more than an implementation-free level: it helps to define theories of the nature of knowledge. The distinction of the knowledge level provides an adequate entry point for the analysis of the nature of knowledge. Such an analysis (or theory of knowledge) permits us to understand, explain and predict behaviour without the need to construct an operational model of the implied process. A theory of knowledge strongly influences the conceptual modelling process (Lucardie, 1992). In Chapter 4 we go into this in more detail.

2.5 CONCLUSION AND DISCUSSION

Computer systems levels provide ways of describing computer systems: they do not provide ways of describing their environments. The identification of the knowledge level as a new and separate level on top of the symbol level, is based on the assumption that a clear conception of knowledge should be logically prior to that of representation. At the knowledge level we are bent on describing the knowledge a knowledge-based system contains. Knowledge used in the service of goals includes semantic issues (about the functionality of a system), but it completely excludes user-interface aspects (how to present the functionality of a system to users) as well as implementation aspects (how to encode the functionality). Knowledge corresponds
with data structures and processes at the symbol level. Knowledge exists by virtue of symbol systems (Figure 2.8):

'The entire field of Artificial Intelligence is, in a sense, devoted to discovering the symbol-level mechanisms that permit a close approximation to the knowledge level.'

(Newell, 1990, p.80)

Newell's theory has served as a guideline for many researchers (Brachman & Levesque, 1986; Bylander & Chandrasekaran, 1988; Schreiber, 1992; Steels, 1990; Steels, 1992). However, we should remember that research at the knowledge level did not arise with the advent of Newell's theory. Long before Newell launched his theory, questions we now classify as knowledge level problems, have been the subject of extensive studies. Furthermore, much research explicitly focusing on knowledge is not presented, and not classified, as knowledge level research.

Broadly speaking, we can identify two lines of knowledge level research. In the first approach, there is an explicit focus on decomposition and distribution of tasks, inference strategies, and models. Examples of this approach are the KADS methodology (Schreiber, 1992) and the Components of Expertise (Steels, 1990; 1992). In the second approach a strong focus can be seen on classification (Clancey, 1983; Clancey, 1985; Goel, Soundararajan, & Chandrasekaran, 1987).

<table>
<thead>
<tr>
<th>Aspects</th>
<th>Knowledge Level</th>
<th>Symbol Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td>Agents</td>
<td>Total Symbol System</td>
</tr>
<tr>
<td>Medium</td>
<td>Knowledge</td>
<td>Symbolic Expressions plus Processes</td>
</tr>
<tr>
<td>Component</td>
<td>Goals; Actions; Bodies</td>
<td>Memories; Operations</td>
</tr>
<tr>
<td>Behaviour Laws</td>
<td>Principle of Rationality</td>
<td>Total Problem Solving Process</td>
</tr>
</tbody>
</table>

(Source: Newell, 1981)

*Figure 2.8: The Reduction of the Knowledge Level to the Symbol Level*

What can we gain from this knowledge level perspective? We think that the utility of the knowledge level is clear enough and that Newell's claim can be confirmed. We found advantages which are not trivial. These advantages relate to the analysis of the nature of knowledge, the reconstruction of implementation-free knowledge level models (or conceptual models), the specification of the role of mathematical logic and to the examination and comparison of representation formalisms.

Furthermore, it appeared that much of the criticism raised against the knowledge level can be rejected. In a sense, Sticklen's criticism is illustrative of critics of the knowledge level. Our analysis and ultimate rejection of Sticklen's observations against the knowledge level help to expose a number of misconceptions. An important misconception is the view that the knowledge level is purely an attempt to shift the emphasis away from representational issues towards implementation-free descriptions of problem solving. If this were true, the knowledge level would not bring us any
further than the realisation that an implementation-free functional specification is necessary for developing computer systems. The proposal of the knowledge level encompasses much more. It is a separate computer systems level that paves the way towards developing theories of the nature of knowledge. Newell's theory of the knowledge level itself is not a theory of the nature of knowledge, but a theory that identifies the knowledge level. Newell's theory sets preconditions by providing the framework for developing and analysing theories of the nature of knowledge. Such an analysis has been performed by Brachman (1985). In it he fulminates against the prototypical theory of the nature of knowledge. In Chapter 5 we discuss the prototypical theory and its relation with other theories of the nature of knowledge. The differences between the knowledge level, a theory of the nature of knowledge and the modelling activities which are based on a certain theory of the nature of knowledge often are not recognised. This arouses the confusion and the criticism Sticklen and others display. We think that Sticklen's observations are somewhat odd and misplaced. The lack of predictive power is, as we argued, no real issue.

Though Newell's ideas have an ever-increasing impact on information technology, there is still great confusion between the knowledge level and the symbol level. Brachman (1985), for instance, points to the problems encountered by Fahlman, Touretzky and Van Roggen (1981) when they tried to ascertain the meaning of their inheritance mechanism and could not find a consistent interpretation; they could not assess the knowledge level import of their mechanism. Conversely, semantic issues are often mixed up with symbol level or implementation concerns. When discussing the functionality of a system, notions of inheritance mechanisms, frames and similar ideas often crop up.

Though Newell's knowledge level theory is an important step forward, much work remains to be done. Especially, two points should be addressed. The first point is that we should attempt to shift the emphasis away from representation issues and redirect it to knowledge. The second point refers to the role of mathematical logic as a representation formalism that is appropriate for the analysis of knowledge. In the next chapters we elaborate on these two points within the scope of integrating AI and DBT, which is the first research issue of the thesis.
3.1 INTRODUCTION

The integration of AI and DBT is being studied in a variety of ways. Some researchers stress the similarities of both research fields such as the common logical basis (Jones, 1991). Others point to differences such as the deductive proof-theoretic inferencing of AI-systems versus model-theoretic query evaluation of DB-systems (Brodie & Jarke, 1986) or emphasise the complementary nature of these systems (Risch, Reboh, Hart, & Duda, 1988). Despite this variety, we can distinguish two basic strategies.

The main thrust of research addressing relationships between AI and DBT is conducted following a *symbol level strategy*. Typical of this strategy is the concentration on the transfer of representation structures and associated processing techniques from AI to DBT and vice versa. At the symbol level a clear distinction exists between AI-systems and DB-systems. This distinction is made up by the different representation formalisms both types of systems employ. In AI-systems rules and accompanying inference mechanisms are widely used, while in DB-systems records and query evaluations constitute an important representation formalism.

An alternative strategy to address the relationships between AI and DBT is exclusively concerned with the knowledge of a knowledge-based system. In this *knowledge level strategy*, the distinction between AI-systems and DB-systems is non-existent. In terms of what knowledge is present, as opposed to representation structures and processing techniques encoding that knowledge, both types of systems are simply collections of knowledge elements serving representational ends (Brachman & Levesque, 1986; Twine, 1989).

The purpose of this chapter is to show that a knowledge level strategy opens up new perspectives and is more fundamental than a symbol level strategy. As mathematical logic plays a key role in a knowledge level strategy, a related purpose is to survey the strengths and weaknesses of mathematical logic.

The structure of this chapter is as follows. First, we characterise the symbol level strategy and discuss the implications of following it (Section 3.2). Second, we describe the knowledge level strategy by the reconstruction of a knowledge universe through mathematical logic (Section 3.3). To illustrate one of the advantages of the knowledge level strategy, we use mathematical logic as a knowledge analysis tool to perform a knowledge level evaluation of record-based representation formalisms (Section 3.4). To conclude this chapter and to set up preconditions for the next one,
we not only summarise the utility of the knowledge level strategy, but also point out an important weakness of mathematical logic (Section 3.5).

3.2 SYMBOL LEVEL STRATEGIES

The prevailing view within research regarding the integration of AI and DBT is that each research field can contribute to the other by the transfer of data structures and processing mechanisms. Examples of such symbol level contributions are the use of a DB-system to store facts and rules of an AI-system (Abarbanel & Williams, 1986; Deering & Faletti, 1986; Herwijnen, et al., 1990; Monarchi & Smith, 1992/1993), the use of frames to store relational data (Chow, 1987) or the addition of rule processing algorithms to databases (Stonebraker, 1984). This focus on representation formalisms is a typical trait of the symbol level strategy, but does not completely characterise it.

To obtain a more precise characterisation, we take a closer look at the work of Herwijnen, et al. (1990). Their point of departure is that the implementation of a knowledge-based system by using a database management system is not only possible, but even yields many important advantages. To illustrate this point, they mention the extensive recovery facilities of a database management system that prevent a system from ending up in an inconsistent state. Further, they point to the query optimisers that take care of an efficient access to large amounts of data. In addition, Herwijnen, et al. (1990) enumerate facilities of database management systems such as screen definition tools and report generators.

<table>
<thead>
<tr>
<th>Client X</th>
<th>Regulation Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Values</td>
</tr>
<tr>
<td>Income</td>
<td>25.000</td>
</tr>
<tr>
<td>City</td>
<td>Enschede</td>
</tr>
<tr>
<td>Age</td>
<td>32</td>
</tr>
<tr>
<td>Civil State</td>
<td>Divorced</td>
</tr>
<tr>
<td>Sex</td>
<td>Female</td>
</tr>
<tr>
<td>Age of Children</td>
<td>2.7 and 9</td>
</tr>
<tr>
<td>Property</td>
<td>Not Known</td>
</tr>
</tbody>
</table>

(Source: Herwijnen, et al., 1990, p.19)

Figure 3.1: The Attribute Table

What strikes one most is their claim that a database management system facilitates the process of modelling knowledge. Herwijnen, et al. (1990) account for this claim by emphasising that rules and data are modelled into a dedicated data model that enables them to model at a higher level of abstraction. They exemplify their argumentation by discussing the functionality of a knowledge-based system that is intended to support
users who advice their customers about government regulations that apply to them. Stated more precisely: 'The task of the system is to find, departing from the attributes of clients, all government regulations that apply to them, and to find, departing from a government regulation, all clients to whom this regulation is possibly applicable.' (Herwijnen, et al., 1990, p.17). A regulation is considered applicable from the standpoint of a client, if that particular client complies with the conditions of the regulation. A client is considered relevant from the standpoint of a regulation, if the client complies with the conditions of that regulation. To perform this task, a knowledge-based system should know the attributes of clients and the conditions of regulations.

The attributes of clients are modelled into three database tables. The Client table contains the individual clients, the Client-attribute table contains the values of each attribute of these clients. The Attribute table holds both attributes of clients and conditions of government regulations (Figure 3.1).

The conditions of the government regulations are modelled into four database tables. Each regulation is composed of one or more condition-parts. The table Government-regulation relates a regulation to its constituent condition-parts. A condition-part consists of regulation-attributes which are connected through the logical operators and and or. Herwijnen, et al. (1990) provide an example of a regulation (we will refer to it as regulation X) that demands that a person is of age, not pensionable, is married or has at least one child. A person is considered not pensionable, if this person is not older than 60 years in case of a woman and not older than 65 years in case of a man. According to Herwijnen, et al. (1990) the conjunctive normal form of regulation X consists of four condition-parts. The condition-parts 2, 3 and 4 each have two regulation-attributes:

1. \((\text{age} \geq 18) \text{ AND} \)
2. \(((\text{sex} = \text{female}) \text{ OR} (\text{age} < 60)) \text{ AND} \)
3. \(((\text{sex} = \text{male}) \text{ OR} (\text{age} < 65)) \text{ AND} \)
4. \(((\text{civil state} = \text{married}) \text{ OR} (\text{number of children} > 0)) \)

The table Condition-part relates the condition-parts to the constituent regulation-attributes. The table Regulation-attribute models these simple regulation-attributes. Finally, the table Attribute-domain contains the possible values of attributes. The complete dedicated data model of the system consists of six tables and is displayed in Figure 3.2.

The principle task of the knowledge-based system is to match object-types (relevant client or applicable regulation) to objects (clients or regulations). It therefore searches through the complete database. The values of the client's attributes are compared to the conditions of the regulations. If, for instance, a solution must be obtained for a client, the system, starting from the client's attributes, searches through all the available regulations and compares the occurring conditions with the values of the client's attributes to obtain a match.

The research briefly described above is a classic example of a symbol level strategy. It shows a combination of two important characteristics of such a strategy. The first characteristic is the focus on representation formalisms and system facilities.
built around them. The majority of the advantages Herwijnen, et al. (1990) put forward, concern a favourable symbol level system i.e. a database management system offering facilities related to the record-based representation formalism.

![Diagram of the dedicated data model](source Fig32.jpg)

(Source: Herwijnen, et al., 1990, p.18)

**Figure 3.2: Tables of the Dedicated Data Model**

The second characteristic is that the researchers do not adduce relevant arguments concerning the nature of knowledge to motivate (a) the choice of the representation formalism and (b) the way knowledge is represented in the tables. In our opinion, the claim that a record-based database management system supports the process of modelling knowledge cannot be understood as a valid knowledge level argument. What plays a central role in the substantiation of this claim is the idea of a standard clause: a condition-part that refers to only one attribute. In their line of reasoning standard clauses facilitate the process of modelling knowledge, because an expert can easily define standard clauses in the dedicated data model. Next, on the basis of these standard clauses, an expert can define a regulation. According to Herwijnen, et al. (1990), the task of the knowledge engineer is largely limited now to familiarising experts with the way the representation formalism is used. They state that experts find it easy to model their knowledge following this approach and conclude that this way of knowledge modelling is in keeping with the way experts think and work.

We do not doubt that experts find it easy, at least in the beginning, to insert knowledge into the tables of the system through standard clauses, but the question is whether the use of standard clauses is really helpful in the process of modelling knowledge and indeed is in keeping with the way experts think and work. Furthermore, we think that it is debatable whether the step from standard clauses to regulations and from a single regulation to a system of regulations can be easily taken by using this dedicated record-based data model.

Let us explain these objections by a short analysis of regulation \( X \). The first thing worth noting is that in this analysis attributes are shown to have several classifications. The attribute *age*, for instance, as a first classification, has the following three categories: (1) \( <18 \), (2) \( \geq 18 \) and \( <60 \) and (3) \( \geq 60 \). The second classification of *age* also has three categories: (1) \( <18 \), (2) \( \geq 18 \) and \( <65 \) and (3) \( \geq 65 \). These classifications of *age* have a conditional nature. They depend on the attribute *sex*: the first classification refers to females and the second classification to males.
The second observation is that the relevance of attributes is conditional as well. For instance, whether the attribute children is relevant, depends on classifications of age, sex and civil state. Further, the relevance of the attribute civil state depends on sex, age and children.

Can standard clauses cope with conditional, flexible classifications and conditional relevance of attributes? We are very doubtful about this. Standard clauses only account for one category of an attribute and ignore other categories and the influence of other attributes. An argument in their defence might be that other categories and attributes will be incorporated in the remaining condition-parts of a regulation. We do not reject this possibility, but think that this way of modelling is error-prone. The previous conjunctive normal form of regulation X showing a part of a table built of standard clauses illustrates several problems that may easily emerge. The second condition-part (((sex = female) OR (age<60)) OR (age<65)) states that a person should be a female or younger than 60 years, whereas the third condition-part (((sex = male) OR (age<65)) OR (age<65)) states that a person should be a male or younger than 65 years. The conjunction of these two condition-parts embodies the knowledge that a person should be a female or younger than 60 and be a man or younger than 65. Nowhere in this representation of regulation X, is it made explicit, that if a person is a man, and thus complies with the second condition-part, it is not possible that the same person can comply with the third condition-part by being a female. Nowhere in this representation of regulation X, is it made explicit, that if a person is younger than 60, and thus complies with the second condition-part, the same person automatically passes the third condition-part. Whether a client is a female or a male is not important in this specific case.

What we are stating is that the categories of each classification of an attribute should be mutually exclusive and that the interdependence between attributes should be made explicit in a model. Both classifications of age we presented above have mutual exclusive categories. This implies that the language in which the model is expressed should provide validation facilities to prevent errors that are a consequence of non-exclusiveness or ignoring interactions between attributes. An example of such an error is displayed in the conjunctive normal form that reflects a part of the condition-part entity of the dedicated data model. The second condition-part now states that a person should be a female or be younger that 60. But what probably is meant is, that if a person is a female, she should be younger than 60 to qualify for regulation X! A similar remark applies to the third condition-part. The right conjunctive normal probably is:

1. (age≥18) AND
2. (((sex = female) AND (age<60)) OR
3. ((sex = male) AND (age<65)) AND
4. ((civil state = married) OR (number of children>0))

Standard clauses not only seem to lack structuring facilities dealing with relations between categories of an attribute, but also seem to have the disadvantage of difficulty in assessing the influence of other attributes on this particular attribute. The representation formalism forces every standard clause to be constructed in strict isolation! If one continues to think in the same line, one will understand the problems
in entities such as the *Attribute* table. No facilities are present to survey and deal with flexible, conditional classifications and conditional relevance. For instance, the conditional relevance of *civil state* cannot be modelled in this table. Connected with this is the problem of validation. Experts are not enabled to validate the knowledge represented because of the fact that the representation formalism does not allow insight in flexible and conditional classifications.

We conclude that flexibility and conditionality of classifications cannot adequately be accounted for in standard clauses and this easily leads to incomplete, inconsistent or incorrect knowledge. Standard clauses may be effective for very simple regulations in which no dependencies between attributes exist, but regulations of this type are rarely found. A survey of the complexities of regulations due to flexibility and conditionality in classifications and the need for structuring facilities to deal with them is extensively described by Overhoff & Molenaar (1991).

A similar approach using a record-based representation formalism can be found in the work of Monarchi & Smith (1992/1993). They show how production rules can be represented in a simple entity-relationship structure by designing a number of relational tables and implementing a simple inference engine in the relational database language SQL. Monarchi & Smith (1992/1993) adduce arguments similar to those of
Herwijnen, et al. (1990). They point to the facilities of a database management system for concurrent processing and for setting up multi-user environments. Furthermore, they point to the security and referential integrity components.

Pursuing the representation of knowledge using a record-based formalism Monarchi & Smith (1992/1993), just like Herwijnen, et al. (1990), have to decompose and distribute rules over several tables of a data model. Their model, displayed in Figure 3.3, shows that the table Dimensions holds the name or description of the occurring dimensions (variables). The '#' denotes sequence numbers. The table D_values contains the values of the dimensions. The Facts table contains the atomic units together with accompanying certainty factors (CFs). The Antecedent and Consequent tables respectively contain the conditions and conclusions of production rules. The Rules table contains the names of the rules.

Though the data model of Monarchi & Smith (1992/1993) looks somewhat different, our previous remarks also apply here. Consider the example of Monarchi & Smith (1992/1993) in which they represent the following rules in a database management system:

1. If F1 or F2, and F3, then F5 and F6.
2. If F1 and F2, or F3, then F6.
3. If F1 and F2, or F3 and F4, then F7.

The representation of the rules in the tables of the data model looks as follows:

### Consequents

<table>
<thead>
<tr>
<th>R#</th>
<th>F#</th>
<th>CFV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

### Facts

<table>
<thead>
<tr>
<th>F#</th>
<th>D#</th>
<th>V#</th>
<th>Qualifier</th>
<th>Fname</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT2</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT3</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT5</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT6</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>FACT7</td>
<td>...</td>
</tr>
</tbody>
</table>

- 37 -
To reconstruct a rule, one has to start in the Rules table to find a unique identifier of a specific rule. Subsequently, keeping the unique identifier in mind, the tables Antecedents and Consequents have to be consulted to 'see' one complete rule. Again, no facilities are provided that help to get an insight into possible interactions between attributes and conditional relevance.

In general, independently of the data model, the record-based data model does not supply structuring, survey and validation facilities to model knowledge that is characterised by conditional relevance and flexible classifications. Kent (1979) made remarks similar to the ones we are making here against the record-based representation formalism. Kent directs his criticism mainly at the lack of structuring and representing facilities to cope with heterogeneous objects that require flexible and conditional descriptions. Kent states that the basic assumptions behind record structures are not adequate to model and represent these types of objects. In Chapter 4 we discuss Kent's criticism more in detail.

Favouring particular symbol level structures and processes is not wrong by definition and does not automatically imply a symbol level approach. We would like to add, however, that preference for a representation formalism should be justified by knowledge level considerations. For at the knowledge level, as we will extensively argue in this thesis, well-founded choices for representation formalisms can be made.
The last advantage mentioned by Herwijnen, et al. (1990), namely that record-based formalisms support the modelling of knowledge, cannot be understood as a knowledge level argument justifying their implementation choice. Rather, it is an attempt to force knowledge into an a priori chosen representation formalism. With regard to knowledge acquisition the approach of Herwijnen, et al. (1990) is a transfer approach: the extraction and modelling of knowledge in the form of the representation formalism a system offers, without any attention to knowledge level considerations. This brings us again to the second characteristic of a symbol level strategy: as far as knowledge is involved its role is limited to that of filling-material for already chosen representation formalisms. The nature of knowledge influences neither the choice nor the use of a specific formalism.

The transfer approach is subject to severe criticism. Schreiber (1992) points to problems related to the transfer approach: difficult (or impossible) mapping of knowledge onto the required representation, bad maintenance and poor explanation facilities. For instance, a maintenance problem in the approach of Herwijnen, et al. (1990) occurs through the fact that they represent a regulation, which is virtually a simple rule, in four tables. When inserting new rules or modifying existing ones integrity constraints are easily endangered. Another problem is that the process of modelling knowledge gets probably more complicated through the fact that we have to model knowledge using a representation formalism without having any knowledge level arguments for applying it in this way.

The main cause underlying these problems is not distinguishing carefully between the knowledge a knowledge-based system contains (or should contain) and its knowledge representation components carrying that knowledge. Smith (1980) made a contribution to describing this difference by formulating constraints for representation formalisms in his knowledge representation hypothesis (Chapter 2). If representation formalisms do not match these constraints, i.e., if the gap between knowledge and the target representation formalism is too wide, an intermediary and structured description of knowledge permitting us to conduct a knowledge level analysis, is needed.

To make our position quite clear we ought to state that we do not deny that AI and DBT can help each other at the symbol level, but emphasise that integration at this level without understanding fundamental issues at the knowledge level leads to serious difficulties. Too much concern with representation mechanisms at the expense of knowing what function a system is computing already led to disadvantageous implementation biases. All too often, discussions about the utility of representation techniques such as inheritance taxonomies, production rules and connection graphs lack correct understanding of what, if anything, these symbol structures indicate (Berg-Cross & Price, 1989; Brachman, 1983; Etherington & Reiter, 1983).

3.3 KNOWLEDGE LEVEL STRATEGIES

From the previous discussion, it follows that for the integration of AI and DBT we should carefully distinguish between the knowledge of a knowledge-based system (this might be an AI-system, a DB-system, an integration of both types of systems or
whatever) and the knowledge representation formalisms putting that knowledge to work. When we focus on the knowledge of a knowledge-based system and thus follow a knowledge level strategy, we are, initially, neither interested in knowledge representation structures that might exist in the knowledge-based system, nor in the processes operating over these structures. How knowledge is represented and made available to a knowledge-based system is of a secondary, symbol level concern. At the knowledge level the only relevant issue is what knowledge is present.

Following a knowledge level strategy has important consequences. The first consequence of a knowledge level strategy is that AI-systems and DB-systems both are simply considered as computer programs containing knowledge. Distinctive symbol level features in processes such as the deductive proof-theoretic inferencing of AI-systems versus the model-theoretic query evaluation of DB-systems and structures such as production rules versus records, are ignored in exchange for an explicit focus on knowledge. As both types of systems simply represent knowledge, no differences can be detected. On the contrary, instead of stressing differences, a knowledge level integration emphasises a deep and significant commonality stemming from fundamental concerns about knowledge.

Another consequence of a knowledge level strategy is the reduced role of knowledge representation formalisms. Though indispensable, they do not exclusively occupy centre stage. Since knowledge should be a logical implication of the content of the representation formalisms and since determining knowledge might require simple retrieval capabilities and inference of some sort, the role of representation formalisms is limited insofar that they should, according to the constraints mentioned in Smith's knowledge representation hypothesis (Chapter 2), enable us to assess their knowledge level import.

The third consequence is the key role that is reserved for mathematical logic. From a knowledge level perspective mathematical logic is a knowledge representation formalism that is uniquely appropriate for the analysis of knowledge. Even when other representation formalisms are applied, the determination of what knowledge is contained in them requires the use of mathematical logic. Though logic may not be useful as an implementation language due to computational disadvantages, there is nothing against viewing mathematical logic as a knowledge representation language that is extremely useful for the analysis of knowledge of knowledge-based systems such as knowledge systems (Walker, 1987), expert systems (Lucas & Van Der Gaag, 1991), expert database systems (Smith, 1986), semantic databases (De Brock, 1989), deductive or logic databases (Das, 1992; Galle, et al., 1984) and decision support systems. A system of mathematical functions normally specifies or underlies the knowledge of these systems. We call such a formal description of knowledge a knowledge universe.

Attention should be paid to the fact that we use mathematical logic to describe knowledge for implementation purposes: we do not use mathematical logic to describe human knowledge, but only to describe 'artificial knowledge' which will be represented in a knowledge-based system. In Chapter 4, we discuss the differences between human knowledge and 'artificial knowledge'. These differences need adequate understanding for us to be able to estimate the possibilities and limitations of knowledge-based systems. In this thesis, mathematical logic is applied to reflect the
knowledge of a knowledge-based system. This is fully compatible with the notion of computer systems levels. Each of these levels provides ways of describing computer systems, not the structure of their environment.

3.3.1 Knowledge Universe

In Chapter 1 we viewed knowledge as a competence to match object-types and objects. To obtain a match a knowledge-based system requires descriptions of object-types or objects. These descriptions can be collected, structured and documented in a knowledge universe. Various more or less synonymous terms are used to denote a knowledge universe: knowledge level model (David & Krivine, 1990; Schreiber, 1992), conceptual model (Di Battista, Kangasallo, & Tammasia, 1989; Oxborrow, 1989) and data base universe (De Brock, 1989). Some of these terms are also used to denote a graphical display of a knowledge universe. Figures 3.2 and 3.3 are examples of graphical displays of knowledge universes. These displays, though providing a survey, cannot be considered as real knowledge universes. Usually, real knowledge universes are formal and much more detailed descriptions of object-types and/or objects. Before defining a system of mathematical functions to describe real knowledge universes, we present a number of basic definitions related to set theory.

3.3.2 Mathematical Preliminaries for Describing a Knowledge Universe

We suppose that the elementary definition of a set as a whole of separate parts is known to the reader. A set does not know any order or duplicates. Thus:

$$\{3, -1, 2, 4, 3\} = \{-1, 2, 4, 3\} = \{-1, 2, 3, 4\}$$

A set can be described by describing the properties of its elements:

$$S_1 = \{x | x \text{ is a real number and is the root of the equation } x^2 = 1\}$$

Another possibility to describe sets is the enumeration of its elements:

$$S_1 = \{-1, 1\}$$ \hspace{1cm} \text{Finite set}

$$S_2 = \{-1, 1, 2, \ldots\}$$ \hspace{1cm} \text{Infinite set}

A number of numerical sets is often used:

$$\mathbb{N} = \{1, 2, 3, \ldots\}$$ \hspace{1cm} \text{Natural numbers (excluding 0)}

$$\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$$ \hspace{1cm} \text{Natural numbers (including 0)}
\[ \mathbb{R} \quad \text{Real numbers}\]

\[ \mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots \} \quad \text{Integers}\]

\[ \mathbb{Z}^+ = \{1, 2, 3, \ldots \} \quad \text{Positive integers}\]

\[ \mathbb{Q} = \{x = \frac{a}{b} \mid a \in \mathbb{Z} \text{ and } b \in \mathbb{N}\} \quad \text{Rational numbers}\]

We make use of the following notation:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall x \in A : )</td>
<td>For every element of ( x ) of ( A ) counts:</td>
</tr>
<tr>
<td>( \exists x \in A : )</td>
<td>There is an element ( x ) of ( A ) for which counts:</td>
</tr>
<tr>
<td>( \Leftrightarrow )</td>
<td>then and only then if</td>
</tr>
<tr>
<td>( D )</td>
<td>by definition then and only then if</td>
</tr>
<tr>
<td>( = )</td>
<td>by definition</td>
</tr>
</tbody>
</table>

We also have to define standard operations for the manipulation of sets. The following definition refers to *set-inclusion* (a up to c) and to the *union* (d), the *intersection* (e), the *difference* (f) the *Cartesian product of two sets* (g) and finally the *powerset of a set* (h). The last definition refers to the *union of a set of sets* (i). If \( A \) and \( B \) are sets, then:

(a) \( A \subseteq B \quad \Rightarrow \quad \forall x \in A : x \in B \)

(b) \( B \supseteq A \quad \Rightarrow \quad A \subseteq B \)

(c) \( A \subset B \quad \Rightarrow \quad A \subseteq B \text{ and } A \neq B \)

(d) \( A \cup B \quad \Rightarrow \quad \{x \mid x \in A \text{ or } x \in B\} \)

(e) \( A \cap B \quad \Rightarrow \quad \{x \mid x \in A \text{ and } x \in B\} \)

(f) \( A - B \quad \Rightarrow \quad \{x \mid x \in A \text{ and } x \notin B\} \)

(g) \( A \times B \quad \Rightarrow \quad \{(x, y) \mid x \in A \text{ and } y \in B\} \)

(h) \( P(A) \quad \Rightarrow \quad \{X \mid X \subseteq A\} \)

(i) \( \bigcup W \quad \Rightarrow \quad \{x \mid \exists A \in W \text{ and } x \in A\} \)

---

To these numbers belong: (a) all finite decimal fractions including integers and (b) all infinite repeating decimal fractions and (c) all finite non-repeating decimal fractions.
The formal definitions to be presented will also make use of the following symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Used for denoting:</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>the restriction of a function to the specified part of its domain</td>
</tr>
<tr>
<td>†</td>
<td>the restriction of a function to the not specified part of its domain</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>⊕</td>
<td>function composition for modification of a single function</td>
</tr>
<tr>
<td>⊗</td>
<td>function composition for modification of a set of functions</td>
</tr>
<tr>
<td>r</td>
<td>the natural join</td>
</tr>
<tr>
<td>↑</td>
<td>tours of a relation</td>
</tr>
</tbody>
</table>

### 3.3.3 The Main Components of a Knowledge Universe

The main components of a knowledge universe to be defined are:

A. Knowledge Elements
B. Knowledge Tables
C. Knowledge States

Many formal definitions concerning these components are derived from De Broek (1989). A number of definitions have been assigned other names or are otherwise modified. The definitions are exemplified with examples which are mainly taken from the field of fire-safety.

#### A. Knowledge Elements

A knowledge element can be taken as a basic element of knowledge. A knowledge element is a means-end relationship that is needed to classify an object as an object-type. A knowledge element can be described by a mathematical function. A formal and precise definition of a function is:

**Definition 3.1: Function**

\[ f \text{ is a function } \iff f \text{ is a set of ordered pairs and} \]
\[ \forall (x; y) \in f : \forall (x'; y') \in f : \text{if } x = x' \text{ then } y = y' \]

As we can see from this definition, a function is a set of ordered pairs with specific properties. An ordered pair with \( x \) as first co-ordinate and \( y \) as second co-ordinate is noted as follows: \((x; y)\). Conversely, if \( p \) is an ordered pair, we can designate the first
co-ordinate of \( p \) with \( \pi_1(p) \) and the second co-ordinate with \( \pi_2(p) \). The domain of a function \( f \) is the set of first co-ordinates of the ordered pairs of \( f \). The range of a function \( f \) is the set of second co-ordinates of the ordered pairs of \( f \). If \( f \) is a set of ordered pairs (i.e., the order of the ordered pairs is not relevant and there are no duplicates), and for every domain element \( x \) of \( f \) exactly one \( y \) exists (i.e. the same \( x \) cannot have different \( y \)'s, but the same \( y \) could be connected to several \( x \)'s), then \( f \) is a function. An example of a function is:

\[
kl = \{(\text{wall - identifier}; 123), (\text{thermal insulation}; 25), (\text{irradiance}; 35)\}
\]

The domain of \( k1 \) is:

\[
\text{dom}(k1) = \{\text{wall - identifier, thermal insulation, irradiance}\}
\]

The range of \( k1 \) is:

\[
\text{range}(k1) = \{123, 25, 35\}
\]

The set \( k1 \) consists of three ordered pairs. Every domain-element is linked to exactly one value. It follows that \( k1 \) complies with definition 1. Thus, a mathematical function \( k1 \) is a recipe that adds to each element from the set \( \text{dom}(k1) \) exactly one element of the set \( \text{range}(k1) \). Another example of a function is:

\[
k2 = \{(\text{wall - identifier}; 124), (\text{thermal insulation}; 25), (\text{irradiance}; 24)\}
\]

Semantically, \( k1 \) and \( k2 \) are knowledge elements each containing three attributes of an object. In this particular situation the object is a wall. In a well-defined knowledge universe these attributes should be helpful in the process of classifying the relevant walls (objects) as object-types. A possible object-type could be a fire-resistant wall. Then, seeing whether a specific wall belongs to the extension of this object-type and thus is fire-resistant, requires knowing the relevant attributes of a wall. These attributes are described in the previous functions.

We say that \( k1 \) is a function over \{wall-identifier, thermal insulation, irradiance\}. The function \( k2 \) is also a function over \{wall-identifier, thermal insulation, irradiance\}. The formal definition of a function over a set is:

**Definition 3.2: A function over a set**

If \( A \) is a set then:

\[
f \text{is a function over } A \overset{D}{\iff} f \text{is a function and } \text{dom}(f) = A
\]

Because not all functions are in the form we want them, we have to define some other operations to modify functions. One of them is the restriction of the domain of a
function $f$ to a set $B$. The restricted and the remaining part of $f$ are respectively defined as:

**Definition 3.3: Restriction**

(a) $f \mid B = \{(x; y) \in f \mid x \in B\}$

(b) $f \upharpoonright B = \{(x; y) \in f \mid x \notin B\}$

An example of a restriction: If $B = \{\text{wall-identifier, thermal insulation}\}$ then:

(a) $k1 \mid B = \{\text{(wall-identifier; 123)}, \text{(thermal insulation; 25)}\}$

(b) $k1 \upharpoonright B = \{\text{(irradiance; 35)}\}$

Restrictions on functions are necessary to formally relate chunks of knowledge. Another operation that serves similar purposes, is the *function composition* of a function into another function. The formal definition is:

**Definition 3.4: Function composition**

If $f$ and $g$ are functions then:

$$g \circ f = \{(x; g(f(x))) \mid x \in \text{dom}(f) \text{ and } f(x) \in \text{dom}(g)\}$$

Suppose we have:

$h_1 = \{(\text{wall-identifier; component-identifier})\}$

$k5 = \{(\text{component-identifier; 123}), (\text{part-identifier; 2765}), (\text{number; 2})\}$

Then:

(a) $k5 \circ h_1 = \{(\text{wall-identifier; 123})\}$

(b) $h_1 \circ k5 = \emptyset$ (because there is no $x \in \text{dom}(k5)$ and $k5(x) \in \text{dom}(h_1)$)

Restriction is a special kind of function composition:

$$f \mid A = f \circ id(A)$$

$id(A)$ is the identical function. The identical function on $A$ adds to each element of $A$ the element itself.
B. Knowledge Tables

If knowledge elements are functions over the same set, they can be grouped into a knowledge table. A knowledge table is a set of functions in which each function has the same domain. \( K_1 \) and \( K_2 \) are knowledge tables. \( K_1 \) is a knowledge table over \{ wall-identifier, thermal insulation, irradiance \} and \( K_2 \) is knowledge table over \{ component-identifier, product-identifier, number \}. \( K_1 \) and \( K_2 \) respectively contain four and two knowledge elements:

\[
K_1 = \{ k_1, k_2, k_3, k_4 \}
\]

\[
k_1 = \{ (\text{wall-identifier}; 123), (\text{thermal insulation}; 25), (\text{irradiance}; 35) \}
\]

\[
k_2 = \{ (\text{wall-identifier}; 214), (\text{thermal insulation}; 35), (\text{irradiance}; 45) \}
\]

\[
k_3 = \{ (\text{wall-identifier}; 329), (\text{thermal insulation}; 20), (\text{irradiance}; 25) \}
\]

\[
k_4 = \{ (\text{wall-identifier}; 491), (\text{thermal insulation}; 25), (\text{irradiance}; 35) \}
\]

\[
K_2 = \{ k_5, k_6 \}
\]

\[
k_5 = \{ (\text{component-identifier}; 123), (\text{part-identifier}; 2765), (\text{number}; 2) \}
\]

\[
k_6 = \{ (\text{component-identifier}; 214), (\text{part-identifier}; 2778), (\text{number}; 2) \}
\]

The formal definition of a knowledge table is:

**Definition 3.5: Knowledge table**

If \( A \) is a set then:

\( K \) is a knowledge table over \( A \) \iff \( K \) is a set and 
\[
\forall k \in K : k \text{ is a function over } A
\]

We can raise the previously defined operation for restricting the domain of one function to the level of a set of functions and thus also to the level of knowledge tables:

**Definition 3.6: Restriction of a set of functions**

If \( F \) is a set of functions and \( B \) is a set then:

\[
F \upharpoonright B = \{ f \upharpoonright B \mid f \in F \}
\]

Now a set of functions can be dealt with in one go:
$K1 \parallel \{(\text{thermal insulation}, \text{irradiance}) = \{
\{(\text{thermal insulation}; 25), (\text{irradiance}; 35)\}, \{(\text{thermal insulation}; 35), (\text{irradiance}; 45)\},
\{(\text{thermal insulation}; 20), (\text{irradiance}; 25)\}, \{(\text{thermal insulation}; 25), (\text{irradiance}; 35)\}\}$

In a similar manner, we can extend the definition of a function composition to let it operate upon a set of functions.

**Definition 3.7: Function composition for a group of functions**

If $G$ is a set of functions and $f$ is a function then:

$G \circ f \triangleq \{ k \circ f | k \in G \}$

---

Example:

$K2 \circ \{(\text{wall - identifier}; \text{component - identifier}) = \{((\text{wall - identifier}; 123)), ((\text{wall - identifier}; 214))\}$

The following operation is called the natural join:

**Definition 3.8: Natural join**

If $F$ and $F'$ are sets of functions then:

$F \bowtie F' = \{ f \cup f' | f \in F \text{ and } f' \in F' \text{ and } f \cup f' \text{ is a function} \}$

---

We quantify thus over all pairs $(f \times f') \in F \times F'$ in the sense that every union of $f$ and $f'$ yields another function. If the union of two functions produces a new function the original functions are joinable.

Example:

If $h_2$ is a function with the following definition:

$h_2(\text{wall - identifier}) = \text{component - identifier}$

$h_2(x) = x$ for each $x \in \{\text{part - identifier, number}\}$

Then:
\[ K_1 \Join K_2 \bowtie h_2 = \]
\[ \{ \{ (\text{wall - identifier}; 123), (\text{thermal insulation}; 25), (\text{irradiance}; 35),
    (\text{part - identifier}; 2765), (\text{number}; 2) \},
    \{ (\text{wall - identifier}; 214), (\text{thermal insulation}; 35), (\text{irradiance}; 45),
    (\text{part - identifier}; 2778), (\text{number}; 2) \} \} \]

The first set of the composition and the natural join express the knowledge that \textit{wall 123} has a \textit{thermal insulation} of 25 and an \textit{irradiance} of 35 and consists of at least two components (products). The second set of the composition and the natural join express the knowledge that \textit{wall 214} has a \textit{thermal insulation} of 35 and an \textit{irradiance} of 45 and consists of at least two components (products).

The restriction, composition and join operations apply to sets of functions. We have, however, a special interest, as will be shown, in using them for the modification of knowledge tables.

\textbf{C. Knowledge States}

Another important class of functions, necessary for the definition of knowledge states, is found in the \textit{set-valued functions}. Every domain-element of a set-valued function is connected to a set. \( S_1 \), for instance, is a set-valued function with two ordered pairs.

\[ S_1 = \]
\[ \{ \langle \text{wall} ; \{ \text{wall - identifier, thermal insulation, irradiance} \} \rangle,
    \langle \text{component} ; \{ \text{component - identifier, part - identifier, number} \} \rangle \} \]

The formal definition of a set-valued function is:

\textit{Definition 3.9: Set-valued function}

\( S \) is a set-valued function \( \iff \) \( S \) is a function and
\[ \forall x \in \text{dom}(S) : S(x) \text{ is a set} \]

The knowledge of \( K_1 \) en \( K_2 \) can be represented by a function \( KS \) so that:

\[ KS(\text{wall}) = K_1 \text{ and} \]
We call $KS$ a knowledge state over $S_1$. In our example, $S_1$ connects objects to their relevant attributes:

$$S_1(\text{wall}) = \{\text{wall - identifier, thermal insulation, irradiance}\}$$

$$S_1(\text{component-parts}) = \{\text{component-identifier, part-identifier, number}\}$$

The formal definition of a knowledge state is:

**Definition 3.10: Knowledge state**

If $S$ is a set-valued function then:

$KS$ is a knowledge state over $S$ $\iff$ $KS$ is a function over $\text{dom}(S)$ and

$$\forall x \in \text{dom}(S) : KS(x) \text{ is a knowledge table over } S(x)$$

A knowledge state is a function with a domain which is equivalent to the domain of the set-valued function $S$. The range of a knowledge state is a set of knowledge tables. Knowledge tables can have several connections with other knowledge tables. These connections can also be described by functions.

**Definition 3.11: Connection**

If $h$ is a function and $K$ and $K'$ are sets of functions then:

$h$ connects $K$ with $K'$ $\iff$ $K \parallel \text{dom}(h) \subseteq K' \circ h$

Two alternative definitions are:

1. $h$ connects $K$ with $K'$ $\iff$ $\{k \mid k \in K \Rightarrow \exists k' \in K' : k' \circ h\}$
2. $h$ connects $K$ with $K'$ $\iff$ $\forall k \in K : \exists k' \in K' : k \circ \text{dom}(h) = k' \circ h$

$h$ is called a variable transformation. If we have the following variable transformation:

$$h_1 = \{(\text{wall - identifier, component - identifier})\}$$

then:
The variable transformation \( h_1 \) connects \( KS(wall) \) with \( KS(component) \)

<table>
<thead>
<tr>
<th>Wall-Identifier</th>
<th>Thermal Insulation</th>
<th>Irradiance</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>25</td>
<td>35</td>
</tr>
<tr>
<td>124</td>
<td>35</td>
<td>45</td>
</tr>
</tbody>
</table>

![Wall-Identifier](image)

<table>
<thead>
<tr>
<th>Component-Identifier</th>
<th>Part-Identifier</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>2765</td>
<td>2</td>
</tr>
<tr>
<td>124</td>
<td>2778</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>3033</td>
<td>4</td>
</tr>
<tr>
<td>127</td>
<td>3035</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 3.5: The Variable Transformation \( h_1 \) Connects \( KS(wall) \) with \( KS(component) \)

When we view \( KS(wall) \) and \( KS(component) \) as tables, this connection can be displayed graphically (Figure 3.5). The restriction of each function of \( KS(wall) \) to \{wall-identifier\} is a subset of the restriction of \( KS(component) \) to \{component-identifier\}. In the second restriction the component-identifier is substituted by a wall-identifier. The variable transformation \( h_1 \) adds the corresponding attribute of \( KS(component) \) to every relevant variable of \( KS(wall) \). The connection between \( KS(wall) \) and \( KS(component) \) expresses the knowledge that every wall with thermal insulation and irradiance attributes, is a component composed of products.

There are many different types of connections to express several types of abstractions which will be dealt with in the next section. In some situations we do not speak of inclusion but of equivalence. In these situations we use another definition.

**Definition 3.12: Bilateral connection**

If \( h \) is a function and \( K \) and \( K' \) are sets of functions then:

\[ h \text{ connects } K \text{ with } K' \text{ bilaterally} \quad \iff \quad K \parallel \text{dom}(h) = K' \parallel h \]

By the function \( h \) we can see how knowledge elements are related to each other. We can express this also in an association:
**Definition 3.13: Association**

If $K$ and $K'$ are sets of functions and $h$ is a function then we call:

$\{(k; k') \in K \times K' | k \mid \text{dom}(h) = k' \circ h\}$

the association on $K \times K'$ induced by $h$

Using the definition of a knowledge state, we can define a knowledge universe:

**Definition 3.14: Knowledge universe**

If $S$ is a set-valued function then:

$\mathcal{D}_{\mathcal{K}U}$ is a knowledge universe over $S$ if $\mathcal{D}_{\mathcal{K}U}$ is a set of knowledge states over $S$

The set-valued function $S$ is a knowledge schema. A domain-element of $S$ is the name of an object-type or an object. An element of a knowledge universe $KU$ is a knowledge state conformable to $KU$. In the next section and the next chapters a number of knowledge universes will be discussed. First, however, we will deal with the logical reconstruction of a knowledge universe.

### 3.3.4 The Reconstruction of a Knowledge Universe: Static Constraints

In the previous sub-section we explained what a knowledge universe is. In this sub-section we describe how a knowledge universe can be reconstructed. The reconstruction of a knowledge universe passes through five phases. These five phases correspond with the determination of the:

1. Knowledge Schema (KSCH)
2. Variable Constraints (VC)
3. Inter-variable Constraints (IVC)
4. Knowledge Table Constraints (KTC)
5. Knowledge Universe Constraints (KUC)

These phases are logical phases. They do not necessarily correspond with phases in time-order. The phases 2 up to 5 aim to assess static constraints. These static constraints together form a hierarchical system of mathematical functions that define a knowledge universe.
1. Knowledge Schema (KSCN)

A knowledge schema relates the name of a knowledge table to the variables occurring in it. Conceptually, a knowledge schema may link object-types to their descriptive conditions or link objects to their relevant attributes. Logically, a knowledge schema can be described by a set-valued function. The domain of this function consists of a set of names and its range represents sets of variables that are relevant for the corresponding object-types or objects. An example of such a set-valued function is $S_2$:

$$S_2 =$$

$\{
\text{component} ; \{\text{component-identifier, type, basic component}\},$

$\text{component - parts} ; \{\text{component-identifier, part-identifier, number}\},$

$\text{basic component} ; \{\text{basic component-identifier, type, buying price, selling price, profit, stock}\},$

$\text{wall} ; \{\text{wall-identifier, thermal insulation, irradiance}\}\}$

$S_2$ is a set consisting of four ordered pairs. Every domain-element represents a name of an object. Every name is linked to exactly one value which is a set. $S_2$ contains knowledge of names and their variables such that:

$$S_2(\text{component}) ; \{\text{component-identifier, type, basic component}\}$$

$$S_2(\text{component - parts}) ; \{\text{component-identifier, part-identifier, number}\}$$

$$S_2(\text{basic component}) ; \{\text{basic component-identifier, type, buying price, selling price, profit, stock}\}$$

$$S_2(\text{wall}) ; \{\text{wall-identifier, thermal insulation, irradiance}\}$$

The schema reflects a hypothetical application in which components of spaces are described. Of every component, the component-identifier, the type of the component and whether a component is a basic component is assessed. Of every part of a component a component-identifier, a part-identifier and the number of constituent parts are assessed. Furthermore, we need to know the basic component-identifier, the type, the buying price, the selling price, the profit and the stock of each basic component. Of every wall a wall-identifier and the attributes thermal insulation and irradiance are assessed.

2. Variable Constraints (VC)

Variable constraints specify the allowed values of variables. They apply to variable-values per variable per object. For every knowledge table $KS(x)$ -where $x$ is an element of $\text{dom}(S)$ and $S$ represents a knowledge schema- the constraints which should be met by the elements of $KS(x)$ can be described by a set-valued function over $S(x)$. This set-valued function establishes the set of allowed values for every variable a
of $KS(x)$. Before determining the variable constraints, we introduce the concept of a row.

**Definition 3.15: Row**

If $n \in N$ then:

$$r \text{ is a row with length } n \iff r \text{ is a function and } \text{dom}(r) = [0, \ldots, n-1]$$

Rows are important when the order of objects is relevant as in character rows. Examples of rows:

- $r_1 = \{(0; 37), (1; 29), (2; 37)\}$
- $r_2 = \{(0; 29), (1; 37), (2; 37)\}$
- $r_3 = \{(0; 29), (1; 37), (2; 37), (3; 29)\}$

Among other things, rows can be used to define the domains of text-variables:

**Definition 3.16: Sets of character strings viewed as rows**

$$Chs(n) = \{r \mid k \in [0, \ldots, n] : r \text{ is a row with length } k \text{ and } \text{range}(r) \subseteq C\}$$

where $C$ is here a yet unspecified set of characters.

Now, we can specify the variable constraints:

$$\text{vcomponent} =$$

$$\{$$

(component - identifier ; $[1, \ldots, 10^3]$),
(type ; \{ceiling, wall, floor, window, door, window frame, door frame\}),
(basic component ; \{yes, no\})

$$\}$$

$$\text{vcomponent - parts} =$$

$$\{$$

(component - identifier ; $[1, \ldots, 10^3]$),
(part - identifier ; $[1, \ldots, 10^3]$),

$$\}$$
(number ; [1, ..., 10^3])
}

desc =
{
(basic component - identifier ; [1, ..., 10^3]),
(type ; Chs(6)),
(buying price ; [1, ..., 10^6]),
(selling price ; [1, ..., 10^4]),
(profit ; [1, ..., 10^4]),
(stock ; [1, ..., 10^4])
}

descwall =
{
(wall - identifier ; [1, ..., 10^3]),
(thermal insulation ; [1, ..., 10^3]),
(irradiance ; [1, ..., 10^3])
}

These set-valued functions are variable constraints describing object characterisations (Remmen, 1982) or object descriptions or object-types. The use of these set-valued functions specifying the variable constraints becomes clear when we define \( \Pi(F) \):

**Definition 3.17: \( \Pi \)**

If \( F \) is a set valued function then:

\[
\Pi(F) = \{ f | f \text{ is a function over } \text{dom}(F) \text{ and } \forall x \in \text{dom}(F) : f(x) \in F(x) \}
\]

\( \Pi(F) \) operates upon a set-valued function. The set defined by \( \Pi(F) \) is a set of functions and every function of the set has the same domain. This domain is equivalent to the domain of the set-valued function \( F \) upon which \( \Pi \) operates. Every function 'chooses' its range from the range of \( F \). As \( F \) is a set-valued function the range of \( F \) is a set. In this way, the number of allowed knowledge elements can be limited. To see whether a knowledge state \( KS \) complies with the constraints mentioned above, we only have to check whether:

\[
KS(\text{component}) \subseteq \Pi(\text{desc})
\]
KS(component - parts) ⊆ Π(vcomponent - parts)
KS(basic component) ⊆ Π(vbasic component)
KS(wall) ⊆ Π(vwall)

Example:

If:

KS(component) =
{
{(component - identifier; 123), (type; wall), (basic component; no)},
{(component - identifier; 214), (type; wall), (basic component; no)}
}

Then:

KS(component) ⊆ Π(vcomponent)

3. Inter-variable Constraints (IVC)

Inter-variable constraints limit the combinations of values of different variables. They further limit the set of knowledge elements by allowing us to impose additional demands on combinations of variables. Suppose we want to express the knowledge that:

(IVC1) a component cannot have itself as part in its structure
(IVC2) the selling price of a basic component is at least twice as much as the buying price
(IVC3) the profit on a basic component is the selling price minus the buying price
(IVC4) the stock of basic components is less or equal to 2 or the stock multiplied by the buying price is smaller than $10^6$
(IVC5) a certain value of the thermal insulation of a wall implicates a higher value for irradiance

This can be done as follows:

ivccomponent =
{k ∈ Π(vcomponent - parts) |
 k(component - identifier) ≠ k(part - identifier)}

(IVC1)
ivc-basic component =
\{k \in \Pi(\text{ivc-basic component}) \mid k(\text{selling price}) \geq 2 \times k(\text{buying price}) \text{ and } \text{(IVC2)}

k(\text{profit}) = k(\text{selling price}) - k(\text{buying price}) \text{ and } \text{(IVC3)}

k(\text{stock}) \leq 2 \text{ or } k(\text{stock}) \times k(\text{buying price}) \leq 10^6 \} \text{ (IVC4)}

ivc-wall =
\{k \in \Pi(\text{ivc-wall}) \mid \text{if } k(\text{thermal insulation}) = X \text{ then } \}

k(\text{irradiance}) < X + Y \text{ } \text{ (IVC5)}

Observe that IVC1 is not strong enough to guarantee that a component cannot have itself as part of its structure. It only accounts for situations wherein a direct part of a component cannot be the component itself. Indirect parts can yet be the component itself. To deal with this problem, an additional knowledge table constraint, as we will see, is required.

4. Knowledge Table Constraints (KTC)

Knowledge table constraints apply to a complete knowledge table. When we have a set of allowed knowledge elements \(Y\), then the set of permitted knowledge tables is a subset of \(P(Y)\) (see Sub-section 3.3.2 on the mathematical preliminaries). The definition of a set of permitted knowledge tables is based on a subset of the domain of the knowledge elements that serves as a unique identifier of a knowledge table. Thus, before specifying knowledge table constraints, we have to define under what circumstances a set \(B\) is uniquely identifying a knowledge table \(K\):

**Definition 3.18: Unique identification**

If \(B\) is a set then:

\(B\) is unique identifying in \(K\) \iff \forall k \in K : \forall k' \in K : \text{if } k \not\in B = k' \not\in B \text{ then } k = k' \]

Now, we can express the knowledge that:

(KTC1) the variable component-identifier of a component is a unique identifier

(KTC2) the variables component-identifier and part-identifier together uniquely identify a component

(KTC3) a component cannot have itself as part in its structure

(KTC4) the variable basic component-identifier of an object is unique in the total set of objects which are basic components
(KTC5) the total selling value of the stock is the product of the number of stock products and the buying price

(KTC6) the variable identifier of a wall is a unique identifier

The formal specifications are:

ktccomponent = \{K \subseteq ivccomponent | \\
\{component\ -\ identifier\} \text{ is unique identifying in } K\} \\
(KTC1)

ktccomponent - parts = \\
\{K \subseteq ivccomponent - parts | \\
\{component\ -\ identifier, part\ -\ identifier\} \text{ is unique identifying in } K \text{ and } (KTC2) \\
\{k(component\ -\ identifier); k(part\ -\ identifier) | k \in K \} \text{ is acyclic}\} \\
(KTC3)

ktcbasic component = \\
\{K \subseteq ivcbasic component | \\
\{basic\ component\ -\ identifier\} \text{ is unique identifying in } K \text{ and } (KTC4) \\
(\Sigma k \in K : k(stock) \ast k(buying\ price)) \leq 10^8\} \\
(KTC5)

ktcwall = \\
\{K \subseteq ivcwall | \{identifier\} \text{ is unique identifying in } K\} \\
(KTC6)

KTC3 not only incorporates IVC1, but also is an extension on IVC1. In contrast to IVC1, KTC3 also prohibits situations wherein an indirect part of a component is the component itself. Understanding the complete effect of KTC3, requires an explanation of the acyclic set. For this reason, we must define how to make a tour of a relation and the transitive closure of a relation:

**Definition 3.19: Tour of a relation**

If \(n \in N\) and \(R\) is a relation then:

\[ R \uparrow n = \{ (r(0); r(n)) | r \text{ is a row of length } n + 1 \text{ and } \forall i \in [0, \ldots, n - 1]: (r(i); (r(i + 1)) \in R} \]

We illustrate this definition with an example from De Brock (1989, p.18).

\[ R1 = \{(1; 2), (2; 3), (3; 4), (1; 4), (3; 1)\} \]

The graphical display of \(R1\) is:
From definition 3.19 and Figure 3.6 we infer that:

\[ R_1 \uparrow 2 = \{(1; 3), (2; 4), (2; 1), (3; 2), (3; 4)\} \] and

\[ R_1 \uparrow 3 = \{(1; 4), (1; 1), (2; 2), (2; 4), (3; 3)\} \] and

\[ R_1 \uparrow 4 = R_1 \]

\( R \uparrow n \) is the set of pairs of tours of exact \( n \) steps that are possible in the graphical display of \( R \). The transitive closure of \( R \) is the set of tours of \( R \) that are not empty. The transitive closure is denoted by \( \text{Tcl}(R) \).

**Definition 3.20: Transitive closure**

If \( R \) is a relation then:

\[ \text{Tcl}(R) = \bigcup \{ R \uparrow n \mid n \in N \} \]

- \( R \) is cyclic if real tours of \( R \) are possible. Otherwise \( R \) is acyclic.

**Definition 3.21: Cyclic and acyclic**

If \( R \) is a relation then:

- (a) \( R \) is cyclic \( \iff \exists (x; y) \in \text{Tcl}(R) : x = y \)
- (b) \( R \) is acyclic \( \iff \forall (x; y) \in \text{Tcl}(R) : x \neq y \)
5. Knowledge Universe Constraints (KUC)

Using the previously defined constraints, the determination of a knowledge universe is reducible to formulating constraints on a set consisting of different sets of object-types or objects. They are called knowledge universe constraints; they define connections between different knowledge states. First, we need an auxiliary function which we call a knowledge characterisation (KC):

\[ KC = \{
(\text{component} ; \text{ktccomponent}), 
(\text{component - parts} ; \text{ktccomponent - parts}), 
(\text{basic component} ; \text{ktcbasic component}), 
(\text{wall} ; \text{ktcwall})
\} \]

A KC is a set-valued function. Every element of range(KC) is a set of knowledge tables. With KC the knowledge universe can be defined using, for instance, the following constraints:

(KUC1) every part of a component is a component

(KUC2) every basic component is a component

(KUC3) every wall is a component which has a thermal insulation and an irradiance variable

(KUC4) the price of a wall is at least the sum of all cost prices of the possible direct parts

\[ KU = \{ KS | KS \in \Pi(KC) \text{ and } \}
\{ (\text{component - identifier; part - identifier}) \} \text{ connects } KS(\text{component - part}) \text{ with } KS(\text{component}) \}
\{ (\text{basic component - identifier; component - identifier}) \} \text{ connects } KS(\text{basic component}) \text{ bilaterally with } \}
\{ k \in KS(\text{component}) | k(\text{basic component}) = \text{yes} \}
\{ (\text{wall - identifier; component - identifier}) \} \text{ connects } KS(\text{wall}) \text{ bilaterally with } \{ k \in KS(\text{component}) | k(\text{type}) = \text{wall} \}
\forall p \in KS(\text{basic component}) : 
(\Sigma k \in KS(\text{component - part}) \gg (KS(\text{wall}) \gg h_2)) \text{ and} \]
\[ k(\text{component-identifier}) = p(\text{basic component-identifier}) : p(\text{price}) \ast\]
\[ k(\text{number}) \leq p(\text{buying price}) \]  
(KUC4)

where:
\[ h_2 = \{ (\text{component-identifier}; \text{wall-identifier}) \}. \]

The connections between the knowledge tables of the knowledge universe can be displayed graphically. In Figure 3.7 KUC4 is not displayed, but it refers to \( \text{KS(component)}, \text{KS(wall)} \) and \( \text{KS(basic component)} \):

\[ \text{Figure: 3.7: A Graphical Overview of the Knowledge Universe} \]

There are different types of connections. The dominant form is:

(A1) \( h \) connects \( \text{KS(S)} \) with \( \text{KS(E)} \)

A stronger alternative form, used less frequently, is:

(A2) \( h \) connects \( \text{KS(S)} \) bilaterally with \( \text{KS(E)} \)

(B1) Sometimes additional constraints have to be specified for the knowledge elements of \( \text{KS(E)} \):

\[ h \text{ connects } \text{KS(S)} \text{ with } \{ k \in \text{KS(E)} \mid \varphi(k) \} \]

\( \varphi(k) \) is a necessary condition which should be met by a knowledge element of \( E \) in order to have associated knowledge elements of \( S \). \( \varphi(k) \) often has the form \( k(a) = v_0 \).

(B2) \( h \) connects \( \text{KS(S)} \) bilaterally with \( \{ k \in \text{KS(E)} \mid \varphi(k) \} \)

\( \varphi(k) \) exactly indicates for which knowledge elements of \( \text{KS(E)} \) knowledge elements of \( \text{KS(S)} \) exist. If \( \varphi(k) \) has the form \( k(a) = v_0 \), \( a \) is called the inspection variable of \( E \) and \( v_0 \) is called the inspection-value for \( S \). From KUC2 it appears that \textit{basic component} is an inspection variable of \( \text{KS(component)} \) and that \textit{yes} is an inspection-value for \( \text{KS(component)} \). From
KUC3 it appears that type is an inspection variable of KS(component) and that wall is an inspection-value for KS(wall).

Knowledge universe constraints enable us to express fundamental abstraction mechanisms such as aggregations, generalisations, specialisations and associations.

![Diagram of Address](image)

\[
\text{ADDRESS} = \Pi \{ \text{STREET} ; \text{CHS}(20), \\
\text{CITY} ; \{15\}, \\
\text{ZIP} ; \text{CHS}(7) \}
\]

(A) Graphical Notation          (B) Mathematical Notation

*Figure 3.8: An Object-type Constructed with Aggregation*

Aggregation denotes the consolidation of object-types/objects into a new object-type/object. For instance, it allows us to focus on the abstract notion of address while ignoring its component parts. An aggregation is a composite object-type/object constructed from other object-types/objects. Figure 3.8 provides an example of an aggregation in a graphical notation often used in semantic database modelling and in a mathematical notation using \( \Pi \).

![Diagram of Generalisation and Specialisation](image)

\[
\{(\text{identifier}; \text{identifier}) \text{ connects:} \\
(1) \text{KS(specialist) bilaterally with} \\
\{k \in \text{KS(employee)} | k \text{ (type) = specialist}\} \\
(2) \text{KS(worker) bilaterally with} \\
\{k \in \text{KS(employee)} | k \text{ (type) \neq specialist}\} \\
(3) \text{KS(medical worker) bilaterally with} \\
\{k \in \text{KS(worker)} | k \text{ (type) = medical worker}\} \\
(4) \text{KS(non - medical worker) bilaterally with} \\
\{k \in \text{KS(worker)} | k \text{ (type) = non - medical worker}\}
\]

(A) Graphical Notation          (B) Mathematical Notation

*Figure 3.9: Generalisation and Specialisation*
Generalisation/specialisation points to particular supertype/subtype relations between object-types/objects. For instance, in a hospital employees may be a generalisation of workers and specialists. Workers, in turn could be a generalisation of medical and non-medical workers. Figure 3.9 shows a graphical and a possible mathematical notation.

Association involves, as does aggregation, the consolidation of object-types/objects into a new object-type/object. Association is used to build sets of elements of an existing object-type/object. The difference from aggregation is that the omission of one of the constituting elements does not lead to the disappearance of the association. Mathematically, association is a finitary powerset. For instance: $P(\text{LANGUAGE})$ where LANGUAGE is a definition.

![Logical Phases in the Reconstruction of a Knowledge Universe](image)

Figure 3.10: Logical Phases in the Reconstruction of a Knowledge Universe

Figure 3.10 depicts the logical phases. The arrows in Figure 3.10 denote 'used-in' relationships.

- $KSCH = \{(N_1; \text{dom}(VC_1)), (N_2; \text{dom}(VC_2)), \ldots, (N_n; \text{dom}(VC_n))\}$
- $IVC_i \subseteq \Pi(VC_i)$
- $KTC_i \subseteq P(IVC_i)$
- $KC = \{(N_1; KTC_1), (N_2; KTC_2), \ldots, (N_n; KTC_n)\}$ and
- $KU \subseteq \Pi(KC)$

- 62 -
3.3.5 The Modification of a Knowledge Universe: Dynamic Constraints

The previous constraints used in the reconstruction of a knowledge universe are static constraints. Static constraints assess the knowledge states that are formally allowed. Static constraints, however, do not constrain the transitions between knowledge states. For this purpose we need dynamic constraints. Dynamic constraints specify the permitted transitions from a knowledge state to a new version of that knowledge state. Dynamic constraints are constraints on changes in pieces of knowledge. The set of permitted transitions can be assessed by a subset $R$ of $KU \times KU$ with the intuitive meaning (De Brock, 1989):

$$(KS; KS') \in R \iff \text{the direct transition from } KS \text{ to } KS' \text{ is allowed}$$

An element of $KU \times KU$ is called a transition and a set of permitted transitions is called a transition relation on $KU$. If $KU$ is a set of knowledge states then:

**Definition 3.22: Transition and transition relation**

(a) $p$ is a transition within $KU$ $\iff p \in KU \times KU$

(b) $R$ is a transition relation on $KU$ $\iff R \subseteq KU \times KU$

An example of a transition relation is:

$KUR = \{(KS; KS') | (KS; KS') \in KU \times KU \text{ and }$

$\forall k \in KS(\text{basic component}) : \forall k' \in KS'(\text{basic component}) :$

$k(\text{buying price}) \leq k'(\text{buying price})\}$

The transition relation expresses the knowledge that the buying price of a basic component cannot be reduced.

3.3.6 Retrieving Knowledge from a Knowledge Universe through Functions

We can retrieve knowledge from a knowledge universe by defining functions over it. In DBT such functions are called queries. The normal approach in DBT is to represent (complex) objects in a knowledge universe and then define queries containing object-types. In this way, it is possible to classify objects as object-types. The same functions, however, can also be defined in AI-systems for retrieving the constraints of an object-type. Queries, for instance, can be added to an AI-system that has been implemented in Prolog by the incorporation of Prolog-definitions. Yet, for reasons of convention, we will adhere to the name 'query' in this thesis.
Definition 3.23: Query

If \( U \) is a set, then:

\[
\text{\( q \)} \text{ is a query over } KU \quad \iff \quad \text{\( q \)} \text{ is a function over } KU
\]

A query that retrieves knowledge from our previously defined knowledge universe is:

\[
\lambda KS \in ktcwall : \{ k \in KS | k(\text{thermal insulation}) \geq 30 \text{ and } k(\text{irradiance}) \geq 30 \}\]

The query represents an object-type. A successful answer to this query should yield the number of walls that meet the constraints of the object-type. This implies that both insulation properties exceed 30 minutes.

Other classes of queries:

(a) \( \lambda KS \in KU : KS(\text{wall}) \)

This query stands for the question: 'Give all values of the knowledge state wall'

(b) \( \lambda KS \in KU : KS(\text{wall}) || \{ \text{identifier} \} \)

This query stands for the question: 'Give all identifier values of the knowledge state wall'

(c) \( \lambda KS \in KU : \{ k \in KS(\text{wall}) | k(\text{identifier}) \neq 123 \} \)

This query stands for the question: 'Give all values of the knowledge state wall except the wall that is identified by 123'

(d) \( \lambda KS \in KU : \{ k \in KS(\text{components}) | k(\text{component - identifier}) \notin \}

\{k'(\text{component - identifier}) | k' \in KS(\text{basic component}) \} || \{ \text{basic component - identifier} \} \)

This query stands for the question: 'Give all identifier values of all components that are not a basic product'

Queries can be very complex when they have to gather knowledge from several knowledge tables. These queries can be formulated by using the natural join and (bilateral) connections. Examples of these more complex queries are provided in Chapter 7.

If queries are applied repeatedly, it might be useful to name them and to use that name instead of continually having to specify the query. A named query is called a view.

---

2 If \( A \) is a set, then \( |A| \) is the number of elements of \( A \). The notation \( \lambda C e \ WW : U_x \) in which \( U_x \) is an expression in \( x \), is shorthand for \( \{ (C; U_x) | Ce \ WW \} \).


Definition 3.24: View

If \( U \) is a set, then:

\[
p \text{ is a view on } KU \quad \iff \quad p \text{ is an ordered pair and } \pi_2(p) \text{ is a query over } KU
\]

If \( p = (n; q) \) then \( n \) is called the name of the query and \( q \) is called the definition of \( p \).

In Chapter 7 an extensive view is given. When we have a set of views, we can gather them in a function. This can only be done if the names of the query are different. Such a set of views is called a \textit{view-system}.

Definition 3.25: View-system

If \( U \) is a set, then:

\[
V \text{ is a view-system on } KU \quad \iff \quad V \text{ is a function and } \forall p \in V: p \text{ is a view on } KU
\]

3.4 A KNOWLEDGE LEVEL INTEGRATION: A VIEW OF DATABASES FROM THE KNOWLEDGE LEVEL

A knowledge level integration of AI and DBT is characterised by a shift in emphasis away from pure representational issues to knowledge. This concern often becomes manifest through a rather intensive use of mathematical logic serving as a tool for the analysis of knowledge. In this way, mathematical logic supports the evaluation of representation formalisms as to their adequacy in representing knowledge. When we discussed the approaches of Herwijnen et al. (1990) and Monarchi & Smith (1992/1993), we performed such an evaluation.

Here, we will conduct a similar, but more technical evaluation that is based on the system of mathematical functions that describe a knowledge universe. In succession, the knowledge characterisation, the variable constraints and the inter-variable constraints were assessed:

\[
S_2 = \\
\{ \\
(\text{component}) ; \{\text{component - identifier, type, basic component}\}, \\
(\text{component - parts}) ; \{\text{component - identifier, part - identifier, number}\}, \\
(\text{basic component}) ; \{\text{basic component - identifier, type, buying price}\}, \\
\} \\
\]
CHAPTER 3

STRATEGIES FOR INTEGRATING ARTIFICIAL INTELLIGENCE AND DATABASE TECHNOLOGY

\[
\text{ivcwall} = \\
\{ \text{wall - identifier ; } [1, \ldots, 10^3], \\
\text{thermal insulation ; } [1, \ldots, 10^3], \\
\text{irradiance ; } [1, \ldots, 10^3] \}
\]

\[
\text{vcwall} = \\
\{ \\
\text{wall - identifier ; } [1, \ldots, 10^3], \\
\text{thermal insulation ; } [1, \ldots, 10^3], \\
\text{irradiance ; } [1, \ldots, 10^3] \}
\]

If \( KS(\text{wall}) = \{ k1, k2 \} \) and:

\[
k1 = \{ \text{identifier; 123}, \text{thermal insulation; 25}, \text{irradiance; 35} \} \\
k2 = \{ \text{identifier; 214}, \text{thermal insulation; 35}, \text{irradiance; 45} \}
\]

then \( KS(\text{wall}) \) complies with the definition of a knowledge table. \( KS(\text{wall}) \) is an example of a knowledge table representing part of the building regulations for construction and industry. It is composed of a set of functions with an explicit concern for selecting fire-resistant walls to limit the spread of fire. Virtually, \( KS(\text{wall}) \) is a knowledge table that is an element of an allowed set of knowledge tables:

\[ KS(\text{wall}) \in ktcwall \]

From a symbol level perspective knowledge table \( KS(\text{wall}) \) could be represented as a table of a database over \{identifier, thermal insulation, irradiance\} with exactly two elements. Each element of \( KS(\text{wall}) \) corresponds to a tuple of a table. As every element of \( KS(\text{wall}) \) is a function, a tuple can be described by a function over the relevant set of field names (Figure 3.11). A second option is the implementation of the elements of \( KS(\text{wall}) \) as production rules. Then \( KS(\text{wall}) \) forms a set of production rules (Figure 3.11). The two types of representation structures, records or production rules, employ search in quite distinct ways. The underlying search mechanism for extracting knowledge from tables is a form of query evaluation and the fundamental access mechanism of a rule-based knowledge base boils down to a sort of deductive inferencing (Brodie & Jarke, 1986, pp.191; Smith, 1986, pp.8-12).

The example shows that a component described by means of mathematical logic to facilitate the assessment of its knowledge level import has several representation options at the symbol level. But this does not imply that the choice of a representation formalism should take place at the symbol level. On the contrary, as we argued and will be arguing again, only at the knowledge level well-founded evaluations and
choices of representation formalisms can be made. One of the advantages of working at the knowledge level, mentioned in the previous chapter, is the availability of a separate, implementation-independent level from which representation structures and access mechanisms can be analysed. An example of such an analysis is the view of databases from the knowledge level as described by Brachman & Levesque (Brachman & Levesque, 1986). They state that it is possible after some theorem proving to assess that one of two conditions is true without saying which one (using disjunction) or to state that an object satisfies a certain condition without saying what that phenomenon is (using existential quantifiers).

**Figure 3.11: Two Symbol Level Representation Structures**

A solution to this intractability problem is to limit the uncertainty expressible in an implementation language. Because record-based data bases are restricted in precisely this way, they can be considered as knowledge-based systems of a limited form. A data base can be compared to the knowledge representation component of a knowledge-based system3. To illustrate a view of databases from the knowledge level we increase the number of functions of the previous example to four.

\[
k_1 = \{ (\text{wall identifier}; 123), (\text{thermal insulation}; 25), (\text{irradiance}; 35) \}
\]

\[
k_2 = \{ (\text{wall identifier}; 214), (\text{thermal insulation}; 35), (\text{irradiance}; 45) \}
\]

\[
k_3 = \{ (\text{wall identifier}; 329), (\text{thermal insulation}; 20), (\text{irradiance}; 25) \}
\]

\[
k_4 = \{ (\text{wall identifier}; 491), (\text{thermal insulation}; 25), (\text{irradiance}; 35) \}
\]

This produces the following table of Figure 3.12.

The range of uncertainty in these functions is quite limited. Disjunctions, negations or existential quantifiers are not present. A consequence of the limitations in

---

3 'A knowledge based system is any system that uses an explicit knowledge base in some capacity. The knowledge representation component is the part of the overall system that manages the knowledge base.' (Brachman & Levesque, 1986, p.72).
expressiveness is a lack of knowledge. This becomes clear when we rephrase the question *How many walls meet the fire-safety requirements concerning fire for limiting the spread of fire?* It appears that the knowledge expressed by the logic functions is not sufficient to answer this question. For example, we also need to know in what way the two heat insulation variables of walls are related to fire-resistance. Assume, that we possess the knowledge that a wall meets the minimal fire-resistance requirements if the heat insulation properties both exceed 30 minutes. These requirements together form the object-type *fire-resistant wall*. As this object-type is not modelled in the knowledge universe, it can be formulated in a query. Then, the formal query to successfully answer the question should be:

\[
\lambda KS \in ktcwall : | \{ k \in KS | k(\text{thermal insulation}) \geq 30 \text{ and } k(\text{irradiance}) \geq 30 \}|
\]

This also means that the answer set only contains walls that meet the fire-resistance requirements. This is denoted by the query above. For instance, we could denote a wall by an identifier. Assume that we have a relation between the identifiers and walls. Then, the query yields the number of walls that meet the requirements. In this case, the number of walls is five.

### Table: Wall Properties

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Thermal Insulation (in minutes)</th>
<th>Irradiance (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>25</td>
<td>35</td>
</tr>
<tr>
<td>214</td>
<td>35</td>
<td>45</td>
</tr>
<tr>
<td>329</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>491</td>
<td>25</td>
<td>35</td>
</tr>
</tbody>
</table>

*Figure 3.12: A Table about the Fire-resistance of Walls*

The computation of the answer should result in a match implying that objects that can be classified as an object-type *fire-resistant wall* will be selected. But it is still not certain that the query yields the right answer. It is possible that the list of functions is incomplete, so that relevant walls might be lacking. Incompleteness can be solved by explicitly naming the positive instances of the present walls. Therefore, to reinterpret the question as the question originally posed, additional knowledge is needed.

\[
\forall[\text{wall}(x) \supset x = 123 \lor x = 214 \lor x = 329 \lor x = 491]
\]

(universal quantifier naming explicitly the positive instances)

Another problem might be that different identifiers refer to the same wall. For instance, one wall could be denoted by the identifiers 123 and 491. However, this is impossible here. In every permitted knowledge table KS(wall), the set \{identifier\} is unique identifying. This guarantees that every tuple represents a unique object.

Because of the extra knowledge, no reasoning is necessary to find out how many walls are present. Inference is simply reduced to calculation. All the system has to do is count how many appropriate tuples appear in the wall relation. It does not have to reason by cases or contradiction. But suppose that the object-type *fire-resistant wall* is modified by covering the relation between exterior walls and the minimal required fire-resistance. Suppose that the minimal required irradiance for *exterior* walls is 30 minutes This implies that the fire-resistance requirements for exterior walls are less stringent than for interior walls: the attribute thermal insulation is not relevant for
exterior walls and, under certain circumstances, this might yield a higher fire-resistance for exterior walls. Before formulating queries and computing answers, we first have to change $KS(wall)$ by adding the attribute wall-type:

\[
k_1 = \{\text{wall-identifier; 123}, \text{thermal insulation; 25}, \text{irradiance; 35}, \text{wall-type; exterior}\}\]
\[
k_2 = \{\text{wall-identifier; 214}, \text{thermal insulation; 35}, \text{irradiance; 45}, \text{wall-type; interior}\}\]
\[
k_3 = \{\text{wall-identifier; 329}, \text{thermal insulation; 20}, \text{irradiance; 25}, \text{wall-type; interior}\}\]
\[
k_4 = \{\text{wall-identifier; 491}, \text{thermal insulation; 25}, \text{irradiance; 35}, \text{wall-type; interior}\}\]

The modified formal query now becomes:

\[
\lambda KS \in \text{ktcwall} : \{k \in KS | \text{if } k(\text{wall-type}) = \text{exterior then } k(\text{irradiance}) \geq 30 \text{ or} \]
\[
\text{if } k(\text{wall-type}) = \text{interior then } k(\text{thermal insulation}) \geq 30 \}
\]

Now, the previous conclusion that there is one fire-resistant wall should be replaced by the inference that two walls are fire-resistant. The wall identified by 123 can still be classified as a fire-resistant wall. The example can be made more realistic by incorporating more attributes of walls and by introducing concepts defining conditional relationships between fire-resistance requirements and different types of spaces adjoining the walls represented in the database. Once again other conclusions will be reached!

This example illustrates that a view of data bases from a knowledge level perspective permits the assessment of the semantic consequences of adding knowledge. We know, for example, that we cannot simply add universally quantified statements without caring about their knowledge level or computational import. Assessments like these enable us to make decisions about the symbol level trade-off between tractability and expressiveness. Actually, the example denotes a more universal problem caused by a symbol level approach: the difficulties occurring while querying a database (De Jonge, Bruijning, Schoemaker, & Otten, 1988; Remmen, 1985; Riet, 1990). One of the specific advantages of following a knowledge level approach is the optimisation of query evaluation based on knowledge which is needed for interactive access to databases (Gallaire, et al., 1984).
3.5 CONCLUSION AND DISCUSSION

In this chapter we dealt with strategies for integrating AI and DBT. Specifically, we investigated the advantages of a knowledge level strategy for integrating AI and DBT in comparison with a symbol level strategy.

It appeared that a symbol level strategy leads to serious difficulties. The process of modelling knowledge is more complicated because of the fact that representation formalisms are used without having any knowledge level arguments for applying them this way. This often yields difficult (or impossible) mapping of knowledge to the chosen representation formalisms, bad maintenance and poor validation facilities.

The observed advantages of a knowledge level strategy form a mirror image of the disadvantages of a symbol level strategy. These advantages refer to an improved process of modelling knowledge which can be attributed to an explicit concentration on knowledge. Levesque (1984) describes the advantages of this approach as follows:

'In particular, a knowledge level view allows us to consider new operations on a knowledge base (that can be explained in terms of existing ones) without necessarily committing ourselves to any particular implementation style.' (Levesque, 1984, p.206).

In this chapter the concentration on knowledge took place by the reconstruction of a knowledge universe using a hierarchical system of mathematical functions. Such a knowledge universe reflects the knowledge of a system more clearly and provides us with arguments for making choices at the symbol level.

Fortunately, knowledge level strategies are spreading rapidly in certain specialised research fields in AI and DBT. In AI integration is studied in the field of expert database systems (Abarbanel & Williams, 1986; Brachman & Levesque, 1986; Brodie & Jarke, 1986; Deering & Faletti, 1986; Fox, 1986; Smith, 1986; Zaniolo, 1986). The counterpart in DBT is the field of semantic database modelling (Hull & King, 1986). Although, these research fields have a somewhat different formulation of their goals, both are interested in similar conceptual constructs aiming at the reconstruction of knowledge universa.

The spreading of knowledge level strategies in the field of expert database systems as well as in the field of semantic database modelling is not surprising, since there is no difference between AI-systems and DB-systems at the knowledge level. We are aware, however, that not everyone agrees with this. At least one view states that AI-systems describe and operate on classes of objects rather than on individual objects and that DB-systems represent and manage facts (see for an example of this view Wiederhold, 1984). AI-systems are said to contain knowledge in the form of material implications and DB-systems contain data in the form of ground atomic assertions. But such differences are rather shallow and it is not even clear whether these are real differences. Many AI-systems deal with material implications as well as with individuals and statistical DB-systems deal almost exclusively with classes of objects. It is also hard to see why a material implication is any less factual than a ground
atomic assertion. In both types of systems objects and/or object-types need to be incorporated to obtain a match.

In a knowledge level strategy mathematical logic plays a key role. Though there is no consensus on the exact value of mathematical logic and its relation to real-world knowledge and common-sense-reasoning, as indicated by the discussion between McCarthy and Minsky, the distinction of the knowledge level helps to assign mathematical logic its proper role. At the knowledge level mathematical logic is an important tool for the analysis of knowledge. We have attempted to cover its use for studying knowledge modelling problems. We have shown how logic applies to the description of object-types and objects by the intensive use of a system of mathematical functions for the reconstruction of a knowledge universe.

This does not automatically imply that mathematical logic can be used as an implementation language. There are many limitations to using mathematical logic as an implementation language, but this is an entirely different issue. For our purposes mathematical functions were used to describe the relevant attributes of objects and conditions of object-types. We have also used mathematical logic to analyse knowledge represented in a record-based formalism.

Knowledge can be written down in such a way that implementation-independent assessment of knowledge is facilitated. Not surprisingly, mathematical logic has already been used in AI for the analysis of deductive manipulations of small sets of facts. In the field of DBT it has been applied for the analysis of integrity constraints, database schemas and query languages (Gallaire, et al., 1984).

But while mathematical logic is a convenient formalism to describe knowledge, it does not provide guidelines for modelling knowledge. It does not help to determine the relevant attributes nor does it help to identify relevant constraints (see the problem described by Kent (1979)). It does not tell us what to keep and what to get rid of. Logic is not a theory of knowledge, it is just a representation formalism with certain properties that are useful for the analysis of knowledge. The only requirement is that mathematical logic is kept in its proper place. We agree with Israel (1983) when he states the following:

'The only requirement is that mathematical logic is kept in its proper place. We agree with Israel (1983) when he states the following:

To acquire 'some better idea about what we want to represent' Newell's theory does not suffice. Newell's description of the knowledge level is a theory describing the knowledge level, but it is not a theory of the nature of knowledge. The formulation of such a theory is in line with Newell's remark that his knowledge level hypothesis requires a technical elaboration. A theory of the nature of knowledge is needed for
identifying the conditions of object-types and the relevant attributes of objects and should help to understand the complexity of knowledge and the mechanisms behind complexity. In brief, the theory should support the reconstruction of a knowledge universe by organizing the mathematical functions that assess a knowledge universe.
CHAPTER 4

FUNCTIONAL OBJECT-TYPES FOR RECONSTRUCTING KNOWLEDGE UNIVERSA

4.1 THE CONCEPTUAL BASIS OF RECONSTRUCTING AND REPRESENTING KNOWLEDGE UNIVERSA

The point of departure of this chapter has been laid in the previous chapters. To deal with the problem of modelling knowledge we argued in these chapters that an integration of AI and DBT should preferably take place at the knowledge level. This implies an explicit focus on knowledge: the competence of matching object-types and objects (Chapter 1 and Chapter 2). Furthermore, we argued that mathematical logic is indispensable for the systematic description of object-types and objects in a knowledge universe (Chapter 3). However, mathematical logic does not automatically lead to a well-defined knowledge universe. On the contrary, one of the main causes underlying the failure of knowledge-based systems is a not well-defined knowledge universe. A knowledge universe is not only indispensable for the design and implementation of knowledge-based systems, but also for their validation, modification, maintenance and enhancement. Experience indicates, however, that in many cases reality is not well reflected in a full-fledged knowledge universe. Even if a knowledge universe is formally described and even when we keep in mind that a knowledge universe 'only' needs to be an approximation of reality (Chapter 2), it appears that a knowledge universe is seldom an acceptable model of reality. Defects in a knowledge universe such as omissions and inconsistencies are by no means exceptional (Davis, 1988, p.1099).

Due to the significance and difficulties of defining a knowledge universe, requirements of a knowledge universe have been specified in the ISO Conceptualisation Principle (Van Griethuysen, 1982). The Conceptualisation Principle prescribes that a knowledge universe should only include conceptually relevant aspects of that part of reality about which we want to communicate knowledge. In addition, the principle emphasises that this part of reality, often denoted as the Universe of Discourse, should be described excluding user presentation and implementation issues.

A valuable element in the principle is the recognition that the specification of a knowledge universe is a conceptual exercise equivalent to the formation of concepts or conceptual modelling. However, apart from lacking instructions how to define a knowledge universe, the ISO Principle does not specify clearly what a well-defined knowledge universe is. The prescription 'conceptually relevant' is not specified and remains unsatisfactorily vague. Falkenberg's attempt (1982) to improve on this by...
adding that conceptual relevant aspects should exclusively refer to the Universe of Discourse, neither yields sufficient specification of what is conceptually relevant nor of what is not.

What is systematically lacking in the ISO principle and in prevailing conceptualisation methods needed for the definition of a knowledge universe, is not the recognition that concepts are important acquisition and classificatory mechanisms responsible for the organisation of the layered system of mathematical functions that define a knowledge universe (Goel, Soundararajan, & Chandrasekaran, 1987). Rather, it is the structural lack of a well-developed theory of knowledge that underlies conceptualisation methods in AI and DBT: a theory that precedes the process of forming meaningful classifications and that precedes the specification of a knowledge universe. In spite of the general acknowledgement of the importance of concepts, theories underlying classification procedures quite remarkably have not received much attention from the AI- and DBT-communities.

Until now, conceptualisation methods are sometimes explicitly but often implicitly based on the probabilistic assumption that, in essence, all conditions necessary for creating a classification, are provided initially and can easily be revealed by utilising mathematical measures of similarity. Another frequently occurring prototypical assumption is that necessary conditions are sufficient to create a classification. Furthermore, it is assumed that the categories of conditions are a priori fixed and unconditional. The fact that conceptualising takes place without any explicit background knowledge about goals of classifications and contextual influences and the fact that categorisations have an unconditional status are largely ignored in these classification approaches.

Recently these basic assumptions have been severely criticised. Theoretical advances in AI state that relevant descriptive attributes are not necessarily a priori given but should be acquired through knowledge about goals of classifications and about contexts (Stepp & Michalski, 1986). It is also argued that an explicit concern for necessary conditions will not suffice for capturing the dynamics of reality. Furthermore, theoretical advances put forward that a goal- and context-oriented strategy leads to the reconstruction of new attributes and categorisations with a dynamic status (Van Der Smagt, 1985). Besides this theoretical criticism, accumulating empirical evidence contributes to the critical discussion by indicating that unconditional categorisations insufficiently account for the dynamics of reality (Carapuça & Fiadeiro, 1988).

The purpose of this chapter is to place both the theoretical criticism and the conflicting empirical evidence in an explanatory knowledge level framework. This conceptual framework will be based on the theory of functional classifications. This theory touches upon fundamental problems computer systems have in the field of classification. We think that this theory can be perceived as a fundamental revision of the way we should address the specification of a knowledge universe and deal with the integration of AI and DBT. Logically, the theory also provides a vantage point to evaluate and compare representation formalisms that should depict a knowledge universe (De Gelder, Van Gorp, & Lucardie, 1993).

The structure of this chapter is as follows. First, we explain three aspects of concepts: terms, object-types and objects (Section 4.2). After a description of several
views on object-types (Section 4.3), we discuss the main elements of the theory of functional classifications referred to as the functional view or as the theory of functional object-types (Section 4.4). Since the practical implications of functional classifications are largely unexplored, we will systematically relate the theory of functional classifications to the current practice of reconstructing a knowledge universe (Section 4.5). To illustrate the fruitfulness of analysing representation formalisms from a functional knowledge level point of view, we discuss the pros and cons of record-based representation formalisms (Section 4.6). We conclude the chapter by discussing implications and perspectives of the functional view (Section 4.7).

4.2 BASIC TERMINOLOGY OF CONCEPTS: TERMS, OBJECT-TYPES AND OBJECTS

The reconstruction of a knowledge universe consists of defining concepts of a problem domain. Computer scientists are often struck by the extreme difficulties of defining concepts. In writing and analysing the Mycin knowledge-based system this happened to Clancey (1985) in connection with concepts such as compromised host and immunosuppression. In daily life, humans do not have any problems when they have to communicate not (well-) defined concepts. The circumstances under which concepts are communicated and the flexibility of humans to process these circumstances in an effortless mental operation provide virtual definitions that are sufficiently clear and self-explaining. However, when we are dealing with definitions for the reconstruction of a knowledge universe, things are quite different. Making definitions accessible for a computer, requires a far-reaching process of formalisation in which mathematical logic plays a key role. During this process the meaning of a concept often appears to become less clear and less self-explaining. Even apparently simple definitions can be surprisingly fuzzy then. Zadeh (1975) formulates this as follows:

'It may be argued rather persuasively that most of the concepts encountered in various domains of human knowledge are, in reality, much too complex to admit of simple or precise definition. This is true, for example, of the concepts of recession and utility in economics; schizophrenia and arthritis in medicine; stability and adaptivity in system theory; sparseness and stiffness in numerical analysis; grammaticality and meaning in linguistics; performance measurement and correctness in computer science; truth and causality in philosophy; intelligence and creativity in psychology; and obscenity and insanity in law.' (Zadeh, 1975, p.147)

A scheme that helps to clarify fuzziness, is the one outlined by Ogden & Richards (1946). Ogden & Richards's triangle, depicted in Figure 4.1, shows that a concept can be 'unravelled' into three basic components: (1) term, (2) object-type and (3) objects.
The left angle of the meaning triangle stands for a term with the synonyms symbol, word or predicate. The top of the triangle stands for an object-type. Other words for object-type are intension, connotation or meaning of a concept. Finally, the right angle stands for objects, the extension, denotation, or the referents of a concept.

Ogden & Richards' scheme illustrates that we must distinguish a term of a concept from its object-type (depicted on the left side of the triangle) and its object-type from its objects (depicted on the right side of the triangle). The relation between a term and an object-type is that a term is merely a label that denotes an object-type. Normally, we use terms to avoid repeating lengthy and complex object-types. The object-type of a concept is a set of conditions establishing its meaning.

A term, an object-type and objects constitute three sides of the same concept. Example: the term diagnosis is only a label that designates the object-type diagnosis. The object-type diagnosis should be a clear description of what it takes to be a diagnosis: it consists of a set of constraints defining under what circumstances an object belongs to the class diagnosis. Processes or activities complying with these constraints are objects (instances or referents) of the object-type diagnosis and thus belong to the extension of the object-type diagnosis.

(Adapted from: Ogden & Richards, 1946)

Figure 4.1: The Basic Components of a Concept

The relation between an object-type and its objects is that objects are referents that should comply with the object-type. Objects are the real-world counterparts, if existent, of the world of an object-type. Objects are anything to which object-types apply and need not be physical phenomena. Objects may also be formed, as in the example, by a sequence of activities. Other examples of objects are an aeroplane, a mechanism in a robot device, a building or the process of writing this line. An object-type stands for a category of objects, whereas an object may be an instance of that object-type.

When an object-type is empirical, the reference is direct. When an object-type is abstractly defined the reference to objects is indirect and passes through one or more levels of more concrete descriptions of object-types and/or objects. Epistemologically, relationships between a theoretical object-type (for instance an electron) and objects in reality are called correspondence rules, co-ordination rules or bridge principles (Derksen, 1980, p.33). Eventually, we need a relation with concrete object-attributes to assess class membership.
From Ogden & Richards's triangle we can derive that computer scientists have to tackle two problems to avoid fuzzy descriptions of object-types. The first problem, the terminological problem, is located at the left side of the triangle where the relation between terms and object-types is depicted. The problem involves the fact that hardly any term is univocal: all terms are polysemic, thus endowed with several meanings or object-types. The terminological problem stems from the confusion of these object-types. This hardly creates a problem when object-types of a term fall into different disciplinary fields or can otherwise be distinguished. That canis applies to a constellation in astronomy and to a dog in zoology and that terminal can apply to a computer terminal or an airport terminal is no matter of concern. Still, in the majority of cases, subtle but essential differences between object-types exist. For instance, the term diagnosis might indicate many 'slightly' different object-types of diagnosis. In this respect, Kuhn and Feyerabend (Kuhn, 1970, p.266) frequently point to the term mass that is supposed to have another meaning in Newtonian dynamics than in relativistic dynamics. While the relativistic mass is dependent on other quantities, the Newtonian mass is an attribute of an object and independent of the behaviour of the object in a co-ordinate system. Kuhn and Feyerabend (Kuhn, 1970, p.266) state that the meaning of a term with a shift from one theory to another often changes in a subtle way. It will be evident that ignoring shifts in meanings of terms is likely to introduce fuzzy elements in the description of an object-type. To state things clearly, the terminological problem does not refer to the multiplicity of object-types that can be related to each term, but to the entanglement or confusion of these object-types.

The second problem is located at the right side of the meaning triangle where an unclearly specified object-type yields a fuzzy set of objects. This problem, designating the occurrence of vague object-types and therefore of vague extensions, is called the denotational problem. It addresses the question of how to find the conditions that should be incorporated in an object-type. An object-type that is not adequately defined and that lacks denotative power necessarily obtains unbounded or fuzzy objects.

Having identified two conceptual problems that underlie defects in a knowledge universe, the terminological problem and the denotational problem, the question is how to cure these problems. Disambiguating the relation term-object-type is relatively simple, but essential. By utilising a nominal or declarative definition we can eliminate the ambiguity caused by confusing object-types of one term. Such a definition can be circumscribed as follows:

'The simplest and most correct notion of a definition is a proposition declaratory of a meaning of a word: namely, either the meaning which it bears in common acceptance, or that which the speaker or writer....intends to annex to it.' (Mill, 1898, p.6)

In the reconstruction of an object-type the nominal definition should solve the terminological problem concerning the question which object-type is annexed to a term. In this respect the nominal definition is a convention that asserts the functions to be fulfilled by objects. Another use of a nominal definition is that it, if necessary, designates other object-types that are needed in the description of the target object-
type. A nominal definition of diagnosis in medicine may be the following: the determination of the main attributes of a type of disease. Another nominal definition of diagnosis may also focus on the causes of the disease. This will lead to a quite different object-type diagnosis and, as a consequence, to another extension.

The denotational problem concerns the question how to describe the meaning annexed to a term. The reconstruction of an object-type in a denotational sense is equal to defining a knowledge universe. The nominal definition is only necessary as a preparation for the denotational or real definition. Denotationally assessed object-types, on their part, provide computers with the competence to assess class membership of objects. This very function is vital for the reasoning and problem solving capacities of knowledge-based systems. Since objects are realised in virtue of object-types and do not exist in their own right, it is not surprising that the modelling of object-types counts as a fundamental issue in object-oriented analysis and design (Martin & Odell, 1992). In AI and DBT object-types such as diagnosis, planning, monitoring, simulation, employee, building, organisation, department, etc., are intensively studied. Of similar importance as the observation that object-types are central in describing a knowledge universe, is the question how object-types can be reconstructed. To answer this question, we will now take a closer look at different views on the nature of object-types.

4.3 VIEWS ON THE NATURE OF OBJECT-TYPES

There are several basic views on how to reconstruct the object-type of a concept (Van Der Smagt, 1985, pp.26-29). Characteristic for the classical view is that it presupposes that an object-type consists of a univocal set of necessary and sufficient conditions. Consequently, classification of an object has a simple all-or-nothing character. One assumes that establishing whether an object is or is not an object-type is a simple operation that presents no problems. For instance, the classification of an animal as a bird, is not considered a problem in the classical view.

Because classification of objects as object-types is less univocal and more complex than the classical view accounts for, alternative approaches have been developed. One of them is the probabilistic view. This view subscribes to the classical idea that an object-type is a set of sufficient and necessary conditions, but exclusively on a theoretical level. The probabilistic approach assumes that all sorts of (random) disturbances at the empirical level cause problems in the delimitation of the extension. In this connection we speak of 'fuzzy sets'. By utilising mathematical measures of similarity between objects, defined over an essentially a priori given set of attributes, the probabilist tries to eliminate the random disturbances, so that at the theoretical level univocal criteria can be proved to underlie the fuzzy extension (Stepp & Michalski, 1986, p.4.). Often, the mathematical techniques are not used to test the validity of a reconstructed object-type, but to inductively reconstruct an object-type on the basis of an a priori assumed similarity of objects. Object-types are considered equivalent if their corresponding extensions have the same objects. Object-types are identified by their members. Then, the line of reasoning is that: 'if it is true that
identical object-types have the same extension, then the reverse, that identical
extensions have identical object-types, will also be true.

In contrast to the classical and probabilistic views, the prototypical or stereotypical
view denies the possibility to exactly assess an object-type by necessary and sufficient
conditions. Instead of using these conditions, object-types are described by means of a
prototype. A prototype shares many attributes of objects that belong to a category. It
reflects a central tendency category of objects. Prototypes are typical object-types: a
typical bird, a typical elephant, a typical building. The description of a prototype
consists of so-called necessary conditions. For instance, a typical elephant could be
described by the necessary conditions grey, mammal, four legs and a trunk. Since no
object will satisfy all the necessary conditions, the question whether an object belongs
to the extension of an object-type depends on the degree of resemblance it has with
the prototype. The best that can be reached is a sort of family resemblance. Inevitably,
the delimitation of the extension is fuzzy, but this vagueness is not ascribed to
empirical disturbances as in the probabilistic view, but to reality which does not let
itself be categorised univocally.

Probabilistic and prototype conceptualisation methods go hand in hand and have
much in common. The probabilistic relation between an inductively derived object-
type and its extension is much the same as the family resemblance relation between a
prototype and its extension. Both conceptualisation methods prevail in AI and DBT
research. This becomes manifest in the knowledge acquisition techniques used to
define a knowledge universe. A survey of knowledge acquisition techniques is
provided by Reitman Olson & Rueter (1987), Kim & Courtney (1988) and Neale
(1988). The majority of knowledge acquisition techniques, including those used in
machine learning, neural networks and fuzzy logic, are scaling or sorting techniques
such as multidimensional scaling, repertory grid analysis and matrix techniques.
These techniques lead to additive or multiplicative models that specify the
reconstructed object-types. These knowledge acquisition techniques are highly
probabilistic and prototypical: the inductive approach delivers a prototypical object-
type and the assignment of objects to this object-type is not univocal but typified by
degrees of family resemblance or by probabilities. In both cases weights and error
terms are used to define object-types.

The central notion in the functional view is that an object-type cannot be defined
univocally on a theoretical level. In this respect the functional view corresponds with
the prototypical view. What is different is that the functional view offers a quite
different explanation of fuzziness. In contrast to the probabilistic and prototype views,
the functional view emphasises that fuzziness has a systematic character. The solution
of fuzziness is neither sought in the elimination of random disturbances (such as
measuring errors), nor in the comparison of objects with a prototypical object-type.
The functional solution is typified by the systematic identification and reconstruction
of several object-types instead of one single object-type. These object-types originate
from functional equivalence: the phenomenon that objects perhaps differing in many
respects are equivalent in achieving a nominally specified function within a certain
context. This leads to the rejection both of defining object-types based on extensions
and of describing object-types by means of prototypes.
4.4 FUNCTIONAL OBJECT-TYPES

In the description of object-types computer scientists should solve terminological fuzziness by means of a nominal definition and denotational fuzziness by means of a real definition. The functional view on object-types can be elaborated upon these two conceptual levels.

4.4.1 The Nominal Definition

Reconstructing object-types is a matter of classification and relies on finding good descriptive conditions. These conditions should help us to determine whether objects have the required attributes to belong to the extension of an object-type. However, the reconstruction of an object-type is not a univocal activity. Normally, it leaves us with an enumeration of seemingly intractable conditions.

Illustrative is the description of the object-type water. An indefinitely large number of conditions potentially qualifies for incorporation in the object-type water. Think for instance of:

(a) at sea level water boils at 100°C
(b) the saturation pressure of water at 60°C is 0.6 cm mercury
(c) water is a liquid with a refraction-index for sodium light of 1.33299 (at 20°C)
(d) liquid water has its maximum density at 3.98°C
(e) the viscosity of water vapour at 20°C is 9.6 x 10^-3 cP
(f) the specific heat of normal ice at 0°C is 2.061/J/°C
(g) water is a set of H₂O molecules
(h) water is a set of T₂O molecules
(i) water is a set of D₂O molecules
(j) water is an easily flowing liquid
(k) water is transparent,
(l) water is capable of dissolving many substances
(m) water is able to become ice when temperature gets lower
(n) water is chemically pure
(o) water is an elementary element of the universe

This enumeration gives rise to the question of how to describe water. Is water H₂O or is it an easily flowing liquid? Should we perhaps describe water by its isotopes T₂O or D₂O? Water is by no means the only object-type that displays an overwhelming number of conditions. Diagnosis, simulation, organisations, employees, persons are just a few other object-types for which an infinite number of conditions is eligible to delimit their extension. In fact, all object-types are describable by a virtually infinite number of conditions.

Using subsets of these conditions we can produce an infinite number of different but equally meaningful candidate object-types. The possibility of a multitude of object-types implies that objects may be instances of several object-types. This phenomenon is called multiple classification. Multiple classifications indicate that for
every arbitrary set of objects it can be shown that they are (dis-)similar concerning an
infinite number of attributes. Well-known examples are the geometrical diagrams of
Popper (1934). These diagrams vary in many respects such as form, size, content,
location and so on. Multiple object-types (and thus multiple classifications of objects)
are possible: object-types may be based on particular conditions such as the same
form, size, content, or they may be reconstructed on the basis of an arbitrary
combination of these conditions. The similarity of the diagrams \(X\) and \(Y\) depends on
which object-type is actual.

Popper postulates that the similarity of objects (and thus the distinction of object-
types) always presupposes the adoption of a point of view: the geometrical diagrams
are similar from a certain point of view, but may be dissimilar from another point of
view. The reconstruction of an object-type therefore requires a selection-principle that
directs abstraction: it governs which conditions will be accepted as relevant and which
will be ignored. The selection-principle is the first step to prevent terminological
entanglement of object-types. Without a selection-principle classification would just
be an undifferentiated chaos.

In the functional approach the operationalisation of a selection-principle takes place
by assuming a goal or function of classification (Stepp & Michalski, 1986). An
important aspect of this approach is the emphasis on the role of functions (or goals or
uses) that should be performed by objects. These functions direct the selection of
conditions of an object-type and consequently determine the relevant attributes of
objects. Every description of an object-type has a functional and intentional status. A
weapon, for example, is anything that can serve to inflict injury. Whether a concrete
object belongs to the extension of the object-type weapon, depends on the attributes of
the object that provide it with the capacity of inflicting injury. Another example of a
functional (nominal) definition may be the following description of an intelligent
object: the capacity to solve new and unexpected problems. Every object that is able
to solve new and unexpected problems and thus complies with the constraints, can be
classified as an intelligent object.

In the field of machine learning Stepp & Michalski (1986) showed the utility of the
availability of (a network of) goals in classification processes. They illustrate that
knowledge of goals is indispensable for the reconstruction of meaningful and useful
object-types. Consider the object-type train. The goal 'find simple geometrical
regularities' leads to the incorporation of conditions involving the number of different
shapes, car shape, engine shape and cargo shape. When other goals underlie the
classification, for instance securing safe transport, the content of the object-type train
changes significantly. The same applies to the object-type water. We can introduce
goals such as 'quench one's thirst' or 'produce H\(_2\)SO\(_4\)'. Whereas the first goal requires
attributes describing the drinkability of water, the second goal requires the evaluation
of the object attribute H\(_2\)O (T\(_2\)O or D\(_2\)O).

Though goal-orientation is an important element of the functional approach, it does
not completely characterise the functional approach. It is just one aspect of the
functional view, though an important one, that helps to solve the terminological
problem by showing that variability in the goals of classification may yield several
object-types. These object-types, which should be systematically distinguished from
each other, may even be incommensurable: a person who is intelligent need not be
intelligent, and a sample of a liquid that is object-type water, need not be watery. An object that can be classified as a wall for fire-safety reasons (Chapter 3) need not to have common attributes with another object that is a wall that is not susceptible to chemical degradation (Chapter 7). Distinct object-types may be incommensurable. Incommensurability occurs when there is no reason why the two object-types (water) should have common descriptive attributes. The goal-orientation is also a partial answer to the question why fuzziness has a systematic character in the functional view. Different goals lead to different object-types with varying contents. Ignoring these differences by gathering knowledge in one single object-type leads to fuzziness and validity errors.

Note that goal-orientation is an elaboration of the argument against inductive, extensional derivation of object-types of probabilistic conceptualisation methods. Assessing similarity of objects and abstracting object-types requires a selection-principle consisting of knowledge of goals or functions:

Functional concepts are not derived from phenomena that are observably similar (look like each other) but from phenomena that are functionally similar (fulfil the same function). The intension never inductively results from (statistical) analysis of observably similar phenomena but from rational arguments about functionally equivalent phenomena. (Van Der Smagt & Lucardie, 1991, p.296)

An important implication for the reconstruction of a knowledge universe is that it should show the flexibility to cope with several, possibly incompatible, goal-dependent object-types. In the following sections we will further analyse fuzziness and argue that even when goals are fixed the distinction of several object-types is often inevitable.

4.4.2 The Real Definition: Context-dependencies

It is true that we can classify objects by their capacity to perform a certain nominally described function. Yet, the actual realisation of that function is also conditional upon the circumstances. The reconstruction of an object-type is indissolubly connected with the whole of circumstances we will from now on call the context. Think, for instance, of the object-type train in the context of a traveller, a railway engineer, a train-spotter, or in the context of securing safe transport. In the first context the object-type will probably incorporate conditions describing the capability of an object to transport a traveller by rail from one place to another within a certain time. In the last context the object-type train probably will include quite different conditions such as radio-activity, corrosivity, explosiveness, flammability and the carrying of toxic chemicals. The remaining two contexts will likewise lead to other object-types train. Putnam (1975) appeals here to the sphere of interests; in some cases impurities are of importance, in other cases they are irrelevant. So in some contexts 'water may mean
chemically pure water, while in another, it may mean the stuff in the Lake Michigan' (p. 239).

To Putnam's observation must be added that chemically pure water is not something that stands on its own. As we devise more clever purification techniques, we will discover more chemically pure waters. Thus, identifying contexts and incorporating them in the object-type is an intellectual activity that keeps going on even when one context is fixed. We should not decide too quickly that an object-type is universally valid over different contexts. In the discussion about object-types, besides context-dependencies, thought-experiments with possible worlds are popular. The stipulation of another world presupposes that object-types are valid in our 'known' world (in all contexts), but may need modification in another world. Though it is not completely clear, how to tell a possible world from a context, it may appear that what originally seems valid in a possible world, is also valid in a certain context in our world! The statement that temperature has an absolute zero of $-273.15^\circ$C seems true independent of a context. Only in a stipulated world lower temperatures are attainable. In 1930 every physician would have agreed with this statement, but now a physical system consisting of magneto-spins may well have a temperature of $-300^\circ$C (Van Brakel, 1986, p.101). What temperatures (objects) can be classified as the absolute zero depends on, for instance, the available series of techniques that keep hydrogen atoms under control and gradually strip their kinetic energy (Van Calnmhout, 1993). Almog (1981) recommends to talk about other contexts in stead of other worlds. We can add numerous examples (compare the scientific discussions that marked the shift from the Aristotelian geocentric system to the Galilean heliocentric system) that show that the definition of object-types and the classification of objects are conditional upon circumstances that need not be immediately known. Whether an object-type is valid, depends on the circumstances under which objects have to perform functions.

Recognising the existence and influence of a context implies that an object-type should include a description of the context. Van Brakel (1986) illustrates this point in his essay on the definition of water. A preliminary object-type water could be defined by attributes mentioned in the previous example. Note that the context is partly included. A number of conditions are only effective at sea level, at a certain temperature and so on. The context, however, is not always mentioned. In (c) up to (f) pressure is not mentioned, while it certainly plays a decisive role in the effectiveness of the associated conditions. Furthermore, (b) is only valid for water vapour with a flat surface. Thus, there are several contexts wherein (a) up to (f) are not true. When these contexts are detected, we have to add them in the definition of the object-type.

Hendriks (1986) exhaustively and systematically describes how we have to account for contexts in the reconstruction of object-types. Object-types established through a concrete interactive relation with a context are called relational. As an example Hendriks (1986) describes a relational object-type dangerous work. Hendriks stresses that it is not possible to classify a specific activity as dangerous work, even when it is nominally described, without concrete references to a context. The context can be formed by the knowledge a person has. Depending on this knowledge, some activities will be classified dangerous and others will not. A member of the explosives disposal services will not consider his daily work dangerous. When he is defusing a bomb, he exactly knows the meaning of blue and green wires. This changes when the same
member is working in a transformer station. Then a green and a blue wire are exactly the same to him. The example indicates that the real connotation of an object-type is relative. The question whether an object is able to perform a nominally assessed function and therefore is an object-type cannot be answered by exclusively looking at the attributes of the object, because an adequate answer requires a systematic account of the context. Variability in contexts will lead to variability in the contents of object-types.

The conclusion is that there may be all kinds of contexts in which object-types vary and have different references. The nominal definition is necessarily relative compared with the ceteris paribus conditions of the contexts we know about and thus relative compared with the real definition. Conditions of object-types become particularly fuzzy when one views them from another perspective, because it is expected that the circumstances under which the conditions are relevant are known. In contrast to daily speech and communication, vaguely specified context-dependencies present a knowledge-based system with problems. A knowledge-based system should exactly know the influence of a context on the content of an object-type. Otherwise, it is not inconceivable that if someone begs for water, a knowledge-based system would advice to direct a stream of steam to a person or throw the person into the sea because both substances are sets of \( \text{H}_2\text{O} \) molecules. Not only the distinction of a goal, but also the distinction of a context can lead to incommensurable object-types.

4.4.3 The Real Definition: Objects

Objects are anything in which we have a special interest. This interest is expressed in an object-type. Since objects are things to which an object-type applies, they are subject to the question whether they belong to the extension of an object-type. We have already argued that classification of objects solely by their attributes is impossible. Yet, the attributes of objects play an essential role. They are co-decisive in the assessment whether an object is an instance of an object-type or not. According to a functional view, to be a referent of a certain object-type, objects should have properties that are functionally required in a certain context.

This section addresses the question of how to determine whether an object is an instance of an object-type and how to pass from the conditions of an object-type to the attributes of the object. A simple way of bridging this gap is direct reference. Once we know the (concrete) conditions of an object-type and the object's attributes and know that the attributes of a certain object match these conditions, direct reference is realised. The required properties correspond with the properties of the object so that we can legitimately view the object in question as an instance of the object-type.

A complication on the level of direct reference is that hardly any objects are completely pure instances and that, consequently, it is not clear what belongs to the extension of an object-type and what does not. Referring to the object-type water, Zemach (1976, p.121) noticed that further developments in science may lead to the discovery that not all \( \text{H}_2\text{O} \) molecules are similar, but that certain \( \text{H}_2\text{O} \) molecules have a deviating structure and are essentially different. Zemach therefore views water as a general term that encompasses all kinds of water. From this observation he comes to
the conclusion that *the* natural kind of water (in our terminology *the* object-type water) does not exist.

From a functional perspective it is not surprising that the object water needs a conditional description and is not always a set of $H_2O$ molecules, not because of the fact that we, indeed, know that $H_2O$ molecules are not necessarily similar, but because of the explicit recognition in the functional view that descriptions of objects can significantly vary under the influence of a goal and a context. In a situation in which a chemist is charged with the task to prepare $H_2SO_4$ (goal) in interaction with $SO_3$ (context), the description of an object as a set of $H_2O$ molecules will be adequate. In a context, however, in which we want to refer to chemically pure water, it is not permitted to describe water as $H_2O$. Then $H_2O$ and $D_2O$ are dissimilar substances. Nevertheless, the conclusion should not be, as Zemach states, that there is no object-type *water*. Rather, we should draw the conclusion that various object-types exist due to variation in goals and contexts. The existence of several object-types leads to the necessity of accounting for various descriptions of objects.

Direct reference is excluded when the description of an objects is not sufficient to see whether an object belongs to the extension of an object-type. We have to bridge the gap by adding more knowledge to the object-type. Knowledge that can be used to infer whether an object is an object-type. Let us exemplify this by considering indirect reference in more detail. Suppose a knowledge-based system has knowledge of an initial description of objects, for instance in terms of their physical attributes, along with a goal of classification and a context. The functionally required properties need not necessarily be present as initial descriptors (note that this conception is in contrast with one of the central probabilistic assumptions mentioned earlier). In that case, we should equip the knowledge-based system with additional knowledge so that it can logically and stepwise infer new and functionally relevant descriptors of objects from the ones initially given.

The derivation of new descriptors can be performed in two ways. First, it is possible to apply cross-referential knowledge. This type of knowledge is often horizontally organised by humans and is connected with multiple classification. Objects can belong to the extension of different object-types. Once it is known that an object is object-type $X$ it can be inferred that it will also be object-type $Y$. The assignment of an international bank director to $X$ and of wealthy people to $Y$ can serve as an example. The assignment may yield a universal quantifier that states that if a person is a bank director, that person is also wealthy. Of course, the inference that an $X$ is a $Y$ has no absolute value and will normally be conditional. Just as not all birds are able to fly, a bank director need not be rich in all conceivable circumstances. He can be arrested for corruption, be left penniless after a divorce, be blackmailed and so on. The specification under what conditions it is legitimate to infer that bank directors are wealthy and that birds can fly should be of a functional nature.

Applying knowledge that is hierarchically organised is another way of deriving new attributes. This vertical organisation of knowledge takes place by generalisation: the transition from a specific to a more general object-type occupies centre stage. Consider the following abstraction steps describing this process (Van Der Smagt, 1985, p.36):
As stated before, it is legitimate to abstract from the differences between the isotopes $T_2O$ and $D_2O$ as long as $SO_3$ constitutes the (interaction) context. The one-level inference chain from $T_2O$ or $D_2O$ to $H_2O$ is permitted and valid, because $T_2O$ and $D_2O$ are functionally equivalent and can, under precisely specified conditions, be perceived as a set of $H_2O$ molecules, though, when a goal or context requires the definition of pure water, $D_2O$, $T_2O$ and $H_2O$ are essentially different.

In discussions about the characterisation of objects the term dispositional attribute often emerges. A dispositional attribute is an attribute that reveals itself only when certain conditions are fulfilled. For example, the solubility of sugar manifests itself when one sprinkles sugar into water and that the fact that glass is breakable manifests itself when we throw it on the floor.

Is it possible to describe objects through dispositional attributes? From a functional perspective the answer is negative. The explanatory power of a dispositional term is of little importance. If you classify a person as an intelligent person, because he shows intelligent behaviour under certain circumstances, you do not give sufficient explanation of his intelligence. It is merely a replacement for a genuine explanation which would be given if the functional object-type intelligent person was explicitly described. The obvious analysis is that dispositional attributes need a definition that specifies the relations between goal-achieving capacities of objects and the circumstances under which these capacities will be revealed. Dispositional terms of themselves have limited application.

4.4.4 The Real Definition: Interaction between Object-types and Objects

There is a continuous interaction between an object-type and objects, so that the distinction between an object-type and objects diminishes and they correct each other mutually. In principle, there is no difference between specifying conditions of an object-type and describing attributes of objects. Object-types and objects interdependently determine the content of an object-type.

Let us assume we have a nominal goal-oriented definition of the term intelligent person. Suppose furthermore, that we have at our disposal an initial sketch of the object-type intelligent person containing one single condition brilliant university student. If the object in question is A. Einstein we encounter the problem that our object does not match the condition. A. Einstein was a moderate student and therefore we cannot classify him as an intelligent person. This is at variance with the classic picture we have of A. Einstein. In other words, our object-type fails to refer. The manoeuvre we perform in this (and similar situations) consists in adapting the object-type and incorporating the condition brilliant scientist. This modification suffices to secure A. Einstein's membership of the object-type intelligent person.
What happened here, is that a potential referent adapted the content of the object-type. Putnam (1975) calls this operation 'a procedure for preserving reference across theory change' (p.181). Putnam and Kripke (1972) have elaborated the influence of referents on the content of the object-type in their theory which states that the extension of an object-type is not assessed by the content of the object-type, but by the fact that we are causally linked to referents that determine the object-type. It is important to note here that in the causal reference theory 1) the content of an object-type is assessed by means of reference and 2) that the referent itself determines what other objects are intelligent too. Note also that the meaning of the object-type fish is also governed by the internal structure of the old fish (Derksen, 1980, p. 278).

We can obtain another solution for our Einstein-classification problem by identifying a student-context and a scientist-context in the reconstruction of intelligent person. Now, A. Einstein will belong to the extension of intelligent person_{scientist}, but not to the extension of intelligent person_{student}. We can view this switch to context as another indication that objects as well as contexts together and interchangeably determine the content of an object-type in a goal-directed classification process.

The interaction of and the fading distinction between object-types and objects is an important point of application for the knowledge level integration of AI and DBT: the distinction between knowledge of constraints (often called knowledge and often represented in AI-systems) and knowledge of objects (often called data and often represented in DB-systems) is a tentative one. It explains why there is a shift in DBT from relational to semantic database systems (Hull & King, 1987) and why expert systems evolve to expert database systems (De Broek, 1989).

\[\text{Source Hendriks, 1986}\]

\(\text{(A) Accounting for a Context}\)

The original conception of functional object-types as formulated by Van Der Smagt (1985) and Hendriks (1986) stresses the significance of a context in the reconstruction of the object-type. Figure 4.2 (A) shows Hendriks's schema of functional object-types. The schema is a one-sided matching-model. Only the context influences the content of
the object-type. In correspondence with the causal reference theory, Lucardie (1988; 1989) points to the potentials of objects to influence, together with the context, the contents of the object-type. This matching-model is two-sided (Figure 4.2. (B)). To prevent misconceptions, we state that our view on functional object-types does not approve of the theory of causal references as long as it does not account for the influence of goals and contexts.

(B) Accounting for Context and Objects

Figure 4.2: Schemas for Concept Analysis

4.4.5 Functional Equivalence

Object-types come into existence in close interaction with a context and with objects which strive to realise/perform a goal/function. According to the functional view an object-type of a concept is established by a goal-oriented reconstruction process in which a disjunction of conjunct sets is modelled. In the treelike schema of Figure 4.3 such a disjunction is pictured. This disjunction consists of three conjunct sets all leading to goal 1:

1. \{(i_1; a), (i_2; k)\},
2. \{(i_1; b), (i_2; r)\} and
3. \{(i_1; b), (i_2; s), (i_3; y)\}
An element of a conjunct set is an *inus-condition*: an Insufficient but Necessary part of the conjunct set which is Unnecessary but Sufficient for the result. Within a conjunct set an inus-condition is indispensable for achieving a goal, but the conjunct set itself, to which the inus-condition belongs, is replaceable by other conjunct sets.

As an object must satisfy one of the conjunct sets\(^1\) of an object-type in order to belong to the extension of a concept, the example, though greatly simplified, shows some interesting features of the functional approach. For instance, it is possible for objects, which at first sight are different, to be identical with respect to a goal and a context. Or, in other words, objects having different attributes, but matching with a conjunct set of an object-type in functionality, are equivalent. An example of two 'different' but functionally equivalent objects is formed by an object 1 characterised by the attributes \(\{(i_1; a), (i_2; k)\}\) and an object 2 having the attributes \(\{(i_1; b), (i_2; s), (i_3; y)\}\). In the context of goal 1 both objects are similar. Here, the notion of *functional equivalence* is essential: objects are identical, fall in the same concept or are similar if they possess - even quite different - attributes to perform the same function. Three mechanisms are responsible for the fact that a goal or a function is attainable by quite different strategies. We will illustrate these strategies by examples taken from the fire-safety regulations, one of the most complex parts of the building regulations, in the Netherlands.

The first strategy of functional equivalence is the mechanism by which, under certain conditions, other attributes (descriptors) may become important for determining class membership. In the third conjunct set \(\{(i_3; y)\}\) becomes a descriptor if \(\{(i_1; b), (i_2; s)\}\). This mechanism is effective in the fire-safety regulations limiting the extension of fire. These fire-safety-requirements for walls are influenced by the conceptualisation of rooms adjacent to the walls. Fire-compartments, (e.g. bath rooms, lavatories and traffic-rooms) are all examples of various types of rooms which lead to different fire-resistance requirements. Therefore, it is important to have adequate definitions of these rooms at one's disposal. If we look at the conceptualisation of a fire-compartment, we can see that quite 'different' rooms can be classified as a fire-compartment: both a heating room and a technical room can function as fire-compartments. But the technical room should have a user surface exceeding 50 m\(^2\). The user surface attribute is virtually a new descriptor, which is only useful for the conceptualisation of fire-compartments in case we are dealing with technical rooms.

The second important mechanism is the fact that categorisations of attributes of objects influence each other. This phenomenon is called *conceptual interaction*. In Figure 4.3 conceptual interaction manifests itself in the mutual influence of the categorisations of the first attribute and the second attribute. If \(\{(i_1; a)\}\), the classification of the second attribute is \((k, l)\). On the other hand, if \(\{(i_1; b)\}\) the classification of the second attribute is \((r, s)\). Referring again to the conceptualisation

---

\(^1\) The definition of the inus-conditionality is not without discussion. Mackie defines a I(NUS)-condition as follows: 'A is an inus-condition of a result P if and only if, for some X and some Y, \((A \text{ and } X) \text{ or } Y)\) is a necessary and sufficient condition of P, but A is not a sufficient condition of P and X is not a sufficient condition of P.' (Mackie, 1965, p.246). The definition of Mackie has been amended by Denise (1984) as follows: 'A is an inus-condition of a result P if and only if, for some X and some Y, \((A \text{ and } X) \text{ or } Y)\) is a necessary and sufficient condition of P, and A is a necessary condition of \((P \text{ and } X)\), but A is not sufficient condition of P and X is not a sufficient condition of P.' (Denise, 1984, p.56)
of fire-compartments conceptual interaction is present between the type of room and the user surface. If we are dealing with an enclosed room, the adequate categorisation of user surface is $\leq 500$ m$^2$ or $> 500$ m$^2$, because the user surface of an enclosed room should not exceed 500 m$^2$. However, if we are dealing with a technical room, the categorisation of user surface is $\leq 50$ m$^2$ or $> 50$ m$^2$. To be classified as a fire-compartment, the user surface of a technical room should exceed 50 m$^2$.

\[\text{Source: Lucardie, 1992, p.105}\]

\textit{Figure 4.3: A Disjunction of Three Conjunct Sets}

The third mechanism refers to the situation that objects may have different attribute values, but that this variation is limited to, or falls within, a goal-constructed category. Two objects characterised by the same $i_1$, but with different $k$-values -object 1 and object 2 respectively have the values $k_1$ and $k_2$ with $k_1$ and $k_2$ both falling in category $k$- are functionally equivalent. For instance, a technical room with a user surface of 55 m$^2$ and a technical room with a surface of 60 m$^2$ are functionally equivalent in the context of defining a fire-compartment.

Of course, reality is much more complex. Our example only concerns limiting fire-extension. It is easy to see that more attributes play a role: external walls, internal walls, thermal insulation, irradiance, fire-compartments, technical rooms, enclosed rooms, traffic rooms and so on. Furthermore, the definitions of different types of rooms also yield different attributes. For instance, the definition of a technical room states that it is an enclosed room for the installation of equipment necessary for the functioning of a building. This definition yields at least two extra attributes.
What can we gain from the previous analysis? Functional equivalence shows that no a priori defined concepts are allowed to describe a knowledge level model, but only goal-constructed concepts. From a functional viewpoint, it is a prerequisite that a knowledge level model should avoid describing its application domain by a priori fixed categorisations. Rather, a knowledge level model should avoid these reifications by accounting for goal-based dynamic, flexible categorisations of the environment. Under the influence of varying goals and changing contexts continuously differing descriptors will be needed to assess class membership. Creating a classification solely by the attributes of objects is practically impossible because objects that are in the same functional class can be vastly different. Not the object properties as such are relevant, but the functionally required properties. Functional equivalence stresses the structural heterogeneity of objects in the sense that in many situations, the presumption that objects are describable by fairly stable characteristics will be a misconception.

Functional equivalence significantly deviates from family resemblance. Just like members of one family, objects that show family resemblance can significantly differ from each other on many attributes. It is even possible that none of the objects share all attributes that lead to family resemblance. Family resemblance between objects can exist without one attribute being necessary or sufficient. Descriptions of objects that are classified on the basis of family resemblance contain overridable attributes. Stegmüller (1973) seems to advocate the notion of family resemblance. Brachman exposes the dangers of this prototypical approach:

'"The lesson here is that in order for a knowledge representation system to be able to handle any reasonable range of descriptions - even the simplest composites constructed from natural kind-like concepts-some type of definitional (i.e., compositional-not of the "typical" kind) - structuring capability is necessary. To form descriptions of a very common sort, necessity and sufficiency are demanded.'

(Brachman, 1985, p.87)

4.5 THE CURRENT PRACTICE OF RECONSTRUCTING OBJECT-TYPES

From the previous exposition on functional object-types, it will be clear that adopting the theory of functional classifications and accepting the occurrences of functional equivalence will have great impact on the reconstruction of an object-type. The primary focus of this section is on analysing the current practice of reconstructing object-types from a functional viewpoint. Since prototypical and probabilistic approaches prevail in AI and DBT, we will turn our analysis to the mathematical techniques that play a central role in these approaches.

A multitude of techniques is available to reconstruct object-types (Kim & Courtney, 1988; Neale, 1988; Reitman Olson & Rueter, 1987). These techniques range from direct techniques such as protocol analysis and interruption-analysis to indirect
techniques such as multiple regression, multidimensional scaling and cluster analysis. The indirect techniques pass through a series of intermediary steps, while the direct techniques follow a more straightforward path to reconstructing object-types. Since especially the indirect mathematical techniques display a number of common features that typify the probabilistic and prototypical approaches, we will take a closer look at these approaches.

By means of *multiple regression* it is possible to study the relation between a dependent variable and a linear combination of independent variables. The central question is how variation in a dependent variable can be traced back to variation in (combinations of) independent variables (Knippenberg & Siero, 1980). The relation between a dependent variable and a set of independent variables is described by an algebraic model of the following form:

$\hat{U} = a + b_1X_1 + b_2X_2 + \ldots + b_kX_k + e$

The model relates an dependent variable $\hat{U}$ to the variables $X_1$ to $X_k$. $b_1$ to $b_k$ are regression coefficients by which the independent variables are weighted. $a$ is the intercept that denotes the value of the independent variable if each independent variable is zero. $e$ is the error-term that should account for measuring errors. The model can be conceived as a description of an object-type. It could, for instance, stand for the object-type *fire-safe wall*. Then, $\hat{U}$ represents the degree of fire-safety and $X_1$ to $X_k$ represent the attributes of a wall such as *thermal insulation* and *irradiance* that influence the degree of fire-safety.

$\hat{U}$ is an estimated variable. The estimation is based on the dependent variables, their regression coefficients (or weights) and the intercept. The value of the variable *in reality* is denoted by the symbol $U$. Commonly, there is a difference between $\hat{U}$ and $U$. For example, the predicted degree of fire-safety of a wall, denoted $\hat{U}$, can deviate from the degree of fire-safety in reality denoted $U$. The difference is called the *residue of $U$*. It represents the part of $U$ that cannot be explained by $\hat{U}$.

$U = \hat{U} + \text{residue of } U$

The part of $U$ that can be explained by a linear combination of independent variables is expressed in a multiple correlation coefficient. This coefficient that relates $U$ to the independent variables $X_1$ to $X_k$, is denoted by the symbol $R$. Often, the square of $R$ is used:

$R^2_{U,1,\ldots,k} = .66$

A $R^2$ of .66 means that 66% of the variation of $U$ is explained by the model.

Independent variables can also be correlated to each other. If two variables are correlated, after the variation in the two variables that can be explained by a third variable has been accounted for, we call the correlation a *partial correlation*. The partial correlation between two variables $X$ and $Y$ is denoted as follows:
(4.7) \[ R_{XY,Z} \]

where \( Z \) is a dependent variable that explains part of the variation in \( X \) and \( Y \).

The reconstruction of an object-type using multiple regression takes place by measuring variables of objects and subsequently estimating the \( b \)-weights. The \( b \)-weights are estimated in such a way that the difference between \( \hat{U} \) and \( U \) is as small as possible. The least-square criterion affords a test for determining the distance.

(4.8) \[ \sum_{i=1}^{n} (u_i - \hat{u}_i)^2 \] is minimal (the least-square criterion)

\( N \) denotes the number of objects observed. A regression model with calculated weights could look as follows:

(4.9) \[ \hat{U} = .62X_1 + .40X_2 + \ldots + .12X_k - 2.0 \]

The model in (4.9) shows that variable \( X_k \) has a low weight and thus contributes little to the dependent variable. The individual contribution of independent variables can also be assessed by a stepwise introduction of each variable. This is called a stepwise multiple regression. Every time that a variable is incorporated in the model, the change of \( R^2 \) can be ascertained. This change of \( R^2 \) can be interpreted as a measure of the importance of the newly incorporated variable.

After having reconstructed the model, the calculated multiple correlation coefficients and the regression coefficients should be tested. This can be done by formulating a null-hypothesis. A null-hypothesis might be: \( R^2 = 0 \). The significance of \( R^2 \) can be determined using the \( F \)-statistic:

(4.10) \[ F = \frac{R^2 / k}{(1 - R^2) / (N - k - 1)} \quad D_f = k \text{ and } DF_{n} = N - k - 1 \]

After calculation of the \( F \)-value, the significance of \( R^2 \) can be looked up in an \( F \)-table. The higher the multiple correlation coefficient \( R^2 \), the \( F \)-value and the significance, the better the reconstructed model will perform. An elaborate description of multiple regression can be found in Pedhazur (1982).

When cluster analysis or factor analysis is used, the reconstruction of an object-type usually starts with a matrix of objects and their attributes. Such a matrix is displayed in (4.11).

(4.11) \[
\begin{bmatrix}
X_{11} & X_{12} & \cdots & X_{1p} \\
\vdots & & & \vdots \\
X_{n1} & X_{n2} & \cdots & X_{np}
\end{bmatrix}
\]
in which $x_{ij}$ represents the value of the $j$th variable for the $i$th object, $n$ represents the number of objects and $p$ the number of variables. The aim of cluster techniques is to determine on the basis of the a priori defined variables whether the objects can be classified as object-types. In most cases the extensions of these object-types $E_1, E_2, \ldots, E_n$ are such that:

\[
E_i \neq \emptyset \text{ and } E_i \cap E_j = \emptyset \text{ and } \bigcup_i E_i \text{ is the complete set of objects}
\]

\[
\begin{align*}
(4.12) & \quad E_i \neq \emptyset \text{ and } E_i \cap E_j = \emptyset \text{ and } \bigcup_i E_i \text{ is the complete set of objects} \\

& \quad \text{Clusters} \quad \begin{cases} \text{X} \\ \text{(n x p)} \end{cases} \quad \begin{cases} \text{Correlation} \\ \text{between} \\ \text{variables} \end{cases} \\
& \quad \text{Clusters} \quad \begin{cases} \text{X} \\ \text{(n x n)} \end{cases} \quad \begin{cases} \text{Similarities} \\ \text{between} \\ \text{objects} \end{cases} \\
& \quad \text{Factors} \quad \begin{cases} \text{R} \\ \text{(p x p)} \end{cases} \quad \begin{cases} \text{Factor} \\ \text{analysis} \end{cases} \\
& \quad \text{Groups} \quad \begin{cases} \text{of} \\ \text{individual objects} \end{cases} \\
& \quad \text{Groups} \quad \begin{cases} \text{of} \\ \text{variables} \end{cases}
\end{align*}
\]

(Source: Everitt, 1983, p.232)

**Figure 4.4: Basic Steps in the Reconstruction of Object-types**

While cluster analysis is concerned with the grouping of rows (i.e. the objects), factor analysis is concerned with the grouping of the columns (i.e. the variables). Cluster analysis and factor analysis do not directly operate on the raw data matrix $X$, but on a matrix derived from $X$, giving a measure of similarity or dissimilarity between each pair of objects (cluster analysis) or between each pair of variables (factor analysis). Figure 4.4. illustrates the basic steps that are involved in the reconstruction of object-types using cluster analysis or factor analysis.

In cluster analysis, the values of a similarity matrix $P$ are calculated by using coefficients such as the simple matching coefficient or Jaccard's coefficient (Everitt, 1983). Instead of a measure of similarity it is also possible to use dissimilarities or distances. The most familiar of these is the Euclidean distance, which is given by:

\[
(4.13) \quad d_{ij} = \left\{ \sum_{k=1}^{p} (x_{ik} - x_{jk})^2 \right\}^{1\over 2}
\]

In (4.13) the Euclidean distance is measured between two individual objects $i$ and $j$. For calculation of the distances between groups of objects summary statistics are needed. The most simple way is to calculate a mean value of a group and then use the
Euclidean distance of these means. Obviously, if the variation between objects of one group is great, this procedure is not appropriate. Then, the distance between objects belonging to different groups may not be well reflected by this measure. A measure that accounts for this problem is:

\[
D^2 = (\overline{x}_A - \overline{x}_B)^TW^{-1}(\overline{x}_A - \overline{x}_B)
\]

where \(\overline{x}_A\) and \(\overline{x}_B\) are the mean vectors of the two groups and \(W\) is a \(P \times P\) matrix of pooled within-groups sums-of-squares and cross-products for the two groups.

Finally, clustering techniques and algorithms are used to group the objects into classes or extensions of object-types. A large number of clustering techniques is available. A basic algorithm is as follows:

(a) Let \(e = n\) and \(E_1, ..., E_n\) each contain a single object
(b) Find the nearest pair of distinct clusters, say \(E_i\) and \(E_j\)
(c) Merge \(E_i\) and \(E_j\), delete \(E_i\) and decrease \(e\) by one
(d) If \(e = 1\) stop, otherwise go to START

In cluster analysis the extensions that are reconstructed in this way, are conceived as if they point to an object-type that underlies the extension. In factor analysis it is not groups of objects but groups of variables that are reconstructed. Such a group, called a factor, forms part of the description of an object-type.

For some time now, the assumptions underlying these algebraic models have attracted strong criticism (Timmermans & Van Der Heijden, 1987; Van Der Smagt & Lucardie, 1991). There are good reasons for this critical discussion. Let us take a look at a number of these assumptions. First, it is assumed that the variables or conditions to describe object-types and objects, are commonly known and defined in advance. The relevant conditions or attributes of objects are provided beforehand and are considered sufficient to reconstruct object-types (multiple regression) or extensions (cluster analysis). Likewise, these techniques do not take into account any goal-oriented principle that seems to underlie many human classification processes and that helps to gather relevant conditions. The goal-oriented and conditional relevance of conditions or variables is lacking.

The second assumption is that an object-type can be reconstructed inductively. While the mathematical techniques in essence are intended for testing purposes, they are often employed inductively to discover knowledge. The dangers of this probabilistic approach are obvious, when we think of the three mechanisms behind functional equivalence. The descriptions of objects used in the inductive reconstruction process, do not account for conditional relevance, conceptual interaction and goal-limited variations of attributes. In contrast to these a priori definitions of objects, the functional view states that descriptions of objects should be defined in a goal-oriented way and should account for functional equivalence. This implies that objects that are similar as to the recorded variables, may be dissimilar when other, functionally relevant variables, are used. When functional equivalence is established by 'heterogeneous' objects, a similarity matrix or a matrix with Euclidean
distances is not of much use. In the mathematical techniques discussed we find no recognition of the fact that objects exist by virtue of object-types and that distinct object-types may have identical extensions. Note that the inductive adaptation of weights in multiple regression when other variables in the description of objects are involved, has nothing to do with conceptual interaction. The adaptation of weights and the use of an error theory, are solutions that should compensate lack of knowledge, part of which is caused by not accounting for functional equivalences.

The third assumption is that the individual contribution of variables to an object-type can be assessed independently (think of the partial correlations in multiple regression) and may be compensatory. This assumption conflicts with our INUS-analysis which states that many variables can only in conjunction be effective in the realisation of a certain goal. Fire cannot be originated in the absence of oxygen or combustibles or an ignition. Nor it is possible to compensate a lack of oxygen by extra combustibles or to compensate the absence of an ignition by the addition of oxygen!

The mathematical techniques are representative for the probabilistic and prototypical methodologies. In many fields of AI and DBT such as machine learning and the discovery of knowledge from databases, these techniques form the cornerstone of knowledge acquisition (Frawley, Piatetsky-Shapiro, & Matheus, 1991). From a functional perspective, the assumptions of the techniques are, however, inherently limiting, not always valid and lack flexibility. This has far-reaching consequences for the validity of descriptions of object-types and objects.

Let us illustrate this observation by taking a global but more concrete look at the algorithms applied in neural network research. A possible starting point for the use of these algorithms is depicted in the matrix of Figure 4.5. It shows that an arbitrary object-type $X$ has three subtypes: $X_1$, $X_2$ or $X_3$. Three conditions are considered relevant in the reconstruction process of object-type $X$. The categorisation of every condition is exhaustive and exclusive. Every conceivable value of a condition can be placed in exactly one category. The conditions are input-units ($I_i$), whereas the subtypes are output units ($O_j$). The matrix reveals that a number of hypothetical objects are already classified by these conditions. When an object simultaneously matches an input-unit and an output-unit the corresponding cell-category is raised by one. Often the resulting matrix of frequencies is displayed in a network of connected nodes.

The learning procedure passes through three phases (Rosch, 1978). First, the cue-validities are determined. A cue-validity is a conditional probability that a specific output-unit will be made active if input-unit $I_i$ is active. This conditional probability is a relative frequency that is calculated by dividing the frequency of $I_i$ ($F_i$) being active during the learning procedure by the frequency of $O_j$ being active ($F_o$) at the same time.

\[
P(O_j = 1|I_i = 1) = \frac{F_i}{F_o}
\]

Example of a cue-validity:
\[(4.16) \quad P(O_1 = 1 | I_1 = 1) = \frac{7}{30} = 0.233 \]

<table>
<thead>
<tr>
<th>OBJECT-TYPE X</th>
<th>(X_1)</th>
<th>(X_2)</th>
<th>(X_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((O_1))</td>
<td>7</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>((O_2))</td>
<td>25</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>((O_3))</td>
<td>18</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>((O_4))</td>
<td>1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>((O_5))</td>
<td>35</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>((O_6))</td>
<td>14</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>((O_7))</td>
<td>10</td>
<td>30</td>
<td>18</td>
</tr>
<tr>
<td>((O_8))</td>
<td>15</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>((O_9))</td>
<td>25</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

*Figure 4.5: Matrix of Frequencies*

<table>
<thead>
<tr>
<th>OBJECT-TYPE X</th>
<th>(X_1)</th>
<th>(X_2)</th>
<th>(X_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((O_1))</td>
<td>0.233</td>
<td>0.433</td>
<td>0.333</td>
</tr>
<tr>
<td>((O_2))</td>
<td>0.498</td>
<td>0.254</td>
<td>0.254</td>
</tr>
<tr>
<td>((O_3))</td>
<td>0.260</td>
<td>0.347</td>
<td>0.391</td>
</tr>
<tr>
<td>((O_4))</td>
<td>0.021</td>
<td>0.489</td>
<td>0.489</td>
</tr>
<tr>
<td>((O_5))</td>
<td>0.432</td>
<td>0.259</td>
<td>0.308</td>
</tr>
<tr>
<td>((O_6))</td>
<td>0.832</td>
<td>0.058</td>
<td>0.117</td>
</tr>
<tr>
<td>((O_7))</td>
<td>0.172</td>
<td>0.517</td>
<td>0.310</td>
</tr>
<tr>
<td>((O_8))</td>
<td>0.319</td>
<td>0.425</td>
<td>0.255</td>
</tr>
<tr>
<td>((O_9))</td>
<td>0.714</td>
<td>0.000</td>
<td>0.286</td>
</tr>
<tr>
<td>Prototype-validity</td>
<td>2.044</td>
<td>1.439</td>
<td>1.190</td>
</tr>
</tbody>
</table>

*Figure 4.6: Matrix of Cue-validities (Prototypical Cue-Validities in Italicics and Prototype-Validity in Bold)*

All cue-validities are depicted in Figure 4.6.
The second step consists of assessing the largest or prototypical cue-validities of every condition for every output-unit. Subsequently, the prototype-validity is calculated. The prototype-validity is the summation of the prototypical cue-validities of every condition per output-unit. Both validities are represented in Figure 4.6.

Finally, the weights are computed by dividing the prototypical cue-validities by the associated prototype-validities.

\[
W_{ij} = \frac{\text{Cue-validity}_{ij}}{\text{Prototype-validity}_j}
\]

Figure 4.7 depicts the matrix of weights. When the network is fed with more objects and an input-unit and an output-unit are simultaneously activated the frequencies and thus the weights are raised. Consequently, the weights between this input unit and other output units are lowered. The adaptation of weights runs along lines similar to those in multiple regression.

<table>
<thead>
<tr>
<th>OBJECT-TYPE X</th>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( X_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_1 )</td>
<td>( O_2 )</td>
<td>( O_3 )</td>
<td></td>
</tr>
<tr>
<td>Condition 1 = F (I1)</td>
<td>0.109</td>
<td>0.300</td>
<td>0.279</td>
</tr>
<tr>
<td>Condition 1 = G (I2)</td>
<td>0.243</td>
<td>0.176</td>
<td>0.213</td>
</tr>
<tr>
<td>Condition 1 = H (I3)</td>
<td>0.127</td>
<td>0.241</td>
<td>0.328</td>
</tr>
<tr>
<td>Condition 2 = K (I4)</td>
<td>0.010</td>
<td>0.339</td>
<td>0.410</td>
</tr>
<tr>
<td>Condition 2 = L (I5)</td>
<td>0.021</td>
<td>0.179</td>
<td>0.258</td>
</tr>
<tr>
<td>Condition 2 = M (I6)</td>
<td>0.407</td>
<td>0.040</td>
<td>0.098</td>
</tr>
<tr>
<td>Condition 3 = P (I7)</td>
<td>0.084</td>
<td>0.359</td>
<td>0.260</td>
</tr>
<tr>
<td>Condition 3 = Q (I8)</td>
<td>0.156</td>
<td>0.295</td>
<td>0.214</td>
</tr>
<tr>
<td>Condition 3 = R (I9)</td>
<td>0.349</td>
<td>0.000</td>
<td>0.240</td>
</tr>
<tr>
<td>Prototype-validity</td>
<td>2.044</td>
<td>1.439</td>
<td>1.190</td>
</tr>
</tbody>
</table>

Figure 4.7: Matrix of Weights

These and other network algorithms are similar to the mathematical techniques discussed previously. The same restrictive assumptions form the basis for the reconstruction process. The assumption that we know approximately what conditions are important for the description of an object-type underlies Rosch's algorithm and other neural network algorithms. They assume that necessary conditions are provided beforehand. The acquisition of conditions is not considered a problem. This sharply contrasts with the functional view. The functional view stresses that conditions are not a priori provided but should be acquired through reasoning about goals of
classification. We have explained that goals can significantly influence the relevance of conditions and that a slight variation in goals may lead to different conditions. It is doubtful whether the initial assessment of conditions preceding the reconstruction process is goal-oriented. However, of more importance is the observation that the learning procedure itself lacks the flexibility to account for (an alteration in) goals. As soon as the learning procedure starts, the possibility of attending selectively to functionally required conditions is excluded.

When we continue the evaluation by analysing how the learning procedure copes with the three mechanisms behind functional equivalence other objections are raised. Consider a set of objects that is typified by the following object data: $C_1 = F$ and $C_2 = K$ and $C_x = O$. Suppose that because of these attributes, the objects are instances of object-type $X_1$. When the objects are being classified the algorithm raises the frequencies, the cue-validities and weights of $C_1(F)$ and $C_2(K)$ of $X_1$. The interpretation of this process is that $C_1(F)$ and $C_2(K)$ are important for the description of object-type $X_1$. It is not recognised that this importance is conditional upon the interaction between object-types and objects that are expected to perform certain functions within certain contexts, for reasons of simplicity, expressed in $C_x = O$. The reconstruction process does not allow the incorporation of this 'new' descriptor into the object-type.

Another objection relates to conceptual interaction. Rosch's algorithm 'tackles' conceptual interaction through an extensional adaptation of weights. This replacement of conceptual interaction by statistical interaction implies that the influence of conditions is assessed independently. It is not possible to have the models include that categories of conditions are dynamic, influence each other and are effective only in conjunction. Categories of conditions are a priori fixed. The learning procedure assesses the contribution of these categories independently of other categories. This excludes conceptual interaction. Since conceptual interaction is abundantly present in human classification processes, we think this a weak point in the reconstruction procedure. The learning procedure seems to deal adequately only with variability that is limited to certain categories. However, this can be solely assessed on the condition that the algorithm is capable of classifying objects with attributes like $C_1 F$, where $F_i$ is an element of $F$.

Furthermore, the functional view states that classification should account for contextual influences. Even when a goal and a context is given, it still appears very difficult for human experts to find the conditions. How can a neural network learn what the content of the object-type water is for implementation purposes when scientists are hardly able to provide an adequate definition for normal communication. How then can a neural network do this job 'without a teacher'? In Chapter 7 we describe an example of the classification of walls which displays the problems met by neural networks in their learning procedures.

We conclude that the derivation of object-types is probabilistic in the sense that the abstraction of object-types is extensional. Since objects can be instances of several object-types (multiple classification) and since object-types need not have common attributes (incommensurability), we think that inductive derivation has inherent dangers. As different object-types may have identical sets of objects (co-extensive object-types) the inductive elimination of random disturbances leads to invalid object-types. As we will see, these disturbances are eliminated by adaptations of weights,
error-terms and correction factors (Lucardie, 1989). Of course, one can object that representative conditions are applied in the reconstruction process and that consequently the picture is not so bad as presented here. Our objection, however, is that what is representative, is not made explicit in a functional manner. This automatically leads to the necessity of adapting weights and error-terms to compensate for lack of knowledge.

How does one assess that an object belongs to the extension of an inductively derived object-type? The classification of objects using Rosch's algorithm consists in measuring with what object-type the objects show the greatest degree of resemblance. The input-unit that corresponds with the attributes of the object will be made active for each object. To see whether an object is an instance of an object-type two steps are necessary. First, for every output-unit the input \( I_j \) is calculated by multiplying the weights of every activated \( U_i \) by activation-values (1 or 0) and by totalling up the products:

\[
I_j = \sum_i W_{ij} U_i
\]

Secondly, the activation value of the output-units is determined. The activation-value indicates whether an object-type is eligible for an object: \( 1 = \text{eligible} \) and \( 0 = \text{not eligible} \).

\[
A_j = \begin{cases} 
1, & I_j > T_j \text{ and } I_j \text{ wins the competition (} T_j \text{ is threshold)} \\
0, & I_j \leq T_j \text{ or } I_j \text{ loses the competition (} T_j \text{ is threshold)}
\end{cases}
\]

\( T_j \) is an arbitrary threshold. Of all eligible object-types, the object-type with the highest \( I_j \) wins the competition. Suppose, we have an object that has the attributes: G, L and Q. This means that the \( I_2, I_3 \) and \( I_8 \) are activated. Using (4.18) \( I_1 \) becomes 0.417, \( I_2 \) becomes 0.650 and \( I_3 \) becomes 0.685. \( I_3 \) wins the competition if \( T_3 = 0.650 \).

In AI and DBT it is common (prototypical) practice to use a continuum of degrees of membership to classify objects as object-types. In fuzzy logic, for instance, membership is assessed through a membership or compatibility function (Zadeh, 1965). A membership function operates upon a set and attaches to each element of this set a compatibility measure which represents the grade of membership. If we have a set \( U \) (4.20) and a membership function (4.21), we obtain a fuzzy set \( A \) (4.23 and 4.24). The integral sign (4.22) stands for the union of the elements of \( U \) after the compatibility function has been applied. A fuzzy set is a collection of ordered pairs which expresses the degree of membership of an element of that set. These elements originate from \( U \). From (4.24) it is clear that the element \( c \) does not belong to the fuzzy set \( A \).

\[
U = \{a, b, c\}
\]

\[
D(\text{omain})\mu_A = U \text{ and } R(\text{ange})\mu_A = [0, 1]
\]
Techniques such as multiple regression and algorithms like Rosch's are based on a prototypical approach. Over the past years the notion of a 'prototype' has caught on in knowledge representation research (Brachman, 1985). A reconstructed object-type is viewed as a stereotype or a prototype. Normally, this prototype is described by necessary conditions. Instances should show some sort of family resemblance with the prototype. Apparently, an object-type viewed as a prototype differs from a functional object-type. Not INUS-conditions, but necessary conditions assess the prototype. Not functional equivalence, but family resemblance determines whether an object is an instance of an object-type.

Because it is difficult to hold that all conditions are valid for all instances of the prototype in all conceivable circumstances, these necessary conditions inevitably have a default status. Due to this vagueness and due to the fact that universal quantified statements are wanted, the necessary conditions of a prototype need to be interpreted as default conditions which can be overridden or cancelled. So, these necessary default conditions can often be eliminated from the description of the prototype. While dealing with the problem of classifying reality, the prototype approach focuses on overridable default conditions.

From a functional perspective it is explainable why a prototype cannot hold for all instances. The three mechanisms of functional equivalence clarify why an object-type cannot be represented through necessary conditions. Objects may be instances of object-types in that they meet one of the conjunct sets that define an object-type. The prototype solution to interpret all conditions as defaults that are cancellable is therefore not adequate. Within a conjunct set none of the conditions is cancellable. The prototype-approach undermines the definitional capabilities needed to model complex object-types. Systems that are built on prototype assumptions, cannot prevent users from mutilating conjunct sets of INUS-conditions. The supposed need for representing exceptions such as three-legged elephants or birds that cannot fly has led to an underestimation of the importance of well-defined object-types by means of necessary and sufficient conditions (Brachman, 1985). But necessary conditions are certainly not sufficient for being an instance of an object-type. The prototype proponent strongly believes that no combination of properties is sufficient to capture an object-type or to define natural kinds of concepts (Putnam, 1977).

Furthermore, in probabilistic and prototypical approaches the process of assessing class membership is reversed. Typically defining properties are not used to see whether an object belongs to the extension of an object-type. On the contrary, objects should be classified as instances and then it is derived that an instance possesses the necessary conditions. For example, a description of an object-type $KS(X)$ that is not explicitly defined by means of necessary conditions $C_i$ will be like this:

(4.22) $A = \sum_{\mu} \mu_A(x) / x$

(4.23) $A = \mu_1 / X_1 + \mu_2 / X_2 + \mu_3 / X_3$

(4.24) $A = 0.3 / a + 0.8 / b + 0 / c$
(4.25) \[ \forall x \in KS(X): C \uparrow F \subseteq \text{dom}(x) \] where:

1. \( F \subseteq \{C_1, C_2, \ldots, C_n\} \)
2. \( x \) is a function representing an object

The degree of resemblance depends on the ratio between \( C \) and \( F \). In the prototype approach an object-type is not applied to recognise instances. Note that this reverse form of reasoning leads to problems. An object that is a bird need not be able to fly. The conditions that define a bird are by no means identical to the conditions that define flying objects. The object-type *bird* and the object-type *flying object* have different contents and need to be distinguished! Many birds being able to fly only implies that objects that are birds also comply with the requirements of the object-type *flying object*. Technically, we can say that when an object complies with one of the conjunct sets (and therefore is an instance of a bird), it is not legitimate to infer that it will also comply with the other conjunct sets of the disjunction (and therefore be able to fly). Viewing objects that are birds as flying objects is a form of *reification*.

Knowledge acquisition based on probabilistic and prototypical assumptions prevail in AI and DBT-research for instance in the 'discovery' of quantitative laws (see for instance: Zytkow & Baker, 1991). Though the demands made on computers can be diminished by letting computers perform rediscoveries instead of discoveries (Koppelaar, 1990), there are other ways of coping with the limitations of the assumptions underlying this type of research. First, we should avoid a purely inductive approach. The dangers of a purely inductive approach are generally known, but not always recognised. Second, the variables that play a role in the process of knowledge acquisition should be of a compensatory nature, that is, a reduction in the value of one variable can be compensated by raising the value of another variable. Third, conceptual interactions should not occur, because an algorithm can hardly detect them, this is especially difficult when conceptual interaction involves more than two variables. In Chapter 7 we describe an example of conceptual interaction and the learning problems of a neural network algorithm in a particular domain of the building and construction industry. Fourth, it is desirable that a supervisor should be present to guide the learning process. Obviously, this supervisor should have functional knowledge at his disposal.

### 4.6 A FUNCTIONAL EVALUATION OF RECORD-BASED REPRESENTATION FORMALISMS

Among other things, the distinction of the knowledge level provides computer scientists with an explicit perspective from which to evaluate knowledge representation formalisms. Clancey (1985) undertook such an evaluation concerning production-rules and Brachman (1985) did the same for frame-based representation formalisms. There is an important difference, though, between Clancey's analysis and Brachman's. Clancey intended to uncover the knowledge that lies behind the production rule structures, while Brachman's goal was to show that a frame-based
knowledge representation formalism forces a knowledge engineer to work according to a disadvantageous prototypical approach. The interesting thing is not that Brachman, just like Clancey, is aware of the importance of distinguishing the knowledge level, but that he evaluates a theory—in this case the prototypical theory—of the nature of knowledge. In his analysis Brachman acknowledges the precondition that a theory defining the nature of knowledge is essential. The preceding sections of this chapter fulfilled this precondition by explaining the theory of functional classifications and comparing it with competing theories. The central purpose of this section is to investigate, from a functional viewpoint, the advantages and disadvantages of representation formalisms that are based on record-structures and processes.

Records are widely accepted as representation formalisms. Most commercial database management systems, applied on a large scale in trade and industry, are implementations of record-based structures. On the one hand, record-structures are generally considered useful due to their efficiency and simplicity, on the other hand, we notice that the basic assumptions behind record-structures are subject to severe criticism. Kent (1979), for instance, points at the limiting assumptions of vertical and horizontal homogeneity that underlie record-based models. By viewing records from a functional perspective we contribute to the critical discussion from a somewhat different angle. The questions which arise are: How can records represent an object-type defined in a goal-oriented way? and: How can records cope with functional equivalence which underlies a goal-oriented definition of an object-type?

A record is a fixed sequence of field values, conforming to an object characterisation (see Chapter 3) usually contained in catalogues or programs. The object characterisation consists mainly of a name, length and type of each field. Every description defines a record-type (or, in relational terms a relation or table).

Functional equivalence also includes situations in which objects indeed have similar attributes for performing a certain task. In many cases, however, a set of objects each matching the definition of an object-type, shows considerable variation in the attributes relevant to each individual in that set. Objects may vary in whatever attributes, but if they match one of the conjunct sets of a goal-constituted object-type, they are all functionally equivalent. So, functional equivalence implies that objects may need different (combinations of) attributes to meet the description of an object-type (Van Der Smagt & Lucardie, 1991). We can illustrate this by an example. Suppose we want to describe an object-type X. Suppose also that the conjunct sets below describe object-type X. Together these conjunct sets indicate that a considerable heterogeneity may exist in the attributes of objects in order to belong to an object-type.

\[(1) \ ((A_1; V_1), (A_2; V_2), (A_3; V_3)) \rightarrow X\]

\[(2) \ ((B_1; V_4), (B_2; V_5), (B_3; V_6)) \rightarrow X\]

\[(3) \ ((C_1; V_7), (C_2; V_8), (C_3; V_9)) \rightarrow X\]

\[A_i \neq B_i \neq C_i\]
When such a goal-dependent variation exists over a population of objects, there are certain techniques for accommodating this variability in a record-based database design (Kent, 1979). One solution is to include all the relevant attributes defining an object-type in a single table multi-field design. In this design not all fields will have values in every record. Many records can even have null-values in many fields. The definition of the knowledge scheme of this design can be assessed in a function $S$. The domain of this function is the name of the object-type and the range represents the relevant attributes:

$$S(X) = \{A_1, A_2, \ldots, C_3\}$$

A selection of the corresponding knowledge table of a knowledge state $KS$ over $S$ is:

$$KS(X) = \{
\{(A_1; V_{1.1}), (A_2; V_{2.1}), (A_3; V_{3.1}), (B_1; null), (B_2; null), (B_3; null), (C_1; null), (C_2; null), (C_3; null)\},
\{(A_1; null), (A_2; null), (A_3; null), (B_1; V_{4.1}), (B_2; V_{5.1}), (B_3; V_{6.1}), (C_1; null), (C_2; null), (C_3; null)\},
\{(A_1; V_{1.1}), (A_2; V_{2.1}), (A_3; V_{3.1}), (B_1; null), (B_2; null), (B_3; null), (C_1; V_7), (C_2; V_8), (C_3; V_9)\}\}$$

$V_{ij} \in V_i$

**Figure 4.8: A Record-Type with Multi-Attribute Objects**

Figure 4.8 shows that a database design in which all attributes are represented in multiple fields of one record-type, is likely to have many records with null-values in many fields. There are many interpretations of a null-value, e.g. not permitted, to be stored, undergoing change, missing etc. However, formal treatment of null-values usually includes the meanings 'not applicable for this object' and 'applicable but value at present unknown'. These two types seem to capture all other interpretations. An approach that handles both types of null-values is described by Vassiliou (1979). Codd (1979) describes extensions of the relational algebra for dealing with the 'value at present unknown' type. Null-values especially pose problems when we would like to use corresponding fields for identification purposes, which is a fairly common situation. Not surprisingly, the database designer is very careful to prevent the appearance of null-values with specialised schema constructions.
and strict rules governing modifications of record values. It appears, however, that such an approach is inappropriate in a number of cases.

Another serious problem of our first database design is that the (explicit) description of an object-type is lost. The conditional relevance of attributes is obscured in patterns of attribute values which exist independently of each other and which do not reflect the underlying conceptual structure of object-type X. As this limited relevance is not defined in the system, no system-facility is likely to be present that will enforce a correct input. Validation gets complicated and this means that the multi-field format endangers the integrity of the model.

Sticking to the single table multi-field design, we can also allow the use of general fields. Taking our example as point of departure, we can have separate fields to store values for $A_1$, $B_1$ and $C_1$ in different records. In this design the same field is allowed to have different meanings. A drawback of this approach is that every field-value has the same meaning for the system. Besides, the field names become unintelligible, because they have to accommodate considerable variety. The records within this design do not convey any meaning. It is only in the buried semantics of application programs that the significance of field values can be made clear. This is in contradiction with the ISO 100% Principle that prescribes that a knowledge universe should explicitly contain all relevant static and dynamic rules and laws of the Universe of Discourse so that a knowledge universe cannot be held responsible for aspects described elsewhere, in particular, those described in application programs (Twine, 1989).

To preserve the semantic structure and to decrease the number of null-values we can switch to another design. In this new design we discern (multi-field) record-types for objects which are instances of an object-type. Each record-type represents objects which match a specific conjunct set of an object-type. The corresponding knowledge scheme is then:

$$
S(X_1) = \{A_1, A_2, A_3\} \\
S(X_2) = \{B_1, B_2, B_3\} \\
S(X_3) = \{C_1, C_2, C_3\}
$$

The knowledge state(s) operating over this scheme, describe(s) the distinct record-types:

$$
KS(X_1) = \\
\{ \\
(A_1; V_{11}), (A_2; V_{21}), (A_3; V_{31}), \\
(A_1; V_{12}), (A_2; V_{22}), (A_3; V_{32}), \\
(A_1; V_{13}), (A_2; V_{23}), (A_3; V_{33}) \\
\}
$$
\[ KS(X_2) = \\
\{ \\
(B_1; V_{4,1}), (B_2; V_{5,1}), (B_3; V_{6,1}), \\
(B_1; V_{4,1}), (B_2; V_{5,2}), (B_3; V_{6,1}), \\
(B_1; V_{4,1}), (B_2; V_{5,1}), (B_3; V_{6,2}) \\
\} \\
KS(X_3) = \\
\{ \\
(C_1; V_{7,1}), (C_2; V_{8,1}), (C_3; V_{9,1}), \\
(C_1; V_{7,2}), (C_2; V_{8,2}), (C_3; V_{9,2}), \\
(C_1; V_{7,1}), (C_2; V_{8,2}), (C_3; V_{9,2}) \\
\} \\
\]

**Figure 4.9: A Multi-Record Multi-Field Design**

At first sight, the multi-table design seems appropriate. The semantic structure is preserved and null-values have disappeared (Figure 4.9). Yet a number of disadvantages are associated with this design. Instead of going to one table to retrieve knowledge of objects which are object-types, a user has to know the name of every table and be prepared to interrogate each of the tables. Interrogation, however, is not simple. Because names of record-types and field names are only placeholders, a database management system cannot provide ways to deliver answers that are field or table names. Records can only supply knowledge by extracting field-values. This normally suffices for the system for matching keys or sequencing, but for representing (functional) knowledge field names table names should convey something intelligible. A second disadvantage is that one object may appear in different record-types. Validation and modification still are problems. Again, integrity pitfalls endanger the suitability of this design.

To extend the previous design, we can aggregate the objects involved into a limited number of 'supertypes' giving them a new identifier. In contrast to the previous design, the states (record-types) are linked now to each other through these new identifiers (Figure 4.10). Each of these additional identifiers needs to be assigned new values which should be recognised in various contexts. This can be formally assessed in the following three knowledge universe constraints (Chapter 3):
Attribute Type is the inspection attribute of $X_0$ and:

1. The value $X_1$ is the **inspection value** for $KS(X_1)$
2. The value $X_2$ is the **inspection value** for $KS(X_2)$
3. The value $X_3$ is the **inspection value** for $KS(X_3)$

The function $KS$ also needs extension:

$$KS(X_0) = \{
(Id; x_1, (Type; X_1),
(Id; x_2, (Type; X_2),
(Id; x_3, (Type; X_2)
... $$

A drawback of the 'supertype' approach is the very nature of functional classifications (Figure 4.10). Objects get classified one way to belong to a certain object-type and another way for another (sub-)object-type. Each object-type is potential ground for other supertypes. Important to note is that we are not dealing with a simple nesting of super- and subtypes: all objects (for instance employees) are an object-type $X$ (for instance people), but some of them also belong to other object-types (for instance customer and stockholder) and others do not. Nor are sub-types mutually exclusive. But to fit into a record-based discipline, the subtypes need to be perceived as if they are not overlapping, distinct objects.

We have seen that one of the advantages of working at the knowledge level is the possibility to analyse representation formalisms at the symbol level. When we examine the basic properties of record-based information models from a functional viewpoint, we gain some interesting insights. If the essential configuration of the conjunct sets leading to the same goal is characterised by identical attributes, each attribute having the same kind of values, records are excellent representation and processing tools. If, on the other hand, these conjunct sets are characterised by heterogeneity caused by having to classify objects in a goal-oriented fashion accounting for different descriptors and new conceptual interactions, records are not appropriate (Kent, 1979; Murdoch & Johnson, 1990). The more a domain deviates from homogeneity, the less appropriate the record configuration is. Unfortunately, each of these designs has its drawbacks. In short, we can state that all these record-
based solutions of coping with heterogeneous but functionally equivalent objects, remain makeshifts. The three basic design configurations of record structures discussed here cannot sufficiently reflect the semantic structure of functional object-types when considerable variation exists. The remarks made here apply to any model based on record-constructs. This includes the hierarchical, network, relational and entity relationship models. Semantic models are also included. In DBT, in the sphere of modelling, semantic models are often used. When working towards implementation the semantic models are converted into record-based relational models. An example of this practice is described by Hull & King (1987).

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$X_1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$X_1$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$X_2$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$X_2$</td>
</tr>
<tr>
<td>$x_5$</td>
<td>$X_3$</td>
</tr>
<tr>
<td>$x_6$</td>
<td>$X_3$</td>
</tr>
<tr>
<td>$x_7$</td>
<td>$X_3$</td>
</tr>
</tbody>
</table>

$V_{ij} \in V_i$

Figure 4.10: A Multi-Record Multi-Field Design with Super/Subtyping

The evaluation of records was not very stringent because of the fact that we used as point of departure a situation in which the description of an object-type was available. However, the majority of practical situations is less ideal. Often, the lack of an explicit functional definition of an object-type leads to continuous modification and design activities. As stated before, these activities increase the probability of integrity violations. Many record-structures need to be modified and redesigned which is, as stated before, a difficult thing to do.

To preserve flexibility the definitions of view-systems are often proposed as solutions. Departing from a general conceptual schema in DBT view-systems can be defined for specific interpretations of a knowledge universe. This is misleading as far as the suggestion is aroused that it concerns functional object-types. A view-system consists of nothing more than selections from basic tables (see Chapter 3 and 7). In the majority of cases, the object-types represented in the tables are not functionally reconstructed.
4.7 CONCLUSION: IMPLICATIONS AND PERSPECTIVES

This chapter emphasised that the reconstruction of knowledge universa boils down to the reconstruction of object-types. Despite general acknowledgement of the importance of object-types and despite the fuzziness that emerges when object-types have to be reconstructed, theories of the nature of object-types are quite remarkably an under-exposed issue in AI- and DBT-communities. This chapter addressed two fundamental questions about the sources of fuzziness:

How do terms relate to object-types?
How do object-types relate to objects?

With respect to answering these questions, probabilistic and prototype theories are dominant in AI and DBT. In these approaches the reconstruction of an object-type appears to be based on extensions. The result of this inductive procedure is an object-type that has the theoretical status of a prototype which is described by overridable necessary conditions. The allocation of objects to object-types takes place by the degree of similarity objects show with a prototypical object-type. In both approaches measures of similarity are goal- and context-free: that is the similarity of two objects A and B depends solely on a priori assessed attributes. The choice and mutual influence of attributes is not influenced by an object-type that accounts for goals and contexts.

The functional attempts show that both theories have some quite serious flaws. They make one crucial type of object-type impossible to model: that of composite descriptions through necessary and sufficient conditions reflecting functional equivalence. From this perspective, it is not surprising that the results of a prototypical approach are very difficult to interpret. This is true, for instance, for the classes that are reconstructed by using cluster analysis (Everitt, 1983). Rather than to group entities on the basis of similarity, the functional approach makes classifications on the basis of functional equivalence. If our main concern is with the reconstruction of a knowledge universe the proper framework is functional rather than probabilistic or prototypical.

A functional perspective does not automatically imply a decline of general object-type (model), it just points to the limited value of many general object-types. The line of reasoning is often as follows: The best way to reconstruct an object-type with general validity is to look mainly at real objects and to incorporate the necessary conditions. If a general object-type is eventually available then the object-type can, if necessary, be adapted for special goals. The model of a building, for instance, can be modified for fire-safety purposes, energy-consumption and so on. The inductive abstraction of a general object-type on the basis of a priori given attributes of

---

2 The fact that the term functional has several connotations in computer science may cause some confusion to the reader. At least three different connotations can be discerned. In the first place, there is the use of functional in functional classification which refers to goal- or function-based conceptualisations of objects. Secondly, functional in: a functional view of knowledge representation which may relate to the service performed by a knowledge representation component in a knowledge based system. (Levesque, 1984; Brachman & Levesque, 1986). Thirdly, functional may refer to functional models demonstrating how a function of a unit follows from the functions of its parts. The first connotation of the term functional is used in this study.
available objects, is problematic from a functional perspective. As explained before, deriving object-types from an extension can lead to object-types that are not valid for certain goals and contexts. The methodology in which the object-type is reconstructed as a prototype will be rejected from a functional point of view as well. Cancelling attributes to deal with exceptions are indications that knowledge of goals or contexts is not included in the general model. A description of an object-type building, reconstructed to deal with fire-safety issues, will differ from a description of an object-type building that should serve to design energy-saving buildings. There is no reason to suppose that these object-types will have common (classifications of) attributes. On the contrary, since the concepts that play a role in fire-safety domains (irradiance, thermal insulation, fire-compartment) are quite different from the ones that are important for energy-savings (emission of materials, supply of raw materials, recycling), the expectation is that these object-types may be incommensurable.

The second implication of the theory of functional classification is the acceptance that there is a continuous interaction between object-types and objects so that the distinction between object-types and objects fades away and they mutually correct each other. By this, the knowledge universe accounts for a unified approach of integrating AI and DBT.

The third implication refers to the suitability and the choice of a language that can be used to reconstruct functional object-types. This language should not only offer facilities for the reconstruction of functional object-types, but also for their validation for properties such as completeness, consistency and correctness. Further, the language should provide facilities for simulating the behaviour of functional-object-types when they are implemented.

The theory offers perspectives for new fields of AI and DBT such as object-oriented analysis & design and semantic database models. What is interesting in object-oriented analysis & design is that the world is modelled in terms of object-types and objects. The theory of functional classifications helps to systematically distinguish object-types and objects. Semantic models were introduced primarily as design tools for conceptual models. Central has been the development of abstraction mechanisms to represent the structural aspects of objects. These mechanisms are comparable to those selected in AI research. In recent years the attention has been turned toward incorporating dynamic aspects in the semantic models. The primary components of semantic models are the explicit representation of objects, attributes, relationships among objects, type constructors, 'is-a' relationships and derived schema components (Hull & King, 1987). Object-types can be modelled as abstract, printable or constructed and can be defined using 'is-a' relationships. Conceptual representation of an object-type once required is possible, but not adequate for maintenance and modification. It is not easy to get to grips with the flexibility of object-types. In this, functional object-types can help to make clear what has to be modelled by the tools and abstraction mechanisms offered.

Besides theoretical perspectives, the theory also offers practical perspectives. Reitsma (1990) developed a Decision Support System for the Shanxi-province in China on the basis of the theory. Lucardie, et al., (1994) developed a computer-based tool to specifically deal with functional object-types. We discuss the tool in Chapter 6.
Functional object-types are very fruitful, but reconstructing them is an arduous task that requires long inference chains of knowledge elements and complex logic expressions. Therefore, we have to invest in methods, techniques and tools to enhance the functional acquisition of knowledge and to simplify these complex logic expressions. Especially, we should pay attention to a formal language to reconstruct knowledge level models. A formal language should support the validation as well as the simulation of a functional knowledge level model. The importance of a formal language is stressed in Cooke (1992) and Balder & Akkermans (1992). Chapters 5 and 6 discuss such a structural investment.
CHAPTER 5

DECISION TABLES AND PROLOG AS A CONCEPTUAL MODELLING LANGUAGE FOR FUNCTIONAL-OBJECT-TYPES

5.1 INTRODUCTION

In Chapter 1 we stressed the importance of integrating the theory and practice of AI and DBT for modelling knowledge. In Chapter 2 the knowledge level and the symbol level were discussed as two computer systems levels at which this integration can take place. In Chapter 3 we argued that an integration of AI and DBT should be realised at the knowledge level rather than at the symbol level. In addition, we explained that a knowledge level integration requires (1) a theory of the nature of knowledge and (2) a language to analyse, represent and simulate knowledge. After proposing, in Chapter 4, the functional view on object-types as a theory of the nature of knowledge, in the present chapter the question remains what modelling language to choose.

Such a language should comply with a number of requirements. It should not only offer adequate expressive power for modelling real-world knowledge, but also provide validation and simulation facilities (Loucopoulos & Karakostas, 1989). For our purposes, the modelling language should not only offer facilities for the representation and reconstruction of functional object-types, but also for their validation on completeness, consistency and correctness. Furthermore, the language should supply facilities to simulate the behaviour of functional object-types (Lucardie, De Gelder, & Huijsing, 1994).

Mathematical logic is an obvious candidate language. As explained, it is suited to the formal and unambiguous specification and representation of functional object-types. For many people, however, mathematical logic may have the disadvantage of being too complex and, consequently, of restricting modelling and validation possibilities. Another disadvantage of mathematical logic is that there are no facilities for simulating the behaviour of functional object-types. What we need is a language that preserves the strong points of mathematical logic without displaying its weak points: a language that allows logic representation, permits easy inspection and offers facilities for the simulation of already specified functional object-types.

The joint application of Decision Tables (DT's) and Prolog seems to meet these requirements. Both have a firm basis in mathematical logic. Some authors claim that, together, they offer a range of powerful formalisms and techniques allowing a formally unambiguous description of real-world phenomena that is close to natural understanding (Reilly, Salah, & Yang, 1987, p.30). DT's are a method for organising and documenting knowledge in a logical manner that permits easy inspection and
analysis. Prolog can be viewed as a logical specification language that admits simulation of specified knowledge (Lazarev, 1989).

The research issue of this chapter is to investigate in more detail whether the joint application of DT's and Prolog indeed possesses the necessary characteristics to act as a language for the representation, reconstruction, validation and simulation of functional object-types.

The organisation of the chapter is as follows. From the perspective of the previous requirements, we first discuss the potentials of DT's as a modelling language on the basis of formal definitions of DT's (Section 5.2). Then, the capabilities of Prolog as a modelling language are explored by referring to its formal background in predicate logic (Section 5.3). Finally, we round off the chapter with a number of conclusions concerning the utility of the joint application of DT's and Prolog as a modelling language (Section 5.4).

5.2 DECISION TABLES

DT-work began over three decades ago. Much of the literature is devoted to the applicability of DT's in all phases of the software engineering life cycle from conceptual modelling (through design, implementation, testing, modification and documentation) to maintenance. The advantages of DT's for software engineering purposes are expressed by Reilly, Salah & Yang:

'...the ubiquitous use of DT's within the life cycle has conferred on them a reputation for compactness, self-documentation, modifiability handling complex logic, redundancy and completeness checking, high degree of non-procedurality, and automatic conversion to code.' (1987, p.191)

We can observe this 'software engineering bias' not only in early literature on DT's (Cantrell, King, & King, 1961; Grad, 1962), but also in more recent publications (Metzner & Barnes, 1977; Mors, 1993; Reilly, et al., 1987; Subramanian, Nosek, Raghunathan, & Kanitkar, 1992; Vanthienen, 1988; Verhelst, 1980). Yet, through the history of DT's, the emphasis on their use in a specific software engineering phase has been subject to change.

After a short period in which DT's were studied as structured alternatives for the classical flowcharts, DT-work soon concentrated on the process of automatically converting DT's into computer programs. A great deal of energy has been spent on algorithms that produce optimal programs (Bayes, 1973; Ganapathy & Rajaraman, 1974; Lew, 1978; Pollack, 1965; Reinwald & Soland, 1966; Reinwald & Soland, 1967; Shwayder, 1971; Verhelst, 1972). The algorithms generally fall into two categories: (i) those attempting to minimise the expected execution time of the program generated from the DT and (ii) those seeking to minimise the storage space required by the generated program.
During the last fifteen years research interest for the use of DT's in the implementation phase has gradually faded. On the one hand, this was related to the emergence of *structured programming* because of which DT's became less necessary for reducing the complexity of programs. On the other hand, DT's were increasingly recognised as means for communicating complex logic, the applicability of which is not restricted to implementation purposes. Furthermore, practical experience indicated that it cannot be taken for granted that well-reconstructed DT's will be available which was the point of departure for much implementation research. Rather, practice revealed that the reconstruction of DT's is a complicated task that should be the central focus in DT-research. For these reasons, attention shifted from converting existing DT's to computer programs, to the effective reconstruction and validation of DT's. By this shift, DT-research was focused again on the first stage of software engineering, namely that of conceptual modelling.

The renewed interest for conceptual modelling corresponds with the central theme of this section: the investigation of the potentials of DT's as a conceptual modelling language for (functional) object-types. This section is structured as follows. First, we provide formal definitions of DT's (Section 5.2.1). Then, we investigate the potentials of DT's for the representation, reconstruction, validation and simulation of functional object-types (Section 5.2.2). Finally, the section is rounded off with a number of conclusions and a discussion concerning the potentials of DT's as a modelling language for functional object-types (Section 5.2.3).

### 5.2.1 Formal Background

A DT can be informally defined as:

> "...a table that represents the exhaustive whole of mutually exclusive conditional statements within an a priori defined problem domain." (Verhelst, 1980, p.9)

An example of a DT is presented in Figure 5.1. The table named *Abstract*, refers to a fictitious domain. The component to the left of the double vertical line is called the *stub*. The first part of the stub, the part that is located above the double horizontal line, contains *condition subjects*. The second part of the stub, located below the double horizontal line, contains *action subjects*. The component to the right of the double vertical line displays six conditional statements about our fictitious domain. These statements are called *Decision Rules* (DR's). They are pictured by means of columns. DR's describe the connection between condition subjects and action subjects. Above the double horizontal line the DR's contain a *condition alternative* for each condition subject. Below the double horizontal line the DR's contain an *action alternative* for each action subject. According to Verhelst's definition a DT must be *exhaustive* and *exclusive*. Exhaustiveness means that, within the domain of the DT, every possible combination of condition alternatives should be accounted for. Exclusiveness means that no situation is permitted to be described in more than one DR.
Verhelst's definition designates the most important characteristics of a DT. A drawback, however, is that it is an informal definition. Since in the past, informal definitions of DT's led to DT's which are inappropriate in some ways, we need a more formal statement to apply DT's adequately and to evaluate their potentials. An important reaction against informal definitions is the report of the Codasyl Decision Table Task Group A Modern Appraisal of Decision Tables (Beitz, Buck, Jorgensen, Larson, Maes, Marselos, et al., 1982). The Task Group's theoretical foundations explained in this report provide definitions to identify various DT components and form an important stimulus to more formal definitions of DT's. The Task Group considers a DT as a tabular presentation of the triplet of a condition set $C$, an action set $A$, and the relations between the condition and action set $R$:

$$DT = (C, A, R)$$

In the following, we present formal definitions (5.1 up to 5.14) for each component of the triplet, which are mainly based upon definitions of A Modern Appraisal of Decision Tables. For this purpose, material of Chapter 3 is employed. As a supplement to this, we also make use of indexed sets to denote families of sets. An indexed set has the form $S = \{S_i | i \in I\}$ where $S$ is a family of sets and where $I$ is a set of (numerical) indices. An alternative way to describe a family of sets is: $S = \{S_1, S_2, S_3, ..., S_n\}$.

**Conditions ($C$)**

A condition consists of a condition subject and an accompanying set of condition alternatives. We can describe a condition as an ordered pair. For the first condition of our example DT, denoted $C_1$, we obtain:

$$C_1 = (\text{Condition Subject 1}; \{A, B, C\})$$

The first co-ordinate of $C_1$, denoted $\pi_1(C_1)$, is condition subject 1. It represents the name or the subject of the first condition. The second co-ordinate of $C_1$, denoted
\[ \pi_2(C_i) \], is the set of alternatives of the first condition: \{A, B, C\}. Each alternative itself is a set of values. For instance, if \( \pi_1(C_i) \) stands for *temperature* alternative A might stand for all temperatures between 0 and 80. This is the interval (0,..., 80). If \( \pi_1(C_i) \) stands for *colour*, alternative A might stand for the colour blue. This is the set \{blue\}. The formal definition of a condition set \( C \) is given below. \( C_{num} \) is, just like in other definitions, the number of conditions.

**Definition 5.1: Condition**

\[
C = \{C_i | i \in [1, \ldots, C_{num}] \text{ and each } C_i \text{ is an ordered pair that consists of a condition subject } CS_i \text{ and a set of condition alternatives } CA_i \}
\]

\[
C \text{ is an indexed set. The set } [1, \ldots, C_{num}] \text{ is the set of numerical indices. Applying the definition of a condition set to the DT Abstract, we get:}
\]

\[
C = \{(Condition \ Subject \ 1; \{A, B, C\}), \ (Condition \ Subject \ 2; \{D, E\})\}
\]

Using the numerical indices, we get:

\[
C_1 = (Condition \ Subject \ 1; \{A, B, C\})
\]

\[
C_2 = (Condition \ Subject \ 2; \{D, E\})
\]

The definition of a set of condition subjects is:

**Definition 5.2: Condition subjects**

\[
CS = \{CS_i | i \in [1, \ldots, C_{num}] \text{ and each } CS_i \text{ is a condition subject}\}
\]

Applying this definition to the DT Abstract, we get:

\[
CS = \{Condition \ Subject \ 1, \ Condition \ Subject \ 2\}
\]

Using the numerical indices, we get:

\[
CS_1 = Condition \ Subject \ 1
\]

\[
CS_2 = Condition \ Subject \ 2
\]

The definition of a set of condition alternatives is:
**Definition 5.3: Condition alternatives**

\[ CA = \{CA_i \mid i \in [1, \ldots, C_{\text{num}}]\} \text{ and each } CA_i = \{CA_{ik} \mid k \in [1, \ldots, n_i]\} \text{ and } n_i \text{ is the number of conditions alternatives of condition } i \]

Applying the definitions of \( CA \) to the DT Abstract, we get:

\[ CA = \{(A, B, C), (D, E)\} \]

With the numerical indices:

- \( CA_1 = \{A, B, C\} \)
- \( CA_{11} = A \)
- \( CA_{12} = B \)
- \( CA_{13} = C \)
- \( CA_2 = \{D, E\} \)
- \( CA_{21} = D \)
- \( CA_{22} = E \)

Every condition has a domain. The domain is the set of all possible values that can be attained by that condition. When, for instance, a condition stands for temperature the domain could be: \((0, \ldots, 120)\). The definition is as follows:

**Definition 5.4: Domain of a condition**

\[ CD = \{CD_i \mid i \in [1, \ldots, C_{\text{num}}]\} \text{ and } CD_i \text{ is the domain of condition } i \]

The condition alternatives of \( CA_i \) are subsets of \( CD_i \). If, for instance, \( CS_i \) stands for temperature and \( CD_i = (0, \ldots, 120) \) \( CA_{i1} \) could be \((0, \ldots, 40)\), \( CA_{i2} \) could be \([40, \ldots, 80]\) and \( CA_{i3} \) could be \([80, \ldots, 120]\). We can view the condition alternatives of \( CA_i \) as partitions, classifications or categorisations of \( CD_i \). The precise relation between the domain of a condition and its alternatives will be explained later in this chapter.

Another important definition concerning the conditions of a DT is that of a condition space denoted as \( \text{SPACE}(C) \):
Definition 5.5: Condition space

\[ \text{SPACE}(C) = CA_1 \times CA_2 \times \ldots \times CA_n \]

For the DT, Abstract \( \text{SPACE}(C) \) is:

\[ \text{SPACE}(C) = \{ \{A, D\}, \{A, E\}, \{B, D\}, \{B, E\}, \{C, D\}, \{C, E\} \} \]

An element of \( \text{SPACE}(C) \) is referred to as a Table Condition Entry (TCE). The condition domain of a table Abstract, denoted as \( \text{DOM(} \text{Abstract} \text{)} \), is a subset of \( \text{SPACE}(C) \) such that each element of the subset appears as a TCE in Abstract. We can succinctly specify completeness for Abstract as \( \text{DOM(} \text{Abstract} \text{)} = \text{SPACE}(C) \) and incompleteness as \( \text{DOM(T)} \subset \text{SPACE}(C) \). If any rule is deleted from a complete table, it becomes incomplete. Note that some condition entries can be filled by a 'don't care'. The 'don't care' is not a condition alternative in the normal sense. It defines a composite rule obtained by merging all the rules that are formed by filling in the possible condition alternatives at the place of the 'don't care'.

To account for exhaustiveness and exclusiveness, the following two constraints must be fulfilled for each \( CA_i \) of a DT:

Definition 5.6: Exhaustiveness

(1) \( \bigcup CA_i = CD_i \)

Definition 5.7: Exclusiveness

(2) \( \bigcap CA_i = \emptyset \)

The two constraints respectively employ the generalised union and the generalised cross-section operating upon sets of sets (see Chapter 3). The first constraint states that the union of \( CA_i \) should be equivalent to \( CD_i \) (exhaustiveness of the condition alternatives). Applied to Abstract this constraint implies that the union of \( A, B \) and \( C \) should be identical to \( CD_1 \) and that the union of \( D, E \) should be identical to \( CD_2 \). The second constraint states that the (sub-)sets of \( CA_i \) must be disjunct: the cross-sections of the (sub-)sets must be empty (exclusiveness of the condition alternatives). Applied to Abstract this means that every cross-section of \( A, B \) and \( C \) should be empty. The same applies to the cross-section of \( D \) and \( E \).
In the following we define the action set. To an important degree this happens analogously to the definition of the condition set with the difference that actions need not to be exhaustive or exclusive. $A_{\text{num}}$ is the number of actions.

**Actions (A)**

The formal definition of action set $A$ is:

**Definition 5.8: Actions**

$A = \{A_i \mid i \in [0, \ldots, A_{\text{num}}] \}$ and each $A_i$ is an ordered pair that consists of an Action Subject $AS_i$ and a set of action alternatives $AA_i$.

The following applies the definition to the DT Abstract:

$A = \{(\text{Action Subject } i; \{X, -\})\}$

Every action consists of an action subject $AS_i$ and a set of action alternatives $AA_i$. Their definitions are given below.

**Definition 5.9: Action subjects**

$AS = \{AS_i \mid i \in [1, \ldots, A_{\text{num}}] \}$ and each $AS_i$ is an action subject.

**Definition 5.10: Action alternatives**

$AA = \{AA_i \mid i \in [1, \ldots, A_{\text{num}}] \}$ and each $AA_i = \{AA_{ik} \mid k \in [1, \ldots, n_i]\}$ and $n_i$ is the number of action alternatives of action $i$.

Another important definition concerning the actions of a DT is that of an action space denoted as $\text{SPACE}(A)$:

---

Note that in case only one condition or action is present in a DT, $\text{SPACE}(C)$ and $\text{SPACE}(A)$ yield an empty set according to the definition of the Cartesian product (Chapter 3). The intention in these cases, however, is that $\text{SPACE}(C)$ is a set consisting of sets where each set contains one of the condition alternatives. Accordingly $\text{SPACE}(A)$ is a set consisting of sets where each set contains one of the action alternatives. Apart from this, it is doubtful whether a DT containing one condition is very useful.
Definition 5.11: Action space

\[ \text{SPACE}(A) = AA_1 \times AA_2 \times \ldots \times AA_{n_{\text{out}}} \]

Applying the definition of \( \text{SPACE}(A) \) to Abstract, we obtain

\[ \text{SPACE}(A) = \{\{X\}, \{-\}\} \]

An element of \( \text{SPACE}(A) \) is called a *table action entry* (TAE).

**Relations between conditions and actions (R)**

A relation is a set of ordered pairs. A DT is a relation that is specified using the sets \( \text{SPACE}(C) \) and \( \text{SPACE}(A) \). Every element of this relation corresponds to a DR.

**Definition 5.12: DT as a relation**

\[ DT \subseteq \{(TCE; TAE) \mid TCE \in \text{SPACE}(C) \text{ and } TAE \in \text{SPACE}(A)\} \]

This definition, however, has a disadvantage. It does not specify that a TCE can only be related to a single TAE. This means that the occurrence of inconsistencies (several TAE's related to one TCE) in a DT is not excluded. To ensure a consistent DT, the following constraint should be matched:

**Definition 5.13: Consistency of a DT**

\[ \forall (TCE; TAE) \in DT : \forall (TCE'; TAE') \in DT : \text{if } TCE = TCE' \text{ then } TAE = TAE' \]

The constraint makes clear that it is preferable to view a DT as a function rather than as a relation.

**Definition 5.14: A DT as a function**

\[ T \text{ is a DT } \Leftrightarrow T = \{(TCE; TAE) \mid \forall TCE \in \text{SPACE}(C) : \exists TAE \in \text{SPACE}(A) \} \text{ and } T \text{ is a function} \]
The domain of the function $T$ is equivalent to $\text{SPACE}(C)$ and is also equivalent to $\text{DOM}(T)$ when $T$ is a complete table. $\text{Range}(T) = \{\text{TAE}\{(\text{TCE}; \text{TAE}) \in T}\}$. An element of $T$ is an ordered pair that represents a DR. As before, a DR relates a TCE to a TAE.

There are some advantages to viewing DT's as functions. First, the definition incorporates completeness. Each element of $\text{SPACE}(C)$ is related to an element of $\text{SPACE}(A)$. Secondly, since each TCE has a unique TAE, the view implies consistency of a DT: the occurrence of several TAE's for one TCE is excluded.

**Goal 1**

<table>
<thead>
<tr>
<th>CS1</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS2</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>CS3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Goal 1</td>
<td>x</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 5.2: Conceptual Interaction**

**New and Modified Definitions**

Though useful, the previous terminology and definitions which are mainly based on research of the Codasyl Decision Table Task Group do not account for conceptual interaction: the phenomenon that a condition can have several classifications due to interdependence with other conditions (Chapter 4). On the one hand, this flexibility is needed to model conditions that can only be effective in conjunction with other conditions. On the other hand, conceptual interaction is needed to avoid combinations of conditions that never arise in practice or are impossible. In relation to a DT, conceptual interaction may manifest itself in two ways. First, the classification of a condition can be flexible and may vary under certain conditions. For instance, the classification of condition $C_2$ of the DT of Figure 5.2 is $k$ and $l$ if $C_1$ is $a$. However, if $C_1$ is $b$, the classification is $r$ and $s$. An interpretation that illustrates this case is: $C_2 = \text{temperature}$, $k \leq 20$, $l > 20$, $r < 30$, $s \geq 30$. Secondly, conceptual interaction may imply that the domain of a condition varies. This occurs when the domain, the set of potential values of a condition, depends on other conditions. An illustrative interpretation may be: $C_1 = \text{type of fruit}$ and $C_2 = \text{colour of fruit}$, $k = \text{yellow}$, $l = \text{green}$, $r = \text{red}$ and $s = \text{orange}$. This interpretation designates that not all colours apply to all kinds of fruit. Therefore, the domain is conditional upon the condition alternatives of $C_1$.

To deal adequately with conceptual interaction, we introduce new definitions and modify a number of definitions that have already been introduced. First, we view a DT as a number of rules depicted by the following set of functions:
\[ RL = \{
  \{(\text{Condition Subject 1; } A), (\text{Condition Subject 2; } D), (\text{Action Subject 1; } X)\},
  \{(\text{Condition Subject 1; } A), (\text{Condition Subject 2; } E), (\text{Action Subject 1; } -)\},
  \{(\text{Condition Subject 1; } B), (\text{Condition Subject 2; } D), (\text{Action Subject 1; } -)\},
  \{(\text{Condition Subject 1; } B), (\text{Condition Subject 2; } E), (\text{Action Subject 1; } X)\},
  \{(\text{Condition Subject 1; } C), (\text{Condition Subject 2; } D), (\text{Action Subject 1; } X)\},
  \{(\text{Condition Subject 1; } C), (\text{Condition Subject 2; } E), (\text{Action Subject 1; } -)\}
\}

Looking at RL, we might be tempted to define a DT as follows:

**Definition 5.15: A DT as a function over a set of conditions and actions**

\[ RL \text{ is a } DT \text{ over set } A \Leftrightarrow \forall DR \in RL: DR \text{ is a function over } A \text{ where } A \text{ is the set of conditions and actions} \]

The ruleset of a DT *Abstract* can be denoted by \( RL(\text{Abstract}) \). Since the definition disregards exclusiveness, exhaustiveness and conceptual interaction, it is not complete yet and requires extension. The first step to be taken yet is assessing the tree structure that underlies every DT and that accounts for conceptual interaction. A *tree* is a special kind of *graph*. A graph is defined as follows (Lew, 1985, p.141):

**Definition 5.16: A graph**

\[ G \text{ is a graph } \Leftrightarrow G = (S; R) \text{ where } S \text{ is a non-empty set and } R \text{ is a relation such that } R \subseteq S \times S \]

The members of \( S \) are called *nodes* (or *vertices*) and the members of \( R \) are called *branches* (also *edges* or *arcs*). An example of a graph is \( G_1 \):

\[ G_1 = (\{a, b, c, d\}; [(a; a), (a; b), (b; a), (b; c), (d; d)]) \]

Graphs have a convenient pictorial representation. Figure 5.3 shows this by displaying the directed graph \( G_1 \). As the directions of the branches of \( G_1 \) are relevant, \( G_1 \) is a *directed* graph (or *digraph*).
Figure 5.3: The Directed Graph $G_1$

We formally define a set of paths $PS$ of length $k$ in a directed graph $G = (S; R)$ as a finite row of $k$ branches (see Chapter 3 for the definition of a row):

**Definition 5.17: The paths of a graph**

If $G = (S; R)$ then:

$PS$ is a set of paths of length $k$ in $G \iff \{(r(0); r(1)), (r(1); r(2)), \ldots, (r(k-1); r(k))\}$

where $r$ is a row with length $k$ and $\forall x \in r : x \in R$.

We say that $k+1$ nodes $\{r(0), r(1), r(2), \ldots, r(k)\}$ are traversed by the path from node $r(0)$ to node $r(k)$. In $G_1$, $\{(a; b), (b; c)\}$ is a path of length 2 from node $a$ to node $c$; it traverses the nodes $a$, $b$, and $c$. A path from a node to itself is called a cycle.

A tree is a graph with certain properties. It is a rooted digraph containing a distinguished node, the root, that does not have a predecessor. Unlike the root node, every other node of a tree has exactly one predecessor. Thus, a tree does not have cycles. In other words, there is no path in which the same node occurs twice. Following Das (1992, pp.27-28) the formal definition that captures these specific properties of a graph is:

**Definition 5.18: A graph as a tree**

A graph $G$ is a tree $\iff G$ is a rooted directed graph with a structure $(S; R)$:

$\forall x \in S : \forall y \in S$ exactly one path exists

$TR_R$ denotes a tree with root $R$. A node adjacent to $R$ is called a child of $R$. Each child of $R$, say $x$, is the root of a subtree induced by the set of nodes reachable from $x$. $TR_x$ is also a tree with root $x$, hence $x$ may have children too. The children of $x$ and these children's children, and so forth, are called the descendants of $x$. The height of a tree is
the length of the longest path in the tree. In this chapter we only consider height-
balanced trees: the subtrees of a node all have equal lengths. The degree of a node is
the number of its children. A tree is t-ary if the degree of its nodes is at most t. The
depth of a node \( x \) in a tree is the length of a path from root \( R \) to the node: \( d(R, x) \).
When we represent the DT Abstract as a (binary) tree, we obtain the following
notation:

\[
TR_{Abstract} = \\
\{ \\
((\text{Condition Subject } 0; \text{Abstract}); (\text{Condition Subject } 1; A)), \\
((\text{Condition Subject } 0; \text{Abstract}); (\text{Condition Subject } 1; B)), \\
((\text{Condition Subject } 0; \text{Abstract}); (\text{Condition Subject } 1; C)), \\
((\text{Condition Subject } 1; A); (\text{Condition Subject } 2; D)), \\
((\text{Condition Subject } 1; A); (\text{Condition Subject } 2; E)), \\
((\text{Condition Subject } 1; B); (\text{Condition Subject } 2; D)), \\
((\text{Condition Subject } 1; B); (\text{Condition Subject } 2; E)), \\
((\text{Condition Subject } 1; C); (\text{Condition Subject } 2; D)), \\
((\text{Condition Subject } 1; C); (\text{Condition Subject } 2; E)), \\
((\text{Condition Subject } 2; D); (\text{Action Subject } 1; X)), \\
((\text{Condition Subject } 2; E); (\text{Action Subject } 1; -)) \\
\}
\]

\( TR_{\text{Condition Subject } 0; \text{Abstract}} \) is a set of ordered pairs. Each of them represents a branch
of the tree. Each ordered pair itself is composed of two ordered pairs which represent
the nodes of a branch. Thus, a node is an ordered pair. The first co-ordinate is a condi-
tion subject and the second co-ordinate is the associated alternative. As a DT does not
contain a root node, we have added an ordered pair \( (\text{Condition Subject } 0; \text{Abstract}) \)
that represents a root node. \( TR_{\text{Condition Subject } 0; \text{Abstract}} \) indeed depicts the DT Abstract.
if, for every DR of the DT, the tree contains exactly one path that contains the same
knowledge as the DR. This implies that not all paths possible in the tree are allowed.
For instance, an illicit path is:

\[
\{((\text{Condition Subject } 1; A); (\text{Condition Subject } 2; D)), \\
((\text{Condition Subject } 2; D); (\text{Action Subject } 1; -)) \}
\]

To check whether \( TR_{\text{Abstract}} \) reflects the rules of RL, we first have to define the set of
paths of a certain tree without the root node:
Definition 5.19: A tree without a root

\[ PS_{TR_{root-root}} = \{ P' \mid P \text{ is a path of length } k \text{ in } TR_{Root} \text{ where } k \text{ is the number of conditions and actions and } P' = P - \{(x; root); (y; z)\} \]

We had to remove the root node from the paths, because RL does not contain the root node either. Now, we can match RL(Abstract) and \( TR_{(Condition \ Subject \ 0; Abstract)} \).

\[ \forall DR \in RL(Abstract) \text{ exactly one path } P \in PS_{TR_{abstract-Abstract}} \text{ exists such that } DR = \{x, y, ..., e\} \text{ and } P = \{(x; y), (y; z), ..., (a; b), (b; e)\} \]

Note that a DR is a set of traversed nodes of a path. When RL(DT) is the ruleset of a table named DT and \( TR_{DT} \) represents the tree structure underlying the ruleset, we can define exhaustiveness and exclusiveness as follows:

Definition 5.20: Exhaustiveness

\( TR_{(CS_i; DT)} \) is exhaustive \( \Leftrightarrow \forall TR_{(CS_i; CA_k)} : TR_{(CS_i; CA_k)} \) is a tree of height 1 in \( TR_{(CS_i; DT)} \)

and \( i \in [0, ..., C_{num-1}] \) and \( k \in [0, ..., N_i] \) where \( N_i \) is the number of alternatives of \( C_i \) and

\[ \bigcup \{ CA_{i+1,k} \mid (CS_{i+1}; CA_{i+1,k}) \text{ is a child of } (CS_i; CA_k) \} = CD_{i+1,j} \]

where \( j \) denotes the relevant domain of \( CS_{i+1} \)

Definition 5.21: Exclusiveness

If \( A \) is a set the elements of which are sets:

\[ \bigcap A \Leftrightarrow \{ x \mid x \in A_i \text{ and } x \in A_j : A_i \in A \text{ and } A_j \in A \text{ and } A_i \neq A_j \} \]

\( TR_{(CS_i; DT)} \) is exclusive \( \Leftrightarrow \forall TR_{(CS_i; CA_k)} : TR_{(CS_i; CA_k)} \) is a tree of height 1 in \( TR_{(CS_i; DT)} \)

and \( i \in [0, ..., C_{num-1}] \) and \( k \in [0, ..., N_i] \) where \( N_i \) is the number of alternatives of \( C_i \) and

\[ \bigcap \{ CA_{i+1,k} \mid (CS_{i+1}; CA_{i+1,k}) \text{ is a child of } (CS_i; CA_k) \} = \emptyset \]

As Figure 5.4 displays, these tree-based definitions describe exhaustiveness and exclusiveness by focusing on the subtrees of height 1 discernible in a DT. Because
actions do not have to be checked on exhaustiveness and exclusiveness, subtrees that have nodes with a depth \( > C_{num} - 1 \) as their roots are excluded. To ensure exhaustiveness, the union of the second co-ordinates of the children of each of the trees should be equal to the relevant domain of the relevant condition. Note that conditions can have several domains due to conceptual interaction. The definition of exhaustiveness denotes a domain of one condition through \( j \). Since both definitions operate upon the subtrees of a DT, they also account for flexible classifications of conditions. To ensure exclusiveness, the second co-ordinates of the children (the condition alternatives) of each of the trees should be disjoint.

\[ \text{exhaustiveness: } CA_j = \bigcup \text{children of } CA_j \]

\[ \text{exclusiveness: } CA_i \cap CA_j = \emptyset, \quad \text{for } i \neq j \]

\[ \text{don't care: } CA_{ik} = \text{parent of } CA_{ik} \]

\[ \text{AND: } \bigwedge \]

\[ \text{OR: } \bigvee \]

\[ \text{negation: } \neg \]

\[ \text{ELSE: } \text{remaining alternatives} \]

\[ \text{trees to be checked on exhaustiveness and exclusiveness} \]

\[ \text{Figure 5.4: Trees of a Decision Table with Height 1} \]

Checking exhaustiveness and exclusiveness, we have to deal with 'don't care' values, AND- and OR-statements, negations and ELSE-statements appearing as condition alternatives. A 'don't care' value in \( CS_i \) counts for \( n_i \) condition alternatives \( CA_{ik} \) under \( CA_{i-1,j} \) where \( CA_{i-1,j} \) is the parent of the condition alternatives. For this reason, a 'don't care' must always appear as the only condition alternative that, on its own, takes care of exhaustiveness and exclusiveness! The arguments of an AND- and an OR-statement must be treated as separate condition alternatives. A negation and an ELSE-statement are equivalent to all remaining, not yet represented, alternatives permitted in
the actual subtree. There are, however, subtle differences. A negation implies that a condition subject has exactly two condition alternatives, the negation itself and its antonym. Conversely, an ELSE-statement indicates that a condition subject has at least two alternatives.

Exhaustiveness and exclusiveness are essential characteristics of a DT. Through these constraints many problems are prevented. Think, for instance, of redundant and overlapping DR's and inconsistencies (Jones, 1991, pp.177-180; Montalbano, 1974, pp.80-91). These problems can completely devaluate the potentials of DT's.

Applying the tree-based definitions to the DT of Figure 5.2, we will see that the union of condition alternatives \( k \) and \( l \) of \( CS_2 \) should be equal to \( CD_2 \) and that the cross-section of these alternatives should be empty. In the same way, the union of condition alternatives \( r \) and \( s \) of \( CS_2 \) should be equal to \( CD_2 \) and these alternatives should be disjoint. The tree-definitions do not demand that condition alternatives of the same condition subjects, but belonging to different trees, should be involved in the verification process for exhaustiveness and exclusiveness. They exclude the possibility of combining a condition alternative of one (sub-)tree with a condition alternative that belongs to another subtree. Condition alternative \( a \) of \( CS_1 \), for instance, cannot be combined with \( r \) or \( s \) of \( CS_2 \). What is conceptually impossible is now accountd for.

Due to conceptual interaction, we must adapt the definitions of \( C, CA \) and \( CD \).

**Definition 5.22: Adapted definition of conditions**

\[
C = \{ C_i | 1 \leq i \leq n \text{ and where } n \text{ is the number of conditions and each } C_i \text{ is an ordered pair that consists of a condition subject } CS_i \text{ and a set of sets of condition alternatives } CA_i \}
\]

Applying the definition of \( C \) to the DT of Figure 5.2, we may get this result:

\[
C = \{ (CS_1; \{a, b\}), (CS_2; \{k, l\}, \{r, s\}), (CS_3; \{_, \{y, not \, y\}\}) \}
\]

Using the numerical indices, we obtain:

\[
C_1 = (CS_1; \{a, b\})
C_2 = (CS_2; \{k, l\}, \{r, s\})
C_3 = (CS_3; \{_, \{y, not \, y\}\})
\]

The second co-ordinate of every ordered pair is now a set the elements of which are sets of condition alternatives:
Definition 5.23: Adapted definition of condition alternatives

\[ CA = \{CA_i | i \in [1, \ldots, C_{num}] \} \text{ and where each } CA_i = \{CA_{ij} | j \in [1, \ldots, S_{num}] \} \]

- \( S_{num} \) is the number of a set of condition alternatives of condition \( i \)
- where each \( CA_{ij} = \{CA_{ijk} | k \in [1, \ldots, n_i] \} \)
- \( n_i \) is the number of condition alternatives of condition \( i \) of set \( j \)

Applying the definition of \( CA \) to the DT of Figure 5.2, we obtain:

\[ CA = \{\{(a, b)\}, \{(k, l)\}, \{r, s\}, \{\_\}, \{y, not \ y\}\} \]

Using the numerical indices, we obtain:

\[ CA_2 = \{(k, l), \{r, s\}\} \]
\[ CA_{22} = \{r, s\} \]
\[ CA_{221} = r \]

Definition 5.24: Adapted definition of the domain of a condition

\[ CD = \{CD_i | i \in [1, \ldots, C_{num}] \} \text{ and each } CD_i = \{CD_{ij} | j \in [1, \ldots, D_{num}] \} \]

- where \( D_{num} \) is the number of the domain of condition \( i \)

Applying the definition of \( CD \) to the DT of Figure 5.2, we obtain:

\[ CD = \{\{(a, b)\}, \{(b, \ldots, t)\}, \{(p, q, r)\}, \{x, y, z\}\} \]

Using the numerical indices, we obtain:

\[ CD_5 = \{(p, q, r), \{x, y, z\}\} \]
\[ CD_{31} = \{p, q, r\} \]

We should also adapt the definitions of \( SPACE(C) \). If we calculate the Cartesian product accounting for every alternative of \( CS_2 \), \( SPACE(C) \) will become very large. The calculation of \( SPACE(C) \) also presents problems in this case, because impossible combinations would be generated: not all colours apply to all types of fruit. Note that conceptual interaction can extend to more than two conditions. So conceptual...
interaction leads to inefficient configurations of $\text{SPACE}(C)$ when the terminology and definitions of the Codosyl Decision Table Task Group are maintained. This causes differences between $\text{SPACE}(C)$ and $\text{DOM}(T)$ and thus leads to incomplete DT's. To solve the completeness problem Vanthienen (1988, p.918) suggests incorporating "impossible" TCE's in $\text{SPACE}(C)$ and denoting them in a special way. We think, however, that this solution pays too little attention to conceptual interaction and can lead to unnecessarily large DT's. So the definition of $\text{SPACE}(C)$ has to be modified too, in the sense that the Cartesian product operates only upon condition alternatives whose combination is permitted. These combinations can be detected in the tree structure of a DT. Then, for the DT of Figure 5.2, we obtain:

$$\text{SPACE}(C) = \{\{a, k, p\}, \{a, k, q\}, \{a, k, r\}, \{a, l, p\}, \{a, l, q\}, \{a, l, r\},$$
$$\{b, r, y\}, \{b, r, \text{not } y\}, \{b, s, y\}, \{b, s, \text{not } y\}\}$$

**Decision Table Systems**

For most applications a single DT is not sufficient. In the majority of cases we need a DT-system: a set of at least two DT's in which each DT is linked to another table belonging to the same system. We can distinguish two types of links. The first type of link is established by the phenomenon that a condition subject with at least one of its alternatives of one DT occurs as an action subject with corresponding action alternatives in another DT. The first table is then called a *head table*, the second table is called a *condition subtable*. The second type of link is established by the phenomenon that an action subject and one of its alternatives of one DT occurs in the same form in another table. The first table again is called a *head table*, the second table is now called an *action subtable*. Usually, the action subject in question, in this action subtable, is further specified by means of other action subjects. Figure 5.5 displays an example of a DT-system containing a head table, a condition subtable and an action subtable.

<table>
<thead>
<tr>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>A1</td>
</tr>
<tr>
<td>R1</td>
</tr>
</tbody>
</table>

(A) A Head Table
Before formally defining a DT-system we first describe, as an example, the head table Abstract and the condition subtable Condition Subject 1 as a set of DR's in which each DR is a function:

$$RL(Abstract) = \{DR_1, DR_2, DR_3, DR_4, DR_5, DR_6\}$$ where:

$$DR_1 = \{(\text{Condition Subject 1}; A), (\text{Condition Subject 2}; D), (\text{Action Subject 1}; X)\}$$

$$DR_2 = \{(\text{Condition Subject 1}; A), (\text{Condition Subject 2}; E), (\text{Action Subject 1}; -)\}$$

$$DR_3 = \{(\text{Condition Subject 1}; B), (\text{Condition Subject 2}; D), (\text{Action Subject 1}; -)\}$$

$$DR_4 = \{(\text{Condition Subject 1}; B), (\text{Condition Subject 2}; E), (\text{Action Subject 1}; X)\}$$

$$DR_5 = \{(\text{Condition Subject 1}; C), (\text{Condition Subject 2}; G), (\text{Action Subject 1}; X)\}$$

$$DR_6 = \{(\text{Condition Subject 1}; C), (\text{Condition Subject 2}; F), (\text{Action Subject 1}; -)\}$$

$$RL(\text{Condition Subject 1}) = \{DR_1, DR_2, DR_3, DR_4\}$$ where:

$$DR_1 = \{(\text{Condition Subject 1.1}; K), (\text{Condition Subject 1.2}; P), (\text{Condition Subject 1}; A)\}$$

$$DR_2 = \{(\text{Condition Subject 1.1}; K), (\text{Condition Subject 1.2}; Q), (\text{Condition Subject 1}; B)\}$$
To specify the link between the head table and the condition subtable, we also need an identical function \( h_1 \):

\[
h_1 = \{ (\text{Condition Subject 1}; \text{Condition Subject 1}) \}
\]

Now, we can formally describe the link using the definition of a (bilateral) connection (see Chapter 3 for a formal description).

\( h_1 \) connects \( RL(\text{Condition Subject 1}) \) with \( RL(\text{Abstract}) \)

The connection states that the restriction of the sets of \( RL(\text{Condition Subject 1}) \) to \( \text{dom}(h_1) \) is a subset of the restriction of the sets of \( RL(\text{Abstract}) \). Because it is not necessary that every (action) alternative of the condition subtable occurs as a condition alternative in the head table, as Figure 5.5 illustrates, we deliberately did not define the connection as a pure bilateral connection. Whereas the head table should contain all the alternatives of the relevant condition, the subtable does not have to contain all the alternatives. Since exhaustiveness constraints only apply to conditions, the exhaustiveness of the condition subtable is not endangered by the omission of one or more alternatives.

We can describe the link between a head table and an action subtable analogously. The notation of the action subtable, then, is as follows:

\[ RL(\text{Action Subject 1}) = \{ DR_1, DR_2, DR_3 \} \]

\( DR_1 = \{ (\text{Condition Subject 3}; S), (\text{Condition Subject 2}; V), (\text{Action Subject 1.1}; H), (\text{Action Subject 1.2}; M), (\text{Action Subject 1}; X) \} \)

\( DR_2 = \{ (\text{Condition Subject 3}; S), (\text{Condition Subject 2}; W), (\text{Action Subject 1.1}; I), (\text{Action Subject 1.2}; N), (\text{Action Subject 1}; X) \} \)

\( DR_3 = \{ (\text{Condition Subject 3}; T), (\text{Condition Subject 2}; -), (\text{Action Subject 1.1}; J), (\text{Action Subject 1.2}; O), (\text{Action Subject 1}; X) \} \)

To describe the relation between the head table and the action subtable, we again need an identical function. We call this function \( h_2 \) and define it as follows:

\[
h_2 = \{ (\text{Action Subject 1}; \text{Action Subject 1}) \}
\]

\( \text{Action subject 1} \) and its alternative \( X \) occur in the action subtable. They should also be present in the head table with the usual difference that the head table contains more alternatives related to the same action-subject. To ensure this, we make use of a connection:
The connection states that the restriction of the sets of $RL(\text{Action Subject 1})$ to $\text{dom}(h_2)$ is a subset of the restriction of the sets of $RL(\text{Abstract})$. Here, we did not use the bilateral connection either. It is true that the action alternative of the action subtable must occur in the head table, but the head table usually contains more action alternatives. Instead the action subtable usually contains other action subjects (1.1 and 1.2 in our example) that further specify the relevant action subjects.

Not everyone would incorporate the action subject and an associated alternative yet to be specified, in the action subtable. Many designers of an action subtable would leave out action subject 1 of our example and only incorporate action subject 1.1 and action subject 1.2. We think, however, that such an approach is not recommendable. When the action subject and the associated alternative are excluded from the action subtable, it is left to the analyst studying the DT structure to see the link between the head table and action subtable. Seeing such a link requires extra knowledge. In this way not only the surveyability of the DT structure at hand is potentially reduced, but we also risk the action subtable becoming less understandable. In addition, omitting an action subject and the associated alternatives deprives us of the possibility to formally describe the link between the head table and the subtable in a neat way. Note that in DBS links between relations are assessed by primary and foreign keys following exactly the same principle as here. As we are working predominantly at the knowledge level at which no distinction exists between AI- and DB-systems, this observation strengthens the idea that omitting an action subject and an alternative to be specified, is not advisable.

When we apply the terminology of Chapter 3 to DT-systems we can easily see that the domain of the knowledge schema is a set of goals. The conditions and actions needed to model these goals form the range of the knowledge schema. The following set-valued function describes the knowledge schema of our example system:

\[
\text{Knowledge Schema}(\text{Abstract}) = \\
\{(\text{Abstract}) \quad : \quad \{(\text{Condition Subject 1}, \text{Condition Subject 2}, \text{Action Subject 1})\}, \\
(\text{Condition Subject 1}) \quad : \quad \{(\text{Condition Subject 1.1}, \text{Condition Subject 1.2}, \text{Condition Subject 1})\}, \\
(\text{Action Subject 1}) \quad : \quad \{(\text{Condition Subject 3}, \text{Condition Subject 2}, \text{Action Subject 1.1}, \text{Action Subject 1.2}, \text{Action Subject 1})\}\}
\]

The DT-system is a function over the domain of the Knowledge Schema(Abstract). The range of the function is a set of DT's or knowledge tables. Calling this function $KS$, we obtain:
KS(Abstract) = RL(Abstract)
KS(Condition Subject 1) = RL(Condition Subject 1)
KS(Action Subject 1) = RL(Action Subject 1)

KS -and thus our DT-system- is a knowledge state. The DT's or knowledge tables are linked by means of connections using the attribute-transformations $h_1$ and $h_2$.

Figure 5.6: Two Connections in a DT-system

Regarding the integration of AI and DBT, it is possible to derive from the previous formal definitions, that DT's and DT-systems have much in common with DBS. The definitions reveal that a DT can be viewed as a relation (or table) of a DBS. A relation is defined by De Broek (1989, p.2) as follows:

**Definition 5.25: Table**

If $A$ is a set then:

$T$ is a table over $A$ $\iff$ $T$ is a set and

$\forall t \in T : t$ is a function over $A$

Set $A$ contains the fields of a relation. In DT-terminology set $A$ corresponds with the stub of a DT. A tuple $t$ of a relation corresponds with a DR of a DT. The conditions form the independent fields. Together, they compose the primary key of the relation. The actions of a DT are the dependent fields of a relation. From this, we can conclude that DT's comply with the definition of a relation. Salah (1986) and Vanthienen (1988) also point to the possibility of viewing a DT as a DB-relation.

Besides exhaustiveness and exclusiveness, the most essential characteristics of a DT, imply that a DT is a relation that meets several types of DB-constraints. To a large extent, they take care of the fulfilment of the attribute constraints (constraints that should be met by the values of the particular attributes of a relation), the tuple constraints (constraints that should be met by combinations of attribute values of each
tuple of a relation) and the table constraints (constraints that should be met by combinations of tuples of one relation). For instance, exhaustiveness and exclusiveness guarantee that the conditions of a DT meet the table constraint of unique identification (see Chapter 3). Every rule of a DT is unique, otherwise a table is not a DT! Another example of a table constraint that is automatically met is, that a relation is not allowed to be cyclic. Because a DT is a tree, cycles are excluded.

The links between the relations of a DB-system can be assessed by means of connections in a similar way as is done to assess the links between DT's in a DT-system. The different types of connections (see Chapter 3) make it possible to assess different types of links between relations. Remember that connections are formalised by subset requirements: the foreign key of a relation of a DB-system, for instance, is an example of a subset requirement. Note that a foreign key is equivalent to a condition occurring as an action in a condition subtable.

Usually, DT's are associated with expert systems. The possibility of viewing a DT and a DT-system as a relation and as a system of relations respectively accentuate that DT's are representation techniques approaching the knowledge level. As DT's are abstractions of mathematical logic, this is not surprising.

### 5.2.2 Functional Object-types and Decision Tables

In this section we analyse the possibilities of representing, reconstructing, validating and simulating functional object-types using DT's. For this purpose, we should bear in mind that a discussion about functional object-types can be reduced to a discussion about functional equivalence.

**Representation**

We return to Figure 5.2 displaying a DT-format of a tree introduced in Chapter 4. We used this tree to clarify the three mechanisms underlying functional equivalence: the conditional relevance of conditions, conceptual interaction and the variation of values of conditions falling within a goal-constructed category. DR 4 and DR 5 illustrate the mechanism of new conditions becoming important under certain circumstances. If CS 1 attains a value that belongs to b and if CS 2 takes a value that belongs to S, it is important to know the value of CS 3. However, if CS 1 attains a value belonging to a, there is no reason to consider the value of CS 3. Conceptual interaction, the second mechanism of functional equivalence, also occurs in the DT of Figure 5.2. It shows the mutual influence of CS 1 and CS 2 on each other’s categorisations. The classification of condition CS 2 is k and l if CS 1 is a. However, if CS 1 is b, the classification is r and s. Also the third mechanism occurs. Different values, say s 1 and s 2 for condition subject CS 2, do not interfere with the realisation of goal 1 if CS 1 is b, because both values are limited to the goal-constructed category s.

From the previous, it is clear that the DR’s of a DT form a disjunction of conjunct sets able to represent functional equivalence. However, DT's also have disadvantages in the representational realm. For instance, Vanthienen points to the limitations of DT's by stating that the representation of iterations in DT's do not lead to a better
surveyability than conventional structuring techniques (Vanthienen, 1988, p.916). This does not imply that it is not possible to represent iterations, recursions or other repeat structures in a DT. De Gelder & Lucardie (1993) showed the possibility of modelling a functional object-type Steel structure connection in a single DT using a (rather complex) repeat structure. However, they also experienced the difficulties of incorporating repeat-structures into DT's.

**Reconstruction**

Besides observing that a DT is capable of representing functional equivalence, it is important to answer the question whether the systematic design of a DT accounts for the reconstruction of functional equivalence. The establishment of the stub of a DT is an essential step in the design of a DT. The stub is structured by first assessing a goal (an action subject) and then finding a relevant condition subject that describes part of the circumstances under which the goal will be realised. This condition subject must be classified in such a way that its alternatives are exclusive and exhaustive. Then a second condition subject is incorporated and classified into alternatives. When conditions are interdependent, this normally leads to a re-categorisation of the first condition subject. This re-categorisation also influences the categorisation of the second condition subject. When a third condition subject is introduced, a re-categorisation of both former condition subjects may prove to be necessary. So it appears that in the design of a DT the modelling of conceptual interaction is explicitly accounted for. Since a condition subject of a DT need not be effective under all circumstances, conditional relevance is also guaranteed. This indicates that somewhere in the DT 'don't cares' are necessary. When too many condition subjects with 'don't care' values appear in a DT or when the DT becomes too large, one should consider redesigning the DT and developing new (sub-)tables. As the condition categories of a DT are goal-constructed, the final mechanism of functional equivalence is also accounted for in the design of the DT. We can therefore conclude that the reconstruction of DT's is goal-oriented and takes care of functional equivalence.

**Validation**

DT's offer extensive opportunities to validate a functional object-type on completeness, consistency and correctness. We previously defined completeness as $\text{DOM}(T) = \text{SPACE}(C)$ and incompleteness as $\text{DOM}(T) \subset \text{SPACE}(C)$. However, our definition assessing the exhaustiveness of a tree automatically accounts for completeness. This is a consequence of the fact that the union of all condition alternatives of every subtree with height 1 starting from a (root-)node that is not an alternative of the last condition subject, should be equivalent to the relevant domain of the condition subject. The DT-format permits easy inspection on completeness: we only have to verify whether all the alternatives are present. We can also have it checked by a computer program provided that the domains of the condition subjects are defined adequately (see Chapter 6).

In a DT two types of inconsistencies can occur. The first type refers to the situation that identical TCE's have different TAE's. This kind of inconsistency, also called
intra-column inconsistency, can easily be detected by a computer program viewing a DT as a function. It is taken care of by our exclusiveness definition which states that every branch is unique. The second type, inter-column inconsistency, pertains to the situation that one or more TCE's are impossible. This type is harder to detect by a program and usually requires inspection of the DT by a human expert. As to this, it is worth noting that the DT-format considerably facilitates the visual inspection of inter-column inconsistency. The easy-reference presentation of complex logic in a DT helps to correct the knowledge and supports the removal of inconsistencies (see for this topic also Montalbano, 1974, pp.85-87).

Finally, we must be able to verify correctness. A DT which is complete and which does not contain any inconsistencies is not by definition correct. Checking correctness using a DT simply comes down to investigating whether every individual DR is correct (Verhelst, 1980, p.18).

Simulation

One can perform a simulation of functional object-types represented in a DT by starting at a goal (action subject) in a DT and tracing the condition subjects and their alternatives in order of appearance. What one actually does is traversing a path of the tree underlying a DT. One can also simulate easily what happens under varying circumstances by traversing several paths of a tree. In this way what-if analyses can be conducted. In case of a DT-system, one needs to consult various condition and action subtables. In these tables the same procedure can be performed repeatedly so that a DT-system allows simulations by hand. The situation changes somewhat when one has to deal with non-trivial DT-systems that contain many layers of (complex) tables. Soon, one will come across a great number of practical difficulties when attempting to navigate through the tables of the system. Then, a clear insight in the structure of the DT-system is preconditional to being able to assess efficiently the knowledge contained in the DT-system.

5.2.3 Conclusion

To analyse the suitability of DT's as a conceptual modelling language for functional object-types, we formally defined DT's as tree structures. Though the definitions presented here deviate from those provided in the Codasyl Report, it is not really unusual to view DT's as tree structures. Tree structures are of fundamental importance in any approach to formalising intelligent processes and have a multitude of applications (Jones, 1991; Lew, 1985). Our tree-based definitions considerably improve the formal assessment of exhaustiveness and exclusiveness which are the two most basic characteristics of a DT.

Our analysis shows that to an important degree DT's are appropriate as a conceptual modelling language for functional object-types. It appears that DT's are capable of representing functional object-types and that there is a close relationship between the design of DT's and the reconstruction of functional object-types. Because DT's vividly show the combinations of conditions which have been anticipated (Metzner, 1977,
p.15), they are also suitable for validation on completeness, consistency and correctness. Finally, the goal-oriented, close and precise grouping of knowledge avoids error-prone (What-if)-simulations. However, if many levels of DT's are involved in the simulation process and one needs to traverse paths through these levels, simulation by hand soon becomes intractable.

Besides these potentials which are specific to functional object-types, we have ascertained that exhaustiveness and exclusiveness enable us to view a DT and a DT-system as a relation and as a system of relations of a DB-system, meeting respectively several types of DB-constraints. This is caused by the fact that a DT is a tree structure and tree structures comply by definition with, for instance, tuple constraints (see Sub-section 5.2.1). Thus, a DT may be viewed as a relation which complies additional requirements. As DT's are normally associated with expert systems, this view shows the lack of fundamental differences between AI- and DBT-systems. This strengthens the argument that a DT is a symbol level technique that resides close to the knowledge level. At the knowledge level no distinction exists between AI- and DBT-systems.

Research shows that DT's in many applications outperform other modelling techniques. Lafleur (1971) compared DT's with flowcharts and concluded that DT's have distinct advantages over them. Jones (1991) found benefits of DT's relative to truth tables and Veitch-Karnaugh maps (K-maps). Van Der Smagt & Lucardie (1991) discussed a number of advantages of DT's over decision plan nets. Palvia & Gordon (1992) observed the advantages of DT's relative to trees and formulas when applied to the analysis of decision problems. The majority of advantages refer to the clear representation of knowledge and (the speed of) checking knowledge for completeness, consistency and correctness. In addition, the fact that we can interpret conditions of DT's as dimensions, factors, causes, causal factors, variables, parameters and so on, and action subjects as goal dimensions from a diagnostic, an interpretation or a monitoring field underlines the broad applicability of DT's. Therefore, it is not surprising that they are used in almost every area of society.

However, DT's do still have drawbacks. For instance, modelling knowledge into DT's remains difficult. Modelling knowledge requires a theory of knowledge, intensive training and experience. In this thesis the theory of functional object-types is proposed as a general approach to mitigating this problem to some extent. Other drawbacks of DT's are:

1. DT's are not fully appropriate for functional object-types that require a frequent use of recursions, iterations or other repeat structures. Also when object-types are extremely simple, DT's are not very effective.
2. Simulation by hand, when dealing with functional object-types, is possible, but not a task which can be undertaken easily. DT's do not generate prototypes that can facilitate simulation processes (Davis, 1988, p.1113).
3. DT's do not provide automated checking facilities (Davis, 1988, p.1113).
4. Drawing DT's is a very time-consuming and difficult process. As long as no advanced graphical tools are available to support the complex drawing of DT's, the acceptation of DT's will be retarded.
The third section searches a solution for the first drawback. The remaining problems are dealt with in the next chapter.

5.3 PROLOG

Under the influence of Kowalski's ideas on logic and theorem proving, Prolog was designed around 1970 by A. Colmerauer of the University of Marseilles. Initially the use of Prolog was limited, but during the last ten years the number of Prolog-applications has been growing rapidly due to the construction of efficient compilers and improved programming environments. Another explanatory factor of Prolog's growing popularity, is that it is a programming language of a conceptual nature. This is not a coincidence since Prolog has been developed from the idea that a good programming language is a powerful conceptual tool for organising, expressing, communicating and executing knowledge.

Why is it that we can regard Prolog as a conceptual language? One of the reasons is revealed by the name. Prolog is an abbreviation of PROgramming in LOGic. Prolog allows users to program by description. We can read Prolog programs of a good style almost always as logical statements. However, an adequate use of Prolog as a conceptual modelling language is not a straightforward matter. In this respect authors such as Das (1992), Kowalski (1979;1984;1985) and Lloyd (1984) emphasise the importance of a proper insight in the formal background of Prolog in Logic Programming.

In this section we have a particular interest in Prolog as a conceptual language for functional object-types. A related point of interest is how Prolog performs in comparison with DT's. To find answers to these questions, we first discuss Prolog's formal background in Logic Programming (Section 5.3.1). Then, Prolog's potentials as a conceptual language for functional object-types are reviewed (Section 5.3.2). Finally, we formulate a number of conclusions regarding Prolog as a conceptual language and Prolog's relation to DT's and round off with a short discussion (Section 5.3.3).

5.3.1 Formal Background

Logic Programming is based on the idea that instead of learning human thinking in terms of operations of the computer, the computer should learn to execute instructions that are easy for us to provide. Logic Programming is highly declarative and in this respect it has strong relations with its sister approach functional programming. Logic Programming began in the early 70s as an outgrowth of automatic theorem proving and artificial intelligence. The development of Logic Programming is governed by the fundamental idea that logic is not only a specification language, but can also be used as a programming language. In the early 70s this idea was quite revolutionary, because until 1972, logic was only used as a specification (or declarative) language in computer science. Chapter 3 demonstrated this declarative use by specifying a
complete knowledge universe through mathematical logic. Logic as a specification \textit{and} a programming language is expressed by the equation of Kowalski (1979):

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure5_7.png}
\caption{The Logic Programming Paradigm}
\end{figure}

Figure 5.7 shows that a Logic Programming system has two components: a logic component and a control component. The distinction of a logic part and a control part disentangles what the computer knows from how the computer uses it. Using mathematical logic, the user develops the logic component that contains the knowledge. In functional programming the logic component is specified by function declarations (Hudak, 1989; Kowalski, 1985; Turner, 1985); in Prolog it is defined by logic expressions.

The control component determines the way this knowledge can be used to arrive at a solution. The two components of a logic program correspond with two types of semantics a logic program can have: \textit{declarative semantics} and \textit{procedural semantics}. As both types of semantics help to evaluate Prolog's potentials as a conceptual modelling language, we discuss each of them thoroughly, before considering Prolog.

\textbf{The Declarative semantics of Logic Programs: First Order Predicate Logic (FOPL)}

Mathematical logic, particularly first order predicate logic (FOPL), is essential in a logic programming system. FOPL is a part of mathematical logic. Another part of mathematical logic is propositional logic. In propositional logic, knowledge is specified by indivisible propositions which are true or false. The expressive power of propositional logic is therefore limited. FOPL overcomes these limitations by logic notations such as terms, predicates and quantifiers.

Declarative semantics of FOPL refers to a syntactic aspect and a semantic aspect. The syntactic aspect is concerned with well-formed formulas admitted by the grammar as well as deeper proof-theoretic issues. The semantic aspect is concerned with the meanings attached to symbols in well-formed formulas.

To clarify the syntactic aspect, we start with the definition of an alphabet. An alphabet consists of seven classes of symbols:

\textit{Definition 5.30: An alphabet}

(a) variables
(b) constants
Classes (a) to (d) may vary from alphabet to alphabet, while classes (e) to (g) are the same for every alphabet. The classes (b) and (c) may be empty for an alphabet. Variables are denoted by the letters $x, y, z, x_1, y_1, z_1$. Constants are denoted by $a, b, c, a_1, b_1, c_1$. Functions of various arities $>0$ are denoted by the letters $f^n, g^n, h^n, f_1^n, g_1^n, h_1^n$. Predicates of various arities $\geq 0$ are denoted by the letters $P^n, Q^n, R^n, P_1^n, Q_1^n, R_1^n$. The connectives are:

- $\neg$ (negation)
- $\wedge$ (conjunction)
- $\vee$ (disjunction)
- $\rightarrow$ (implication)
- $\leftrightarrow$ (bi-implication)

The quantifiers are:

- $\forall$ (Universal Quantifier: $\forall x$ means for all $x$)
- $\exists$ (Existential Quantifier: $\exists x$ means there exists an $x$)

The scope or range in $\forall x(F)$ and in $\exists x(F)$ is $F$. A variable is bounded if it occurs within the scope of a quantifier, otherwise it is a free variable. In the formula $\forall xP(x, y) \land Q(x)$ the first two occurrences of $x$ are bound, while the third occurrence is free. A formula is closed if it contains no free variables.

The punctuation symbols consist of commas and parentheses. To avoid cluttering of formulas with brackets the following precedence hierarchy will be used:

- $\neg, \forall, \exists$
- $\vee$
- $\wedge$
- $\rightarrow \leftrightarrow$

Now, we can define terms.
Definition 5.31: Terms

(a) A variable is a term.
(b) A constant is a term.
(c) If \( f \) is an \( n \)-ary function and \( t_1, \ldots, t_n \) are terms then \( f(t_1, \ldots, t_n) \) is a term.

A first order language consists of the set of all well-formed formulas constructed from the symbols of the alphabet. A well-formed formula is defined as follows:

Definition 5.32: Well-formed formula

(a) If \( P \) is an \( n \)-ary predicate and \( t_1, \ldots, t_n \) are terms then \( P(t_1, \ldots, t_n) \) is an atomic formula (or atom). An atomic formula or its negation, is also called a literal.
(b) If \( F \) is a formula and \( x \) is a variable then \( \forall x(F) \) and \( \exists x(F) \) are formulas.
(c) If \( F \) and \( G \) are formulas then \( \neg F, F \lor G, F \land G, F \rightarrow G, F \leftrightarrow G \) are formulas.

Examples of well-formed formulas are:

\[- P(a) \land \forall x(P(x) \rightarrow \exists y(P(y) \land Q(x, y))) \]
\[- Q(a) \land \forall x(Q(x) \rightarrow Q(R(x))) \]

Definitions 5.30 to 5.32 covered the syntactic aspect. It will be clear that a system of well-formed formulas based upon these definitions is meaningless, since we cannot determine the truth-values of the formulas. Therefore, it is necessary to attach some meaning to each of the symbols of the formulas. This brings us to the semantic aspect of FOPL. The various quantifiers, connectives and punctuation symbols have a fixed meaning, but the meaning attached to variables, constants, functions and predicates can vary. In other words, we have to interpret formulas. This requires a structure.

Definition 5.33: Structure

Structure \( S \) is a tuple of the following form:
\(<D, \{f^n_i : D^n \rightarrow D, \}, \{P^m_i : D^m \rightarrow \{true, false\}\} \). \( n \) and \( m \geq 0 \). \( S \) has the following properties:

1. \( D \), the domain of \( S \), is a non-empty set of elements.
2. \( \{f^n_i : D^n \rightarrow D\} \) is a set of functions defined on \( D \). \( n \geq 0 \).
3. \( \{P^m_i : D^m \rightarrow \{true, false\}\} \) is a non-empty set of predicates from \( D^m \) to the set of truth values. \( m \geq 0 \).
The basic elements of a structure must be assigned to constants, function-symbols and to predicate-symbols. However, when a (meaningless) set of formulas, a structure \( S \) and the assignments mentioned are available, we still cannot assess the truth values of the formulas as long as the occurring variables are not assigned a value. Thus, the \textit{interpretation} of the formulas in a structure \( S \) boils down to:

1. an assignment of variables ranging over \( D \).
2. an assignment of each individual constant to a fixed element of \( D \) and of an \( n \)-ary function with domain \( D^n \) and range \( D \) to each \( n \)-ary function symbol of \( S \).
3. an evaluation whether a formula evaluates to true or false. This includes:
   a. assignment to each \( n \)-ary predicate symbol of an \( n \)-ary relation in \( D \)
   b. evaluation of the connectives in combination with formulas
   c. evaluation of \( \exists \) in combination with formulas
   d. evaluation of \( \forall \) in combination with formulas

The formal definition of an interpretation of a set of formulas in structure \( S \) under a variable assignment \( \nu \) (notation: \( I^\nu \)) is:

\textit{Definition 5.35: Interpretation}

1. \( I^\nu(x_i) = \nu(x_i) \). Several variable assignments are possible within one structure.
2. \( I^\nu(f^n(t_1, \ldots, t_n)) = \overline{f^n}(I^\nu(t_1), \ldots, I^\nu(t_n)) \) in which \( f^n \) is the function that is associated in \( S \) with \( f^n \) (including constants).

Finally, the truth value of a formula in a structure under an interpretation and under a variable assignment is assessed:

1. \( I^\nu(P^m(t_1, \ldots, t_m)) = \overline{P^m}(I^\nu(t_1), \ldots, I^\nu(t_m)) \): this means that an atom \( P^m(t_1, \ldots, t_m) \) is true in structure \( S \) under interpretation \( I^\nu \) and variable assignment \( \nu \) if and only if \( P^m(I^\nu(t_1), \ldots, I^\nu(t_m)) \) is true.
2. If the truth values of \( F \) and \( G \) are determined, the truth values of \( \neg F, F \lor G, F \land G, F \rightarrow G, F \leftrightarrow G \) are determined according to the table below:

<table>
<thead>
<tr>
<th>( F )</th>
<th>( G )</th>
<th>( \neg F )</th>
<th>( F \lor G )</th>
<th>( F \land G )</th>
<th>( F \rightarrow G )</th>
<th>( F \leftrightarrow G )</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

- 143 -
(3) \( \exists x(F) \) is true under \( v \) if there is a \( v' \) which only differs in \( x \) from \( v \), such that \( F \) is true under \( v' \).

(4) \( \forall x(F) \) is true under \( v \) if for every \( v' \) which only differs in \( x \) from \( v \), it is true that \( F \) is true under \( v' \).

To clarify the previous formal aspects of an interpretation, we revert to a simple well-formed formula \( F \):

\[
F = \exists x(P(x) \leftarrow Q(x))
\]

For the interpretation of \( F \) we can make use of the structure \( S \) and the variable assignments \( v_1 \) and \( v_2 \):

\[
S = \{ \{123, 214\}, \{\text{Fireproof}(123) = \text{True}, \text{Fireproof}(214) = \text{False}\}, \{\text{Fire-resistant}(123) = \text{True}, \text{Fire-resistant}(214) = \text{False}\} \}
\]

\( V_1(x) = 123 \)

\( V_2(x) = 214 \)

We can have the following interpretation and evaluation of the predicates \( P \) and \( Q \):

\[
I'_{v_1}(P(x)) = \overline{P}(I'_{v_1}(x)) = \text{Fireproof}(123) = \text{True} \quad \text{and} \quad I'_{v_1}(Q(x)) = \overline{Q}(I'_{v_1}(x)) = \text{Fire-resistant}(123) = \text{True}
\]

Under this interpretation and valuation \( v_1 \), \( F \) is true. Under valuation \( v_2 \), \( F \) is false, because the predicates do not both evaluate to true:

\[
I'_{v_2}(P(x)) = \overline{P}(I'_{v_2}(x)) = \text{Fireproof}(214) = \text{True} \quad \text{and} \quad I'_{v_2}(Q(x)) = \overline{Q}(I'_{v_2}(x)) = \text{Fire-resistant}(214) = \text{False}
\]

The truth value of a closed formula does not depend on the variable assignment. Note that \( G \) is false with respect to \( S \):

\[
G = \forall x(P(x) \leftarrow Q(x))
\]

At this point, we can introduce the concepts of model, satisfiability and logical consequence.
Definition 5.36: Model

(1) A model of a closed formula is an interpretation under which the formula is true.
(2) If $S$ is a set of closed formulas, then an interpretation $I$ is a model for $S$ if $I$ is a model for each formula of $S$.

In our previous example $I_{11}$ is a model of the closed formula $F$. $I_{12}$ is not a model of $F$.

Definition 5.37: Satisfiability of formulas

A set of formulas $S$ is satisfiable if $S$ has a model. $S$ is unsatisfiable if it has no models.

Definition 5.38: Logical consequence

A closed formula $F$ is a logical consequence of a set formulas of $S$, if for every interpretation of $I$ of $S$, $I$ is a model of $S$ implies that $I$ is a model of $F$.

So $F$ is logical consequence of $S$ if and only if $S \cup \{\neg F\}$ is unsatisfiable. Thus, in general, the basic problem is to show that $S \cup \{\neg F\}$ is unsatisfiable and to assess $F$. According to the definition this implies showing that every interpretation of $S \cup \{\neg F\}$ is not a model. This is a considerable problem. However, it so happens that there is a much smaller and more convenient class of interpretations being all that needs to show unsatisfiability: the Herbrand Interpretations.

Definition 5.39: Herbrand universe

Let $L$ be a first order language\(^2\). The Herbrand Universe $U_L$ is the set of all ground terms (ground terms are terms that do not contain variables), which can be formed out of the constants and functions appearing in $L$. In case, there are no constants, we add a constant, $a$.

\(^2\) A first order language consists of the well-formed formulas of a first order theory. A first order theory consists of an alphabet, a first order language, a set of axioms and a set of inference rules.
Definition 5.40: Herbrand basis

Let $L$ be a first order language. The *Herbrand Basis* $B_L$ for $L$ is the set of all ground atoms (ground atoms are atoms that do not contain variables), which can be formed by using predicates from $L$ with ground terms from $U_L$ as arguments.

Consider the formulas:

$P(a) \leftarrow$

$P(x) \leftarrow Q(f(x), G(x))$

$R(y) \leftarrow$

the Herbrand Universe $U_L$ is:

$\{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), \ldots\}$

and the Herbrand Basis $B_L$ is:

$\{P(a), Q(a, a), R(a), P(f(a)), Q(a, f(a)), Q(f(a), a), Q(f(a), f(a)), R(f(a)), \ldots\}$

When we return to our example program:

$\text{Fireproof}(123) \leftarrow \text{Fire - resistant}(123)$

the Herbrand Universe $U_L$ is:

$\{123\}$

and the Herbrand Basis $B_L$ is:

$\{\text{Fireproof}(123), \text{Fire - resistant}(123)\}$

The reader can obtain more information about the Herbrand Universe and the Herbrand Base in Lloyd's *Foundations of Logic Programming* (Lloyd, 1984) and in Sterling's and Shapiro's *The Art of Prolog* (Sterling & Shapiro, 1986).
**Definition 5.41: Herbrand interpretation**

Let \( L \) be a first order language. An interpretation for \( L \) is a Herbrand Interpretation if the following conditions are satisfied:

(a) The domain of the interpretation is \( U_L \).
(b) Constants in \( L \) are assigned to themselves.
(c) If \( f \) is an \( n \)-ary function in \( L \), the \( f \) is assigned to mapping from \( (U_L)^n \) into \( U_L \) defined by \( (t_1, \ldots, t_n) \mapsto f(t_1, \ldots, t_n) \).

**Definition 5.42: Herbrand model**

Let \( L \) be a first order language and \( S \) a set of closed formulas of \( L \). A Herbrand model for \( S \) is a Herbrand Interpretation for \( L \) which is a model for \( S \).

A good deal of the theory of logic programming is concerned only with *clauses* which are a special kind of formulas, and for them, Herbrand Interpretations suffice. If there is a model then there is a Herbrand model.

**The Procedural Semantics of Logic Programs: SLD-derivations**

Until now, we discussed the declarative semantics of logical formulas expressed in FOPL. The declarative semantics helps us to ascertain whether a formula is a logical consequence of a set of formulas. An alternative way to ascertain a logical consequence is to apply *inference rules* or *deduction rules*. A well-known example of an inference rule is the *resolution principle*. This principle is a machine-oriented way of viewing *modus ponens* (from \( F \) and \( F \rightarrow G \) follows \( G \)). The resolution principle operates upon clauses:

**Definition 5.43: Clause**

A *clause* is a formula of the form:

\[
\forall x_1 \ldots \forall x_s (A_1 \lor \ldots \lor A_k \lor \neg B_1 \lor \ldots \lor \neg B_n)
\]

where each \( A_1 , \ldots , A_k \) and \( B_1 , \ldots , B_n \) are atoms and where \( \forall x_1 \ldots \forall x_s \) are all the variables occurring in these atoms.

Because clauses are so common in Logic Programming, it is convenient to adopt a special clausal notation. Therefore, we rewrite the clause
∀x₁…∀xₙ(A₁ ∨…∨Aₖ ∨¬B₁ ∨…∨¬Bₙ)

to the equivalent clause

∀x₁…∀xₙ(A₁ ∨…∨Aₖ ← B₁ ∧…∧Bₙ).

The clausal notation of this clause then is:

A₁,...,Aₖ ← B₁,...,Bₙ

Thus, in clausal notation, all variables are assumed to be universally quantified, the commas in A₁,...,Aₖ denote disjunction and the commas in B₁,...,Bₙ denote conjunction.

Definition 5.44: Program clause

A program clause is a clause of the form: A ← B₁,...,Bₙ. The positive literal A is called the head and B₁,...,Bₙ is called the body.

Definition 5.45: Unit clause

A unit clause is a program clause with an empty body. It has the form: A ← . So a unit clause is an unconditional clause.

Definition 5.46: Logic program

A logic program is a finite set of clauses.

Definition 5.47: Definition of clauses

The set of all program clauses with the same predicate P in the head is called the definition of P.
Definition 5.48: Goal clause

A goal clause is a clause of the form \( \leftarrow \neg B_1, \ldots, \neg B_n \). It is a program clause with an empty consequent. Each element \( B_i \) (\( i = 1, \ldots, n \)) is called a subgoal.

If \( y_1, \ldots, y_n \) are the variables of the goal clause, then

\[
\leftarrow B_1, \ldots, B_n
\]

is shorthand for:

\[
\forall y_1 \ldots \forall y_n \left( \neg B_1 \lor \ldots \lor \neg B_n \right)
\]

or an equivalent of

\[
\neg \exists y_1 \ldots \exists y_n (B_1 \land \ldots \land B_n)
\]

Definition 5.49: Horn clause

A Horn clause is either a program clause or a goal clause.

Another important type of clause is the empty clause:

Definition 5.50: Empty clause

An empty clause denoted by \( \square \) is a clause with an empty consequent and an empty antecedent. Such a clause is to be understood as a contradiction.

Resolution on clauses works as follows. Suppose we have a set of logical formulas \( S \). Attempting to determine whether \( F \) is a logical consequence of \( S \) is equivalent to proving that \( S \cup \{ \neg F \} \) is unsatisfiable. Resolution on \( S \cup \{ \neg F \} \) proceeds as follows: first, it is checked whether \( S \cup \{ \neg F \} \) contains the empty clause \( \square \). If this is the case, \( S \cup \{ \neg F \} \) is inconsistent and we can assess that \( F \) is a logical consequence of \( S \). If \( S \cup \{ \neg F \} \) does not contain the empty clause \( \square \), the resolution principle selects an appropriate pair of clauses and tries to derive a new clause according to the following procedure:

If we have two clauses \( C_1 \) and \( C_2 \) and if a positive literal \( Q \) occurs in \( C_1 \) and the negation of \( Q \) occurs in \( C_2 \), then a new clause \( C_1 \lor C_2 \) follows in which \( Q \) and \( \neg Q \) are eliminated.
The new clause \( C_1 \lor C_2 \), obtained by resolution, is called the \textit{resolvent}. The clauses \( C_1 \) and \( C_2 \) are the \textit{parent clauses} of the resolvent.

The elimination of a literal \( L \) from a clause \( C \) can be noted down as follows: \( C \setminus L \). Every clause that can be derived from a pair of clauses is added to the set of formulas \( S' \cup \{ \neg F \} \) and there is a new check to see whether this new set contains the empty clause \( \Box \). This process continues until the resulting clauses contain the empty clause \( \Box \). To show the resolution principle we can examine the following three clauses. The predicates do not contain variables:

\[
\begin{align*}
C_1 & : P \leftarrow Q \\
C_2 & : Q \leftarrow S \quad (\text{equivalent to } \neg Q \lor S) \\
C_3 & : S
\end{align*}
\]

From the parent clauses \( C_1 \) and \( C_2 \) we obtain the resolvent by eliminating \( Q \) from the antecedent in the first clause and from the consequent in the second clause:

\[
P \leftarrow S
\]

The general form of resolution is:

\begin{verbatim}
procedure Resolution(S)
    clauses ← S;
    while the empty clause ∉ S do
        \{c_i, c_j\} ← SelectResolvable(clauses);
        resolvent ← Resolve(c_i, c_j);
        clauses ← clauses ∪ resolvent
    end
end
\end{verbatim}

(Source: Lucas & Van Der Gaag, 1991, p.58)

\textit{Figure 5.8: The General Form of Resolution}

Since resolution is a \textit{syntactical operation}, it is less simple to derive new clauses in case variables occur in the clauses. The solution of this problem consists of making a literal of one clause identical to the negation of this literal in another clause by some appropriate substitution for their variables. This operation is known as \textit{unification}. Unification is a general method that compares clauses and that determines, if possible, the substitution necessary to make clauses syntactically identical. For a proper
understanding of unification, we need to define substitutions, expressions, compositions and unifiers.

**Definition 5.51: Substitution**

A substitution \( \theta \) is a finite set of the form \( \{v_1 / t_1, \ldots, v_n / t_n\} \) where each \( v_j \) is a variable, each \( t_i \) is a term distinct from \( \theta \) and the variables \( v_1, \ldots, v_n \) are distinct. Each element \( v_j / t_i \) is called a binding for \( \theta \). \( \theta \) is called a ground substitution if the \( t_i \) are all ground terms. \( \theta \) is called a variable-pure substitution if the \( t_i \) are all variables.

**Definition 5.52: Expression**

An expression is either a term, a literal or a conjunction or disjunction of literals. A simple expression is either a term or an atom.

A substitution can be applied to an expression, so that we obtain a new expression \( E\theta \).

**Definition 5.53: Composition**

Let \( \theta = \{u_1 / s_1, \ldots, u_m / s_m\} \) and \( \sigma = \{v_1 / t_1, \ldots, v_n / t_n\} \) be substitutions. Then the composition \( \theta \sigma \) of \( \theta \) and \( \sigma \) is the substitution obtained from the set \( \{u_1 / s_1 \sigma, \ldots, u_m / s_m \sigma, v_1 / t_1, \ldots, v_n / t_n\} \) by deleting any binding \( u_i / s_i \sigma \) of which \( u_i = s_i \sigma \) and deleting any binding \( v_j / t_j \) for which \( v_j \in \{u_1, \ldots, u_m\} \).

**Definition 5.54: Most general unifier**

Let \( S \) be a finite set of simple expressions. A substitution \( \theta \) is called a unifier for \( S \) if \( S\theta \) is a singleton. A unifier \( \theta \) is called a most general unifier (mgu) for \( S \), if for each unifier \( \sigma \) for \( S \), there exists a substitution \( \gamma \) such that \( \sigma \gamma = \theta \).

The unification-algorithm takes a finite set of simple expressions as input and returns an mgu if the set is unifiable. Otherwise, it reports that the set is not unifiable. When the unification-algorithm attempts to unify two simple expressions, it puts two imaginary pointers at the two leftmost symbols of these expressions. If these symbols are identical, both pointers move to the right. But if these symbols are different, the unification-algorithm attempts to unify these sub-expressions by making a substitution. If the attempt is successful, the process is continued with the two sub-
expressions obtained by applying the substitution. If not, the sub-expressions are not unifiable. If the pointers reach the end of the two expressions, the composition of all the substitutions made is a mgu of the two expressions.

An important notion to properly understand unification is the disagreement set of $S$.

**Definition 5.55: Disagreement set**

Let $S$ be a finite set of simple expressions. The **disagreement set** of $S$ is defined as follows. Locate the leftmost symbol position in which not all expressions in $S$ have the same symbol and extract from each expression in $S$ the sub-expression beginning at that symbol position. The set of all sub-expressions is the disagreement set.

A formal definition of the unification-algorithm is given below.

**Definition 5.56: Unification-algorithm**

1. Put $k = 0$ and $\sigma_0 = \epsilon$ (the empty substitution).
2. If $S\sigma_k$ is a singleton, then stop; $\sigma_k$ is mgu of $S$. Otherwise, find the disagreement set $D_k$ of $S\sigma_k$.
3. If there exist $v$ and $t$ in $D_k$ such that $v$ is a variable that does not occur in $t$, then put $\sigma_{k+1} = \sigma_k \{v / t\}$, increment $k$ and go to 2. Otherwise, stop; $S$ is not unifiable.

Logic programming systems use resolution as the only inference rule. Resolution is mostly applied as *refutation*: the derivation of an empty clause from a set of unsatisfiable formulas. Suppose we want to prove that the formula

$$\exists y_1 \ldots \exists y_n (B_1 \land \ldots \land B_n)$$

is a logical consequence of logic program $P$. As resolution theorem provers are refutation systems, the negation of the formula to be proved is added and a contradiction is derived. The **negation** of our example clause is added to $P$ and thus we obtain the goal:

$$\leftarrow B_1, \ldots, B_n$$

The logic programming system derives successive goals. If the empty clause is eventually derived and the system has obtained a contradiction it inferences that

$$\exists y_1 \ldots \exists y_n (B_1 \land \ldots \land B_n)$$
indeed is a logical consequence of the program \( P \). From a \textit{theorem proving point of view} the only interest is to demonstrate logical consequence. From a \textit{programming point of view} we are much more interested in the bindings that have been made, because they provide us with the output of an executed logic program.

There are several forms of resolution. Most widely used is a variant of the linear resolution: \textit{the SLD-resolution}. Prolog is based on a limited form of the SLD-resolution. SLD stands for: Linear Resolution with Selection function for Definite Clauses. The word linear designates the form of the graphical display of an SLD-derivation.

\textit{Definition 5.57: Computation rule}

A \textit{computation rule} is a function from a set of goals to a set of atoms, such that the value of the function for a goal is always an atom, called the \textit{selected} atom, in that goal.

\textit{Definition 5.58: Derivation}

Let \( G_i \) be \( A_1, A_2, \ldots, A_k, C_{i+1} \) be \( A \leftarrow B_1, \ldots, B_q \) and \( R \) be a computation rule. Then \( G_{i+1} \) is derived from \( G_i \) and \( C_{i+1} \) using \textit{mgu} \( \theta_{i+1} \) via \( R \), if the following conditions hold:

(a) \( A_m \) is the selected atom given by the computation rule \( R \).
(b) \( A_m \theta_{i+1} = A \theta_{i+1} \) (that is, \( \theta_{i+1} \) is an \textit{mgu} of \( A_m \) and \( A \)).
(c) \( G_{i+1} \) is the goal \( \leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k) \theta_{i+1} \). \( G_{i+1} \) is the resolvent of \( G_i \) and \( C_{i+1} \).

\textit{Definition 5.59: SLD-derivation}

Let \( P \) be a program, \( G \) a goal and \( R \) a computation rule. An \textit{SLD-derivation} of \( P \cup \{G\} \) via \( P \) consists of a (finite or infinite) sequence of \( G_0 = G, G_1, \ldots \), a sequence \( C_1, C_2, \ldots \) of variants of program clauses of \( P \) and a sequence \( \theta_1, \theta_2, \ldots \) of \textit{mgu}'s, such that each \( G_{i+1} \) is derived from \( G_i \) and \( C_{i+1} \) using \( \theta_{i+1} \) via \( P \).

SLD-derivations can be finite or infinite. A finite SLD-derivation can be successful or it can fail. A successful SLD derivation is one that ends in an empty clause. It is just a refutation.
Definition 5.60: SLD-refutation

An SLD-refutation of \( P \cup \{G\} \) via \( R \) is a finite SLD-derivation of \( P \cup \{G\} \) via \( R \) which has the empty clause \( \Box \) as the last goal in the derivation. In \( G_i = \Box \) the refutation has length \( n \).

Definition 5.61: R-computed answer substitution

Let \( P \) be a program, \( P \) a goal and \( R \) a computation rule. An R-computed answer substitution \( \theta \) for \( P \cup \{G\} \), is the substitution obtained by restricting \( \theta_1 , \ldots , \theta_n \) to the variables of \( G \), where \( \theta_1 , \ldots , \theta_n \) is the sequence of mgus used in an SLD-refutation of \( P \cup \{G\} \) via \( R \).

Definition 5.62: SLD-tree

Let \( P \) be a program, \( G \) a goal and \( R \) a computation rule. Then, the SLD-tree for \( P \cup \{G\} \) via \( R \) is defined as follows:

(a) Each node of the tree is a goal (possibly empty).
(b) The root node is \( G \).
(c) Let \( \leftarrow A_1 , \ldots , A_m , \ldots , A_k (k \geq 0) \) be a node in the tree and suppose that \( A_m \) is the atom selected by \( R \). Then this node has a descendant for each input clause \( \leftarrow B_1 , \ldots , B_q \) such that \( A_m \) and \( A \) are unifiable. The descendant is \( \leftarrow (A_1 , \ldots , A_{m-1}, B_1 , \ldots , B_q, A_{m+1}, \ldots , A_k) \theta \) where \( \theta \) is an mgu of \( A_m \) and \( A \).
(d) A node which is an empty clause has no descendants.

Definition 5.63: Search rule

A search rule is a strategy for searching SLD-trees to find success branches. An SLD-refutation procedure is specified by a computation rule together with a search tree.

We will demonstrate the resolution principle including a computation rule and a search rule by looking at Prolog's search engine.

PROLOG

The declarative and procedural semantics of FOPL having been explained, it is a small step to evaluate the significance of Prolog as a conceptual modelling language. Knowledge modelling in Prolog comes down to the formulation of knowledge in a
syntax acceptable to Prolog and in such a way that the inference engine makes the right deductions. We already know that Prolog-systems use the SLD-resolution with the computation rule that selects the leftmost axiom and with the search rule that works depth-first. What remains to be described is Prolog’s syntax and a number of extra-logical features of Prolog. We first introduce the syntax of Prolog for dealing with normal Prolog-programs and goals:

(1) Constants, functions and predicates begin with a lower-case letter or may comprise any string of characters enclosed in single quotes.
(2) Variables begin with at least one upper-case letter or the underline symbol '_'.
(3) The translation of the FOPL-connectives into the syntax of Prolog is:

<table>
<thead>
<tr>
<th>Connective</th>
<th>FOPL</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>( \neg )</td>
<td>not</td>
</tr>
<tr>
<td>Conjunction</td>
<td>( \wedge )</td>
<td>;</td>
</tr>
<tr>
<td>Disjunction</td>
<td>( \vee )</td>
<td>;</td>
</tr>
<tr>
<td>Implication</td>
<td>( \leftarrow )</td>
<td>:-</td>
</tr>
</tbody>
</table>

(4) Arithmetic function symbols (for example + - * /) may be written in their infix form (for example 3+2 rather than +(3,2).
(5) Arithmetic assignment is achieved by the predicate is.
(6) The unification of two terms is performed using the symbol =.
(7) A special data structure is the list. The list is an ordered sequence of elements that can have any length. A list with the three constants \( a, b, c \) is denoted through \([a, b, c]\). A list has a head and tail. The tail is always a list. \([HT] = [a, b, c] \) evaluates to \( H = a \) and \( T = [b, c] \).

As an example the clause in the syntax of FOPL:

\[
\text{Fire-resistant(wall)} \leftarrow \neg \text{Interior(wall)} \wedge \text{Thermal-Insulation(wall,sufficient)}. 
\]

is expressed in Prolog-syntax as:

\[
\text{fire_resistant(Wall)} :- \\
\quad \text{not interior(Wall), thermal_insulation(Wall,sufficient)}. 
\]

Let us now consider a Prolog-program, a Prolog search-tree, a Prolog-trace and a computed answer substitution in order to combine the declarative and procedural semantics of Prolog. The Prolog-program contains seven Horn clauses:

\[
\begin{align*} 
C_1. \quad & \text{fire_resistant(123)}. \\
C_2. \quad & \text{fire_resistant(Wall)} :- \\
& \quad \text{exterior(Wall), thermal_insulation(Wall,sufficient)}. 
\end{align*}
\]
C3,  fire_resistant(Wall) :-
    interior(Wall), thermal_insulation(Wall,sufficient),
    irradiance(Wall,sufficient).

C4,  exterior(123).

C5,  interior(124).

C6,  thermal_insulation(124,sufficient).

C7,  irradiance(124,sufficient).

The Prolog search-tree based on the Herbrand interpretations and the SLD-derivation is displayed in Figure 5.9. A search-tree describes the search space of a goal in relation to a given program. The goal $G_i$ represents the question which walls are fire-resistant. The leftmost branch shows that $C_1$ is the atom that is first selected by Prolog's computation rule. The mgu $\theta_{i+1}$ of $G_i$ and $C_1$ is $\{X/123\}$. $C_1\theta_{i+1} = fire\_resistant(123)$. This yields a contradiction and thus leads to success. The computed answer substitution is $\{X/123\}$ which is the first answer to the question originally posed to the Prolog-program. So the first branch of the tree is a successful SLD-derivation. If we indicate that we want more answers, Prolog will construct the second branch of the tree. Now the head of $C_2$, the atom $fire\_resistant(Wall)$, is selected by Prolog's computation rule. The mgu $\theta_{i+1}'$ of $G_i$ and the head of $C_2$ is $\{Wall/X\}$. The resolvent of $G_i$ and $C_2$ using $\theta_{i+1}'$ is $\leftarrow exterior(X)$, $thermal\_insulation(X, sufficient)$ Applying the substitution $\theta_{i+2}' = \{X/123\}$ to $exterior(X)$ yields $exterior(123)$. This leads to refutation and therefore to success. The resolvent of $G_i\theta_{i+1}'$ and $C_4$ using $\theta_{i+2}'$ is $\leftarrow thermal\_insulation(123, sufficient)$. This resolvent, however, fails. Then Prolog tries, by means of backtracking, to find another conclusion for $exterior(X)$, but this second attempt fails and Prolog has to traverse the third branch. The resolution process is identical to that in the second branch with the difference that now a refutation is eventually realised delivering the answer substitution $\{X/124\}$.

To develop complex Prolog-programs one often uses a search tree together with a Prolog-trace. The trace is a report of the inference process of Prolog. It provides information that for a part is complementary to the search-tree. Whereas a search-tree should be reconstructed and drawn by a user, the trace is a product of a Prolog-system running a logic program. A number of words occurring in the trace have special meanings:

<table>
<thead>
<tr>
<th>WORD</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL &lt;Goal&gt;</td>
<td>Trying to solve &lt;Goal&gt;</td>
</tr>
<tr>
<td>EXIT &lt;Goal&gt;</td>
<td>&lt;Goal realised&gt;</td>
</tr>
<tr>
<td>REDO &lt;Goal&gt;</td>
<td>Backtrack to &lt;Goal&gt;</td>
</tr>
<tr>
<td>FAIL &lt;Goal&gt;</td>
<td>&lt;Goal&gt; failed</td>
</tr>
<tr>
<td>_&lt;Number&gt;</td>
<td>Variable</td>
</tr>
</tbody>
</table>
For our example the trace looks as follows:

call fire_resistant(_881)
UNIFY 1 [_881 = 123]
exit fire_resistant(123)
redo fire_resistant(123)
UNIFY 2 []
    call exterior(_881)
    UNIFY 1 [_881 = 123]
    exit exterior(123)
    call thermal_insulation(123, sufficient)
    fail thermal_insulation(123, sufficient)
    redo exterior(123)
    fail exterior(_881)
UNIFY 3 []
    call interior(_881)
    UNIFY 1 [_881 = 124]
exit interior(124)
call thermal_insulation(124, sufficient)
UNIFY 1 []
exit thermal_insulation(124, sufficient)
call irradiance(124, sufficient)
UNIFY 1 []
exit irradiance(124, sufficient)
ex\it\it i\tt\textrm{t} fire_resistant(124)
redo fire_resistant(124)
  redo irradiance(124, sufficient)
  fail irradiance(124, sufficient)
  redo thermal_insulation(124, sufficient)
  fail thermal_insulation(124, sufficient)
  redo interior(124)
  fail interior(_881)
fail fire_resistant(_881)

The computed answer substitutions representing a set of fire-resistant walls, are reported in the trace as follows:

No\textsuperscript{\textcopyright}1 \( X = 123 \)

No\textsuperscript{\textcopyright}2 \( X = 124 \)

No more solutions

What makes Prolog valuable is the incorporation of \textit{recursion search} within a deductive framework. We can illustrate recursion search by the representation of a graph in Prolog, shown in Figure 5.10.

\textit{Figure 5.10: A Graph}

The representation in Prolog is:
connected(a,b,25).
connected(a,c,25).
connected(a,d,25).
connected(b,e,25).
connected(c,f,25).
connected(c,j,40).
connected(d,e,25).
connected(d,f,25).
connected(d,h,25).
connected(e,i,25).
connected(f,g,25).
connected(g,h,25).
connected(h,i,25).

\[
\text{path}(\text{Node1, Node2, Nodelist, Distance}) :- \\
\text{path}(\text{Node1, Node2, [Node1, Nodelist, Distance]}). \\
\text{path}(\text{Node1, Node2, Nodelist, [Node1, Nodelist, Distance]}). \\
\text{path}(\text{Node1, Node3, Nodelist, [Node1, Node3, Nodelist, Nodelist, Distance]}). \\
\text{path}(\text{Node1, Node2, Node3, [Node1, Node2, Node3, Nodelist, Nodelist, Distance]}). \\
\text{path}(\text{Node2, Distance}) = \text{Distance} + \text{Distance}_2. \\
\]

We can ask the system whether there is a path between the nodes \( e \) and \( j \), and if so, which nodes are located between these nodes and what is the distance between \( e \) and \( j \). The question becomes:

\[ ? - \text{path}(e, j, \text{Nodelist, Distance}) \]

The system generates the following answers:

\( \text{N°1 Nodelist} = \{e, i, h, d, f, c, j\}, \text{Distance} = 165 \)

\( \text{N°2 Nodelist} = \{e, i, h, d, a, c, j\}, \text{Distance} = 165 \)

\( \text{N°3 Nodelist} = \{e, i, h, g, f, c, j\}, \text{Distance} = 165 \)

\( \text{N°4 Nodelist} = \{e, i, h, g, f, d, a, c, j\}, \text{Distance} = 315 \)

\( \text{N°5 Nodelist} = \{e, b, a, c, j\}, \text{Distance} = 115 \)

\( \text{N°6 Nodelist} = \{e, b, a, d, f, c, j\}, \text{Distance} = 165 \)

\( \text{N°7 Nodelist} = \{e, b, a, d, h, g, f, c, j\}, \text{Distance} = 315 \)
The observation that Prolog is able to represent graphs is not trivial. Bakker (1987) stressed the importance of graphs for the representation and structuring of knowledge. Since a DT is a special kind of graph, this implies that functional object-types can also be captured in graph-structures. Besides this, it is important to focus on the fact that Prolog allows graph-representations to be defined recursively as is shown in the example graph. The reader should note that this is one of the weak points of DT's.

From the above, we are inclined to draw the conclusion that Prolog has considerable representational power. But not everyone will completely agree with that observation. Butrick (1987) and Deville (1990) point to a number of limitations of Prolog:

- **Prolog is incomplete.** The following program displays this:

  ```prolog
  path(a,b).
  path(c,b).
  path(X,Y) :- path(X,Z), path(Z,Y).
  path(X,Y) :- path(Y,Z).
  ```

  It is obvious that `path(a,c)` is a logical consequence of this logic program. There is a successful derivation for the goal `path(a,c)`, but Prolog will never find it. Generally a logic programming language with a depth-first search rule is incomplete. To obtain completeness a breadth-first rule is necessary. From a programming point of view a breadth-first rule is too restrictive because of its ineffectiveness. So it is in the nature of any practical logic programming system to be incomplete. Another aspect of incompleteness is illustrated below:

  ```prolog
  q(a) :- r(a).
  q(a) :- not r(a).
  r(X) :- r(f(X)).
  ```

  Independent of the computation and search rules, there is no successful derivation of `q(a)`, while `q(a)` obviously is a logical consequence. In order to derive `r(X)` Prolog has to search an infinite derivation tree. This is a consequence of the undecidability of FOPL: the derivation of `p` will not
always terminate if \( p \) is not a logical consequence. Incompleteness can occur in such cases.

- **Prolog is unfair.** If an existentially quantified variable occurs in a logical consequence, Prolog is required to succeed and to compute values for that variable. With a depth-first rule not all the possible values will necessarily be found. Such behaviour is called unfair. Consider the following program:

\[
\begin{align*}
\text{append}([],L,L). \\
\text{append}([H|T1],L,[H|T2]) :&- \\
\text{append}(T1,L,T2). \\
\text{append}(List1,List2,List3,List4) :&- \\
\text{append}(List1,List2,List5), \\
\text{append}(List5,List3,List4).
\end{align*}
\]

The program shows the implementation of the relation append/3 which appends two lists (the first two arguments) to a new list (the third argument). It also contains the relation append/4 which concatenates three lists (the first three arguments) to a new list (the fourth argument). The table of Figure 5.11 presents the sequence of successive instantiations of the goal append(List1, List2,[2], List4) where List1 and List2 and List4 are implicitly existentially quantified.

<table>
<thead>
<tr>
<th>List1</th>
<th>List2</th>
<th>List4</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[1]</td>
<td>[2]</td>
</tr>
<tr>
<td>[]</td>
<td>[1]</td>
<td>[1,984]</td>
</tr>
<tr>
<td>[]</td>
<td>[1,984,988]</td>
<td>[1,984,988,2]</td>
</tr>
<tr>
<td>[]</td>
<td>[1,984,988,992]</td>
<td>[1,984,988,992,2]</td>
</tr>
<tr>
<td>[]</td>
<td>[1,984,988,992,996]</td>
<td>[1,984,988,992,996,2]</td>
</tr>
</tbody>
</table>

*Figure 5.11: Unfairness in a Prolog Execution*

List1 is always the empty list, while other correct instantiations exist. Just like incompleteness, unfairness can be solved by applying a breadth-first search rule. As a breadth-first search rule is too inefficient, unfairness is standard in the nature of a practical logic programming language.

- **Prolog is unsound.** Unsoundness occurs when there is a successful computation of a goal which is not a logical consequence of a program. Many Prolog-implementations use a unification algorithm without occur check. This
means that when testing whether or not a variable \( X \) unifies with a term \( t \), Prolog does not check whether \( X \) occurs in \( t \). A goal \( p(X, X) \) leads to success when a program \( p(Y, f(Y)) \) is present. As \( p(X, X) \) is not a logical consequence, soundness is destroyed. When a Prolog-implementation does not work with an occur check the goal \( \text{append([], List, [List])} \) will unfortunately succeed.

- Prolog uses negation as failure inference rule. The idea of the negation as failure is to infer \( \neg q \) when \( q \) cannot be derived. This leads to the phenomenon that Prolog computes \( \neg q \) while \( \neg q \) is not a logical consequence of a logic program. Butrick (1987, p.31) proposes a number of solutions to handle negative information. The negation-by-failure is also called the closed-world-principle.

- Prolog needs control information to reduce the search space during the execution of a logic program. The cut, denoted by ("!"), is the main control primitive. It allows derivation trees to be pruned. Misplaced cuts can therefore lead to incompleteness and unfairness.

- Prolog needs extra logical features just like any other programming language. Besides the cut, we can mention input- and output-primitives and \( \text{assert(a)}, \text{retract(a)}, \text{bagof}, \text{setof} \) and \( \text{univ} \). They are outside the scope of FOPL, but are useful from a pragmatic point of view.

- Prolog is not completely multi-directional. It is often claimed that a Prolog definition can be used in more ways than one. What is meant by multi-directionality is that arguments of a predicate can be used as input and/or output results. For instance, for our first path program we can formulate a goal containing the begin- and end nodes of a path and ask whether this path exists, what nodes make up this path and what the length of this path is. The first two arguments thus function as input arguments. The third and the fourth argument will contain the results of a SLD-derivation (by computed answer substitutions) as we have seen previously.

\[
? - \text{path(e,j,Nodelist,Distance}).
\]

It is also possible to formulate a goal in which the nodes are variables:

\[
? - \text{path(Nodel,Node2,Nodelist,Distance}).
\]

But now the Prolog-systems comes with an unwanted answer substitution:

\[
X = _884, Y = _884, \text{Nodelist} = [_884], \text{Distance} = 0
\]

This can be repaired by incorporating a check, e.g. \( \text{var(Nodel)} \) on the type of the arguments in the first statement of path/4. But there are two more
fundamental restrictions on multi-directionality: efficiency and undecidability. Suppose that \( p(X, Y) \) is a relation where \( X \) is an integer of the domain of a function that decomposes \( X \) into its prime factors. So, if \( X \) serves as input the predicate will compute \( f(x) = y \). But if \( Y \) serves as input the logic program has to compute the inverse function \( y = f(x) \) from a given one! Computations like these are usually too inefficient or are undecidable.

This list of items underlines the gap between the declarative and procedural semantics. If we wish to use Prolog as a conceptual modelling language for the representation of knowledge, more specifically for the representation of knowledge contained in functional object-types, we will have to understand this gap.

5.3.2 Functional Object-types and Prolog

In this section we analyse the possibilities to represent, reconstruct, validate and simulate functional object-types using Prolog. As before, we reduce the discussion about functional object-types to a discussion about functional equivalence.

**Representation**

We revert again to the DT of Figure 5.2 displaying functional equivalence. The DT can be captured in the following Prolog-clauses:

\[
\begin{align*}
goall(X) & :- cs1(a), cs2(k). 
goall(X) & :- cs1(b), cs2(r) 
goall(X) & :- cs1(a), cs2(s), cs3(y).
\end{align*}
\]

The statements of the definition correspond with the DR's of the DT. Another example in which functional equivalence as a form of conceptual interaction occurs is:

\[
\begin{align*}
\text{fire\_compartment}(\text{SPACE}(Sp)) & :- 
is\_a(\text{SPACE}(Sp), \text{enclosed\_room}), 
surface(\text{SPACE}(Sp), Su), 
Su \geq 500. \nonumber 
\text{fire\_compartment}(\text{SPACE}(Sp)) & :- 
is\_a(\text{SPACE}(Sp), \text{technical\_room}), 
surface(\text{SPACE}(Sp), Su), 
Su \geq 50.
\end{align*}
\]

The programs show that it is possible to represent functional equivalence by Prolog. In the previous section, we noticed that DT's were also able to represent functional equivalence. However, there is no reason to think that DT's and Prolog have the same representational power in this respect. There are a number of differences between Prolog and DT's.
The first difference is that in Prolog a condition that is not relevant for the realisation of a goal can simply be omitted. In the Prolog-program with the head goal(x) CS does not occur in the first two clauses, but only in the third clause. In a DT such a condition occurs in the stub as a condition subject with 'don't cares' at the appropriate places in SPACE(C). It is possible to get rid of a condition in a DT by rearranging the knowledge in a different DT-system, but eventually almost always 'don't cares' are needed. In Prolog these 'don't care' conditions need not be mentioned. The second difference is that Prolog is very suitable for effectively representing complex repeat structures. Especially recursion makes Prolog very powerful.

Reconstruction

To what extent does Prolog support the reconstruction of functional equivalence? The first thing to mention is that the conceptualisation of knowledge by Horn clauses in most cases will start with the assessment of a goal. Reverting to a previous example a goal could be the determination whether a certain space, say SPACE(Sp), is a fire_compartment.

fire_compartment(SPACE(Sp)).

The definition of the functional object-type fire_compartment requires knowledge of constraints. From the goal we can find out that the space in question should be an enclosed room and that its surface should be equal to or larger than 500 (m²). The Prolog program now is:

fire_compartment(SPACE(Sp)) :-
  is_a(SPACE(Sp), enclosed_room),
  surface(SPACE(Sp), Su),
  Su >= 500.

Other, functionally equivalent possibilities, may be captured in the following piece of program:

fire_compartment(SPACE(Sp)) :-
  is_a(SPACE(Sp), technical_room),
  surface(SPACE(Sp), Su),
  Su >= 50.

So, the head of a Prolog-definition is a goal. The reconstruction of the body supports the goal-oriented classification of conditions. Each element of the body of a Prolog-program represents a classification of a condition that is useful regarding a goal. Prolog stimulates this, because the elements of the body are gathered in relation to the head. What should be noted is that Prolog does not help to reconstruct conceptual interaction. The mutual influence between a type of room (technical or enclosed) and required surface is hard to detect, because the categorisations of these conditions are spread over the various statements of Prolog definitions. The same is true for the conditional relevance of a condition. In a Prolog-program we only have to incorporate
a condition in a statement when it is relevant. This prevents the occurrence of 'don't cares' in the program.

Validation
Validating a functional object-type represented in Prolog requires a fundamental insight in the declarative and procedural semantics of Prolog. Only then, the knowledge level import of a piece of Prolog code can be assessed. However, Prolog does not provide (non-computational) facilities to perform checks on completeness, correctness and consistency. This is reinforced by the phenomenon that logic programs often suffer from what we call the sliver effect, in which isolated lines of code are difficult to relate. In this respect it is not strange that it is often proposed to carry out the validation of a conceptual model by using the extensive computational facilities Prolog offers for these purposes (see for such a proposal for instance Loucopoulos & Karakostas, 1989, pp.92-93).

Thus simulation is an important basis for validation. We can, for instance, gather a variation of objects and check them against the definitions of the Prolog program. In the fire-compartment example we can check all kinds of spaces to see whether they indeed are instatiations of the object-type fire-compartment. When a 'scenario' illegitimately succeeds or fails, tracing of the execution, if desired in combination with a search tree, will usually reveal the defect and the conceptual model can be modified. Usually, a large number of tests of varying degrees of abstraction should be carried out. Prolog has proved to be invaluable for these types of validations.

Simulation
Simulation in Prolog comes down to formulating an object-type in Prolog-syntax (the declarative semantics) accounting for the procedural semantics (SLD-resolution + search rule + computation rule). In the previous section we discussed the use of simulations for validation purposes. Another use is the consultation of knowledge of a functional object-type by formulating goals for a Prolog program as we did before. A precondition to take into account the non-logical elements of the Prolog language.

5.3.3 Conclusion

Prolog's declarative semantics is claimed to be ideally suited to the analysis of knowledge, while Prolog's procedural capabilities are well suited for software implementation by making a specification 'executable'. Especially in the area of executable specifications and rapid prototyping, Prolog is regarded as an attractive tool for software engineering purposes (Davis, 1982; Kowalski, 1984; Lazarev, 1989). Kowalski (1985) formulates this as follows:

Logic sufficiently blurs the distinction between program and specification that many programs can just as well be regarded as executable specifications. On the one hand, this can give the impression that Logic Programming lacks a programming methodology; on the other, it may imply that many of the
software engineering techniques that have been developed for
c conventional programming languages are inapplicable and
unnecessary for logic programs. (Kowalski, 1985, p.23)

Concerning the goal of this section we should conclude that Prolog is capable of
representing functional object-types. Prolog’s representational power is impressive
and can add a lot to the representational potentials of DT’s. This explains why it is
fairly easy to represent DT’s in Prolog (Reilly, et al., 1987). However, the
transformation of definitions represented in Prolog to DT’s is not always possible.
especially when definitions are recursive difficulties arise. However, the
representational power of Prolog is only adequately employed if we account for the
gap between the declarative and procedural semantics.

Unfortunately, Prolog does not impose a methodology for reconstructing object-
types and lacks facilities for the reconstruction of functional object-types.
Programming in Prolog requires much discipline. Therefore, Deville (1990)
describes a methodology how to use Prolog. On the other hand, Prolog admits the execution of
specified knowledge and thus provides facilities to validate and consult (functional)
object-types. Validation of functional object-types, however, is to an important degree
limited to the simulation facilities offered by Prolog.

Furthermore, we can argue that Prolog-based systems subsume DB-systems with
databases containing rules as well as explicitly stored data. This logically fits with the
observations of the previous chapters. FOPL and Prolog reside close to the knowledge
level and at that level no distinction exists between knowledge-based systems and
DB-systems! Another attractive feature of Prolog is that it allows metalevel-
programming so that it can be augmented relatively easily (Sterling & Shapiro, 1986).

A potential solution of Prolog’s mentioned weak points is to add techniques to
Prolog that intercept the lack of structuring capabilities. Ideally, these techniques
should not conflict with the use of Prolog as a conceptual tool. In the next chapter we
will study such a solution in relation to DT’s.

5.4 CONCLUSION

The central question of this chapter was to investigate whether the joint application of
DT’s and Prolog possesses the necessary characteristics to act as a language for the
representation, reconstruction, validation and simulation of functional object-types.
The question can be answered positively. We have seen that DT’s and Prolog jointly
form a powerful conceptual modelling language that is compatible with the functional
view.

DT’s and Prolog can serve as a pivot for a conceptual integration of AI and DBT.
Both are strongly related by their common descent from mathematical logic. To
understand the step from mathematical logic (FOPL) to Prolog and to acquire
fundamental insight in the use of Prolog as a conceptual language, requires
understanding Prolog’s declarative and procedural semantics. Therefore, both types of
semantics are extensively explained. Analysing these two types of semantics, Prolog
appears to be a logic programming language that admits of simulation of specified knowledge. One of Prolog's weak points is that it does not compel the user to structure knowledge (functionally). Fortunately, it seems that this weak point of Prolog, can be compensated by using DT's which deliver a method for organising and documenting knowledge in a logical manner that permits easy inspection and analysis. The formal definitions of the Codasyl Decision Table Task Group and our modifications of these definitions, help to see that DT's are goal-oriented and stimulate the modelling of functional categorisations. A disadvantage of DT's is that they do not offer efficient facilities to incorporate knowledge into a conceptual model requiring recursive or small definitions for its assessment. In these cases Prolog scores considerably better.

The general conclusion is that DT's and Prolog are complementary and that their joint application yields a powerful modelling language, especially for functional object-types. However, the language does not meet all the requirements mentioned in Section 5.1: DT's lack automated validation and automated simulation facilities and essential graphical facilities for drawing DT's are lacking. The next chapter addresses the remaining drawbacks.
CHAPTER 6

THE ADVANCED KNOWLEDGE TRANSFER SYSTEM

6.1 INTRODUCTION

In spite of the high complementarity of DT's and Prolog as conceptual languages to functionally reconstruct knowledge, the analysis in the previous chapter also revealed that their joint application still yields a language which has certain limitations. This language does, for instance, not offer facilities for automated validation and automated simulation when DT's are employed. Furthermore, essential graphical facilities for drawing DT's are lacking.

To overcome these limitations, the Netherlands Organization for Applied Scientific Research (TNO), five years ago, started the development of a computer-based tool that, besides taking advantage of the complementarity of DT's and Prolog to functionally reconstruct knowledge, was to extend the language with validation, simulation and drawing facilities. The entire development of this tool, from specification up to implementation and maintenance, is based on the theoretical findings of the previous chapters. By now, these efforts have led to the emergence of a tool called AKTS: the Advanced Knowledge Transfer System.

This chapter aims to describe the functionality of AKTS to see (1) to what extent it exploits the complementarity of DT's and Prolog to functionally reconstruct knowledge and (2) to what extent it overcomes the limitations observed. The chapter first provides an overview of the main functions of AKTS (Section 6.2). Subsequently, each main function is described in more detail (Sections 6.3 up to 6.5). We round off the chapter with a number of conclusions and a brief discussion of the performance of AKTS relative to other tools, the performance of AKTS in the daily practice of reconstructing, designing and simulating knowledge universa and, finally, of the future perspectives of AKTS (Section 6.6).

6.2 MAIN FUNCTIONS OF AKTS

AKTS offers main functions to:

• reconstruct a knowledge universe
• design a knowledge universe
• simulate a knowledge universe

These functions are accessible through a menu bar (Figure 6.1).
The reconstruction function of AKTS helps to build a knowledge universe. To fully exploit the complementarity of DT's and Prolog, AKTS supports an integrated use of DT's and Prolog for reconstructing a knowledge universe. To a large extent, a knowledge universe is reconstructed graphically. During the reconstruction process, AKTS provides validation facilities and facilities to remove redundant knowledge elements. In addition, AKTS helps to survey (parts of) the knowledge universe already reconstructed (The reconstruction function is discussed in Section 6.3).

The design function of a knowledge universe is an intermediary function that involves the incorporation of additional information in a reconstructed knowledge universe to simulate that knowledge universe. The information is added in the form of properties of DT's and of conditions and actions. Strictly speaking, AKTS does not really need this information, but the information helps AKTS to make simulation more user-friendly and smooth (The design function is discussed in Section 6.4).

The simulation function refers to consulting a reconstructed and (possibly) designed knowledge universe. It gives users immediate access to the knowledge present in a reconstructed knowledge universe. AKTS has facilities to simulate parts of a knowledge universe. Furthermore, AKTS offers several possibilities for performing What-if analyses. To be able to perform these simulation functions AKTS is equipped with a powerful inference machine (The simulation function is discussed in Section 6.5).

Figure 6.2 shows the main functions of AKTS together with the paths along which one can traverse from one function to another (see also Huijsing, 1992). It should be pointed out that these functions reflect the main steps that are to be taken in each software engineering process. In AKTS these functions are tailored to developing knowledge-based systems.
6.3 RECONSTRUCTING A KNOWLEDGE UNIVERSE

In this section we discuss the reconstruction facilities offered by the Graphical Decision Table Editor, the validation facilities offered by the Integrity Control Subsystem, the reduction mechanisms of AKTS and the interaction between Prolog models and models reconstructed in DT-systems. Finally, AKTS' facilities to provide surveys of a knowledge universe are dealt with.

6.3.1 Reconstruction Facilities: The Graphical Decision Table Editor

To take full advantage of the structuring facilities of DT's, AKTS stimulates a user to start the reconstruction of a knowledge universe with the creation of a DT-system and not with the creation of a Prolog model. In the previous chapter we, however, observed that the adoption of DT's is retarded because of the fact that drawing and modifying tables is a complicated and time-consuming process. This is mainly due to the fact that the modification of a part of a DT often has consequences for other parts of the DT, so that the complete DT has to be redrawn. Even a slight modification may require redrawing a complete DT.

To intercept this problem, AKTS is equipped with a DT-editor that provides a multitude of convenient graphical facilities. This editor has a carefully designed multi-window, menu-driven mouse-oriented interface for optimal communication with users. It allows users to quickly reconstruct and modify a complete DT-system. The editor shows 'intelligent behaviour': it knows what a correct table looks like and applies this knowledge to support the user in the drawing process of a DT. The editor also knows how to obtain a DT that occupies a minimal amount of space. This knowledge is used, if necessary, to calculate the measures of a minimal table after each user's action.

When AKTS receives a command to create a DT, the Graphical Editor displays a window on the screen and draws the four quadrants of an empty DT in this window: a quadrant for the stub with action subjects, a quadrant for the stub with condition subjects, a quadrant for the condition space and finally a quadrant for the action space (see Chapter 5 for information on these quadrants). Figure 6.3 displays the creation of a DT the content of which is discussed in Chapter 7. Initially, the quadrants leave room for one action subject (goal) and for one condition. An edit-box is automatically positioned in the action part of the empty DT to stimulate a goal-oriented strategy. A user can simply insert a goal by typing the name of the goal in the edit-box and clicking the mouse outside the DT but within the window. The Graphical Editor calculates whether the name of the goal fits in the quadrant and draws, if necessary, new quadrants. Consequently, a user can move the mouse-pointer to the condition stub and click once. An edit-box is then opened in the condition stub and the name of a condition can be typed in. Following similar procedures, condition alternatives and action alternatives can be inserted. When one condition alternative is inserted, the editor automatically adds space for a second condition alternative and positions an edit-box in it. The editor does this because it knows that each condition has more than
one condition alternative. There is one exception: if a condition alternative is a 'don't care', no other alternatives are allowed.

![Figure 6.3: Creating a Decision Table](image)

Of course, a DT consisting of one condition and one action will not suffice. Three functions are available to add other conditions, condition alternatives and actions.

- **Add Condition**
  adds an empty row in the condition stub below the condition already inserted. An edit-box is automatically opened in the condition stub. 'Don't cares' are automatically placed in the corresponding parts of the condition space. The user can type in the name of the condition and its condition alternatives by using the edit-box. When an edit-box is already active in the condition stub or in the condition space, the empty row is added below the row in which the edit-box is active.

- **Add Condition Alternative**
  adds space for entering a condition alternative to the right of the position of an active edit-box. The edit box is moved to this new condition alternative. Below the condition alternative 'don't cares' are automatically filled in for the conditions beneath the actual condition. Actually, a complete branch of a tree is added! In Figure 6.4 R3 is added this way. Obviously, using this function requires an active edit-box in the condition space. AKTS alerts the user if the function is called for with no active edit-box in the condition space.

- **Add Action**
  adds an empty row in the action stub below the actions already inserted. If an edit-box is active already in the action stub or in the action space the empty row is added just below the row in which the edit-box is active. An edit-box is opened in the action stub of the new row to allow the user to type in the name
of the action subject. 'Don't cares' are automatically placed in the corresponding parts of the action space.

These functions enable users to draw a DT much more quickly than when using a drawing program. Yet, we decided to equip the editor with a few other functions to improve the speed of the drawing process. Currently, the Graphical Editor also offers powerful editing functions to Select, Cut, Copy, Paste and Clear the following components of a DT: text, condition and condition alternatives, subtrees and finally actions and action alternatives. The effect of these functions is similar to the effect of an average word processor, but the effect of every function differs for every component of a DT. The first component that can be handled by these functions is a piece of text that occurs in a DT. An edit-box can be opened by clicking in an arbitrary part of a DT. Then, the user can select a piece of text by dragging the mouse over the text. The text is marked by a colour. The text can also be added to design dialogues (see Section 6.4). Reversibly, text from these dialogues of AKTS or from other computer programs, can be copied and pasted into a DT.

![Figure 6.4: Adding a Condition Alternative with AKTS](image)

A condition can be selected by positioning the cursor in the corresponding part of the condition stub and dragging the mouse pointer horizontally over the double vertical line. The condition subject and its alternatives will consequently be selected (Figure 6.5). If one moves the pointer upward or downward crossing horizontal lines more conditions will be selected. The Clear command removes the selected condition(s). Every split in the DT caused by the selected condition alternatives is removed! The DT becomes less deep and less wide. Of every split only the first one remains.

Condition alternatives can be selected by clicking in a condition alternative and dragging the mouse to the right or to the left. Every alternative that is (partly) covered by the marqui and that belongs to the same subtree is marked by a colour. Clear removes the condition alternatives. The subtrees of which a selected condition
alternative is a root will disappear. By Copy and Paste the selected condition alternatives are first copied into a clipboard and then pasted to the left of an active edit-field. Below the condition alternatives an empty column appears. This column consists of a part of a table condition entry and of a part of a table action entry.

![Figure 6.5: Selecting a Condition and Its Alternatives for Editing](image)

If a user clicks in a condition alternative and drags the mouse downward, the subtree whose root is the condition alternative, will be selected. When the mouse is moved to the left or to the right adjacent subtrees are selected. Clear deletes the subtrees. Through Copy and Paste subtrees can be added into a DT. Figure 6.6 provides an example of selecting, copying and pasting a subtree. Precondition is that the height of the subtree corresponds with the available space at the insertion point!

The functions Select, Cut, Copy, Clear and Paste operate for actions and action alternatives in much the same way as with conditions. There is one difference. Actions do not show the ramifications that are usually present in trees. Thus, the functions cannot operate upon these structures concerning actions.

Besides speed, another argument induced us to develop these functions: the Graphical Editor should behave in such a way that a user can intuitively operate it with minimal use of a manual (see for instance Figure 6.6). Since many software packages, for instance word processors, make use of similar functions, they are generally known. There is a difference: the functions of the Graphical Editor operate on components of DT's and not on, for instance, pieces of text. Practical experience of the last two years clearly indicates that the Graphical Editor is satisfactory in every respect. The Graphical Editor even allows a user to employ his own language. The only prerequisite is that the user is consistent in naming conditions and actions.

The editor not only focuses on a single DT but also supports the reconstruction of a complete system of hierarchically and logically related DT's. A condition subtable is created by opening an edit-box of a condition subject and entering (by means of a menu item) the command Make subtable. Then, the Graphical Editor creates an initial condition subtable. The condition subject in question appears as an action subject in
the new table and the condition subtable can be finished following the procedures just explained. From this condition table, other condition tables can be reconstructed. The creation of an action subtable is done in an analogous way. By the creation of condition and action tables, one can very easily set up a complete DT-system with several levels within a few minutes!

(A) Selecting a Tree Structure in a Decision Table and Copying It into the Clipboard of AKTS

(B) Clicking in the 4th Alternative of C₂ (see A) and Pasting the Copied Tree Structure from the Clipboard into the Decision Table

Figure 6.6: Graphically Editing (Tree Structures of) Decision Tables. The Entire Operation Only Takes 2 or 3 Seconds

The variety of (bilateral) connections can easily be represented by such a system of DT's. This indicates that AKTS permits users to deal in a graphical way with generalisation, specialisation, association and aggregation. The meaning and necessity of these abstractions in the process of modelling knowledge are explained in Section 3.3.4. To guarantee the integrity of a DT-system, AKTS also offers facilities to
6.3.2 Validation Facilities: The Integrity Control Sub-System

AKTS also offers validation facilities. They fall under the topic of integrity control. Integrity control is a notion that predominantly originates from DBT. It involves the verification of so-called integrity constraints. Integrity constraints are properties that the data of a database are required to satisfy and they are expected to be satisfied after each transaction performed on the database (Das, 1992, p.276; Grefen & Apers, 1993, p.188). Integrity constraints exist in all shapes and sizes, but most of them have already been discussed as static constraints. We formally defined the following types of static constraints: variable constraints, inter-variable constraints, knowledge table constraints and knowledge universe constraints.

The Integrity Control Sub-system of AKTS is responsible for maintaining integrity of a knowledge universe. It deals with all static constraints by providing possibilities to check (1) the exhaustiveness and exclusiveness of every DT of the knowledge universe and (2) the formal links between head tables and subtables. The exhaustiveness and exclusiveness integrity constraints are based on the formal definitions of exhaustiveness and exclusiveness concerning trees (Chapter 5). The integrity constraints for links are based on the formal definitions of (bilateral) connections (Chapter 3).

Checking Exhaustiveness and Exclusiveness

Checking the exhaustiveness and exclusiveness of a DT requires the domains of the condition subjects in the table to be known. As explained, a domain is a set that describes the allowed values for a subject. Domains are inserted using the design function of AKTS (See Section 6.4). To enable AKTS' Integrity Control Sub-system to verify several types of condition subjects, the domain of a condition subject must be one of the following types:

- Text
- Enumeration
- Interval

A domain of the type Text is specified as a set in which each element is a string of letters of a specified maximum length. The domain is denoted by \( Chs(n) \). \( Chs(3) \) denotes the set of all strings that belong to some alphabet with a maximum length of 3: \( a, \text{why} \) and \( \text{yes} \) are 3 elements of this set. This type of domain helps to simulate a knowledge universe by, for instance, enabling AKTS to deal with names of objects which cannot be known in advance. This means that it is not possible to enumerate the elements of the domain of the condition subject that should represent one of these names. The only possible way to define the domain is to restrict the lengths of the...
strings. These strings can be used to personalise messages in a consultation session or to retrieve information from a database or a formula.

A domain of the type *Enumeration* is a set of elements that are enumerated one by one. An element can be of type *word*, an *integer* or a *real*. Often, enumeration domains only have words as possible elements. We have introduced the type integers and reals to deal with specific situations in which a condition subject has a limited number of values that are discontinuous integers or discontinuous reals, so that intervals cannot be used. Possibly these condition subjects should be used in formulas. A few condition subjects and their enumeration domains are described through in the following ordered pairs:

(transport; {car, vehicle, bycicle, plane})  (word)
(distances to be run; {5, 10, 21})  (integer)
(widths; {1.12, 2.24, 3.36})  (real)

A domain of the type *Interval* is a set the elements of which are not enumerated, but indicated by an interval. The element are reals or integers.

![Figure 6.7: Two Classifications of One Condition Subject To Be Checked](image)

As stated previously, the exhaustiveness and exclusiveness checking procedures should operate upon the nodes of each subtree that can be distinguished in a DT. The subtrees to be checked have a height 1 and their nodes have a depth > $C_{num}$. This tree-approach implies that several classifications of one and the same condition subject should be subject to control. The *general checking principle* works as follows. Figure 6.7 graphically displays two subtrees of a tree. The first half of each child of the subtrees in the tree contains the well-known formal indication of a condition alternative. The second half contains the literal condition alternative. To check the exclusiveness and exhaustiveness of a DT, AKTS evaluates each second half of a node of a relevant subtree to a set. For instance, *red* is evaluated to the set {red}. Remember that according to the formal definitions of the previous chapter the *union* of all nodes of a subtree should equal the relevant domain to ensure exhaustiveness and that the cross-section of all nodes of a subtree should be empty to ensure exclusiveness.
Suppose we have a condition $C_i$ with $CD_i = \{\text{red, orange, yellow, green}\}$. Then we can see, in Figure 6.7, that the first classification of this condition below node $X$ is exhaustive and exclusive, because:

\[
\bigcup\{\{\text{red}\}, \{\text{orange}\}, \{\text{yellow}\}, \{\text{green}\}\} = CD_i \quad \text{and} \quad 
\bigcap\{\{\text{red}\}, \{\text{orange}\}, \{\text{yellow}\}, \{\text{green}\}\} = \emptyset
\]

However, the second classification below node $Y$ is not exhaustive and not exclusive, because:

\[
\bigcup\{\{\text{green}\}, \{\text{orange}\}, \{\text{yellow}\}, \{\text{green}\}\} \neq CD_i \quad (\text{red is lacking}) \quad \text{and} \\
\bigcap\{\{\text{green}\}, \{\text{orange}\}, \{\text{yellow}\}, \{\text{green}\}\} \neq \emptyset \quad (\text{green occurs two times}).
\]

In both cases, the Integrity Control Sub-system alerts the user that the integrity of the knowledge universe is endangered. So far, integrity control seems to function reasonably well. It may be clear, however, that many cases show much more complexity. Complex cases may emerge as a consequence of using $\text{OR}$, $\text{AND}$, $\text{NOT}$ and $\text{ELSE}$ as operators in the condition alternatives of a DT. The following example displays the condition $\text{height}$ that is supposed to have the following interval domain of integers: $[0, \ldots, 50]$. Suppose that the condition is the first condition in the DT, so that we do not have to display a tree structure. The displayed situation containing an $\text{OR}$-operator is not accepted by AKTS, because the values $[10, \ldots, 14]$ are lacking. The Integrity Control Sub-system of AKTS classifies the DT as being not exhaustive:

```
\text{Cl} \quad \text{Height} \quad X < 5 \quad 5 \leq X \leq 10 \quad \text{OR} \quad 15 \leq X \leq 20 \quad X \geq 20
```

\text{evaluates to:} \quad \{0, \ldots, 5\} \quad \text{evaluates to:} \quad \{5, 6, 7, 8, 9, 15, 16, 17, 18, 19, 20\} \quad \text{evaluates to:} \quad \{20, \ldots, 50\}

*Figure 6.8: A Non-Exhaustive Condition*

The connective $\text{AND}$ can only be used for *multi-valued* variables. A multi-valued variable can attain several values at the same time. The verification of multi-valued variables is not supported by the Integrity Control Sub-system for two reasons. First, practical experience does not indicate a need for automatic support for the verification of multi-valued variables. Second, the verification procedures for multi-valued variables are of a tremendous complexity and have limited value because they are only able to perform simple logical checks.

The $\text{NOT}$-operator can be used in a condition alternative to cover values that cannot be used elsewhere. The range of the $\text{NOT}$-operator often contains the $\text{OR}$-connective.
Below, we present two examples containing the NOT-operator. The first example refers to a condition subject colour with the domain \{red, yellow, green\}. The second example refers to the condition subject height again with the domain: \([0, ..., 50]\). Both situations are not accepted by AKTS. The first situation is not exclusive because green occurs twice. The second situation is not exhaustive because of the absence of 15.

![Figure 6.9: A Non-Exclusive Condition with NOT-operators](image)

![Figure 6.10: A Non-Exhaustive Condition with NOT-operators](image)

The Integrity Control Sub-system of AKTS also deals with ELSE and 'don't cares'. ELSE always evaluates to the remaining values of the domain that are not yet present in a DT, whereas a 'don't care' always equals the complete domain. An example with ELSE is displayed by Figure 6.11. Apart from the evaluation of ELSE to \{Yellow, Green\}, AKTS does not accept this condition, because Blue does not match the relevant variable constraint.

![Figure 6.11: A Condition Containing One Condition Alternative with ELSE](image)
The integrity constraints that are covered by the exclusiveness and the exhaustiveness constraints are *deferred* constraints. The user determines whether and when these constraints are verified. *Immediate* constraints, on the contrary, have to be satisfied after every user action. The constraints referring to the formal connections between levels of DT's are immediate constraints. They are explained in the next section.

<table>
<thead>
<tr>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>A1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

(A) A Head Table

<table>
<thead>
<tr>
<th>Condition Subject 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>A1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

(B) A Condition Subtable

<table>
<thead>
<tr>
<th>Action Subject 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>A1</td>
</tr>
<tr>
<td>A2</td>
</tr>
<tr>
<td>A3</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

(C) An Action Subtable

Figure 6.12: A Decision Table System
Checking Formal Connections in a Decision Table System

The facilities that check the hierarchical connections occurring in a DT-system rely on the formal definition of (bilateral) connections as given in Chapter 3 and applied in Chapter 5. We revert again to the following example of Figure 6.12. The formal definition of this DT-system is:

\[ RL(\text{Abstract}) = \{ DR_1, DR_2, DR_3, DR_4, DR_5, DR_6 \} \]

\[ DR_1 = \{ (\text{Condition Subject 1; A}), (\text{Condition Subject 2; D}), (\text{Action Subject 1; X}) \} \]

\[ DR_2 = \{ (\text{Condition Subject 1; A}), (\text{Condition Subject 2; E}), (\text{Action Subject 1; \textit{--}}) \} \]

\[ DR_3 = \{ (\text{Condition Subject 1; B}), (\text{Condition Subject 2; D}), (\text{Action Subject 1; \textit{--}}) \} \]

\[ DR_4 = \{ (\text{Condition Subject 1; B}), (\text{Condition Subject 2; E}), (\text{Action Subject 1; X}) \} \]

\[ DR_5 = \{ (\text{Condition Subject 1; C}), (\text{Condition Subject 2; G}), (\text{Action Subject 1; X}) \} \]

\[ DR_6 = \{ (\text{Condition Subject 1; C}), (\text{Condition Subject 2; F}), (\text{Action Subject 1; \textit{--}}) \} \]

\[ RL(\text{Condition Subject 1}) = \{ DR_1, DR_2, DR_3, DR_4 \} \]

\[ DR_1 = \{ (\text{Condition Subject 1; 1; K}), (\text{Condition Subject 1; 2; P}), (\text{Condition Subject 1; A}) \} \]

\[ DR_2 = \{ (\text{Condition Subject 1; 1; K}), (\text{Condition Subject 1; 2; Q}), (\text{Condition Subject 1; B}) \} \]

\[ DR_3 = \{ (\text{Condition Subject 1; 1; L}), (\text{Condition Subject 1; 2; P}), (\text{Condition Subject 1; B}) \} \]

\[ DR_4 = \{ (\text{Condition Subject 1; 1; L}), (\text{Condition Subject 1; 2; Q}), (\text{Condition Subject 1; A}) \} \]

\[ RL(\text{Action Subject 1}) = \{ DR_1, DR_2, DR_3 \} \]

\[ DR_1 = \{ (\text{Condition Subject 3; S}), (\text{Condition Subject 2; V}), (\text{Action Subject 1.1; H}), (\text{Action Subject 1.2; M}), (\text{Action Subject 1; X}) \} \]

\[ DR_2 = \{ (\text{Condition Subject 3; S}), (\text{Condition Subject 2; W}), (\text{Action Subject 1.1; I}), (\text{Action Subject 1.2; N}), (\text{Action Subject 1; X}) \} \]

\[ DR_3 = \{ (\text{Condition Subject 3; T}), (\text{Condition Subject 2; \textit{--}}), (\text{Action Subject 1.1; J}), (\text{Action Subject 1.2; O}), (\text{Action Subject 1; X}) \} \]

Further:

\[ h_1 = \{ (\text{Condition Subject 1; Condition Subject 1}) \} \text{ and:} \]

\[ h_1 \text{ connects } RL(\text{Condition Subject 1}) \text{ with } RL(\text{Abstract}) \]

The connection states that the restriction of the sets of \( RL(\text{Condition Subject 1}) \) to \( \text{dom}(h_1) \) is a subset of the restriction of the sets of \( RL(\text{Abstract}) \). To describe the relation between the head table and the action subtable, the identical function \( h_2 \) is defined as follows:

\[ h_2 = \{ (\text{Action Subject 1; Action Subject 1}) \} \text{ and} \]
There are several ways in which a user might, unintentionally, break off a connection between two tables. First, the user may change the name of parameters involved in the connection (conditions and actions are collectively called parameters). When this happens, the Integrity Control Sub-system of AKTS immediately asks the user whether breaking off the connection is indeed intended. Second, the user may add or delete parameter alternatives involved in the connection. Adding alternatives only in a head table or only in a subtable, in case of a bilateral connection, is not allowed. If, on the contrary, in both tables the same alternative is added or deleted the bilateral connection remains intact, so no immediate danger is caused. Adding an alternative only to a head table is allowed in case of a non-bilateral connection, whereas deleting an alternative only in the head table is not allowed if it concerns an alternative that is an action alternative in the connected subtable. For instance, when a user deletes condition alternative \( A \) from the DT Abstract, the subset relationship is broken off. As long as alternatives are added that also occur in the head table to only a subtable, adding is allowed in case of a non-bilateral connection. In this situation deleting alternatives in the subtable and adding and deleting in both types of tables is also allowed as the subset relation remains intact.

6.3.3 Reduction Mechanisms

AKTS also has a facility to optimise DT's. The optimisation facility is based on the phenomenon that identical goal-patterns in a DT are reducible to common condition alternatives. This occurs when:

a. neighbouring rules have identical goal patterns and the similarity of these patterns is reducible to common condition alternatives
b. neighbouring groups of rules have identical goal patterns and the similarity of these patterns is reducible to common condition alternatives

There are several reduction possibilities. Figure 6.13 shows a first reduction possibility. The first tree represents a DT with four DR's each containing two conditions and one action. The actions show two patterns: 00 and 11. The first pattern is reducible to the second condition. The pattern indicates that there is no functional difference between the condition alternatives \( CA_{21} \) and \( CA_{22} \). Since both alternatives are functionally equivalent, (the reduction module of) AKTS removes the differences, assuming that \( CA_{21} \) and \( CA_{22} \) are exclusive and exhaustive, by replacing the alternatives by a 'don't care'. In this way the first two rules are automatically reduced to one.

Almost the same procedure can be applied to the second pattern 11. This pattern is reducible to the differences between \( CA_{12} \) and \( CA_{13} \). A difference with the previous reduction is that the condition alternatives causing differences are located at another level. Another difference is that now it is not allowed to replace \( CA_{12} \) and \( CA_{13} \) by a 'don't care', because another alternative, \( CA_{11} \), is present. For this reason AKTS
applies the OR-operator and reduces two rules to one rule. The complete operation yields a reduction of 50%.

Figure 6.13: Reduction of Neighbouring Rules Using 'Don't Cares' and the OR-Connective

Figure 6.14 displays another type of reduction. It shows that groups of patterns are present. The first group of patterns is 01 and 01. This pattern is reducible to the condition alternatives CA_{21} and CA_{22}. Since AKTS assumes that the condition subject to be reduced is exhaustive and exclusive, it replaces these alternatives by a 'don't care'. The operation yields a reduction of two rules. The second pattern is 10 and 10. The repetition of patterns is reducible to CA_{12} and CA_{13}. As there is again a third condition alternative, AKTS uses the OR-connective and reconstructs the condition alternative B OR C. This operation also yields a reduction of 50%.

A quite different type of reduction is exemplified in Figure 6.15. It should be noticed that the first pattern 00 cannot be reduced, because the first 0 belongs to another subtree of the first condition. But note that the first rule and the fourth rule are exactly the same except for condition 1. AKTS can automatically merge these rules by combining the relevant alternatives CA_{11} and CA_{13} through an OR-connective. This reduction operation deletes one rule. AKTS is now in a position to discover another reduction possibility. Two patterns emerge: 01 and 01 which are reducible to differences between CA_{12} and CA_{14}. AKTS again uses the OR-connective. The complete procedure again reduces the number of rules by 50%.

Note that a tree is reduced from below, so that also subtrees which are dissimilar in shape but logically equivalent, are reduced. Figure 6.16 demonstrates a reduction with the OR-connective in practice. The domain is discussed in Chapter 7.
Figure 6.14: Reduction of Neighbouring Groups of Rules Using 'Don't Cares' and the OR-Connective
Figure 6.15: Reduction of Non-Neighbouring Groups of Rules

(A) A Decision Table in AKTS Before Reduction

(B) A Decision Table After Reduction

Figure 6.16: Reduction in Practice
Part of the knowledge universe can be specified through Prolog. For didactic reasons, we postpone the discussion of using Prolog in AKTS to the next section dealing with the design of a knowledge bank using AKTS.

6.3.4 Reconstructing in Prolog

The lack of structuring capabilities of Prolog imposes much discipline on the modeller. This weak point of Prolog can be circumvented in two ways. First, the modeller in Prolog should be trained to develop skills to reconstruct a(n executable) knowledge universe in Prolog. Second, formal techniques can be added to provide the necessary structuring facilities. The two ways are complementary and reinforce each other. The skills needed to reconstruct knowledge universa in well-styled Prolog models can be gained by practice together with consultation of specific textbooks on this topic.

In AKTS the second way is implemented by taking advantage of the structuring capabilities of DT's. The reconstruction of a (part of a) knowledge universe in Prolog predominantly takes place from a reconstructed DT-system: when the option of DT's is not effective anymore, part of the knowledge universe can be modelled in Prolog. This Prolog model can be inserted in AKTS. A call to this model takes place from (1) a DT property or a (2) a parameter property. Such a call can be used to retrieve the answer to the question whether a certain wall is susceptible to the formation of a chemical product. This example is elaborated in Chapter 7.

\[
susceptible\_to(Wall,Chemical\_Product,Answer).
\]

Input arguments and output arguments are placed between exclamation marks. Input and output respectively means input and output for the Prolog model. Thus the variable Wall may be a parameter that occurs in a DT or in another Prolog model. When the call is made, Wall should already be instantiated. The variable Chemical Product and Answer could be output variables that return values from the Prolog model. A Prolog model representing the object-type susceptible to ettringite might look as follows:

\[
susceptible\_to(Wall, the\_formation\_of\_ettringite,yes) :-
\begin{align*}
&\text{contains}(Wall,\text{gypsum}), \\
&\text{contains}(Wall,\text{hydrated\_calcium\_aluminates}), \\
&\text{contains}(Wall,\text{moisture}).
\end{align*}
\]

Using Prolog's inference engine (Chapter 5) Chemical Products can be bound to ettringite and Answer to yes. These values can subsequently be used to proceed the problem solving process.
6.3.5 Navigating through a Knowledge Universe

AKTS also provides facilities to survey the structure of the knowledge universe. Figure 6.17 shows two surveys of parts of a knowledge universe. The knowledge universe belongs to a knowledge-based system that is currently being developed. It contains knowledge coming from several European experts; with it damage to a building caused by deterioration processes can be diagnosed. In Chapter 7 we discuss this knowledge-based system in relation to functional object-types.

(A) Survey of the Knowledge Universe 'Brick Masonry Susceptible to the Formation of Ettringite'

(B) Survey of the Same Knowledge Universe. Now the Survey Zooms in on 'Gypsum as an in Situ Product of a Chemical Reaction'

Figure 6.17: Two Surveys of the Structure of a Knowledge Universe.

6.4 DESIGNING A KNOWLEDGE UNIVERSE

When a knowledge universe is reconstructed using DT's and Prolog, it should be designed. Normally, the design phase is an intermediary step that should preferably be
executed before a knowledge universe is implemented into a knowledge-based system. The design step focuses on the design of the architecture, the modules to be built, the user interface and on the design of the implementation formats of a knowledge-based system.

For our purposes design looks somewhat different. In AKTS the design phase implies adding extra information to the reconstructed knowledge universe. This information is useful for simulating the knowledge universe. It refers to properties of DT's and to properties of conditions and actions of DT's.

6.4.1 Designing Decision Tables

We can specify three properties of a DT. A designer can type in a text that explains the reason of existence of the DT and that clarifies the domain for which the DT is reconstructed. This is the Remarks property. Furthermore, a designer can specify a Do Before property and a Do After property. A designer can mention an already specified Prolog model as the value of these properties. Do Before and Do After then imply the execution of the Prolog model respectively before or after executing the DT. These properties can be applied to give a message to a user, to query a database, to conduct calculations and so on.

Figure 6.18: Designing Decision Tables

6.4.2 Designing Parameters

A designer can also assign values to parameter properties. A parameter can have several properties. Explanation is a piece of text that serves as clarification of the parameter. If necessary, the text can be shown on the screen during the execution of a knowledge universe. The value of the Prompt property is text representing a question. This question can be posed to a user so that the user can provide AKTS with a value for the parameter. The When Needed property is a Prolog command or the name of a Prolog model. At this moment AKTS has the following predefined (Prolog system predicates are always present) Prolog commands available:
do_table(<tablename>)
The table name filled in as an argument will be executed.

repeat_table
The actual DT or the most recently executed DT will again be executed.

reset(<parameter>)
The parameter mentioned will lose its value if it has received one.

reset([<parameterlist>])
The parameters mentioned will lose their values if they have received one.

reset_all
All parameters with values will be reset.

reset_conditions
All conditions of the actual DT or of the last DT executed lose their values.

reset_from(<parameter>)
All parameters that received a value since the tracing of the parameter mentioned will lose their values if they have received one.

show_picture(Resource file, ResourceNumber, Window, Top, Left, Depth, Width)
Displays a picture from the specified resource file in the graphical window with the specified size.

show_picture(Resource file, ResourceNumber, Window)
Displays a picture from the specified resource file in the window.

value(Parameter, Value)
Retrieves the value of a parameter. The first argument of this predicate is a parameter. The second argument is the value of that parameter.

Together these Prolog commands form an important extension for using DT's in the knowledge modelling process.

To specify a Prolog model a special window is available. A Prolog model may contain formulas. These formulas can be used to calculate, for instance, the stiffness of a column. The When Needed property can also function as a demon. Before a parameter is evaluated, AKTS checks the When Needed property of the parameter. The value of the When Found property is executed after that the relevant parameter has received a value. The value of the Default property is the default value of the parameter. The default value will be assigned to the parameter when no ways are open anymore to find the value of a parameter.
The Domain Type property assesses the allowed values of the parameter involved. It can, as already discussed, have the following types: text, enumeration, interval. A text parameter is a parameter the value of which is a string of text of a certain length. An enumeration parameter is a parameter the possible values of which are enumerated in a set called the domain of the parameter. The difference between a text parameter and an enumeration parameter is that the text parameter can attain every potential value that is a string of text with a limited length, while an enumeration parameter can only
attain values that are specified in the domain. Text parameters do not play a role in the realisation of a goal of a DT. An application of AKTS may need a text parameter for creating messages with a personal flavour. For instance, the name of a person can be used to 'personalise' messages. The parameter name then is a text parameter. Enumeration parameters can be single- or multi-valued. The meaning of interval will be clear. Parameters of type Enumeration or Interval can have integers or reals as their values (see Figure 6.19).

The Ask first property denotes that the user must be asked first about the value of the parameter. Finally, the property Goal denotes whether the parameter is a main goal of the knowledge-based system-application. If a parameter has a Goal property we call it a goal parameter. Every application must have at least one goal parameter. Goal parameters are placed in the goal stack. When AKTS starts the execution of a knowledge universe, AKTS begins tracing the goals of the goal stack.

6.5 SIMULATING A KNOWLEDGE UNIVERSE

If a knowledge universe is reconstructed AKTS offers several facilities to simulate the knowledge universe. For this purpose AKTS possesses an inference machine (a meta-interpreter) that takes care of the execution of the knowledge universe.

6.5.1 The Inference Machine

The inference machine of AKTS is goal-oriented. It departs from a list of goal parameters. The inference machine attempts to trace these goal parameters in order of appearance. Tracing a parameter is the process of trying to find a value for a parameter. The inference machine of AKTS uses a backward chaining strategy, though it is possible to influence the inference machine. During this inference process parameters that are not goal parameters also must be traced. These non-goal parameters are relevant for tracing the goal parameters. There are several ways to find a value for a parameter; they are described below.

1. If the parameter to be traced has an Ask first property, its value is simply asked. Whether the user has to make a selection out of a list of possible values or type in the exact value of the parameter depends on the Type property of the parameter (see Figure 6.19). If the value of the parameter can be found in other DT's, the user has the option of indicating that he does not know the answer.

2. Another way of finding a value for a parameter is provided by the When Needed property. The When Needed property is often used if the value of a parameter has to be calculated and the necessary formulas are available in the form of the When Needed property value. Another use of this property is to
design a query to a database as a *When Needed* property value. The calculation of the query then returns the value of the parameter.

3. The value of a parameter can also be found by tracing DT's. AKTS then tries to look for a DT that contains the parameter to be traced as an action parameter. The value of the action parameter depends on the condition parameters of the DT. In order of appearance the condition parameters are traced until the value of the action parameter is found. When the condition parameters are traced again other DT's can be searched through. In this way a whole system of DT's can be traced by the inference machine of AKTS.

4. If still no value is found the inference machine uses the *Default* property if present. The value of the *Default* property becomes the value of the parameter.

5. If the above strategies fail, the user is finally asked (again) to provide a value. But now the user is forced to provide an answer and does not have the possibility to answer 'Don't Know'.

When one of the ways succeeds the inference process for the parameter stops. The parameters that receive values during the inference process, the so-called facts are stored in a list. The user can modify this list by explicitly resetting these parameters. Prolog commands can also be used to change the list of facts.

The inference machine of AKTS applies four basic strategies: the determination of parameter values, the inference of values, the evaluation of a DT and the execution of Prolog models (Figure 6.21). These strategies interactively co-operate with each other to realise a general and efficient reasoning process.

*Reasoning strategy 1: Determining a Value of a Parameter*

Figure 6.21 (A) shows a general overview of the strategy of determining a value of a parameter. The point of departure is that there is a parameter to be traced. If its value is known, the inference machine of AKTS has finished its work. Otherwise, the inference machine checks whether the parameter has an *Ask first* property. By means of assigning *Ask first* properties to parameters and adding subtables, AKTS asks the value and checks, if necessary and possible, whether the value belongs to the domain of the parameter. If the value cannot be asked or if the answer is not right, AKTS proceeds by checking whether there is a task and, if so, attempts to execute it. If this procedure does not lead to success, AKTS attempts to infer the value (Figure 6.21 (B)). If inferring does not yield a value either, AKTS searches for a possible default value. If all these possibilities do not yield a value, AKTS, as a last resort, (again) asks the user for a value.

*Reasoning strategy 2: Inferring a Value*

To infer the value of a parameter, AKTS searches through the database of DT's to find a table that contains the parameter as a goal. If AKTS is not able to find such a table, it appeals to other reasoning strategies. If, on the contrary, a table is found, AKTS verifies whether a Prolog model is associated and, if this is the case, it executes this
model (Figure 6.21 (D)). Subsequently, AKTS evaluates the table that was found according to the evaluate a DT strategy (Figure 6.21 (C)). Finally, AKTS again checks whether a Prolog model is associated with the table and, if this is the case, it executes this model (Figure 6.21 (D)).

Reasoning strategy 3: Evaluating a Decision Table
For evaluating a DT, AKTS first selects a condition. If the selected condition is a 'don't care', the inference machine of AKTS jumps to the next available condition in the table. If the selected condition is not a 'don't care', AKTS attempts to determine the value of the parameter and, if possible, selects a subtable that contains the parameter in question as a goal (Figure 6.21.A). If no condition is available the right action value is selected. If there is a Prolog model associated with the action alternative that model is executed.

Reasoning Strategy 4: Executing a Prolog Model
One of the first things AKTS does to execute a Prolog model, is to verify whether the model contains parameters that have to be traced first. Parameters that constitute an input for the model are selected. Since their values should be used in the model, their values are traced using the strategies just explained. Subsequently, the Prolog model is executed according to the Prolog inference machine as explained in Chapter 5. Normally, the execution of Prolog models is used to retrieve knowledge from databases, to use the knowledge of object-types that require recursive definitions, or to access formulas.

6.5.2 Performing What-If Analyses

Using the inference machine, What-if analyses can be performed. When a user is simulating a knowledge universe and has already given at least one answer, AKTS offers the possibility to review the answers. A parameter that has received a value can be selected (Figure 6.22). Next, the associated answer can be made visible and changed. If the user indicates that the simulation should be continued (by clicking the button Reconsult, see Figure 6.22), AKTS retracts the values of the parameters that logically follow the parameter the answer of which has been changed by the user. Since the parameters that now follow the changed parameters may be different, other parameters may be traced. This may imply that (a) parameters that already have attained values are not useful anymore and (b) parameters that have been passed in a previous 'round' can yet be asked or otherwise traced for their value.

By means of the What-if facility, AKTS offers the possibility to study the consequences of changing one (or more) parameters if the values of the other variables are fixed. Since, AKTS deals with functional equivalence, the What-if analyses are in accordance with the theory of functional object-types. This means that the influence of changing an INUS-condition within a conjunct set can be assessed. It is also possible to check the influence of a changed conjunct set. In this way, complete functional analyses can be performed.
Figure 6.21: The Inference Machine of AKTS
6.5.3 Debugging a Knowledge Universe

A knowledge universe can be debugged. This implies that AKTS, when asked, shows the complete reasoning process in a special consultation window. Search trees that are built are displayed by indents. The depths of the indents corresponds with the depths of the traversed tree. AKTS tells the user whether a parameter value has been asked, inferred (i.e. found in other tables), determined by performing a task (for instance accessing a database or a formula) or by default.

6.5.4 Tracing a Part of the Knowledge Universe

In order to prevent a user from being continuously obliged to simulate a complete knowledge universe, it is possible to select a parameter and let the inference machine find a value for it. Then, AKTS by means of its inference machine only uses knowledge needed to find a value for the requested goal parameter. Another way to trace a part of the knowledge universe is to select a non-goal parameter and ask AKTS to trace it. The procedure that is followed is similar to the procedure for tracing goal parameters (see the menu in Figure 6.22).

6.5.5 Explanation Facilities

During simulation, the user has the possibility to ask for explanation. Two types of explanations are shown then: first, the text inserted as the Remarks property of the DT and second, the text that belongs to the Explanation property of the parameter in question. These pieces of text can help a user to give an answer. A third explanation
possibility is to ask AKTS to display the DT to which the parameter belongs. By this, a user is able to see the background of AKTS asking the value of a particular parameter.

6.6 CONCLUSION AND DISCUSSION

The conclusion is justified that, to a considerable degree, AKTS takes advantage of the complementarity of DT's and Prolog. AKTS not only uses the strong points of DT's (their structuring capabilities, their well-organised representation of knowledge permitting easy validation and simulation by hand, their goal-orientation and tackling of functional equivalence), but also intercepts their weak points (lack of facilities for automated simulation, lack of possibilities to incorporate object-types that require small or recursive definitions) by using Prolog (see Chapter 5). By the integration of DT's and Prolog AKTS offers facilities to represent, reconstruct, validate and simulate functional object-types.

It is also justified to draw the conclusion that AKTS, in a convenient way, overcomes the three limitations observed of jointly applying DT's and Prolog: lack of facilities for automated validation and automated simulation and lack of facilities for drawing DT's. Firstly, AKTS permits automated validation. By dealing with exhaustiveness and exclusiveness and (bilateral) connections between DT's, the Integrity Control Sub-system of AKTS helps to validate that the intended system is specified. Together with the inference machine it also makes the validation that the program code satisfies the model superfluous. Secondly, AKTS permits automated simulation. The inference machine of AKTS provides extensive facilities to simulate a reconstructed knowledge universe including facilities for conducting What-if analyses. Thirdly, the Graphical Decision Table Editor of AKTS offers advanced graphical support to quickly reconstruct and modify DT-systems.

Undoubtedly, the two previous conclusions will not prevent the reader of still having at least a few questions. For instance: 'How does AKTS relate to other packages?' AKTS differs from tools that work with DT's such as PROLOGA (PROcedural LOGic Analyser; see Vanthienen, 1988 and the DT-tool of Mors (1993). In PROLOGA, the individual DR's of a DT have to be typed in. Only if every rule is correct and the rules together form a tree, PROLOGA is able to draw a DT. If the table has to be modified, a user is forced to correct the individual DR's. Also in the DT-tool, the tree structure is absent during the edit procedure leading to an error-prone reconstruction process. In AKTS this can be done graphically and much faster. In comparison with a tool such as TheME (Balder & Akkermans, 1992), AKTS is much more direct and much easier to use. In contrast to TheME, AKTS offers a high-level language that shields the user from the low level complexities of mathematical logic. For these reasons, and other reasons already clarified in Chapter 5, we think that AKTS outperforms other packages.

1 A number of TNO-reports contain a detailed, formal functional specification and technical design concerning the architecture of the modules, module descriptions and the data structures of AKTS. Also the Prolog code (a small piece of it is displayed in Appendix A) is documented. These documents are, for understandable reasons, not available.
Another readers' question might be: 'How does AKTS perform in the daily practice of reconstructing, designing and simulating knowledge universa?' By now, it appears that the utility and ease-of-use of the representational power of AKTS have not pass unnoticed in the building and construction industry (and beyond). Currently, AKTS is operational in the following domains:

- Checking designs of office building on the fire-safety regulations
- Damage assessment of buildings (see Figure 6.23)
- Diagnosis of humidity problems in houses
- Diagnosis of Indoor Environments
- Design of connections in steel structures
- Matching office building profiles concerning location and quality for real estate agents
- Urban planning and architecture

![Figure 6.23: AKTS in Full Operation with Knowledge about the Diagnosis of Brick Masonry Walls (see Chapter 7). Bottom right: Parts of the Structure of the Knowledge Universe. Top Left: A Decision Table Containing Knowledge about Rising Damp. Bottom left: Consultation Output. Middle: Consult Dialogue for Questioning Users. The Wall and Diagram are Displayed Using the Graphical Description Language of AKTS](image-url)
CHAPTER 6

THE ADVANCED KNOWLEDGE TRANSFER SYSTEM

Besides this, new AKTS-projects are being set up to improve knowledge transfer in many other fields such as that of environmental regulations and of insurance and finance.

However, the near future will show how well the integration of DT's and Prolog works in various knowledge domains. Many questions concerning the functional combination of AKTS and the people using it, need to be answered. Is AKTS really adequately equipped to model knowledge of the heterogeneous outside world? Does it have the right validation facilities? Do people, working with AKTS, have the discipline and the abilities to reconstruct DT's and to apply Prolog in the right way; can they bridge the gap between the declarative and procedural semantics of Prolog? To what extent are people capable of modelling their own knowledge using AKTS.

The first signs are positive. However, experience points out that modelling knowledge still requires a theory of the nature of knowledge and explicit experience and capacities for modelling knowledge. So, AKTS does not make knowledge-engineers superfluous, but facilitates and improves their activities. We think that AKTS is a step forward in the development of knowledge-based systems. It will not surprise the reader that extensions on AKTS are intended to make it yet more appropriate to serve its purpose: improving the process of modelling knowledge to develop (complex) knowledge-based systems that will become vital tools on any company's shop floor.
CHAPTER 7

CHEMICAL DEGRADATION AND RESTORATION OF ANCIENT BRICK MASONRY WALLS

7.1 INTRODUCTION

To substantiate the central argumentation of the thesis we discuss a case-study. The subject of it originates from the field of chemical degradation and restoration of brick masonry walls. Specifically, the case-study deals with sulphate salt reactions causing the formation of ettringite, a specific form of chemical degradation. We do not aim at deepening our understanding of this field or to present a laborious case. Rather, this chapter attempts to exemplify the three main methodological points at issue in this thesis that deal with the problem of modelling knowledge: (1) the value of a knowledge level integration of AI and DBT, (2) the value of the theory of functional classifications to accomplish such an integration and (3) the value of applying DT's and Prolog in the form AKTS makes them available as a modelling language for functional object-types.

We first provide an overview of the construction materials that play a role in the chemical degradation and restoration of brick masonry walls (Section 7.2). To attain a knowledge level integration of AI and DBT, we reconstruct implementation-free descriptions of object-types (delineating walls susceptible to the formation of ettringite) and objects (referring to brick masonry walls). These descriptions, based on the theory of functional classifications, are represented by means of DT's (and Prolog in 7.4) using AKTS (Section 7.3). Then, we descend to the symbol level by transferring the reconstructed object-types and objects to representation formalisms of AI and DBT. To prevent a bias toward a specific representation formalism, we use mathematical logic to assess the technical design of each of these representation formalism (Section 7.4). We conclude the chapter by going through the three methodological points again (Section 7.5).

7.2 GENERAL DESCRIPTION

The case-study is derived from the EC Environment-project entitled Expert System for Evaluation of Deterioration of Ancient Brick Masonry Structures (1992-1995). The main goal of this project is to improve the availability and accessibility of knowledge of ancient brick masonry structures by the development of a knowledge-based system. This system should contain knowledge with regard to (a) the determination of types of
damage in the masonry walls of historical buildings and (b) the determination of (environmental) causes of these damages.

The original construction materials play an essential role in determining the types of damages due to chemical degradation. The construction materials originally used in the masonry walls of historical buildings are (1) mortar composed of binding materials and sands and (2) bricks and stones to be bound.

**Binding materials and sands**

Gypsum (\(\text{CaSO}_4 \cdot 2\text{H}_2\text{O}\)) is a binding material that is able to harden in the air when mixed with water and sand. Because it can be washed away by rain water, gypsum is used in the form of rendering mortar, mostly at the interim of a wall. It is also used in the form of jointing mortar to bind bricks and stones. Gypsum has a low strength, but is easy to produce.

A binding material that is more difficult to produce, but possesses a higher strength, is lime (\(\text{Ca(OH)}_2\)). Lime hardens in the air when mixed with water. It also hardens under water: lime has the property of becoming hydraulic when it is mixed with water and natural or artificial pozzolan or pozzolanic sand. That is, it is able to react with the pozzolanic material in the presence of water to form a cementitious binder. This is called the pozzolanic activity of pozzolan sand. Lime that of its own has the ability to harden under water independently of the presence of pozzolan sand is called hydraulic lime.

**Figure 7.1: Jointing and Rendering Mortar in a Brick Masonry Wall**

**Bricks and stones**

Bricks are obtained by firing clay previously shaped and dried. Bricks may contain salts (e.g. sulphates) that are soluble in water. Water can penetrate the bricks, dissolve salts and on evaporation, deposit the salts on the brick surface. Salty efflorescences may occur if the brick is exposed to view and this may lead to the disjunction of mortar. An even more serious deterioration may occur when the brick is coated with
rendering mortar that is able to react with the salts.

Stones do not show special problems as to the interaction with the binders originally used in the construction of historical buildings apart from the alkali-aggregate reaction between the alkalis of the binder and some minerals reactively present in the stones (Collepardi, 1990, p.84).

In masonry works (bricks and mortars) three types of degradation may occur: (1) the reaction between alkalis supplied by the bricks and the reactive aggregate of the mortar, (2) the reaction between the sulphates and the hydrated calcium silicates (production of thaumasite) and (3) the reaction between the sulphates and hydrated aluminates of the hydraulic limes. The latter reaction refers to the formation of ettringite.

### 7.3 FUNCTIONAL OBJECT-TYPES (KNOWLEDGE LEVEL ANALYSIS)

An infinite number of attributes or descriptors of brick masonry walls are potentially relevant for detecting degradation of and restoring masonry walls. Using these attributes numerous classifications of masonry walls are possible. As stated in the previous chapters, the process of forming meaningful classifications is a difficult task which requires a theory underlying classification construction. We described the theory of functional classifications as a theory for classification construction. A central point of departure for the theory, is the assumption of a goal or function that acts as a guiding principle to identify relevant attributes and to create meaningful, functional classifications. For example, the goal 'find masonry wall that is susceptible to chemical degradation' helps to classify masonry walls according to all the possible combinations of the available original materials of historic buildings: lime, gypsum, hydraulic lime, stone, brick, etc. On the basis of these attributes, Collepardi (1990, p.85) distinguishes 50 types of masonry walls; 20 of them are brick walls.

Figure 7.2 is a tree that for reasons of economy is represented in a nested Prolog list. The list is based on the S(string)-expression notation (Lew, 1985, pp.181-184) and represents 20 types of brick masonry walls. The codes in the terminal nodes of the tree are the types of walls. The code $Bmgl'$, for instance, denotes a brick masonry wall ($B$) that is coated with rendering mortar ($m$) and that contains gypsum in the inside binder ($g$) and lime and pozzolanic sand in the outside binder($l'$), while the code $Bmlg$ denotes a brick masonry wall ($B$) that is coated with rendering mortar ($m$) and that contains lime in the inside binder ($l$) and gypsum in the outside binder($g$). Figure 7.2 shows that 5 dichotomous attributes are used in the classification. In theory, this leads to $2^5$ types of brick masonry walls. However, if a wall has jointing mortar and thus is exposed to view, no outside binder is present. In this case, the materials of the outside binder are 'impossible' attributes. So the actual classification only yields 20 types of brick masonry walls.

On the one hand, the proposed classification of Figure 7.2 seems to be of a completely functional nature. The classification criteria are derived from the materials used and seem to be directly related to the chemical degradation process. On the other hand, the classification does not show any form of flexibility and lacks functional
equivalences. To evaluate the functional nature of the classification we focus on brick masonry walls susceptible to chemical degradation. One of the main chemical reactions that cause masonry degradation is the formation of ettringite. Therefore, we concern ourselves with the reconstruction of the object-type Brick masonry walls susceptible to the formation of ettringite. For the time being, we describe the context as the circumstances under which no restoration has (yet) taken place. This context indicates under what conditions the object-type is valid.

<table>
<thead>
<tr>
<th>Types of brick masonry wall,</th>
<th>Jointing mortar,</th>
<th>Gypsum inside binder,</th>
<th>Lime inside binder,</th>
<th>Rendering mortar,</th>
<th>Gypsum inside binder,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brick masonry walls</td>
<td>Mortar</td>
<td>Ordinary sand, [Beg]</td>
<td>Pozzolanic sand, [Beg']</td>
<td>Lime inside binder,</td>
<td>Ordinary sand, [Bel]</td>
</tr>
<tr>
<td>susceptible to the formation</td>
<td>Mortar</td>
<td></td>
<td></td>
<td>Lime inside binder,</td>
<td>Ordinary sand, [Bel']</td>
</tr>
<tr>
<td>of ettringite</td>
<td>Mortar</td>
<td>Gypsum in outside binder,</td>
<td>Pozzolanic sand, [Bmg]</td>
<td>Lime inside binder,</td>
<td>Gypsum in outside binder,</td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td></td>
<td></td>
<td>Lime inside binder,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td>Lime in outside binder,</td>
<td>Pozzolanic sand, [Bmg']</td>
<td>Lime inside binder,</td>
<td>Gypsum in outside binder,</td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td></td>
<td></td>
<td>Lime inside binder,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td>Lime in outside binder,</td>
<td>Pozzolanic sand, [Bmg'g]</td>
<td>Lime inside binder,</td>
<td>Gypsum in outside binder,</td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td></td>
<td></td>
<td>Lime inside binder,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td>Lime in outside binder,</td>
<td>Pozzolanic sand, [Bmg'g']</td>
<td>Lime inside binder,</td>
<td>Gypsum in outside binder,</td>
</tr>
<tr>
<td></td>
<td>Mortar</td>
<td></td>
<td></td>
<td>Lime inside binder,</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.2: A Classification of Brick Masonry Walls

The object-type Brick masonry walls susceptible to the formation of ettringite is pictured in the DT of Figure 7.3. Rule 1 is a conjunct set containing three abstract...
INUS-conditions that play a role in the production of ettringite: (1) gypsum, (2) hydrated calcium aluminates and (3) moisture. Each condition on its own is insufficient for the formation of ettringite, but within the conjunction each is indispensable. The chemical reaction underlying this conjunct set is (Collepardi 1990, p.89):

$$3(CaSO_4 \cdot 2H_2O) + 3CaO \cdot AL_2O_3 \cdot 6H_2O + 20H_2O \rightarrow 3CaO \cdot AL_2O_3 \cdot 3CaSO_4 \cdot 32H_2O$$

gypsum hydrated calcium aluminates moisture ettringite

**Figure 7.3: The Object-type 'Brick Masonry Wall Susceptible to the Formation of Ettringite'

<table>
<thead>
<tr>
<th>C1</th>
<th>gypsum</th>
<th>present</th>
<th>not present</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>hydrated calcium aluminates</td>
<td>present</td>
<td>not present</td>
</tr>
<tr>
<td>C3</td>
<td>moisture</td>
<td>present</td>
<td>not present</td>
</tr>
<tr>
<td>A1</td>
<td>Brick masonry walls susceptible to the formation of ettringite</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

R1 R2 R3 R4

**Figure 7.4: Gypsum as a Binder or as a Product of a Chemical Process

**Ad 1. Gypsum**

The DT of Figure 7.4 shows that gypsum may be present in a wall as a binder (C1) or as a product of a chemical process (C2). It is a condition subtable related to the DT of Figure 7.3. Viewing each of these DT's as a set of knowledge elements (see Chapter 3), the formal link between the condition subtable and the head table is the following subset-requirement:

$id(gypsum)$ connects $KS(gypsum)$ with $KS(brick masonry walls susceptible to the formation of ettringite)$

The DT of Figure 7.5 describes the presence of gypsum as a binder. The DT displays
the knowledge that if the type of mortar is rendering, gypsum may be present in the
inside or in the outside binder. If, on the contrary, the type of mortar is jointing
mortar, no outside binder is present. In this case, the availability of gypsum depends
solely on its presence in the inside binder. The formal link between this table and the
table of Figure 7.4 is the following subset-requirement:

\[ id\{\text{gypsum as binder}\} \text{ connects } KS(\text{gypsum as binder}) \text{ with } KS(\text{gypsum}) \]

<table>
<thead>
<tr>
<th>Gypsum as binder</th>
<th>jointing mortar</th>
<th>rendering mortar</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 type of mortar</td>
<td>gypsum</td>
<td>gypsum</td>
</tr>
<tr>
<td>C2 type of inside binder</td>
<td>lime OR hydraulic lime</td>
<td>gypsum</td>
</tr>
<tr>
<td>C3 type of outside binder</td>
<td>gypsum</td>
<td>lime OR hydraulic lime</td>
</tr>
<tr>
<td>A1 gypsum as binder</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

**Figure 7.5: Gypsum As Binder**

<table>
<thead>
<tr>
<th>Gypsum as an in situ product of a chemical reaction</th>
<th>sulphates</th>
<th>chlorides OR nitrates OR carbonates</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 type of salts</td>
<td>lime OR hydraulic lime</td>
<td>gypsum</td>
</tr>
<tr>
<td>C2 type of binder</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>C3 moisture</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>A1 gypsum as an in situ product of a chemical reaction</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

**Figure 7.6: Gypsum as an In Situ Product of a Chemical Reaction**

The DT of Figure 7.6 reflects the following chemical reaction (Collepardi 1990, p.89):

\[ \text{SO}_4^{2-} + \text{Ca(OH)}_2 + 2\text{H}_2\text{O} \rightarrow \text{CaSO}_4 \cdot 2\text{H}_2\text{O} + 2\text{OH}^- \]

The sulphate may originate from bricks which have been contaminated with pyrite
(FeS₂) during the manufacturing process. Under specific circumstances pyrite can
change into sulphate. Sea water can also be a source of sulphate. These two sources
are not incorporated in DT's, but are shown in Figure 7.8. The formal link between the
table of Figure 7.6 and the table of Figure 7.4 is the following subset-requirement:
id{gypsum as an in situ product of a chemical reaction}
connects KS(gypsum as an in situ product of a chemical reaction) with KS(gypsum)

Ad 2. Hydrated Calcium Aluminates
The second INUS-condition (Figure 7.3), hydrated calcium aluminates, is itself described by two other conjunct sets as Figure 7.7 displays: (1) the presence of hydraulic lime and (2) the presence of normal lime in conjunction with (natural or artificial) pozzolanic sands from the jointing or rendering mortar. The formal link between this table and the table of Figure 7.3 is the following subset-requirement:

id{hydrated calcium aluminates} connects KS(hydrated calcium aluminates) with KS(brick masonry wall susceptible to the formation of ettringite)

Natural pozzolan or artificial pozzolan (pounded earthenware) is capable of changing an air-hardening lime into a lime with hydraulic properties. In gypsum-based mortars, the difference between normal or pozzolanic sand is unimportant as pozzolan can only react with lime. However, the pozzolan in a gypsum mortar can react with the lime or hydraulic lime of adjacent mortars.

<table>
<thead>
<tr>
<th>Hydrated calcium aluminates</th>
<th>C1 type of binder</th>
<th>C2 type of sand</th>
<th>A1 hydrated calcium aluminates</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 type of binder</td>
<td>hydraulic lime</td>
<td>lime</td>
<td>gypsum</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2 type of sand</td>
<td>-</td>
<td>natural pozzolan OR artificial pozzolan</td>
<td>normal</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1 hydrated calcium aluminates</td>
<td>present</td>
<td>present</td>
<td>not present</td>
<td>not present</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.7: Hydrated Calcium Aluminates

Ad 3. Moisture
The third INUS-condition that plays a basic role in the chemical reaction that produces ettringite, is moisture. Moisture is not only involved in the chemical interaction under discussion, but it also exerts the function of carrying one component of a wall towards another component by means of which (other) chemical reactions can take place. Capillary water, for instance, can take up salts and put them in contact with a component of rendering mortar. Generally, historical buildings are only subject to significant deterioration in the presence of moisture. Moisture can originate from 'wetting' rain or from the capillary flow of water.

Figure 7.8 displays an overview of the structure of the object-type Brick masonry walls susceptible to the formation of ettringite represented by means of a system of DT's. The reader should note that the identification of the object-type leads to a
reclassification of the objects originally presented in Figure 7.2. In principle two classes of objects can be distinguished: walls that deteriorate due to the formation of ettringite and walls that do not. Our object-type shows that every wall of the tree that somehow manages to match the conjunct set at abstraction level II (Figure 7.8) is an instance of the object-type and thus belongs to the class of objects that is susceptible to the formation of ettringite.

These walls may differ a great deal. Some walls contain gypsum in the inside binder, some contain gypsum in the outside binder. Other walls do not contain gypsum, but contain lime in the inside binder or outside binder. In contrast, some walls contain hydraulic lime, whereas others contain lime in their components combined with pozzolanic sand. Whatever differences these walls may show, all of them are functionally equivalent, because they somehow comply with the constraints of the reconstructed object-type.

Let us assume that the INUS-condition of moisture is met. Then, a moist wall complies with the conjunct set at level II if gypsum is directly present in the inside binder and hydraulic lime is applied in the outside binder. The conjunct set of hydraulic lime is replaceable by the combination of pozzolanic sand in the inside binder and lime in the outside binder as is clearly shown in Figure 7.7. As the differences between $g^l$ and $g^l$ are not functional, we can abstract from them and reconstruct the following set of functionally equivalent walls:

\[ FeC_1 = \{Bmg^l, Bmg^l\} \]

---

1 FeC stands for Functional Equivalence Class.
An alternative way of realising a match with the conjunct set at level II is by means of the direct presence of gypsum in the outside binder walls if hydraulic lime is applied. As hydraulic lime is replaceable by the combination of lime and pozzolanic sand, the following set of walls therefore also matches the conjunct set at level II:

\[ FeC_2 = \{Bmlg', Bml'g', Bml'g\} \]

A third way for a wall to meet the conjunct set at level II is shown by Rule 2 of the DT of Figure 7.4. It reveals that the absence of gypsum in the inside or outside binder can be compensated by the presence of gypsum as an in situ product of a chemical reaction. Rule 1 in the table of Figure 7.6 shows that the conjunct set: sulphates, lime and moisture, leads to the formation of gypsum as an in situ product. Now, another attribute is necessary: sulphate in bricks. Note that this attribute is lacking in the tree of Figure 7.2. To provide hydrated calcium aluminates the lime should be combined with pozzolanic sand or it should be hydraulic itself. The following walls do not possess gypsum as a binder, but may deteriorate all the same when gypsum is formed through sulphate interacting with lime and moisture:

\[ FeC_3 = \{Bel', Bml'l', Bml'l, Bml'l'\} \]

Many other functional classifications can be created. We can, for instance, use the criteria lime in inside binder or lime in outside binder, pozzolanic sand in inside binder or in outside binder etc. The only relevant criterium, however, is the question whether a wall complies with the conjunct set at level II. Later, we will see that the precise mechanisms behind functional equivalence should also be known.

The complete class of functional equivalent objects that is susceptible to chemical degradation can be computed using the generalised union:

\[ \bigcup\{FeC_1, FeC_2, FeC_3\} = \{Bmlg', Bml'g, Bmlg', Bml'g', Bel', Bml'l', Bml'l, Bml'l'\} \]
Though we recognise the utility of this object-type, we must not forget to point out, as we did in the previous chapters, the relative validity of the object-type. This can be illustrated by incorporating several types of restoration material in the object-type. This addition changes the meaning of our initial object-type. Knowledge about the mechanisms behind functional equivalence becomes important to be able to describe the interaction between the original materials and the restoration materials. The condition-alternatives of the first condition of the DT of Figure 7.9 display the following types of restorations: reparation by hydraulic binders (cement, hydraulic lime, lime-pozzolan), reparation by non-hydraulic mortars (lime and ordinary sand) and reparation by non-hydraulic mortars (gypsum). The DT illustrates what attributes of walls (C_2 up to C_4) may lead to deterioration if a certain type of restoration is carried out (C_1). Conceptual interactions occur between C_1 and C_2. The type of binder is conditionally classified: gypsum, lime OR hydraulic lime versus gypsum, lime, hydraulic lime!

Other objects are instances of the object-type Brick masonry walls susceptible to the formation of ettringite. New functional classifications of objects arise. If restoration is carried out through hydraulic binders, all walls containing gypsum may degrade:

$$FeC_4 = \{Beg, Beg', Bmgg, Bmg'g', Bmgl, Bmg'l, Bmgl', Bmg'g, Bmgl', Bmg'g'\}$$

If restoration is carried out by applying non-hydraulic mortars (lime and ordinary sand), all walls containing gypsum and pozzolanic sand can deteriorate:

$$FeC_5 = \{Beg', Bmgl, Bmgl', Bmgl', Bmg'l, Bmgl', Bmg'g'\}$$

<table>
<thead>
<tr>
<th>C1 restoration material</th>
<th>cement OR hydraulic lime OR lime-pozzolan</th>
<th>lime AND ordinary sand</th>
<th>gypsum</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 type of binder</td>
<td>gypsum</td>
<td>lime OR hydraulic lime</td>
<td>gypsum</td>
</tr>
<tr>
<td>C3 type of sand</td>
<td>gypsum</td>
<td>lime OR hydraulic lime</td>
<td>gypsum</td>
</tr>
<tr>
<td>C4 moisture</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>A1 Restorated brick masonry wall susceptible to formation of ettringite</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 7.9: A Shift to Another Context
If restoration is carried out by applying non-hydraulic mortars (gypsum), all walls containing pozzolanic sand and (hydraulic) lime may deteriorate. In this situation, the difference between lime and hydraulic lime is not relevant anymore, so we obtain the following class of functionally equivalent objects:

\[ FeC_0 = \{Bel', Bmg'l', Bmg'l, Bmgll, Bml'g, Bml'l, Bml'l'\} \]

The reader should note that a functionally reconstructed object-type underlines the need for flexible classifications. Consequently, *multiple classifications* frequently occur. For instance, a brick masonry wall that is exposed to view with lime and pozzolanic sand in the inside binder \((Bel')\) is an instance of the first object-type and simultaneously an instance of the second object-type.

\[ \text{Object-type 1} \quad \text{Object-type 2} \]
\[ \text{instance of} \quad \text{instance of} \]
\[ Bel' \quad \text{instance of} \quad Beg \]

\[ \text{Figure 7.10: Multiple Classifications} \]

Since the materials employed in the restoration work may interact negatively with the original components of a wall, the differences between the two object-types should be clearly understood. Collepardi (1990, p.92) points out the implications of the absence of this knowledge.

'It frequently occurs that, after the first period of apparent improvement due to the restoration and consolidation of masonries, historical buildings deteriorate even more severely than before the intervention.'

This quotation underlines the importance of knowing object-types. The object-types show that similarity of objects depends on other conditions that should be known for effective classifications. Knowing these classifications implies knowing the object-types. The object-types described mainly serve illustration purposes. Though not trivial at this moment, the object-type will become more complex and additional object-types are needed. Not only are there more forms of chemical degradation and more chemical products, such as thaumasite, that lead to deterioration, but there are also other causes of degradation, for instance degradation due to physical causes. There are also more types of elements that should be bound such as stones. Finally, we can also distinguish other functions. For instance, we can reconstruct object-types...
using the 'prevention of leaking' or 'washing away through rainwater' as modelling purposes. Though an intellectually difficult and time-consuming process, the reconstruction of functional object-types is a worthwhile and necessary activity as a basic step toward the implementation of complex knowledge-based systems.

What do we gain from this knowledge level analysis? At any rate, it teaches us that the reconstructed object-types are conducive to more functional classifications of the walls. These classifications are conditional reclassifications of the classifications originally proposed. The knowledge specified in the object-type not only assesses the conditional relevance of attributes of brick masonry walls, but also identifies conceptual interactions that occur between attributes. Therefore, this knowledge is ideal for damage diagnosis purposes especially for the detection of chemical reactions leading to degradation. In addition, the classifications are needed to be able to determine whether specific restoration materials will interact negatively with original materials.

The implication of the functional view for neural networks is another thing that requires attention. In Chapter 4 we critically discussed the competence of neural networks from a functional perspective. From what we stated there and from the contents of this chapter it is highly questionable whether a neural network is capable of reconstructing functional classifications. Within the framework of this chapter we could ask: How can a neural network learn from the 'different' objects $Smg/l$ versus $Bel$?

7.4 KNOWLEDGE-BASED SYSTEMS (SYMBOL LEVEL ANALYSIS)

In the previous section we came across two types of knowledge: (1) knowledge of constraints and (2) knowledge of objects. Knowledge of constraints is contained in an object-type. The constraints must be matched by objects in order to be instances of an object-type. Knowledge of objects refers to the attributes of objects that need to be known for this matching procedure. For instance, the object-type *Brick masonry wall susceptible to the formation of ettringite* is a set of constraints that must be matched by a brick masonry wall to be an instance of the object-type. If so, the object or wall at issue is susceptible to the formation of ettringite.

Conventionally, object-types are perceived as knowledge and knowledge of objects as data. Therefore, many computer scientists associate object-types with AI-representation formalisms and knowledge of objects with representation formalisms of database technology. At the symbol level, they consider AI-formalisms -and the systems built around these formalisms- appropriate for describing and operating on object-types, whereas database formalisms and systems are considered effective to represent objects. Consequently, AI- and DB-systems have traditionally been applied to problems that are viewed as being different. Applying this view to the field of chemical degradation would mean that the object-types are represented in an expert system and the objects (the data) in some kind of database system.

Roughly speaking, AI-formalisms are expected to represent object-types by means of universal quantified statements:
∀x(F(x) ⊃ G(x))

A universal quantified statement defining an object-type in the field of chemical degradation might be represented in a Prolog program clause.

Example 1

```prolog
susceptible_to(wall(identifier(X)), ettringite) :-
    contains(wall(identifier(X)), gypsum),
    contains(wall(identifier(X)), hydrated Calcium aluminates),
    contains(wall(identifier(X)), moisture).
```

The Prolog program clause is a universally quantified one statement definition of the object-type walls susceptible to the formation of ettringite. The head is the conclusion that a certain wall is indeed susceptible to the formation of ettringite. The body represent the constraints. The clause depicts the knowledge that every wall with gypsum, hydrated calcium aluminates and moisture, is susceptible to the formation of ettringite. It shows the kind of knowledge that AI-formalisms are presumed to contain: intensional knowledge at a certain level of abstraction.

DB-systems, as opposed to AI-systems, are expected to contain knowledge of objects at a lower level of abstraction as expressed by the following predicate assertion:

```
F(x)
```

Expressed in Prolog program clauses with empty bodies, such a ground assertion regarding the susceptibility to ettringite of two walls might look like:

Example 2

```prolog
susceptible_to(wall(identifier(1)), ettringite).
contains(wall(identifier(2)), gypsum).
contains(wall(identifier(2)), hydrated Calcium aluminates).
contains(wall(identifier(2)), moisture).
```

The Prolog clauses tell us that the object wall (identified by 1) is susceptible to the formation of ettringite and that the object wall (identified by 2) contains gypsum, hydrated calcium aluminates and moisture. Generally, Prolog can be perceived as a database language (Brodie & Jarke, 1986; Li, 1985; Rowe, 1988). The table of Figure 7.11 describes the analogy between Prolog and Database concepts.

DB-systems are said to represent a collection of data representing objects. In general, the objects represented by the data are assumed to be of the 'same' kind. This view, distinguishing knowledge from data, often goes hand in hand with an explicit focus on representational issues. This is not strange, because only at the symbol level, as stated in a previous chapter, can differences be seen. This view forms the basis for much research studying the relationships between AI-systems and DB-systems (see for instance Wiederhold 1984; Murdoch and Johnson 1990). A logical consequence of
this view is that knowledge and data are stored separately in different systems. While this is perhaps a sensible point from which to start considering how AI-formalisms and systems relate to database formalisms and systems, our contention is that these differences are rather shallow. The traditional distinction between universal quantified statements and ground atomic assertions may be relevant, but it is not a matter of universal quantified statements being knowledge and atomic assertions being data (Brachman & Levesque, 1986, p.77). Obviously these types of knowledge are closely intertwined. Both are legitimate forms of knowledge necessary to classify or recognise objects as instances of reconstructed object-types. The falsehood of the assumption that data is by definition homogeneous is another indication that the difference between knowledge and data is rather shallow. The various functional equivalence classes of the previous section showed that relevant attributes may significantly vary (Figure 7.12) and may even be inferred one from the other. This implies that complex abstraction mechanisms are also needed for the representation of data! In the Prolog clauses representing walls also the clause of the first example is needed to perform a match of attributes of objects to constraints (object-types)!

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground Clause (Fact)</td>
<td>Base Relation Tuple</td>
</tr>
<tr>
<td>Predicate Argument</td>
<td>Attribute</td>
</tr>
<tr>
<td>Horn Clause</td>
<td>View Definition</td>
</tr>
<tr>
<td>Theorem</td>
<td>Query</td>
</tr>
<tr>
<td>Hypothesis</td>
<td>Program</td>
</tr>
<tr>
<td>Assert/Retract</td>
<td>Insert/Delete</td>
</tr>
<tr>
<td>Query (Returning True)</td>
<td>Assertion (Constraint)</td>
</tr>
<tr>
<td>Predicate</td>
<td>Trigger</td>
</tr>
<tr>
<td>Set Of Facts To Prove A Theorem</td>
<td>Set Valued Query Result</td>
</tr>
</tbody>
</table>

(Source: Brodie & Jarke, 1986, p.198)

**Figure 7.11: The Analogy between Prolog and Database Concepts**

Knowledge is principally a complex form of object-type or concept matching (Goel, Soundararajan, & Chandrasekaran, 1987). For this reason, descriptions of object-types and descriptions of objects should explicitly be present in any knowledge-based system. The descriptions influence each other. If an object-type is described in a detailed way, the description of an object may be kept simple. The other way round, if an object is extensively described by means of universal quantified statements and by means of ground atomic assertions, the description of the object-type becomes easy. It is not a coincidence that AI and DBT are increasingly working on similar topics, as is manifested, for instance, by the close relationship between semantic data modelling and research on knowledge representation (Hull & King, 1987, p.212). Several approaches can be distinguished. One approach stresses the complex interrelationships between the attributes of objects; the other emphasises the
representation of object-types. Within AI as well as within DBT both approaches are found. For instance, within AI the frame-oriented community is an exponent of the first orientation and the rule-based community of the second.

![Diagram](image)

**Figure 7.12: A Goal-oriented Object-type and Functional Equivalence**

### 7.4.1 Knowledge of Objects

Following an orientation toward attributes of objects, a computer scientist is focused on representing the complex relationships between objects by means of their attributes. Often, the relational data model is used for these purposes. The relational model for formatted databases was conceived 25 years ago (Codd, 1970), primarily as a tool to free users of having to deal with the clutter of storage representation details (Codd, 1979, p.397). Even in semantic data modelling, which aims at capturing more of the meaning of the data, while preserving independence of implementation, conceptual data models are reduced to relational schemes and, in the end, implemented in record-based representation formalisms. Because of its widespread use in attribute-oriented approaches, we will use the relational data model and the record-based representation formalism to represent objects and their attributes.

As indicated by the first object-type, at the highest level three attributes of the object wall should be incorporated to be able to determine the susceptibility of a wall to ettringite: gypsum, hydrated calcium aluminates and moisture. These attributes are necessary to see whether a certain wall matches the conjunct set at level II of the object-type brick masonry wall susceptible to the formation of ettringite when no restoration has been carried out yet. The reader should note that this knowledge is absent in the original tree-structure of Figure 7.2 presenting a number of attributes of the walls. The first object-type clarifies the meaning of (the interdependence of) the attributes! Apart from these main attributes, another attribute that serves as an unique identifier for each wall is required. The ordered pair \( W \) represents the object wall.
(π₁(W)) and the associated attributes (π₂(W)).

\[ W = \{\text{wall - identifier, gypsum, hydrated calcium aluminates, moisture, susceptible to the formation of ettringite}\} \]

\[ \pi_1(W) = \text{wall} \]

\[ \pi_2(W) = \{\text{wall - identifier, gypsum, hydrated calcium aluminates, moisture, susceptible to the formation of ettringite}\} \]

A number of objects stored according to the ordered pair in a single relational table are presented below. As indicated by the first object-type, the following functional dependency must be present:

\[ \{\text{gypsum, hydrated calcium aluminates, moisture}\} \rightarrow \{\text{susceptible to the formation of ettringite}\} \]

<table>
<thead>
<tr>
<th>Wall-Identifier</th>
<th>Gypsum</th>
<th>Hydrated Calcium Aluminates</th>
<th>Moisture</th>
<th>Susceptible To The Formation Of Ettringite</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>present</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>present</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>present</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>no</td>
<td>not present</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

*Figure 7.13: A Table in a Relational Model*

For our first object-type this description will do. However, in at least two situations additional knowledge is needed and the utility of this initial description is questionable. The first situation refers to a user of our knowledge-based system who has to fill the system with objects and who may not know whether a concrete wall contains gypsum or hydrated calcium aluminates or moisture. In attempting to arrive at values for these attributes, the user has to get a grip on the functional mechanisms that conceptually operate behind the values: the user should know the conjunct sets that underlie and determine the values. The conjunct sets at level III can help here to fill in the right values for the object-attributes.

The second situation refers to restoration purposes as described in the second object-type (Figure 7.9). Masonry can be repaired by hydraulic binders such as cement, hydraulic lime or lime-pozzolan. As we still need to know whether a wall contains gypsum, the original description in the table is useful. However, if non-hydraulic binders (such as lime or ordinary sand) are applied for the restoration work, it does not suffice for a knowledge-based system to know whether a wall contains gypsum. In this particular situation, the system should not only know whether a wall...
contains gypsum, but also whether a wall is composed of pozzolanic sands or not. This flexibility is made clear by the object-type reconstructed: on the condition that enough moisture is present, ettringite can be formed and lead to the deterioration of the wall after all. Another example of an unexpected degradation occurs when gypsum is used for restoration purposes. Then, a masonry wall may not be composed of lime-pozzolan or hydraulic-lime mortars! Analogous to the first situation, a knowledge-based system should have explicit knowledge of the mechanisms behind chemical degradation due to the formation of ettringite.

Considering both object-types, the knowledge schema (in attribute-oriented approach often called database scheme) of the wall in our example might have this form:

The Knowledge Schema

\[
\begin{align*}
BMWSE = \\
\{ & \{ \text{wall identifier, gypsum, hydrated calcium aluminates, moisture, susceptible to the formation of ettringite} \} , \\
& \{ \text{wall identifier, gypsum as binder, gypsum as an in situ product of a chemical reaction, gypsum} \} , \\
& \{ \text{wall identifier, gypsum as binder} \} , \\
& \{ \text{wall identifier, type of mortar, type of inside binder} \} , \\
& \{ \text{wall identifier, type of outside binder} \} , \\
& \{ \text{wall identifier, type of salts, type of binder, moisture, gypsum as an in situ product of a chemical reaction} \} , \\
& \{ \text{wall identifier, restored} \} , \\
& \{ \text{wall identifier, restoration materials} \} \}
\end{align*}
\]

The scheme resembles the structure of the DT's. That is, the name of a DT corresponds with a domain element of the scheme and the set of conditions and actions of a DT correspond with the range of the domain element. There are, however, some deviations from the structure of the DT's. The scheme shows that the knowledge of the DT \text{gypsum as binder} is spread over three tables. The table \text{gypsum as binder} contains the general conclusion whether a wall has gypsum as a binder. The tables \text{inside binder} and \text{outside binder} respectively deal with the inside and the outside binder of a wall. The distinction is motivated by the fact that only walls with
rendering mortar have an inside and an outside binder, whereas a wall with jointing mortar only has an inside binder. A design in which both objects are modelled as one object, yields null-values (in the meaning of not applicable) for the type of outside binder fields for every wall which has only jointing mortar. A second deviation is displayed by the tables restored and restoration materials. They serve to store knowledge of walls that have been subjected to restoration. As much of the knowledge regarding restored walls, is assessed through connections with other parts of the scheme, several conditions and actions occurring in the associated DT are left out from the scheme.

In database theory a scheme like this one is called an object-type. This seems misleading, but should be interpreted as just another indication that object-types and objects are difficult to distinguish.

**Variable Constraints (VC)**

The following auxiliary function is needed to consider the binders as an aggregation of binding material and sands (see the functions vcinside binder and vcoutside binder). The aggregation is represented by means of nestings. Not all DB-systems can cope with such nestings.

\[
F1 =
\{
(binding materials ; \{lime, hydraulic lime, gypsum\}),
(type of sand ; \{normal, pozzolan\})
\}
\]

The variable constraints are described in the following set-valued functions

\[
vcewall =
\{
(wall - identifier ; [1, ..., 10^4]),
(gypsum ; \{present, not present\}),
(hydrated calcium aluminates ; \{present, not present\}),
(moisture ; \{present, not present\}),
(susceptible to the formation of ettringite ; \{yes, no\})
\}
\]

\[
vcegypsum =
\{
(wall - identifier ; [1, ..., 10^4]),
(gypsum as binder ; \{yes, no\}),
(gypsum as an in situ
\]
product of a chemical reaction = {
(gypsum as binder = {
(wall - identifier = [1, ..., 10^4]),
(gypsum as binder = {yes, no})
}
}

vcinside binder = {
(wall - identifier = [1, ..., 10^4]),
(type of mortar = {jointing mortar, rendering mortar}),
(type of inside binder = \( \pi(F1) \))
}

vcoutside binder = {
(wall - identifier = [1, ..., 10^4]),
(type of outside binder = \( \pi(F1) \))
}

vgypsum as an in situ product of a chemical reaction = {
(wall - identifier = [1, ..., 10^4]),
(type of salts = {sulphates, chlorides, nitrates, carbonates}),
(type of binder = {gypsum, lime, hydraulic lime}),
(moisture = {yes, no}),
(gypsum as an in situ product of a chemical reaction = {present, not present})
}
vrestored =
{
(wall - identifier ; [1, ..., 10^4]),
(restored ; {yes, no})
}

vrestoration material =
{
(wall - identifier ; [1, ..., 10^4]),
(restoration material ; {cement, hydraulic lime, lime - pozzolan, lime - ordinary sand, gypsum})
}

**Inter-variable Constraints (IVC)**
The inter-variable constraints are defined using the \( \Pi \) that operates upon the variable constraints:

\text{ivcwall} = \{ t \in \Pi (vcwall) | \\
    \text{if: } t(\text{gypsum}) = \text{present and } \\
    t(\text{hydrated calcium aluminates}) = \text{present and } \\
    t(\text{moisture}) = \text{present} \\
    \text{then: } t(\text{susceptible to the formation of ettringite}) = \text{yes} \\
    \text{else: } t(\text{susceptible to the formation of ettringite}) = \text{no} \}

Furthermore:

\text{ivcgypsum} = \{ t \in \Pi (vcgypsum) | \\
    \text{if:} \\
    t(\text{gypsum as an in situ product of a chemical reaction}) = \text{present or} \\
    t(\text{gypsum as binder}) = \text{yes} \\
    \text{then: } t(\text{gypsum}) = \text{present and} \\
    \text{else: } t(\text{gypsum}) = \text{no} \}

ivcgypsum as binder = \{ t \in \Pi (vcgypsum as binder) \}

ivcinside binder = \{ t \in \Pi (vcinside binder) \}
ivcoutside binder = \{ t \in \Pi(ivcoutside binder) \}

ivcgypsum as an in situ product of a chemical reaction =
\{
    t \in \Pi(ivcgypsum as an in situ product of a chemical reaction)
\}

ivrestored = \{ t \in \Pi(ivrestored) \}

ivrestoration material = \{ t \in \Pi(ivrestoration material) \}

Knowledge Table Constraints (KTC)
The knowledge table constraints assess the space of allowed tables for each object:

ktewall = \{ T \subseteq ivcwall | \{wall - identifier\} is unique identifying in T \}

ktegypsum = \{ T \subseteq ivcgypsum | \{wall - identifier\} is unique identifying in T \}

ktcgypsum as binder = \{ t \subseteq \Pi(ivcgypsum as binder) | \{wall - identifier\} is unique identifying in T \}

ktcinside binder = \{ T \subseteq ivcinside binder | \{wall - identifier\} is unique identifying in T \}

ktcoutside binder = \{ T \subseteq ivcoutside binder | \{wall - identifier\} is unique identifying in T \}

ktcgypsum as an in situ product of a chemical reaction =
\{
    T \subseteq \Pi(ktcgypsum as an in situ product of a chemical reaction)
\}

kterestored = \{ T \subseteq ivrestored | \{wall - identifier\} is unique identifying in T \}

kterestoration material = \{ T \subseteq (kterestoration material) | \{wall - identifier\} is unique identifying in T \}

Before defining the universe, an auxiliary function is necessary:
HBMWE =
{
  (wall ; ktcwall),
  (gypsum ; ktcgypsum),
  (gypsum as binder ; ktcgypsum as binder),
  (inside binder ; ktcinside binder),
  (outside binder ; ktsoutside binder),
  (gypsum as an in situ product of a chemical reaction ; ktcgypsum as an in situ product of a chemical reaction),
  (restored ; ktcrestored),
  (restoration materials ; ktcrestoration materials)
}

**Knowledge Universe Constraints (KUC)**
The knowledge universe then is:

\[ UBMWE = \]
\[
\{ 
  KS | KS \in \Pi(HBMWE) \text{ and } id(\{\text{wall} - \text{identifier}\}) \text{ connects } \\
  KS(\text{gypsum}) \text{ with } KS(\text{wall}) \text{ and } \\
  id(\{\text{wall} - \text{identifier}\}) \text{ connects } \{k \in KS(\text{gypsum as binder})\} \\
  k(\text{gypsum as binder}) = \text{yes} \text{ with } \\
  \{k \in KS(\text{gypsum}) | k(\text{gypsum as binder}) = \text{yes}\} \text{ and } \\
  id(\{\text{wall} - \text{identifier}\}) \text{ connects } \{k \in KS(\text{inside binder})\} \\
  k(\text{type of inside binder})(binding materials) = \text{gypsum} \text{ with } KS(\text{gypsum}) \text{ and } \\
  id(\{\text{wall} - \text{identifier}\}) \text{ connects } \{k \in KS(\text{outside binder})\} \\
  k(\text{type of outside binder})(binding materials) = \text{gypsum} \text{ with } \{k \in KS(\text{gypsum}) | k(\text{gypsum as binder}) = \text{yes}\} \text{ and } \\
  id(\{\text{wall} - \text{identifier}\}) \text{ bilaterally connects } KS(\text{outside binder}) \text{ with } \\
  \{k \in KS(\text{inside binder}) | k(\text{type of mortar}) = \text{rendering mortar}\} \\
  id(\{\text{wall} - \text{identifier}\}) \text{ bilaterally connects } \\
  \{k \in KS(\text{gypsum as binder}) | k(\text{gypsum as binder}) = \text{yes}\} \text{ with } \{k \in KS(\text{gypsum as an in situ product of a chemical reaction}) \text{ and } id(\{\text{wall} - \text{identifier}\}) \text{ bilaterally connects } \\
  \{k \in KS(\text{gypsum}) | k(\text{gypsum as binder}) = \text{yes}\} \text{ with } k(\text{type of binder}) = \text{gypsum} \text{ and } id(\{\text{wall} - \text{identifier}\}) \text{ bilaterally connects } \\
  \{k \in KS(\text{inside binder}) | k(\text{type of inside binder})(binding materials) = \text{lime}\} \text{ with } \{k \in KS(\text{gypsum as an in situ product of a chemical reaction}) \text{ and } k(\text{type of binder}) = \text{lime}\} \text{ and }
\]
Querying

Querying a database is the way to perform matches and to classify an object as an
instance of an object-type. A relevant query might be:

- 'Give the identification codes of the walls that do not contain gypsum as a binder, yet are susceptible to the formation of ettringite'

Translated into a symbol level language, the logical structure of the query might look like:

$$\lambda \text{KS} \in \text{UBMWE}:$$

$$\{ k \mid (\text{wall - identifier}) \} \in \text{KS}(\text{wall}) \land k'(\text{susceptible to the formation of ettringite}) = \text{yes} \land (k'' \in \text{KS}(\text{gypsum as binder}) \land k''(\text{gypsum as binder}) = \text{no}) \}$$

![Figure 7.14: A Graphical Overview of the Knowledge Universe](image)

Other structures are possible. For instance:

$$\lambda \text{KS} \in \text{UBMWE}:$$

$$\{ k \mid (\text{wall - identifier}) \} \in \text{KS}(\text{wall}) \land k'(\text{susceptible to ettringite}) = \text{yes} \land \exists k'' \in \text{KS}(\text{gypsum as binder}) : k''(\text{gypsum as binder}) = \text{no} \land k'(\text{wall - identifier}) = k \}$$

Another query could be:

- 'Give the identification codes of the walls that are exposed to view and that are
The formulation of queries for a data base is not a trivial matter (Remmen, 1985). The knowledge of the reconstructed object-types is essential for the adequate formulation of a query. Suppose we want to know what walls are susceptible to the formation of ettringite after restoration. If this question is frequently posed to a system, it will be worthwhile to consider designing a view. A view is a named query. Logically, a view is a function in which the first co-ordinates of the ordered pairs denote the names of the queries and the second co-ordinates denote the queries (for a formal definition of a view, see Chapter 3). In the present case, the formulation of a view not only requires knowing what restoration materials have been applied, but also what possible negative interactions may occur with materials originally used in the walls. For these purposes, the knowledge of the second object-type is essential. On the basis of the second object-type, the view becomes:

\[
\text{View} = \\
\{ \\
\quad (\text{susceptible after restoration}_1; \lambda k \in UBMWE: \\
\quad \{(p \mid (\text{wall - identifier}) \mid p \in \\
\quad \{k \in KS(\text{restored}) \mid k(\text{restored}) = \text{yes}\}) \bowtie \\
\quad \{k \in KS(\text{gypsum as binder}) \mid k(\text{gypsum as binder}) = \text{yes}\}) \bowtie \\
\quad \{k \in KS(\text{restoration material}) \mid k(\text{restoration material}) = \\
\quad \text{cement OR hydraulic - lime OR lime - pozzolan}), \\
\quad (\text{susceptible after restoration}_2; \lambda k \in UBMWE: \\
\quad \{(p \mid (\text{wall - identifier}) \mid p \in \\
\quad \{k \in KS(\text{restored}) \mid k(\text{restored}) = \text{yes}\}) \bowtie \\
\quad \{k \in KS(\text{restoration materials}) \mid k(\text{restoration materials}) = \\
\quad \text{lime - ordinary sand}) \bowtie \\
\quad \{k \in KS(\text{gypsum as binder}) \mid k(\text{gypsum as binder}) = \text{yes}\}) \bowtie \\
\quad (\{k \in KS(\text{inside binder}) \mid k(\text{inside binder})(\text{type of sand}) = \text{natural or artificial pozzolan}\}) \text{ OR} \\
\quad (k \mid (\text{wall - identifier}) \mid k \in KS(\text{type of inside binder}) \mid k(\text{type of mortar}) = \text{rendering mortar and } \exists k' \in KS(\text{outside binder}) \mid k'(\text{outside binder})(\text{type of sand}) = \text{natural or artificial pozzolan and } k' \mid (\text{wall - identifier}) = k \mid (\text{wall - identifier})\}),
\]

- 223 -
(susceptible after restoration3; \( \lambda KS \in UBMWE \):
\[
\{p \mid (\text{wall - identifier}) \} p \in \\
\{k \in KS(\text{restored}) \mid k(\text{restored}) = \text{yes} \} \Rightarrow \\
\{k \in KS(\text{restoration materials}) \mid k(\text{restoration materials}) = \\
gypsum\} \Rightarrow \\
((\{k \in KS(\text{inside binder}) \mid k(\text{inside binder})(\text{type of binder}) = \text{lime} \} \lor \\
\{k \mid (\text{wall - identifier}) \mid k \in KS(\text{type of inside binder}) \mid k(\text{type of mortar}) = \text{rendering} \} \land \\
\exists k' \in KS(\text{outside binder}) : k'(\text{outside binder})(\text{type of binder}) = \text{lime} \\
\land k' \mid (\text{wall - identifier}) = k \mid (\text{wall - identifier}) \}) \Rightarrow \\
((\{k \in KS(\text{inside binder}) \mid k(\text{inside binder})(\text{type of sand}) = \text{natural or artificial pozzolan} \} \lor \\
\{k \mid (\text{wall - identifier}) \mid k \in KS(\text{type of inside binder}) \mid k(\text{type of mortar}) = \text{rendering} \} \land \\
\exists k' \in KS(\text{outside binder}) : k'(\text{outside binder})(\text{type of sand}) = \text{natural or artificial pozzolan} \land \\
k' \mid (\text{wall - identifier}) = k \mid (\text{wall - identifier}) \})
\]

Note that the four ordered pairs correspond with the four conjunct sets represented by
R1, R4, R8 and R11 of the DT named Restored brick masonry wall susceptible to
degradation due to the formation of ettringite.

In an orientation toward objects and their attributes, the reconstructed object-types provide indications for designing an appropriate database scheme. This relates to finding criteria to decide which relations should be base relations and which should be derived relations (Gallaire, Minker, & Nicolas, 1984). Developing the scheme is not a trivial task (De Broek, 1989, pp.92). Furthermore, the object-types provide an insight into defining integrity constraints and finding more efficient means for the detection of the violation of integrity constraints. Finally, they improve the formulation of queries and views.

Search in a database supports a model-theoretic strategy (Brodie & Jarke, 1986, p191.). The scheme, also called the theory, is provided by the definition of data structures and integrity constraints. A database state is an interpretation that must be a model of the theory as expressed in the definition of the knowledge universe. A query is a formula with free variables to be bound at runtime. Query evaluation is the
computation of a truth value for the formula over the current database state. The result is a set of variable-free instantiations provable from the underlying structures.

7.4.2 Knowledge of Object-types

Production rules and fact sets on the one hand and the associated backward, forward or hybrid chaining inference mechanisms on the other hand, constitute a popular representation formalism for encoding knowledge of object-types.

A production rule has the structure:

$$IF <\text{antecedent}> \ THEN <\text{consequent}>$$

The antecedent of a rule is a conjunction of (one or more disjunctions of) conditions. In the antecedent AND- and OR-statements connect the conditions to form conjunctions or disjunctions. A condition is built from definite statements such as same, notsame or less_than, about object-attribute-value triples or $<o, a, v>$ triples. An example of a condition is:

$$\text{same} < \text{wall, gypsum, present}>$$

Note that the objects of a $<o, a, v>$ triple, are distinct from the objects that should comply with the constraints of an object-type. The former are implementation constructs or symbol level objects, whereas the latter are knowledge level objects. Both types of objects are related though. Symbol level objects should represent the knowledge level objects by means of a specific language of implementation.

The consequent of a rule has the form $\text{conclude} <o,a,v>$. An example of a production rule is:

$$IF \ \text{same} < \text{wall, gypsum, present}> \text{ AND } \text{ < wall, hydrated calcium aluminates, present> AND < wall, moisture, present>}$

$THEN \ \text{conclude} < \text{wall, brick masonry walls susceptible to the formation of ettringite, yes}>$

Production rules can also display a structure in which relational operators replace the definite statements, and in which the objects and the conclude statement are left out respectively from the $<o,a,v>$ triples and the consequent. This yields:

$$IF \ \text{gypsum} = \text{present AND hydrated calcium aluminates} = \text{present AND moisture} = \text{present}$$
THEN  brick masonry walls susceptible to the formation of ettringite = yes

A fact set has the following structure:

\[ F = \{ (\text{gypsum}; \text{present}), (\text{hydrated calcium aluminates}; \text{present}), (\text{moisture}; \text{present}) \} \]

An inference mechanism selects and evaluates production rules from the rule base. If the conditions of a selected rule are met, the inference mechanism will execute the actions. Subsequently, the fact set is updated by adding, modifying or retracting facts. Two basic forms of inference can be distinguished. Backward chaining or top down inference starts with one or more goals. A goal may match with the conclusion of one or more production rules in the rulebase. Each of the selected production rules is applied by considering its conditions. These conditions are the new subgoals. A conclusion can be satisfied by the facts of the fact set or by users answering questions posed by the rule based knowledge base. If all conditions of a rule are met, its actions will be executed. Forward chaining or bottom up inference, starts with one or more facts which are matched against the conditions of the production rules. Again, if all conditions of a rule are met, its actions will be executed and the fact set will adapted accordingly. In case, both type of inference mechanisms are available, the inference mechanism is called hybrid.

Structuring the production rules in such a way that a knowledge-based system is able to display the desired problem solving behaviour, is a task with many complexities. Therefore, in this section, the system of logic structures that organises the production rules is extensively described. Successively, the following logic structures and their relations will be dealt with (1) the knowledge schema, (2) the variable constraints, (3) the inter-variable constraints, (4) the knowledge table constraints and (5) the knowledge universe constraints.

The Knowledge Schema

The following set-valued function displays a part of the knowledge schema underlying the first object-type. It reflects the system of DT's that represent the first object-type. In each case, the first co-ordinate of every ordered pair is the name of a DT and the second co-ordinate is the set consisting of the conditions and actions of the same DT. Moisture is an exception that can be derived from Figure 7.8.

\[ BMWSE = \]

\{ (Brick masonry wall susceptible to the formation of ettringite ; (gypsum, hydrated calcium aluminates, moisture, brick masonry wall susceptible to the formation of ettringite)),

(Gypsum ; {gypsum as binder, gypsum as an in situ product of a ;

- 226 -
chemical reaction, gypsum}), (Gypsum as binder
(type of mortar, type of inside binder, type of outside binder, gypsum as binder)),
(Gypsum as an in situ product of a chemical reaction
(type of salts, type of binder, moisture, gypsum as an in situ product of a
chemical reaction)),
(Hydrated calcium aluminates
(type of binder, type of sand, hydrated calcium aluminates)),
(Moisture
(wetting rain, capillary flow from groundwater, moisture))
}

Variable Constraints (VC)
For every element of every element of the range of BMWSE, attribute constraints are
defined. The variable constraints describe the domains of conditions and conclusions
occurring in production rules. The variable constraints below, for example, indicate
that gypsum, hydrated calcium aluminates and moisture can only attain the values
present or not present and that brick masonry wall susceptible to the formation of
ettringite can only attain the values yes or no.

vcbrick masonry wall susceptible to the formation of ettringite =
{
(gypsum ; {present, not present}),
(hydrated calcium aluminates ; {present, not present}),
(moisture ; {present, not present}),
(brick masonry wall susceptible to the formation
of ettringite ; {yes, no})
}

Following similar lines, the other variable constraints are:

cgyypsum =
{
(gypsum as binder ; {yes, no}),
(gypsum as an in situ product of a chemical reaction ; {yes, no})

vcgypsum as binder =
{
(type of mortar ; {jointing mortar, rendering mortar})
(type of inside binder : {gypsum, lime, hydraulic lime}),
(type of outside binder : {gypsum, lime, hydraulic lime}),
gypsum as binder : {yes, no})

vcgypsum as an in situ product of a chemical reaction =
{
(type of salts ; {sulphates, chlorides, nitrates, carbonates}),
(type of binder ; {gypsum, lime, hydraulic lime}),
(moisture ; {yes, no}),
gypsum as an in situ product of a chemical reaction ; {yes, no})
}

vhydrated calcium aluminates =
{
(type of binder ; {gypsum, lime, hydraulic lime}),
(type of sand ; {natural pozzolan, artificial pozzolan, normal sand}),
(hydrated calcium aluminates ; {present, not present})
}

vcmoisture =
{
(wetting rain ; {yes, no}),
(capillary flow from groundwater ; {yes, no}),
(moisture ; {yes, no})
}

**Inter-variable Constraints (IVC)**

Inter-variable constraints do not have to be defined, because the mutual dependence of variables is automatically included in the table constraints.

**Knowledge Table Constraints (KTC)**

If \( PS(\text{TR}_\text{BMWSE - Root}) \) denotes the set of paths that do not contain the root node and \( \text{TR}_\text{BMWSE} \) is the exclusive and exhaustive tree (see Chapter 5) underlying the corresponding DT, then:

\[
\text{WBMWSE} =
\]
\( KT \subseteq \Pi(\text{brick masonry wall susceptible to the formation of ettringite}) \)
\( \forall PR \in KT : \text{exactly one path } P \in PS(\text{TR}_{BMWSE-Root}) \text{ exists and} \)
\( \forall P \in PS(\text{TR}_{BMWSE-Root}) : \text{exactly one } PR \in KT \text{ exists such that} \)
\( PR = \{x, y, \ldots, e\} \text{ and } P = \{(x; y), (y; z), \ldots, (a; b), (b; e)\} \)

An element of \( BMWSE \) is a rule group. An example of an element of \( BMWSE \)
expressed in mathematical logic is:

\[ RL(\text{brick masonry wall susceptible to the formation of ettringite}) = \{
\{(\text{gypsum; present}), (\text{hydrated calcium aluminates; present}), (\text{moisture; present}),
(\text{brick masonry wall susceptible to ettringite; yes})\},
\{(\text{gypsum; present}), (\text{hydrated calcium aluminates; present}), (\text{moisture; not present}),
(\text{brick masonry wall susceptible to ettringite; no})\},
\{(\text{gypsum; present}), (\text{hydrated calcium aluminates; not present}), (\text{moisture; -}),
(\text{brick masonry wall susceptible to ettringite; no})\},
\{(\text{gypsum; not present}), (\text{hydrated calcium aluminates; -}), (\text{moisture; -}),
(\text{brick masonry wall susceptible to ettringite; no})\}
\] 

The same example but now expressed in a typical rule language is:

**IF** gypsum = present AND
hydrated calcium aluminates = present AND
moisture = present
**THEN** brick masonry walls susceptible to the formation of ettringite = yes

**IF** gypsum = present AND
hydrated calcium aluminates = present AND
moisture = not present
**THEN** brick masonry walls susceptible to the formation of ettringite = no

**IF** gypsum = present AND
hydrated calcium aluminates = not present
**THEN** brick masonry walls susceptible to the formation of ettringite = no

**IF** gypsum = not present

- 229 -
**THEN** brick masonry walls susceptible to the formation of ettringite = no

Because the demand is that each possible rule group is a representation of an exhaustive and exclusive tree, other demands are already included and need not to be repeated. For example, such a demand is:

(gypsum, hydrated calcium aluminates, moisture) is u.i. in KT

Similarly, the other rule groups can be defined.

If $PS(\text{TR}_{\text{Gypsum}}-\text{Root})$ denotes the set of paths that do not contain the root node and $\text{TR}_{\text{Gypsum}}$ is the exclusive and exhaustive tree underlying the corresponding DT, then:

$$\textbf{WGYPSUM} =$$

\[
\{ \text{KT} \subseteq \Pi(\text{vcgyypsum}) | \forall PR \in \text{KT} : \text{exactly one path } P \in PS(\text{TR}_{\text{Gypsum}}-\text{Root}) \text{ exists and } \\
\forall P \in PS(\text{TR}_{\text{Gypsum}}-\text{Root}) : \text{exactly one } PR \in \text{KT} \text{ exists such that } \\
PR = \{x, y, \ldots, e\} \text{ and } P = \{ (x; y), (y; z), \ldots, (a; b), (b; e) \} \}
\]

If $PS(\text{TR}_{\text{Gypsum as binder}}-\text{Root})$ denotes the set of paths that do not contain the root node and $\text{TR}_{\text{Gypsum as binder}}$ is the exclusive and exhaustive tree underlying the corresponding DT, then:

$$\textbf{WGYPSUM AS BINDER} =$$

\[
\{ \text{KT} \subseteq \Pi(\text{vcgyypsum as binder}) | \\
\forall PR \in \text{KT} : \text{exactly one path } P \in PS(\text{TR}_{\text{Gypsum as binder}}-\text{Root}) \text{ exists and } \\
\forall P \in PS(\text{TR}_{\text{Gypsum as binder}}-\text{Root}) : \text{exactly one } PR \in \text{KT} \text{ exists such that } \\
PR = \{x, y, \ldots, e\} \text{ and } P = \{ (x; y), (y; z), \ldots, (a; b), (b; e) \} \}
\]

If $PS(\text{TR}_{\text{Gypsum as an in situ product of a chemical reaction}}-\text{Root})$ denotes the set of paths that do not contain the root node and $\text{TR}_{\text{Gypsum as an in situ product of a chemical reaction}}$ is the exclusive and exhaustive tree underlying the corresponding DT, then:

$$\textbf{WGYPSUM AS AN IN SITU PRODUCT OF A CHEMICAL REACTION} =$$

\[
\{ \text{KT} \subseteq \Pi(\text{vcgyypsum as an in situ product of a chemical reaction}) | \\
\forall PR \in \text{KT} : \text{exactly one path } P \in PS(\text{TR}_{\text{Gypsum as an in situ product of a chemical reaction}}-\text{Root}) \text{ exists and } \\
\forall P \in PS(\text{TR}_{\text{Gypsum as an in situ product of a chemical reaction}}-\text{Root}) : \text{exactly one } PR \in \text{KT} \text{ exists such that }
\]

- 230 -
If \( PS(T_{Hvdrated\ calcium\ aluminates - Root}) \) denotes the set of paths that do not contain the root node and \( T_{Hydrated\ calcium\ aluminates} \) is the exclusive and exhaustive tree underlying the corresponding DT, then:

\[
WHYDRATED\ CALCIUM\ ALUMINATES = \\
\{ \\\n\quad KT \subseteq \Pi(v_{hydrated\ calcium\ aluminates})| \\
\quad \forall PR \in KT: \text{exactly one path } P \in PS(T_{Hydrated\ calcium\ aluminates - Root}) \text{ exists and} \\
\quad \forall P \in PS(T_{Hydrated\ calcium\ aluminates - Root}): \text{exactly one } PR \in KT \text{ exists such that} \\
\quad PR = \{x, y, \ldots, e\} \text{ and } P = \{(x; y), (y; z), \ldots, (a; b), (b; e)\} \\
\} \\
\]

If \( PS(T_{Moisture - Root}) \) denotes the set of paths that do not contain the root node and \( T_{Moisture} \) is the exclusive and exhaustive tree underlying the corresponding DT, then:

\[
WMOISTURE = \\
\{ \\\n\quad KT \subseteq \Pi(v_{moisture})| \\
\quad \forall PR \in KT: \text{exactly one path } P \in PS(T_{Moisture - Root}) \text{ exists and} \\
\quad \forall P \in PS(T_{Moisture - Root}): \text{exactly one } PR \in KT \text{ exists such that} \\
\quad PR = \{x, y, \ldots, e\} \text{ and } P = \{(x; y), (y; z), \ldots, (a; b), (b; e)\} \text{ and} \\
\} \\
\]

Knowledge Universe Constraints (KUC)

Before defining the knowledge universe constraints, an auxiliary function is required that links each object to a space of knowledge tables or rule groups.

\[
ABMWE = \\
\{ \\\n\quad \text{(brick masonry wall susceptible to the} \quad ; WBMWSE), \\
\quad \text{formation of ettringite} \quad ; WGYPSEM), \\
\quad \text{gypsum} \quad ; WGYPSEM AS BINDER), \\
\quad \text{gypsum as binder} \quad ; WGYPSEM AS AN IN SITU PRODUCE OF A \quad DUCT OF A CHEMICAL REACTION), \\
\quad \text{gypsum as an in situ product of a} \quad \text{chemical reaction} \\
\} \\
\]

- 231 -
(hydrated calcium aluminates ; WHYDRATED CALCIUM ALUMINATES),
(moisture ; WMOISTURE)

The knowledge universe then is:

\[
BMWEU = \\
\{ \text{id((hydrated calcium aluminates)) bilaterally connects } KS(\text{Hydrated calcium aluminates}) \text{ with } KS(\text{Brick masonry wall susceptible to the formation of ettringite}) \text{ and} \\
\text{id((moisture)) bilaterally connects } KS(\text{Moisture}) \text{ with } KS(\text{Brick masonry wall susceptible to the formation of ettringite}) \text{ and} \\
\text{id((gypsum as binder)) bilaterally connects } KS(\text{Gypsum as binder}) \text{ with } KS(\text{Gypsum}) \text{ and} \\
\text{id((gypsum as an in situ product of a chemical reaction)) bilaterally connects } KS(\text{Gypsum as an in situ product of a chemical reaction}) \text{ with } KS(\text{Gypsum}) \}\n\]

The system of logic structures for production rules representing the restoration object-type:

Knowledge Schema
\[
RBMWSE = \\
\{ \\
(\text{restored brick masonry wall susceptible to the formation of ettringite}) ; \\
(\text{restoration material, gypsum, type of sand, type of binders, moisture}) \\
\}
\]

Variable Constraints (VC)
\[
VC \text{ } RBMWSE = \\
\{ \\
(\text{restoration material}) ; \{\text{cement, hydraulic lime, lime - pozzolan sand, lime - normal sand}\}, \\
(\text{gypsum}) ; \{\text{yes, no}\}, \\
\}\n\]
(type of sand) \{pozzolan, normal\},
(type of binders) \{lime, hydraulic lime, gypsum\},
(moisture) \{yes, no\}

**Inter-variable Constraints (IVC)**
The inter-variable constraints need not be defined, because the mutual dependence of variables is included in the table constraints.

**Knowledge Table Constraints (KTC)**
If $PS(TR_{RBMWSE-Root})$ denotes the set of paths that do not contain the root node and $TR_{RBMWSE}$ is the exclusive and exhaustive tree underlying the corresponding DT, then:

\[ WRBMWSE = \]
\[
\{ KT \subseteq \Pi(VC_{RBMWSE}) \forall PR \in KT : \text{exactly one path } P \in PS(TR_{RBMWSE-Root}) \text{ exists and} \\
\forall P \in PS(TR_{RBMWSE-Root}) : \text{exactly one } PR \in KT \text{ exists such that} \\
PR = \{x, y, ..., e\} \text{ and } P = \{(x; y), (y; z), ..., (a; b), (b; e)\} \}
\]

**Knowledge Universe Constraints KUC**
The auxiliary function is:

\[ ARBMWE = \]
\[
\{ \text{restored brick masonry wall susceptible to the formation of ettringite} \quad \; \text{; WRBMWSE} \}
\]

Then:

\[ RBMWEU = \{KS \mid KS \in \Pi(ARBMWE)\} \]

**7.4.3 Knowledge of Object-types and Objects Reduced to the Symbol Level**
The representation formalisms of the two approaches differ in search strategy and structure. In an object or attribute oriented approach, search supports query evaluation (*model-theoretic inferencing*), whereas in an object-type oriented approach search supports deductive inferencing (*proof-theoretic inferencing*). As the structures
described in mathematical logic show, in deductive inferencing complex knowledge is represented by means of a system of groups of rules (axioms), whereas in query evaluation knowledge representation is restricted to predicate clauses that do not contain negations, disjunctions or quantified statements. It is in this respect that Brachman & Levesque (1986) state that data bases are knowledge bases of a simple and limited form. Queries (Hsieh, 1993, p.65) and integrity constraints, however, are transformed into more complicated forms that should represent complex knowledge. So, the overall effect of moving from one knowledge representation formalism to another, is the transfer of the complexity of knowledge from the rules to the queries and integrity constraints. The named query (view) that retrieves walls that have been the subject of restoration and that are susceptible to the formation of ettringite and the integrity constraints in the form of formal connections, indeed display a considerable degree of complexity.

For these reasons, the structures of the object-type oriented approach are not the same as the structures in the object orientation. One difference is that the identifiers of the objects are left out from the structures in the former orientation. These identifiers are not needed, because in the object-type oriented approach the attributes of objects are expected to be filled in interactively by users answering questions of the knowledge-based system. In the object oriented approach the knowledge needed for matching objects as instances of object-types is represented by storing attributes of objects using abstractions and relations between attributes. In the object-type oriented approach the knowledge needed for matching objects as instances of object-types is represented by storing the object-types directly and the user is expected to insert the attributes of the objects. However, this must not lead the reader to think that databases deal with individual objects and knowledge bases with classes of objects. Many knowledge bases deal with both objects and classes of objects, while many databases, for instance, statistical databases, deal with classes of objects.

How are we to choose rationally between the different representation formalisms considering that both formalisms are able to represent knowledge? At this point, we must realise that the availability of knowledge in the form of functional object-types is essential. As stated before, functional object-types provide a vantage point from which to compare and evaluate representation formalisms. We already examined the basic properties of record-based information models from a functional viewpoint and concluded that if the essential configuration of the conjunct sets leading to the same goal, is characterised by identical attributes, each attribute having the same kinds of values, records are excellent representation and processing tools. If, on the other hand, these conjunct sets are characterised by heterogeneity caused by having to classify objects in a goal-oriented fashion accounting for different descriptors and new conceptual interactions, records are not appropriate (Kent, 1979; Murdoch & Johnson, 1990). Functional object-types show that the presumption that objects have a homogeneous nature is not true by definition.

The theory of functional classifications as a basis for knowledge-based systems implies that functional object-types should underlie a knowledge-based system whether it is object-type oriented or not:

- to provide users with sufficient knowledge to classify an object as an instance
of an object-type. As users may have several levels of expertise, the system should have several description levels of object-types and objects, to provide the necessary flexibility.

* to define efficient structures (rule groups, tables, integrity constraints, views) which together with the available computational processes display intelligent behaviour for matching object-types and objects.

To a considerable degree the Advanced Knowledge Transfer System (AKTS) complies with these demands. AKTS adopts a tight coupling strategy to integrate both knowledge of object-types and knowledge of objects in a unified framework. Knowledge representation in AKTS is based on systems of DT's which can be consulted by a meta-interpreter and Prolog-definitions which can be operated upon by SLD-resolution. Both inference engines are tightly coupled, because both types of knowledge are stored in the same domain, not as in conventional approaches in which knowledge of object-types and objects are stored separately. For example, by means of AKTS it is possible to ask the values of functionally relevant attributes of walls. Using Prolog, these attributes can also be stored at certain levels of abstraction.

7.5 CONCLUSION

To exemplify the central argumentation of this study we presented a case concerning the chemical degradation and restoration of ancient brick masonry walls. The case-study illustrates three main methodological points: (1) the value of a knowledge level integration of AI and DBT, (2) the value of the theory of functional classifications to accomplish such an integration and (3) the value of applying DT's and Prolog in the form AKTS makes them available as a modelling language for functional object-types. Each of these three values finds expression in knowledge level and symbol level advantages.

First, by concentrating on knowledge level descriptions of object-types and objects, the case-study reveals the advantages of a knowledge level integration of AI and DBT. Since these descriptions are not obstructed by a bias toward a specific representation formalism, problems due to using a representation formalism to model knowledge are avoided. On the one hand, the implementation-free descriptions of object-types and objects yield knowledge level advantages such as a better view of knowledge and enriched possibilities to improve explanation and maintenance. On the other hand, the reduction of the object-types and objects to symbol level structures and computational processes, is facilitated by the availability of clear descriptions. Codd (1979, p.398) already described the associated advantages for DB-systems:

> '... a meaning-oriented data model stored in a computer should enable it to respond to queries and other transactions in a more intelligent manner. Such a model could also be a more effective mediator between the multiple external views employed by application programs and end-users on the one
Second, the case-study reveals the advantages of the theory of functional classifications to accomplish a knowledge level integration. At the knowledge level, many traditional classifications of masonry walls are possible. For instance, Adam (1984) describes a classification based on architectural-structural criteria. However, functional classifications are conducive to a better organisation of the knowledge of objects (the masonry walls) and knowledge of constraints (the requirements these masonry walls should meet to be resistant to chemical degradation). At the symbol level, functional object-types foster the transfer of knowledge into representation formalisms of AI and DBT. The functional reconstruction of object-types and objects supports the organisation and adaptation (for technical design purposes) of the mathematical functions underlying representation formalisms. It not only helps to structure production rules, but also supports the representation of views.

Third, the utility of the joint application of DT's and Prolog is exemplified. Especially in the form they are offered by AKTS. Together, DT's and Prolog have the capability of describing object-types that delineate walls susceptible to the formation of ettringite and objects that delineate brick masonry walls. Besides showing that Prolog can be used for recursive definitions (Chapter 5), the case, in addition, reveals the utility of Prolog as a database language. By this, the expressive power of AKTS is emphasised.

Knowledge-based systems based on traditional classifications are easier to develop, but less useful, whereas knowledge-based systems based on functional classifications are difficult to develop, but better geared to the concrete needs of preventing chemical degradation and carrying out restorations on brick masonry walls.
CHAPTER 8

EPILOGUE

8.1 INTRODUCTION

Knowledge-based systems are pervading nearly all fields of trade and industry. As volume and complexity of knowledge are increasing in the daily practice of organisations, it is expected that a growing number of organisations will be in need of these knowledge-based systems (see for instance Cohen, 1989, p.3; Smith, 1986, p.22). Unfortunately, the development of knowledge-based systems causes many problems. Especially modelling knowledge is a major obstacle in the development of these systems. This thesis has extensively dealt with the following dimensions underlying this problem:

1. Confusion of knowledge and knowledge representation formalisms
2. Lack of an adequate theory of the nature of knowledge
3. Lack of an adequate formal language

To complete the thesis, this chapter briefly states the principal conclusions concerning these dimensions (Section 8.2). Subsequently, to provide a broader perspective, we briefly discuss the relation of qualitative object-types and quantitative object-types (Section 8.3). Finally, the chapter is rounded off with the discussion of some implications of adopting the proposals made here as a remedy in dealing with these dimensions (Section 8.4).

8.2 PRINCIPAL CONCLUSIONS

In characterising modelling knowledge as an activity of reconstructing complex object-types and complex objects, we explained the need to integrate AI (to deal with complex object-types) and DBT (to deal with complex objects). The three dimensions that interfere with the modelling of knowledge and thus with the reconstruction of object-types and objects led to the following research issues (Chapter 1):

1. What are the advantages of integrating AI and DBT at the knowledge level compared to symbol level integration? Does a knowledge level integration contribute to the process of modelling knowledge?
II. Does the theory of functional classifications constitute a conceptual advance for realising a knowledge level integration, and, if so, in what respects? Does the theory help to define object-types and objects to enable knowledge-based systems to perform matches?

III. Can the joint application of Prolog and DTs be considered as an adequate knowledge level language for describing functional object-types? That is, does it comply with the general requirements applied to modelling languages and the particular requirements for functional object-types?

The principal conclusions with regard to these issues can be stated briefly.

The investigation of the first research issue yielded a number of advantages that a knowledge level integration of AI and DBT has over a symbol level integration. These advantages refer to the availability of a specific computer systems level to define the nature of knowledge, to the improved competence to reconstruct knowledge level models and to the availability of a computer systems level from which to analyse and compare representation formalisms. It also appeared that the role of mathematical logic can be specified better by a knowledge level integration: from a knowledge level perspective mathematical logic is perceived as a formal language that is uniquely appropriate for the analysis of knowledge. Mathematical logic thus helps to reconstruct object-types and objects and to assess them in a knowledge universe. However, it is also concluded that mathematical logic is not enough, because it has no ontology. A theory of the nature of knowledge is needed to provide this.

The second research issue concerns the theories of the nature of knowledge to intercept the observed weakness of mathematical logic. We argued that, compared to competing probabilistic and prototypical theories, the theory of functional object-types offers promising perspectives, because it accounts for fundamental concerns of modelling functional object-types and objects. A central element in this theory is the fact that the problem of modelling knowledge is dealt with by systematically identifying several object-types which emerge by explicitly accounting for functional equivalence. As such the theory constitutes an integration of AI and DBT at the knowledge level. This theory does not only touch upon the activity of knowledge modelling and the choice of methodologies and modelling techniques, but also on the evaluation and choice of knowledge representation formalisms. It is claimed that this theory can be viewed as a framework for knowledge modelling. We also concluded that a language is required to formally describe object-types and objects in accordance with the theory of functional object-types.

The third research issue concerns the evaluation of the joint application of Prolog and DTs as a formal language to describe, validate and simulate functional object-types and objects. It is argued that DTs provide some methodological advantages compared to e.g. decision plan nets, as advocated by Timmermans (1987). Furthermore, it is argued that, though Prolog compensates a number of weak points of DTs, some drawbacks remain. The Advanced Knowledge Computer Transfer System (AKTS) that offers extensive facilities for working with Prolog and DTs to reconstruct functional object-types has been developed to solve these problems. In comparison with DTs, the validation facilities are to an important degree enhanced in
correspondence with the proposed tree-based definitions of a DT. AKTS offers facilities to check the exhaustiveness and exclusiveness of a complete DT-system. Furthermore, AKTS provide possibilities to check the connections between DTs using the formal definitions of connections. Finally, AKTS has extensive facilities to simulate a reconstructed functional object-type. In brief, we can state that AKTS takes advantage of the complementarity of DTs and Prolog and eliminates remaining drawbacks such as the lack of prototypes of DTs, automated checking facilities and the lack of advanced graphical facilities for drawing DTs.

8.3 QUANTITATIVE OBJECT-TYPES

The views of object-types that are presented here are mainly qualitative object-types. As in physics the aim is often to introduce quantitative object-types such as length, time, mass which permit the use of exact figures on measuring scales. If adequate measuring techniques are not available, one is satisfied with qualitative object-types (Stegmüller, 1973).

Our approach is somewhat different. We always define an object-type by first assuming a goal and finding relevant conditions. Obviously, we account for the occurrence of functional equivalence. If it is clear, however, that the conditions needed to describe an object-type independently of each other contribute to the definition of the object-type and that these conditions are metric, we will consider the possibility of using quantitative object-types. Whether this is possible, depends on the nature of knowledge: if the conditions of the object-type comply with the principle of additivity or the principle of multiplicativity, it is permitted to use quantitative object-types. The principles are displayed below:

(1) \( \dot{U} = X_1 + X_2 + \ldots + X_k \) (object-type described by additions)

(2) \( \dot{U} = X_1 \times X_2 \times \ldots \times X_k \) (object-types described by multiplications)

An example of a multiplicative model is:

(3) If \( F = 0 \) then: \( \frac{d(m,v)}{dt} = 0 \) (the first motion law of Newton)

\( F \) is the sum of the vectors of the exogenous forces that operate over an object. \( m \) is the mass of that object and \( v \) is its velocity. \( F \) and \( v \) are vectors (have a direction and orientation) and \( m \) is a scalar (without a direction and orientation). If we assume that the law is true the principle of multiplicativity can be applied and the use of quantitative object-types is allowed.

The reader should note that the knowledge represented in this type of models can be represented using AKTS in several ways. Firstly, but this is an artificial solution, it is possible to use a single DT (for an example see: De Gelder, Van Gorp, & Lucardie,
1993). Secondly, it is possible to incorporate the formula in the design phase of AKTS by means of DT-properties or parameter properties (see Chapter 6). Thirdly, it is possible to use Prolog to represent the formula. In fact, this means that functional object-types subsume quantitative object-types.

8.4 IMPLICATIONS

When we review the main approaches towards modelling in computer science, we can see that most of the activities have prototypical and probabilistic traits. The line of reasoning is that the reconstruction of generic object-type is possible and valuable. A further specification of the object-type is supposed to be performed afterwards to tune the object-type for specific applications and implementations.

A sad example of this approach is the development of the Building Information Model (BIM) that was finished in 1989. BIM is the result of a large-scale Innovative-oriented project that aimed at contributing to the information-technical infrastructure for the building and construction industry. BIM indicates what attributes of building objects (including building processes) are necessary for effectively dealing with these objects. The description of the attributes was made in accordance with international developments in order to develop a standard. Because BIM should be appropriate to exchange all kinds of attributes, a generic model was developed. BIM proved to be a severely limited solution. The disappointments came rapidly when attempts were made to implement systems based on the models of BIM. It appeared that the models were not adequate for use as a basis for building computer systems and this led to a significant loss of money.

From a functional perspective, this is not surprising. We think that the main reason for the failure of BIM was that the models were not goal-oriented and did not account for contextual influences. As a consequence the occurrence of functional equivalences was ignored. We have already described the object-types wall. From the object-type wall we can derive that a description based on the goal 'fire-safety' is completely different from a description of the same object-type wall from the goal 'susceptible to the formation of ettringite'. In both description totally different conditions or attributes are needed. There is no way to deduce one description from the other. Both object-types are incommensurable.

However, things are improving in computer science applied to the building and construction industry. In the world of Building and Construction Research DT's arc applied (see for instance Garrett & Fenves, 1987) and ideas that confirm the theory of functional object-types are slowly spreading. Examples of these ideas can be found in De Scheemaker (1994) and in Gui & Mäntylä (1994). De Scheemaker (1994) stressed the importance of describing a building from different viewpoints by accounting, for instance, for several parties interested in the building or for the life-cycle phases of the building. As a consequence De Scheemaker also declined the reconstruction of general models. Similar signals can be found in Gui & Mäntylä (1994) with respect to assembly modelling.
We hope that we have provided enough convincing arguments for the reader to agree with the scheme of Figure 8.1. It conveys the message that the theory of functional classifications offers promising perspectives to improve the development of knowledge-based systems, because it accounts for fundamental concerns of reconstructing object-types and objects. These knowledge-based systems are now known under the headings of expert database systems, decision support systems and deductive, object-oriented, logic databases or semantic databases. They all have in common that they can somehow cope with both types of knowledge to classify an object as an instance of an object-type. These systems, which inevitably have a complex nature due to functional equivalences, form a real integration of AI- and DB-systems. We hope that functional object-types will be their basis and that AKTS will be applied to reconstruct them.
REFERENCES


REFERENCES


REFERENCES


APPENDIX A

/* Table Drawing
The drawing of DT's in AKTS is done by the predicate draw/1. Draw/1 first performs a number of computations, constructs picture descriptions in the Graphic Description Language (GDL) and finally draws the table in a window, using a GDL-processing predicate. Basically, it works as follows. First of all the title, if any, is being placed by title/10. The predicate lines_and_label/12 places the condition and action names, i.e. the labels, together with the horizontal and some vertical lines. The information for the label width and the row heights is taken from the table_measures/3 term returned by get_table/7. The vertical lines between the fields of the table are being placed in fill_tree/10. The fields of the table, consisting of the condition alternatives and the action values, are processed by fill_tree/10. The predicate fill_tree/10 uses the predicates 'create_alts&line'/11 to place the alternative of the current field together with the vertical line separating it from the next field; and fill_action_values/10 to fill in the action values. The necessary information about field width is also drawn from the table_measures/3 term. The actual drawing is done by process_figures/3 which calls either add_pic/3 or record_pic/3, both GDL-processing predicates.

In the following, only the major predicates are preceded by a specification of their argument types, success and/or failure conditions and whether they are deterministic.
*/

/* draw(+Table) :-
Table is an atom.
Succeeds after drawing the table identified by Table.
Fails otherwise.
Deterministic.
*/

draw(Table) :-
    get_table(Table, _, Conditions, Actions, [Decison_trees],
    table_measures(Label_width, [Tree_width|Subwidth_trees],
    Row_heights), layout(Max_title, _, _, Font, Style, Size),
    test_layout(numbering, Nr_test),
    test_layout(titles, Title_test),
    test_layout(change, Layout_change),
    test_table_drawing(Firsttime, New_table),
    test_table(new_name, Name_change),
    font_height(Font_height),
    compute_label_width(Nr_test, Font, Style, Size, Width_labellnrs),
    Tablewidth is Label_width + 2 * Tree_width,
    X is Width_labellnrs,
compute_table_window(Firsttime, New_table, Nr_test, Title_test, Table, Title_height, Row_heights, Tablewidth, Width_labelnrs, Font_height),
length(Conditions, Nr_Conditions),
append(Conditions, Actions, Labels),
lines_and_label(Labels, Nr_test, Nr_Conditions, Table, Tablewidth, Label_width, Font_height, Row_heights, X, 0, 1, Fig2),
fill_tree(Decision_trees, Subwidth_trees, X_offset, Row_heights, Font_height, Nr_test, 1, Fig3),
flatten({Fig1, Fig2, Fig3, Table_figures}),
deall_all(Table),
refresh_now(Table),
scroll_drawing(Firsttime, Layout_change, Name_change, Title_test, Table, Title_height, Width_labelnrs),
process_figures(Layout_change, Table, Table_figures),
remember(first_time_drawing, no),
remember(name_change, no).

/* compute_table_window(+Firsttime, +New_table, +Nr_test, +Title_test, +Table, +Title_height, +Row_heights, +Tablewidth, +Width_labelnrs, +Font_height) :-
Firsttime, New_table, Nr_test and Title_test are atoms and either 'yes' or 'no'.
Table is an atom, representing the name of a table.
Title_height, Width_labelnrs and Font_height are integers.
Row_heights is a list of integers.
Succeeds after computing the minimal sizes for the window in which the table Table
will be drawn, considering the current screen size.
Deterministic. */

compute_table_window(no, _, _, _, _, _, _, _, _).
compute_table_window(yes, yes, yes, no, no, Title_test, Title_height, Reno_heights, Table_width, Width_labelnrs, Font_height).
compute_table_height(Row_heights, Font_height, Table_height),
compute_dxy(Nr_test, Title_test, Title_height, Width_labelnrs, Font_height, Dx, Dy),
Windowheight is Table_height + Dy + 35,
Windowwidth is Table_width + Dx + 35,
Max_windowheight is Screenheight - 45,
Max_windowwidth is Screenwidth - 6,
min(Windowheight, Max_windowheight, N_windowheight),
min(Windowwidth, Max_windowwidth, N_windowwidth),

- 254 -
compute_dy(+Nr_test, +Title_test, +Title_height, +Width_label_numbering, +Font_height, -Dx, -Dy) :-
    Nr_test and Title_test are atoms and either 'yes' or 'no'.
    Title_height, Width_label_numbering, Font_height, Dx and Dy are integers.
    Succeeds after computing the sizes Dx and Dy for the optional label numbers and title
    of a table respectively.
    Deterministic.
/*/ 

compute_dy(yes, yes, Title_height, Width_labelnrs, Font_height,
    Width_labelnrs, Dy) :-
    Dy is Title_height + Font_height + 7.
compute_dy(yes, no, _, Width_labelnrs, Font_height, Width_labelnrs,
    Dy) :-
    Dy is Font_height + 7.
compute_dy(no, yes, Title_height, _, _, 0, Title_height).
compute_dy(no, no, _, _, _, 0, 0).

compute_table_height(Row_heights, Font_height, Table_height):-
    compute_table_height(Row_heights, Font_height, 2, Table_height).
compute_table_height([], _, Table_height, Table_height).
compute_table_height([H_row_height,T_row_heights], Font_height, Accu,
    Table_height) :-
    NAccu is Accu + H_row_height * Font_height + 7,
    compute_table_height(T_row_heights, Font_height, NAccu, Table_height).

compute_label_width(no, _, _, _, 0).
compute_label_width(yes, Font, Style, Size, Width_labelnrs, -text_width('A99', Font, Style, Size, Width_numbers),
    Width_labelnrs is Width_numbers + 10,
    remember(width_label_numbering, Width_labelnrs).

/* title(+Title_test, +Table, +Max_title, +Font_height, +Font, +Style, +Size,
    +Width_labelnrs, -Figures, -Title_height) :-
Title_test is an atom and either 'yes' or 'no'.
Table is an atom representing the name of a table.
Max_title, Font_height, Style, Size, Width_labelnrs and Title_height are integers.
Font is an atom, representing a font.
Figures is a list with GDL-elements.
Succeeds after constructing the GDL-elements representing the title of Table and computing Title_height.
Deterministic.
*/
title(yes, Table, Max_title, Font_height, Font, Style, Size,
Width_labelnrs, [Title_box, Title_text], Title_height):-
Max_title_box is Max_title - 10,
text_width(Table, Font, Style, Size, Title_width),
min(Title_width, Max_title_box, Final_title_width),
Final_title_box_width is Final_title_width + 10 + 1,
lines_needed(Table, Final_title_width, Font, Style, Size,
Title_lines),
Title_height is Title_lines * Font_height + 7 + 1,
Y_offset is 1 - Title_height,
X_offset is - Width_labelnrs,
Title_box = box(Y_offset, X_offset, Title_height, Final_title_box_width),
write_text(Table, Font_height, Y_offset, X_offset, Title_lines,
Final_title_box_width, 0, Title_text),
succeed(retractall(title_box(Table, _))),
assert(title_box(Table, Title_box)).
title(no, Table, Max_title, Font_height, Font, Style, Size, _, {}, Title_height):-
Max_title_box is Max_title - 10,
text_width(Table, Font, Style, Size, Title_width),
min(Title_width, Max_title_box, Final_title_width),
Final_title_box_width is Final_title_width + 10 + 1,
lines_needed(Table, Final_title_width, Font, Style, Size,
Title_lines),
Title_height is Title_lines * Font_height + 7 + 1.

/* scroll_drawing(+Firsttime, +Layout_change, +Name_change, +Title_test, +Table, +Title_height, +Width_labelnrs) :-
Firsttime, Layout_change, Name_change and Title_test are atoms and either 'yes' or 'no'.
Table is an atom, representing the name of a table.
Title_height and Width_labelnrs are integers.
Succeeds after scrolling the drawing in such a way that the upperleft corner of the
Table is in the upperleft corner of the window.
Deterministic.
*/

scroll_drawing(no, no, no, _, _, _, _) :- !.
scroll_drawing(_, _, _, yes, Table, Title_height, Width_labelnrs) :-
X_scroll is -10 - Width_labelnrs,
Y_scroll is -10 - Title_height,
gscroll_to(Table, Y_scroll, X_scroll),
scroll_drawing(_, _, _, no, Table, Title_height, Width_labelnrs) :-
/* lines_and_label(+Labels, +Key, +Conditions, +Table, +Tablewidth, +Label_width, +Font_height, +Row_heights, +X, +Y, +Nr, -Figures) :-
Labels is a list of atoms.
Key is an atom and either 'yes' or 'no'.
Conditions, Tablewidth, Label_width, Font_height, X, Y and Nr are integers.
Table is an atom, representing the name of a table.
Row_heights is a list of integers.
Figures is a list with GDL-elements.
Succeeds after constructing the picture elements in GDL of the labels and most of the lines in Table.
Deterministic.
*/

lines_and_label([], yes, _, Table, Tablewidth, Label_width, Font_height, [], X, Y, _, Figures) :-
  !,
  succeed(retractall(right_down_corner(Table, _, _))),
  assert(right_down_corner(Table, Y, Tablewidth)),
  N_y is Y + Font_height + 7,
  remember(bottom_line, N_y),
  X_2nd_line is Label_width + 2,
  Figures = [line((Y,X), (Y,Tablewidth)),
             line((0,0), (Y,0)), line((0,Tablewidth), (N_y,Tablewidth)),
             line((0,Label_width), (N_y,Label_width)),
             line((0,X_2nd_line), (N_y,X_2nd_line)),
             line((N_y,X), (N_y,Tablewidth)), line((0,X), (N_y,X))].
line((0,0), (Y,0)), line((0,Tablewidth), (Y,Tablewidth)),
line((0,Label_width), (Y,Label_width)),
line((0,X_2nd_line), (Y,X_2nd_line))].

lines_and_label(Label, Key, 0, Table, Tablewidth, Label_width,
Font_height, Row_heights, X, Y, _, [line((Y, X), (Y,
Tablewidth))|T_figures}):-
% Conditions == 0
N_y is Y + 2,
!,
lines_and_label(Label, Key, -1, Table, Tablewidth, Label_width,
Font_height, Row_heights, X, N_y, 1, T_figures).
lines_and_label([Term|T_label], Key, T_Conditions, Table, Tablewidth,
Label_width, Font_height, [Row_height|T_row_heights], X, Y, Nr, Figures):-
number_label(Key, T_Conditions, Nr, X, Y, Row_height, Font_height,
N_nr, Nr_fig),
Figure1 = line((Y, X), (Y, Tablewidth)),
write_text(Term, Font_height, Y, 0, Row_height, Label_width, 0,
Figure1),
N_T_Conditions is T_Conditions - 1,
N_y is Y + Row_height * Font_height + 7,
( Key == yes :-
    Figures = [Figure1, Nr_fig, Figure2|T_figures]
  /* ELSE */
    Figures = [Figure1, Figure2|T_figures]
),
!,
lines_and_label(T_label, Key, N_T_Conditions, Table, Tablewidth,
Label_width, Font_height, T_row_heights, X, N_y, N_nr, T_figures).

number_label(no, _, _, _, _, _, _, _, _, _).
number_label(yes, Nr_Conditions, Nr, X, Y, Row_height, Font_height,
N_nr, Nr_fig) :-
determine_label_character(Nr_Conditions, Character),
concat(Character, Nr, Nrterm),
Nr_width is -X,
write_text(Nrterm, Font_height, Y, X, Row_height, Nr_width, 0,
Nr_fig),
N_nr is Nr + 1.

determine_label_character(Conditions, 'C') :-
    Conditions > 0,
!,
determine_label_character(_, 'A').
APPENDIX

/* fill_tree(+Decision_tree, +Width_tree, +X, +Y, +Heights, +Font_height, +Key, +Nr, -N_nr, -Figures):-
Decision_tree is a nested list.
Width_tree is a nested list of integers.
X, Y, Font_height, Nr and N_nr are integers.
Heights is a list of integers.
Key is an atom and either 'yes' or 'no'.
Figures is a list with GDL-elements.
Succeeds after computing the total number of rules in the table and after constructing
the picture elements in GDL for the fields of the table. The contents of the fields are
represented in Decision_tree, the width of each field in Width_tree, and the height of
each row in Heights.
Deterministic. */

fill_tree([], [], _, _, _, _, _, Nr, Nr, []).
fill_tree([N (Dec_trees)], T_widthtrees, X, Y, [H_height], T_height, Key, Nr1, Nr2, [Fig1, Fig2, Actionfig'), T_fig]) :-
  !,
  'create_altsaline'(Alts, Font_height, Y, X, H_height, Width, 2, Ny, Nr, Fig1, Fig2),
fill_action_values(Action_values, Width, Font_height, Key, Nr1, X, Ny, T_height, Nr2, Actionfig),
fill_tree(T_dec_trees, T_widthtrees, Nx, Y, [H_height], T_height, Font_height, Key, Nr2, N_nr, T_fig).
fill_tree([N (Subtrees)], T_dec_trees),
  T_widthtrees, X, Y, [H_height], T_height, Key, Nr1, N_nr, [Fig1, Fig2, Subfig'], T_fig) :-
  !,
  'create_altsaline'(Alts, Font_height, Y, X, H_height, Width, 0, Ny, Nr, Fig1, Fig2),
fill_tree(Subtrees, Subwidth, X, Ny, T_height, Font_height, Key, Nr1, Nr2, Subfig),
fill_tree(T_dec_trees, T_widthtrees, Nx, Y, [H_height], T_height, Font_height, Key, Nr2, N_nr, T_fig).

fill_action_values([], Columnwidth, Font_height, yes, Nr, X, Y, [], N_nr, [Figure]) :-
  !,
concat('R', Nr, Numberterm),
write_text(Numberterm, Font_height, Y, X, 1, Columnwidth, 1, Figure),
N_nr is Nr + 1.
fill_action_values([], _, _, _, Nr, _, _, [], Nr, []).
fill_action_values([N (Action_value)], T_action_values), Columnwidth, Font_height, Key, Nr, X, Y, [Row_height], T_row_heights, N_nr, [Figure], T_figures) :-
write_text(Actionvalue, Font_height, Y, X, Row_height, Columnwidth, 1, Figure),
N_y is Y + Row_height * Font_height + 7,
fill_action_values(T_action_values, Columnwidth, Font_height, Key, Nr,
X, N_y, T_row_heights, N_nr, T_figures).

/* 'create_alts&line'(Alts, +Font_height, +Y, +X, +Height, +Width, +Double, -Ny,
-Nx, -Fig1, -Fig2):-
Alts is a list of atoms.
Font_height, Y, X, Height, Width, Double, Ny and Nx are integers.
Fig1 and Fig2 are GDL-elements.
Succeeds after computing new Ny and Nx coordinates and after constructing the
GDL-elements for the contents of a field and for the separating line between the
current field and the next.
Deterministic.
*/

'create_alts&line'([H_alt, T_alt], Font_height, Y, X, Height, Width,
Double, Ny, Nx, Fig1, Fig2),
alternatives_string([H_alt, T_alt], String),
write_text(String, Font_height, Y, X, Height, Width, 1, Fig1),
division_line(Y, X, Width, Fig2),
Ny is Y + Height * Font_height + 7 + Double,
Nx is X + Width,
!
alternatives_string(Alts, String) :-
alternatives_string(Alts, '', String).

alternatives_string([Alt], Accu, String):-
concat([Accu, Alt], String),
!.
alternatives_string([H_alt, T_alt], Accu, String):-
concat([Accu, H_alt, ' OF ', NAccu],
alternatives_string(T_alt, NAccu, String).

division_line(Y, X_offset, Tree_width, line((Y,X), (Bottom_line,X))) :-
X is X_offset + Tree_width,
recall(bottom_line, Bottom_line).

write_text('', Font_height, Y_offset, X_offset, Height, Width,
Alignment, Figure) :-
!,
construct_textbox('', Font_height, Y_offset, X_offset, Height, Width,
Alignment, Figure),
write_text(Text, Font_height, Y_offset, X_offset, Height, Width,
Alignment, Figure) :-
    construct_textbox(Text, Font_height, Y_offset, X_offset, Height, Width, Alignment, Figure).

construct_textbox(Text, Font_height, Y_offset, X_offset, Height, Width, Alignment, Figure) :-
    Top is Y_offset + 4,
    Left is X_offset + 5,
    Fieldheight is Height * Font_height,
    Fieldwidth is Width - 10,
    front_table(Table),
    wfont(Table, Font, Face, Size),
    Figure = textbox(Font, Size, Face, Top, Left, Fieldheight, Fieldwidth,
        Alignment, Text).

test_layout(numbering, yes) :-
    marked_item('Layout Settings', 'Numbering'), !.

test_layout(titles, yes) :-
    marked_item('Layout Settings', 'Titles'), !.

test_layout(change, yes) :-
    recall(layout_change, yes), !.

test_layout(Layout, no) :-
    atom(Layout).

test_table(new, yes) :-
    recall(new_table, yes), !.

test_table(new_name, yes) :-
    recall(name_change, yes), !.

test_table(Aspect, no) :-
    atom(Aspect).

test_table_drawing(yes, New_table) :-
    recall(first_time_drawing, yes),
    test_table(new, New_table), !.

test_table_drawing(no, New_table) :-
    test_table(new, New_table), !.
SUMMARY

Chapter 1. A central problem in the development of knowledge-based systems is the process of modelling knowledge. To deal with this problem, we characterise knowledge as the competence of matching object-types and objects. In this way, the process of modelling knowledge is reduced to reconstructing these object-types and objects. Since object-types and objects display a high complexity, this characterisation is closely connected with the recognition that integrating Artificial Intelligence (AI) and Database Technology (DBT) constitutes an essential step to cope with the complexities of object-types and objects. Three dimensions underlying the problem of reconstructing object-types and objects are addressed: (1) confusion of knowledge and knowledge representation formalisms, (2) lack of an adequate theory of the nature of knowledge and (3) lack of an adequate formal language. Attending to these dimensions leads to the following research issues assessing the value of:

I. The integration of AI and DBT at the knowledge level
II. The theory of functional object-types as a theory of the nature of knowledge to accomplish a knowledge level integration
III. The joint application of Decision Tables (DT's) and Prolog as a formal language

1. THE INTEGRATION OF AI AND DBT AT THE KNOWLEDGE LEVEL

Chapter 2. To oppose confusing knowledge and knowledge representation and the minor role of mathematical logic, Newell introduces the knowledge level. We investigate Newell's claim that the distinction of the knowledge level leads to a comprehensive and consistent view of knowledge and knowledge representation and helps to assign to mathematical logic the role it deserves.

Newell views the knowledge level as a separate computer systems level just as the device level, the circuit level, the register-transfer level and the symbol level. The introduction of the knowledge level is primarily intended to have a separate computer systems level for the definition of the nature of knowledge. Though Newell views knowledge as a competence to select actions to realise goals, accomplished by an intelligent system called the agent, and though Newell provides three principles to define the nature of knowledge (the Principle of Rationality, the Principle of Equipotence of Acceptable Actions and the Principle of Joint Goal Satisfaction), our opinion is that his theory should not be viewed as a theory of the nature of knowledge, but as one describing the knowledge level. Two main elements prevail in Newell's characterisation of the knowledge level; (1) at the knowledge level, the explicit aim to analyse the knowledge of a computer system without any reference to knowledge
representation formalisms or user interface issues and (2) mathematical logic playing a key role as a representation formalism uniquely suited to the analysis of knowledge.

The investigations in this chapter deal with advantages and disadvantages claimed for the knowledge level. The outcome is that the identification of the knowledge level yields important advantages, so that we can confirm Newell's claim. We point out the advantages of a specific computer systems level to define the nature of knowledge, the reconstruction of implementation-free knowledge level models, the specification of the role of mathematical logic and the possibilities of examining and comparing representation formalisms.

Though Newell's theory is an important step forward, much work remains to be done for the integration of AI and DBT. Two points deserve special attention. The first is attempting to shift the emphasis away from representation issues and to redirect it to knowledge. The second point is the role of mathematical logic as a representation formalism appropriate for the analysis of knowledge. Within the scope of integrating AI and DBT we examine both points in Chapter 3.

Chapter 3. Though the integration of AI and DBT can be studied in various ways, in Chapter 3 two basic strategies are distinguished: a symbol level strategy concentrating on transferring data structures and processes from AI to DBT and vice versa and a knowledge level strategy concentrating on the knowledge present in an AI- or DB-system. In a symbol level strategy a distinction is made between AI- and DB-systems, whereas in a knowledge level strategy this distinction is absent. Distinctive symbol level features in processes such as the deductive proof-theoretic inferencing of AI-systems versus the model-theoretic query evaluation of DB-systems, and in structures such as production rules versus records, are ignored in exchange for an explicit focus on knowledge.

Analysis of symbol level strategies reveals that the symbol level approach interferes with the process of modelling knowledge. The main cause -confusion of knowledge and knowledge representation- becomes manifest in the use of representation formalisms to model knowledge without any serious arguments for doing so.

Instead of stressing differences a knowledge level integration emphasises a deep and significant commonality of AI- and DB-systems stemming from a fundamental common concern about knowledge. Besides the focus on knowledge and the reduced role of representation formalisms, a knowledge level strategy is characterised by the major role reserved for mathematical logic. It is the basic language for formal definition of knowledge in a knowledge universe. As we explained, a knowledge universe is a set of knowledge states. The range of a knowledge state is a set of knowledge tables describing object-types or objects. A knowledge universe can be defined by the description, in logical order, of the knowledge schema, the variable constraints, the inter-variable constraints, the knowledge table constraints and the knowledge universe constraints.

To illustrate one of the advantages of a knowledge level integration, we perform a knowledge level evaluation of a data base using a system of mathematical functions describing a knowledge universe. We conclude that such a system of mathematical functions describing a knowledge universe is convenient, but lacks a theory of the nature of knowledge to take care of the organisation of the mathematical functions.
II. THE THEORY OF FUNCTIONAL OBJECT-TYPES TO ACCOMPLISH A KNOWLEDGE LEVEL INTEGRATION

Chapter 4. As the reconstruction of a knowledge universe consists of defining object-types of a problem domain, theories of the nature of object-types play an indispensable role. The probabilistic theory, subscribes to the classical idea that an object-type is a set of sufficient and necessary conditions, but exclusively on a theoretical level. The probabilistic approach assumes that all sorts of random disturbances at the empirical level cause problems in the delineation of the extension (fuzzy sets). By using mathematical measures of similarity between objects, defined over an essentially a priori given set of attributes, the probabilist tries to eliminate the random disturbances, so that univocal criteria can be proved to underlie the fuzzy extension at the theoretical level.

In the prototypical or stereotypical theory, object-types are described by means of a prototype. A prototype shares many attributes of objects so it reflects a central tendency category of objects. The description of a prototype consists of so-called necessary conditions. Since no object will satisfy all the necessary conditions, the answer to the question whether an object belongs to the extension of an object-type depends on the degree of resemblance with the prototype. Inevitably, the delimitation of the extension is fuzzy, but this vagueness is not ascribed to empirical disturbances as in the probabilistic view, but to reality which does not let itself be categorised univocally. Probabilistic and prototype conceptualisation methods have much in common and prevail in AI and DBT.

In the theory of functional classifications, the reconstruction of an object-type takes place through a goal- or function-controlled process. What is essentially different is that the functional theory offers a totally different explanation of fuzziness. In contrast to the probabilistic and prototype theory, the functional theory emphasises that fuzziness has a systematic character. The solution of fuzziness is neither sought in the elimination of random disturbances (such as measuring errors), nor in the comparison of objects with a prototypical object-type. The functional solution is typified by the systematic identification of several object-types. These object-types originate through functional equivalence: the phenomenon that objects, possibly differing in many respects, are equivalent in achieving a nominally specified function in a certain context. Consequently object-types are neither abstracted from extensions nor described through prototypes.

Furthermore, analysis of the current practice of reconstructing knowledge universa which is mainly based on probabilistic and prototypical assumptions, uncovers deficiencies in dealing with functional equivalences. To round off the chapter we again perform a knowledge level evaluation of the record-based representation formalism, but this time the theory of functional object-types is used. The evaluation shows a number of weaknesses of the record-based formalisms. If the essential configuration of the conjunct sets leading to the same goal is characterised by identical attributes, each attribute having the same kind of values, records are excellent representation and processing tools. If, on the other hand, these conjunct sets are characterised by heterogeneity caused by having to classify objects in a goal-oriented fashion accounting for different descriptors and new conceptual interactions, records are not appropriate.
III. THE JOINT APPLICATION OF DT'S AND PROLOG

Chapter 5. The central aim of this chapter is to assess to what extent the joint application of DT's and Prolog is useful as a conceptual modelling language for functional object-types. From this perspective, we analyse the strong and weak points of DT's and Prolog separately with special concern for the representation, reconstruction, validation and simulation of functional object-types. The analysis is based on (1) formal definitions (amended by us) of DT's and (2) the formal background of Prolog.

Our analysis shows that DT's and Prolog can compensate their mutual weak points to a considerable degree. DT's have deficiencies in representing functional object-types that require recursive or small definitions that are no problem at all in Prolog. DT's stimulate the reconstruction of functional object-types, while Prolog hardly enforces users to reconstruct functional object-types and generally lacks a modelling methodology. Prolog offers no facilities to validate functional object-types on completeness, consistency and correctness. DT's, on the other hand, provide extensive facilities for these validation purposes. Finally, functional object-types represented in DT's are not executable, which limits their simulation facilities. In contrast, functional object-types represented in Prolog are executable and this yields extensive simulation facilities. The general conclusion is that DT's and Prolog are complementary and that their joint application yields a powerful modelling language, especially for functional object-types.

Chapter 6. Despite the high complementarity of DT's and Prolog, our analysis also reveals that their joint application still yields a language with certain drawbacks. The language does, for instance, not offer facilities for automated validation and automated simulation when DT's are employed. Furthermore, essential graphical facilities for drawing DT's are lacking. We attempt to take advantage of the complementarity of DT's and Prolog and eliminate the drawbacks by developing a computer-based knowledge modelling tool: the Advanced Knowledge Transfer System (AKTS). AKTS provides functions to reconstruct, design and simulate knowledge universa in line with the theory of functional object-types by integrating DT's and Prolog. The Integrity Control Sub-system, the Inference-Machine and the Graphical Decision Table Editor respectively offer facilities for automated validation, automated simulation and drawing DT's. Consequently, it is concluded that AKTS does not only take advantage of the complementarity of DT's and Prolog, but also removes the remaining drawbacks.

Chapter 7. In this chapter the methodological argumentation of the thesis is exemplified with a case-study from the field of chemical degradation and restoration of ancient brick masonry walls. The case-study specifically deals with sulphate salt reactions causing the formation of ettringite, a specific form of chemical degradation. It refers to the development of a knowledge-based system for the diagnosis whether (1) brick masonry walls are susceptible to the formation of ettringite and how (2) to restore brick masonry walls such that no negative interactions between the restoration materials and the original materials occur. Since we view knowledge as a competence
of matching object-types and objects, this requires equipping the system with object­
types that describe walls susceptible to the formation of ettringite and objects that
describe the brick masonry walls under consideration.

The case-study exemplifies three main methodological points. First, by
concentrating on knowledge level descriptions of these object-types and objects, it
illustrates the advantages of a knowledge level integration of AI and DBT. Besides
the fading distinction and interaction between object-types and objects, it confirms the
value of knowledge level descriptions of object-types and objects: a better view of
knowledge, avoidance of biases toward specific representation formalisms and
improved explanation and maintenance.

Second, the case-study reveals the advantages of the theory of functional
classifications in comparison with probabilistic and prototypical theories to
accomplish a knowledge level integration. Many classifications of brick masonry
walls are possible on the basis of the original materials. However, the reconstruction
of functional object-types is conducive to more effective classifications of brick
masonry walls. At the symbol level, functional object-types foster the transfer of
knowledge to representation formalisms made available by AI and DBT and simplify
the evaluation of these representation formalisms by means of mathematical logic.

Third, the utility of the joint application of DT's and Prolog, especially in the form
they are offered by AKTS, is exemplified. Together, DT's and Prolog have the
capability of describing object-types that delineate walls susceptible to the formation
of ettringite and objects that delineate brick masonry walls. Besides the fact that
Prolog can be used for recursive definitions, the case, in addition, reveals the utility of
Prolog as a database language. By this, the expressive power of AKTS is emphasised.

The main conclusions of the thesis are that (1) an integration of AI and DBT at the
knowledge level helps to deal with the problem of modelling knowledge (2) the
theory of functional object-types significantly contributes to such a knowledge level
integration and that (3) the joint application of DT's and Prolog does not only provide
facilities for reconstructing and representing functional object-types, but also for
validation and simulation purposes.

Chapter 8. Here a broader perspective is provided together with a discussion about
implications of adopting the proposals made in the thesis.
SAMENVATTING

Hoofdstuk 1. Een centraal probleem bij de ontwikkeling van kennis-gebaseerde systemen vormt het proces van kennis-modellering. Om dit probleem hanteerbaar te maken, omschrijven we kennis als het vermogen om object-typen en objecten te matchen. Het proces van kennis-modellering is hiermee herleid tot het reconstrueren van deze object-typen en objecten. Omdat object-typen en objecten een hoge mate van complexiteit vertonen, is deze omschrijving nauw verbonden met de erkenning dat de integratie van Artificial Intelligence (AI) en Database Technologie (DBT) een essentiële stap vormt om de complexiteit van object-typen en objecten het hoofd te kunnen te kunnen bieden. Drie dimensies liggen ten grondslag aan het reconstructieprobleem van object-typen en objecten: (1) de verwarring van kennis en kennisrepresentatie-formalismen, (2) het ontbreken van een adequate theorie betreffende de aard van kennis en (3) het ontbreken van een adequate formele taal. Het centraal stellen van deze drie dimensies leidt tot drie onderzoeksthema's, gericht op het vaststellen van de waarde van:

I. De integratie van AI en DBT op het kennisniveau
II. De theorie van functionele object-typen als een theorie over de aard van kennis gericht op het tot stand brengen van een integratie op het kennisniveau
III. De gezamenlijke toepassing van beslissingstabellen (BT's) en Prolog als formele taal.

I. DE INTEGRATIE VAN AI EN DBT OP HET KENNISNIVEAU

Hoofdstuk 2. Newell introduceert het kennisniveau om het verwarren van kennis en kennisrepresentatie en de ondergeschikte rol van mathematische logica tegen te gaan. We onderzoeken Newell's claim dat het onderscheiden van het kennisniveau leidt tot een inzichtelijke en consistent kijk op kennis en kennisrepresentatie en helpt bij het toekennen van een passende rol aan de mathematische logica.

Newell beschouwt het kennisniveau als een afzonderlijk computersysteem niveau vergelijkbaar met het instrumentele niveau, het circuit niveau, het registratie-transfer niveau en het symbool niveau. De introductie van het kennisniveau is primair bedoeld om een afzonderlijk computersysteem niveau te hebben ten behoeve van de definiëring van de aard van kennis. Hoewel Newell kennis beschouwt als een, door een intelligent systeem - de agent -, bewerkstelligde competentie acties te selecteren ter realisatie van doelen, en hoewel Newell drie principes formuleert om de aard van kennis te definiëren (het Rationaliteitsprincipe, het Principe van Equipotentie van Acceptabele Acties en het Principe van Gezamenlijke Doel Satisfactie), zijn we van mening dat zijn theorie niet beschouwd moet worden als een theorie over de aard van

- 269 -
kennis, maar als een theorie die het kennisniveau beschrijft. Twee belangrijke elementen treden in Newell's beschrijving van het kennisniveau op de voorgrond: (1) het expliciete doel op het kennisniveau de kennis van een computersysteem te analyseren zonder verwijzing naar kennisrepresentatie-formalismen of zaken het user interface betreffende en (2) mathematische logica die een sleutelrol speelt als een representatie-formalisme dat uitzonderlijk geschikt is voor de analyse van kennis.

Het onderzoek in dit hoofdstuk heeft betrekking op de geclaimde voor- en nadelen van het kennisniveau. De conclusie is dat de identificatie van het kennisniveau belangrijke voordelen heeft zodat we Newell's claim kunnen bevestigen. We wijzen op de voordelen van een specifiek computersysteem niveau om de aard van kennis, de reconstructie van implementatie-vrije kennisniveau-modellen en de specificatie van de rol van mathematische logica te definiëren en de mogelijkheden van representatie-formalismen te onderzoeken en te vergelijken.

Hoewel Newell's theorie een belangrijke stap voorwaarts inhoudt, vergt de integratie van AI en DBT nog veel inspanning. Twee aspecten verdienen speciale aandacht, ten eerste de poging om de nadruk op representatie aspecten te verminderen en te verschuiven in de richting van kennis. Het tweede aspect betreft de rol van mathematische logica als een geschikt representatie-formalisme voor de analyse van kennis. In Hoofdstuk 3 besteden we -binnen het kader van de integratie van AI en DBT- aandacht aan beide aspecten.


Analyse van symboolniveau-strategieën maakt duidelijk dat de symboolniveau- benadering interfereert met het proces van kennis-modellering. De hoofdoorzaak -verwarring van kennis en kennisrepresentatie- wordt manifest bij het gebruik van representatie-formalismen om kennis te modelleren zonder steekhoudende argumenten te hebben voor deze handelwijze.

Een kennisniveau-integratie benadrukt niet de verschillen tussen maar de significante overeenkomsten van AI- en DB-systemen. Deze overeenkomsten komt voort uit een fundamentele gemeenschappelijke gerichtheid op kennis. Naast de nadruk op kennis en de gereduceerde rol van representatie-formalismen, kenmerkt de kennisniveau-strategie zich door de belangrijke rol die zij toekent aan mathematische logica. Het is de basistaal voor de formele definitie van kennis in een kennisuniversum. Een kennisuniversum omvat een set kennistoestanden. Het bereik van een kennistoestand wordt gevormd door een set kennistabellen die object-typen of objecten beschrijven. Een kennisuniversum kan gedefinieerd worden door de
beschrijving, in logische volgorde, van het kennisschema, de variabele restricties, de inter-variabele restricties, de kennistabel restricties en de kennisuniversum restricties.

Ter illustratie van een van de voordelen van een kennisniveau-integratie, evalueren we een database met behulp van een systeem van mathematische functies voor het beschrijven van een kennisuniversum. We concluderen dat een dergelijk systeem van mathematische functies voor de beschrijving van een kennisuniversum handig is, maar dat een theorie over de aard van kennis ontbreekt; een theorie die zorgt voor de organisatie van de mathematische functies.

II. DE THEORIE VAN FUNCTIONELE OBJECT-TYPEN VOOR HET REALISEREN VAN INTEGRATIE OP KENNISNIVEAU

Hoofdstuk 4. Aangezien de reconstructie van een kennisuniversum bestaat uit de definiering van object-typen van een probleemdomein, zijn theorieën over de aard van object-typen onmisbaar. De probabilistische theorie onderschrijft, uitsluitend op een theoretisch niveau, het klassieke idee dat een object-type beschreven kan worden door middel van een set voldoende en noodzakelijke voorwaarden. De probabilistische benadering veronderstelt dat allerlei soorten random verstoringen op empirisch niveau problemen veroorzaken bij de afbakening van de extensie (fuzzy sets). Door gebruik te maken van mathematische maten van gelijkheid tussen objecten, gedefinieerd over een in hoofdzaak a priori gegeven set van attributen, probeert de aanhanger van de probabilistische theorie de random verstoringen te elimineren, zodat bewezen kan worden dat op theoretisch niveau eenduidige criteria ten grondslag liggen aan de fuzzy extensie.

In de prototypische of stereotypische theorie worden object-typen beschreven door middel van een prototype. Doordat een prototype veel overeenkomstige eigenschappen van objecten bezit, vertegenwoordigt het een 'centrale tendentie' exemplaar in de categorie van objecten. De beschrijving van een prototype bestaat uit zogenaamde noodzakelijke condities. Aangezien geen enkel object zal voldoen aan alle noodzakelijke condities, hangt het antwoord op de vraag of een object behoort tot de extensie van een object-type af van de mate van gelijkenis dat het object vertoont met het prototype. De afbakening van de extensie is onvermijdelijk fuzzy. Deze fuzziness wordt niet toegeschreven aan empirische verstoringen zoals in de probabilistische benadering, maar aan de realiteit die zich niet eenduidig laat categoriseren. Probabilistische en prototypische conceptualiseringsmethoden hebben veel gemeen en overheersen in de AI en de DBT.

In de theorie van functionele classificaties vindt de reconstructie van een objecttype plaats middels een doel- of functie-gecontroleerd proces. Een essentieel verschil is dat de functionele theorie een totaal andere verklaring biedt voor fuzziness. In tegenstelling tot de probabilistische en prototypische theorie, benadrukt de functionele theorie dat fuzziness een systematisch karakter heeft. De oplossing voor fuzziness wordt derhalve niet gezocht in de eliminatie van random verstoringen, noch in de vergelijking van objecten met een prototypisch object-type. De functionele oplossing wordt gekenmerkt door de systematische identificatie van meerdere object-typen.
Deze object-typen komen voort uit *functionele equivalentie*: het verschijnsel dat objecten die mogelijk in velerlei opzicht verschillen, identiek zijn in het vervullen van een nominaal gespecificeerde functie in een bepaalde context. Object-typen worden derhalve niet afgeleid van extensies noch beschreven door prototypen.

Analyse van de huidige praktijk van het reconstrueren van kennisuniversa, welke voornamelijk gebaseerd is op assumpties uit de probabilistische en prototypische theorie, onthult bovendien de gebreken in het hanteren van functionele equivalenties. Ter afronding van het hoofdstuk evalueren we opnieuw de record-gebaseerde representatieformalismen vanuit het kennisniveau-perspectief. Ditmaal wordt de theorie van functionele object-typen gebruikt. De evaluatie laat een aantal zwakke punten van de record-gebaseerde formalismen zien. Als de essentiële configuratie van conjuncte sets leidend naar hetzelfde doel, gekenmerkt wordt door identieke eigenschappen, waarbij elke eigenschap dezelfde soort waarden heeft, zijn records uitstekende representatie- en processing tools. Als deze conjuncte sets daarentegen worden gekenmerkt door heterogeniteit, veroorzaakt doordat objecten op een doelgerichte manier geclassificeerd moeten worden rekening houdend met verschillende descriptors en nieuwe conceptuele interacties, dan zijn records niet geschikt.

### III. DE GEZAMENLIJKE TOEPASSING VAN BT'S EN PROLOG

*Hoofdstuk 5.* De beoordeling in welke mate de gezamenlijke toepassing van BT's en Prolog bruikbaar is als een conceptuele modellertaal voor functionele object-typen vormt de voornaamste doelstelling in dit hoofdstuk. Vanuit dit perspectief, analyseren we de sterke en zwakke punten van BT's en Prolog afzonderlijk, met speciale aandacht voor de representatie, reconstructie, validering en simulatie van functionele object-typen. De analyse is gebaseerd op (1) formele (door ons geamendeerde) definities van BT's en (2) de formele achtergrond van Prolog.

De analyse toont aan dat BT's en Prolog elkaars zwakke punten in hoge mate kunnen compenseren. BT's hebben tekorten bij de representatie van functionele object-typen waarvoor recursieve of kleine definities vereist zijn. Voor Prolog vormen dit type definities geen enkel probleem. BT's stimuleren de reconstructie van functionele object-typen, terwijl Prolog haar gebruikers nauwelijks stimuleert functionele object-typen te reconstrueren en in het algemeen een methodologie van modelleren ontbreekt. Prolog biedt geen faciliteiten om functionele object-typen te valideren op volledigheid, consistentie en correctheid. BT's daarentegen, bieden uitgebreide faciliteiten voor deze valideringsdoeleinden. Tot slot zijn door BT's gerepresenteerde functionele object-typen, niet executable. Dit beperkt de simulatiefaciliteiten van BT's. Functionele object-typen gerepresenteerd in Prolog, aan de andere kant, zijn executable; hetgeen leidt tot een ruim aanbod van simulatiefaciliteiten. De algemene conclusie is dat BT's en Prolog complementair zijn en dat hun gezamenlijke toepassing een krachtige modellertaal oplevert, met name voor functionele object-typen.

*Hoofdstuk 6.* Onze analyse laat, ondanks de grote complementariteit van BT's en
Prolog, zien dat hun gezamenlijke toepassing nog steeds een taal oplevert met een aantal schaduwzijden. De taal biedt bijvoorbeeld geen faciliteiten voor geautomatiseerde validering en geautomatiseerde simulatie als BT's worden gebruikt. Daarnaast ontbreken essentiële grafische faciliteiten voor het tekenen van BT's. Door de ontwikkeling van een computer gebaseerde kennis-modelleren tool: de Advanced Knowledge Transfer System (AKTS) trachten we de voordelen van de complementariteit van BT's en Prolog te benutten en de nadeelen op te heffen. AKTS biedt, overeenkomstig de theorie van functionele object-typen, functies om kennisuniversa te reconstrueren, te ontwerpen en te simuleren door BT's en Prolog te integreren. Het Integriteitscontrole Sub-systeem, de Inferentie Machine en de Grafische Beslissingstabel Editor bieden faciliteiten voor respectievelijk geautomatiseerde validering, geautomatiseerde simulatie en voor het tekenen van BT's. Derhalve kan geconcludeerd worden dat AKTS niet alleen de voordelen van de complementariteit van BT's en Prolog benut maar ook de resterende bezwaren opheft.

Hoofdstuk 7. In dit hoofdstuk wordt de methodologische argumentatie van het proefschrift geïllustreerd aan de hand van een case-study op het gebied van de chemische degradatie en restauratie van historische, gemetselde stenen muren. De case-study richt zich speciaal op sulfaatreacties die leiden tot de vorming van ettringiet, een specifieke vorm van chemische degradatie. De studie heeft betrekking op de ontwikkeling van een kennis-gebaseerd systeem voor de diagnose of (1) gemetselde stenen muren gevoelig zijn voor de vorming van ettringiet en hoe (2) gemetselde stenen muren te restaureren opdat geen negatieve interacties optreden tussen de restauratie-materialen en de oorspronkelijke materialen. Aangezien we kennis beschouwen als het vermogen om object-typen en objecten te matchen, vereist dit dat het systeem uitgerust wordt met object-typen die muren beschrijven welke gevoelig zijn voor de vorming van ettringiet en met objecten die de gemetselde stenen muren beschrijven die voorwerp van onderzoek zijn.


Ten tweede maakt de case-study de voordelen van de theorie van functionele classificatie voor de totstandkoming van een kennisniveau-integratie zichtbaar in vergelijking met probabilistische en prototypische theorieën. Op basis van de oorspronkelijke materialen zijn vele classificaties van gemetselde stenen muren mogelijk. De reconstructie van functionele object-typen draagt echter bij aan een effectievere classificatie van gemetselde stenen muren. Op het symboolniveau ondersteunen functionele object-typen de transfer van kennis naar representatie-formalismen, zoals beschikbaar gesteld door de AI en de DBT en vergemakkelijken zij de evaluatie van deze representatie-formalismen door middel van mathematische logica.

Ten derde wordt de bruikbaarheid van de gezamenlijke toepassing van BT's en
Prolog, met name in de vorm zoals deze door AKTS geboden wordt, geïllustreerd. BT's en Prolog hebben gezamenlijk het vermogen object-typen te beschrijven die muren definiëren die gevoelig zijn voor de vorming van ettringiet en objecten te beschrijven die gemetselde stenen muren definiëren. Naast het feit dat Prolog gebruikt kan worden voor recursieve definities, toont de case-study bovendien de bruikbaarheid aan van Prolog als database-taal. Dit benadrukt de expressieve kracht van AKTS.

De hoofdconclusies van het proefschrift zijn dat (1) integratie van AI en DBT op het kennisniveau ondersteunend werkt bij het hanteren van het probleem van het modelleren van kennis (2) de theorie van functionele object-typen significant bijdraagt tot een dergelijke kennisniveau-integratie en dat (3) de gezamenlijke toepassing van BT's en Prolog niet alleen faciliteiten biedt voor de reconstructie en representatie van functionele object-typen, maar ook voor validerings- en simulatiedoeleinden.

_Hoofdstuk 8_. Tot slot schetsen we een breder perspectief en bediscussiëren we enkele implicaties van de in dit proefschrift gedane voorstellen.
CURRICULUM VITAE

The author was born on April 1960 in the city of Rheden. He completed grammar school in 1978 and afterwards studied in Tilburg and Nijmegen where he graduated in 1988 on the subject of theories, methods and techniques of developing knowledge-based systems. From 1988-1989 the author was employed as a computer scientist at the University of Nijmegen. From 1989 until 1991 he was employed at the Institute of Applied Computer Science of the Netherlands Organization of Applied Scientific Research (Expertise Centre Knowledge-Based Systems). From 1991 the author is employed at the Institute of Building and Construction research of the Netherlands Organization of Applied Scientific Research (Advisory Group Knowledge-Based Systems).
THESES ADDED TO THE DISSERTATION 'FUNCTIONAL OBJECT-TYPES AS A FOUNDATION OF COMPLEX KNOWLEDGE-BASED SYSTEMS'

Thesis 1
Newell's idea about the theory of the knowledge level providing 'a definition of representation, namely, a symbol system that encodes a body of knowledge.' and not 'a theory of representation...' (Newell, 1981, p.14) is inconsistent with Brachman and Levesque's view about a knowledge level perspective allowing us 'to examine computational properties of representational formalisms that will continue to hold no matter what Symbol Level decisions are made.' (Brachman & Levesque, 1986, p.77)

Thesis 2
When Hull and King remark that 'little work has been directed at providing methodological support for selecting an appropriate semantic model or for integrating the various modeling capabilities found in semantic models.' (Hull & King, 1987, p.211), they thereby stress the importance in Database Technology of a knowledge level perspective.

Thesis 3
Davis (1988) is right when exposing the deficiencies of text in a Software Requirements Specification (SRS) as follows: 'While I am not opposed to large SRSs, I am opposed to large SRSs written in natural language. This is analogous to building a 100-story skyscraper. I am not opposed to 100-story office buildings, just to those entirely constructed of wood. The solution to the skyscraper problem is simple: use steel. The solution to the SRS problem is also simple: use a formal technique. Nevertheless, wood does have its proper place, even in a 100-story building. And natural language does have its proper place in an SRS. Where do you apply a formal technique and where do you apply something else? The answer is: use a formal technique when you cannot afford to have the requirement misunderstood.' (Davis, 1988, p.1100)

Thesis 4
'Users don't know what they want; and often, when they do, they do not need what they want.' (Kowalski, 1986, p.94)

Thesis 5
People who are able to solve cryptograms are probably bad knowledge engineers.

Thesis 6
Looking up the translation of Dutch 'kennisbank' in 'van Dale', we find: 'knowledge bank, data bank'. From this we can conclude that 'van Dale' follows a knowledge level approach: they do not distinguish a knowledge bank from a data bank.
The intended reduction of subsidies to technological institutes in the Netherlands not only contrasts with government policy of other European countries but also with the Dutch Government wish to use budget windfalls to create more jobs.

Representation of the Technical University of Eindhoven promotion regulations in a Decision Table system and Prolog, using the Advanced Knowledge Transfer System would significantly reduce the time spent in setting up a promotion commission and assessing when to hand in what and in what way.

When life-expectancy for women has decreased to that for men, women will have attained their ultimate emancipation.

A logician will write 'If I come home, I take off my shoes.', whereas a linguist will write 'When I come home, I take off my shoes.'

Knowledge is expensive, but not nearly as expensive as ignorance (Cornelis, 1993).