

# **The Design of Spatial Information Systems Part 2: Knowledge Representation**

Max J. Egenhofer  
Andrew U. Frank  
Douglas L. Hudson

©1997

This version does not contain  
any references to pertinent literature.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Data Model . . . . .	2
1.2	Database Schema . . . . .	2
<b>2</b>	<b>Designing Information Systems</b>	<b>5</b>
2.1	What is Included in an Information System? . . . . .	6
2.2	Elementary Representation . . . . .	7
2.2.1	Predicates . . . . .	7
2.2.2	Deduction Rules . . . . .	8
2.3	Databases as Formal Models . . . . .	9
2.4	Organization of Information . . . . .	11
2.5	Uniform Representation of Data and Meta-Data . . . . .	14
2.6	Using the Same Data Model for Data and Meta-Data . . . . .	14
<b>3</b>	<b>Structural Components</b>	<b>17</b>
3.1	Entities . . . . .	18
3.2	Relations . . . . .	19
3.3	Properties . . . . .	21
3.4	Values . . . . .	22
<b>4</b>	<b>Advanced Features</b>	<b>23</b>
4.1	Relations between Relations . . . . .	24
4.2	Combinations of Relations . . . . .	26
4.3	Implementations of Relations . . . . .	26

4.4	Caveats . . . . .	27
<b>5</b>	<b>Abstraction Methods</b>	<b>29</b>
5.1	Object-Orientation . . . . .	31
5.2	Classification . . . . .	32
5.3	Generalization . . . . .	35
5.4	Aggregation . . . . .	38
5.5	Concurrent Abstraction Processes . . . . .	39
<b>6</b>	<b>Modeling Behavior</b>	<b>41</b>
6.1	Inheritance . . . . .	41
6.1.1	Single Inheritance . . . . .	43
6.1.2	Multiple Inheritance . . . . .	44
6.2	Propagation . . . . .	47
<b>7</b>	<b>The Relational Data Model</b>	<b>53</b>
7.1	The Relational Model Concept . . . . .	53
7.2	Relational Operators . . . . .	55
7.2.1	Union . . . . .	56
7.2.2	Set Difference and Set Intersection . . . . .	57
7.2.3	Cartesian Product . . . . .	58
7.2.4	Selection . . . . .	59
7.2.5	Projection . . . . .	59
7.2.6	Join . . . . .	60
7.2.7	The Composition of Relational Operators . . . . .	61
7.3	Functional Dependency . . . . .	62
7.3.1	Transitive Dependencies . . . . .	65
7.3.2	Partial Dependencies . . . . .	65
7.3.3	Multi-Valued Dependency . . . . .	65
7.4	Normalization Rules . . . . .	65
7.4.1	First Normal Form . . . . .	66
7.4.2	Second Normal Form . . . . .	66
7.4.3	Third Normal Form . . . . .	68
7.4.4	Other Normal Forms . . . . .	68
7.5	Practical Consideration of the Relational Database . . . . .	68

<b>8</b>	<b>Extending the Data Model with Logic</b>	<b>71</b>
8.1	Domain Closure Axiom . . . . .	73
8.2	Unique Name Assumption . . . . .	73
8.3	Equality Relation . . . . .	74
8.4	Closed World Assumption . . . . .	74
<b>9</b>	<b>Consistency of a Database</b>	<b>77</b>
9.1	Terminology . . . . .	78
9.2	Formal Definition for Consistency . . . . .	78
9.3	Definition of Consistency of a Database . . . . .	79
9.4	Preconditions of Programs . . . . .	81
9.5	Consistency Constraints as Conditions on a Database . . . . .	82
9.5.1	Checking a Database . . . . .	82
9.5.2	Checking An Input . . . . .	83
9.5.3	More General Considerations for Keeping a Theory Consistent . . . . .	84

# Chapter 1

## Introduction

Having learnt about the importance of formal systems in general, our focus can now shift towards building databases that are appropriate for the representation of real-world phenomena. Geographic information systems are typically models of the real world and users will try to interact with them as if they would interact with the actual objects. The decisions people make based on an information system are decisions either about the real world, or with immediate implications on the real world—granting a building permit, deciding where to locate a hazardous waste dump, selecting a route to transport some good quickly. In order to let people make the best use of an information system, it is necessary to represent knowledge about geographic space, stored in a computer system, in a way that comes close to their thinking.

The discussion concentrates on knowledge representation, and not on processing. This reflects the general finding that the organization of the knowledge to be used is the crucial step in building computerized systems. The procedural, algorithmic part is important, especially in designing systems with high performance, but performance should be considered *after* a clear understanding of what a system should do is reached. The system which produces wrong results, but very rapidly, is obviously of little use to anyone.

## 1.1 Data Model

A notion that will follow us throughout this part is the concept of a *data model*. A data model is a (formal) conceptual construction which provides the rationale for organizing data to be stored in a database. There are a small number of conventionally recognized data models such as hierarchical, network, and the relational.

DATA MODEL = The rationale for organizing data to be stored in a database.

The hierarchical model in its pure form demands that the data be arranged in a hierarchy which is a simple and highly efficient storage scheme. This model, however, is clearly too restrictive for most applications, particularly spatial ones. The conceptual schema for this data model rarely fits reality satisfactorily and, therefore, this model is currently seldom used.

The network model concept is to link data elements in a network structure. The idea is very general and powerful, but is not easy to implement. Network models were first proposed quite some time ago, and most major database constructions of this type follow a standard set in the early 1970's by the CODASYL committee.

The most prominent data model in today's (1992) commercial applications is the relational data model, to which we will dedicate more attention in a later chapter. There is evidence that the innovative concepts of object-oriented data models will overhaul the relational data model in engineering applications. The abstraction mechanisms present in object-oriented models come closer to the humans mental organization of complex situations. They are also more general than the restricted, but well-defined view of the relational data model. This part will elaborate on the distinct properties of object-oriented data models and their differences to the relational data model. Complementary concerns of implementing object-oriented models will be covered in Part 3, where the relevant software engineering aspects will be discussed.

## 1.2 Database Schema

Usually, the description of all predicates or all relations (i.e., meta-data), in a database is called the *database schema*. According to ANSI/X3/SPARC, we differentiate

three types of schemas:

- The *conceptual schema* describes the data included in a database from the conceptual view, meaning an overall picture which concentrates as much as possible on the real world meaning of the data. The conceptual schema (at least a part of it) may be that the real world relation “X is the father of Y” and “X is the grandfather of Z” can be determined to some extent.
- The *internal schema* describes how the data from the conceptual schema are mapped into computer storage. It contains all the details of storage structure necessary for a complete implementation, but this part of the design should have no influence on the interpretation of the data. Changes in the internal schema must not be noticed by the users of the database (except for perhaps improved performance). The internal schema may be the storage of all axioms by hash coding, keyed on the first three characters of the consequent predicate’s name.
- Not all users need to know and understand the complete conceptual schema and it seems a good idea to present a certain user or user class only that part which is of interest to them and leave away parts and details they need not know (and, perhaps, may not be allowed to access). The *user schema* has the same level of world orientation that the conceptual schema has, but it includes only part of it and may also arrange things differently, to serve this user group better. It is assumed that the database can internally, and invisibly for the user, translate the data from the internal format to whatever form is described in the user schema. The user schema may be that the information on a user’s own family is accessible, but that not that of other user’s families, (perhaps that information is restricted by a password). Furthermore, the family data is always presented as full sentences, such as: “Henri is the grandfather of Stella,” in lieu of “X = henri; Z = stella.”

Most commercially available DBMS do not include extensive facilities for user schemas and permit only very limited changes; however, the use of the predicate calculus, as we employ it for prototyping, has significant power in this respect.

This part is exclusively interested in the conceptual schema. Aspects relevant to the internal chapter will be addressed in Part 4 dealing with the architecture of

spatial information systems.

## Chapter 2

# Designing Information Systems

Designing a valid formal model of reality is the primary problem of designing an information system. The solution must not only include decisions as to what part of reality will be modeled, but also how the different things modeled will be related. Certain significant parts of reality must be selected for modeling; the selection will determine what questions may be asked by future users of the system. Not surprisingly, the design of the system will influence the user interface. Experience shows that the design of the data model most often decides whether an information system is useful for and accepted by the intended users or not.

This chapter approaches the design of an information system from the point of view currently prevailing in database research. It is based on methods for knowledge representation as used in artificial intelligence and implemented in first-order logic. It is, therefore, more general than the implementation-oriented methods that have been employed in more traditional database systems.

Traditionally, the design of a data structure for use in an information system is begun with a list of “fields” in records, which represent information that should be included. It is then necessary to find out which “fields” are dependent on which others (functional and multi-valued dependencies). A breakdown into independent units (independent in the sense of normalization rules) follows.

We will, in this chapter, ignore all considerations necessary for realization on present day computers and we will use predicate calculus as a formal vehicle for our discussion. Using a PROLOG system will allow us to build and test examples

quickly (*rapid prototyping*). This will permit students to experience different design alternatives without a large programming effort.

Over the last couple of years, the area of *deductive databases* has been an intense field of research in computer science, and some commercial products have reached the market. Although these techniques demonstrate performance problems when confronted with very large (spatial) data sets, we believe that it is important that the two concerns, “being true to reality” and “performance issues,” are kept separate. Overall we assess that the “true to reality” is the more important consideration and we agree with Nick Chrisman’s demand that “Systems should not be built for our imperfect technology, but should reflect the inherent structure of the problem.” Performance depends on many technical solutions and changes drastically in a few years; it should not be valued too highly and influence the structure of our solution. Some of the problems we have to deal with (e.g., recording ownership of land tracts) have not been changed substantially over more than a thousand years.

In order to prepare for designing a database, a theoretical understanding of database behavior and the users’ expectations is necessary. Only afterwards can informed decisions for the design be made. The theoretical approach presented here introduces students to basic information management concerns before requiring them to study more specific data handling problems.

## 2.1 What is Included in an Information System?

An information system is built for an organization to fulfill some needs. It is generally thought that it is cost effective to collect information only once and to make it available to all concerned users. If this is done, all parts of the organization make their decisions based upon the same information, which may be advantageous, even if the information is sometimes wrong.

In this chapter we will only discuss methods to describe the data to be included in an information system. There are no set rules about what to include and what to leave out. The structure of the descriptions can sometimes be used to detect holes or redundancies which require modification to make the system more usable. There are no clear rules about what is to be included in an information system.

## 2.2 Elementary Representation

In an information system we have to store the theory that represents the model. This is often called “knowledge” as it seems to represent an understanding of reality. The problem we face then is how to best represent knowledge.

We have selected first-order predicate calculus (in Horn clause form) as a knowledge representation scheme. This is a very general approach, which induces a minimal amount of artifacts. A large variety of types of information can be represented by this method.

### 2.2.1 Predicates

In the formal system of a first-order predicate calculus a predicate is usually written as:

$$p(A, B).$$

This can be read as “the relationship  $p$  holds between  $A$  and  $B$ .” The predicate is assigned a value of either true or false depending on whether or not the relation holds.

Often (but not always) the truth value cannot be discerned for a predicate until after specific values for its arguments have been assigned. When all variables have been assigned (i.e., bound to) constant values, logicians refer to the predicate as having an *interpretation*. For instance, the predicate  $fa$  may be true under the interpretation  $fa(an, st)$ , but false under the interpretation  $fa(he, st)$ . The reference is only to the symbolic representation of a formal predicate instance.

We will not use the term interpretation in this way. We will reserve the term for less formal situations where we want to refer to the meaning attached to the symbols. For instance, the predicate  $fa(an, st)$  can be interpreted as “andrew is the father of stella.”

A predicate has a defined number of arguments, called the *arity* of the predicate. In the example above,  $fa$  is said to be a binary predicate (i.e., a predicate with two arguments). Predicates can have any number of arguments, including none. These would be considered constants, always true or always false. Unary is the name for a predicate with one argument, binary for two, ternary for a predicate with

three arguments. Logicians permit the use of the same predicate name for different predicates with different arities, e.g., `father(a)` to state that `a` is a father, `father(a, b)` to state that `a` is the father of `b`; we will, however, restrict a predicate name to be unique for one predicate with a defined arity.

In all of our examples, argument symbols starting with lower case letters stand for constants, symbols starting with capitals are variables. All predicate symbols will be lower case. Simple predicates of the form `p(x, y, z, ...)`, with `x, y, z, ...` all being constants, are called *facts* or *ground axioms*.

Some examples of facts with possible interpretations:

<code>brown(table).</code>	“The table is brown.”
<code>color(table, brown).</code>	“The color of the table is brown.”
<code>book(melville, mobyDick).</code>	“Melville wrote Moby Dick.”

First-order languages may have constants or variables for the arguments of a predicate, however, the predicate itself must be a constant. Higher-order languages allow for variables over predicate names as well. While this extension provides for additional power, the logic associated with such a language is extremely difficult and even inconsistent.

### 2.2.2 Deduction Rules

These rules will be presented as Horn clauses. This limits the expressive power of the knowledge representation scheme somewhat, since certain situations cannot be described in Horn clause form (indefinite expressions, e.g., `A or B if C`).

A rule:

$$p(A, B) \text{ if } q(D, E), r(F, G).$$

The statement above is an example of a composition of predicates that represents a more extensive statement than those described for a simple  $n$ -ary predicate type. It can be read as: `p(A, B)` can have a value true assigned if both `q(D, E)` and `r(F, G)` are found to be true. (We use “if”, not “iff” (i.e., if and only if); “if” is the logical converse of “implies.”)

<code>chore(washClothes)</code>	if	<code>day(monday)</code> and <code>weather(clearAndWarm)</code> .
<code>grandfather(X, Z)</code>	if	<code>father(X, Y)</code> and <code>father(Y, Z)</code> .

The second rule can be read as “X is the grandfather of Z if X is the father of Y and Y is the father of Z.” This is only one rule for describing a grandfather relationship (a paternal grandfather), but there are others; a maternal grandfather rule would be:

$$\text{grandfather}(X, Z) \quad \text{if} \quad \text{father}(X, Y) \text{ and mother}(Y, Z).$$

or a more general grandfather rule:

$$\text{grandfather}(X, Z) \quad \text{if} \quad \text{father}(X, Y) \text{ and parent}(Y, Z).$$

## 2.3 Databases as Formal Models

Previously we discussed how formal systems can be interpreted as a model for reality. This is done by selecting an interpretation for each symbol in the formal system that maps that symbol onto some part of reality, such that the interpretations of provable sentences in the formal system are judged as true in the real world by a knowledgeable human observer. The formal system:

$$\begin{aligned} & \text{fa}(a, s). \\ & \text{fa}(h, a). \\ & \text{fa}(g, h). \\ \text{gfa}(X, Z) & \text{if } \text{fa}(X, Y) \text{ and } \text{fa}(Y, Z). \end{aligned}$$

with the interpretations:

$$\begin{aligned} \text{fa}(X, Y) &= x \text{ is father of } Y \\ \text{gfa}(X, Y) &= x \text{ is grandfather of } Y \\ a &= \text{Andrew Frank} \\ h &= \text{Henri Frank} \\ g &= \text{George Frank} \\ s &= \text{Stella Frank} \end{aligned}$$

This is a valid model of part of the Frank family, because the interpretation of the sentences above and all logical conclusions from them correspond to true statements about the Franks.

Be careful to note the difference: the wff  $\text{father}(a, s)$  is a ground axiom in a formal theory with a truth value “true” assigned; the sentence “Andrew is the father of Stella” is information about reality (at least a small part of it) and is judged (or observed) to be true (Figure 2.1). The above is a valid model, because we have a systematic mapping and we assume that each wff that has the value true corresponds to a correct sentence about reality.

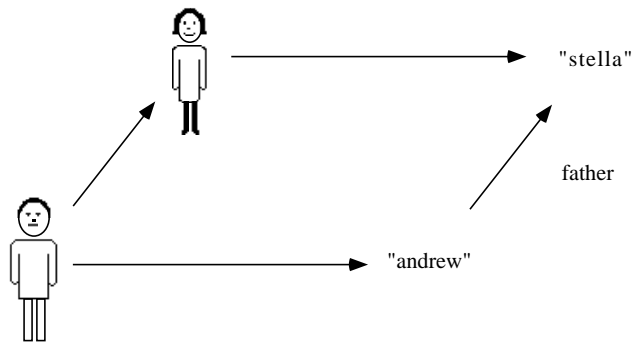


Figure 2.1: Interpretation of the model  $\text{fa}(a, s)$ .

Using only formal symbol manipulation (like modus ponens) and without reference to any meaning of the symbols involved, we can conclude from the theory presented above that:

$$\begin{aligned} & \text{gfa}(h, s). \\ & \text{gfa}(g, a). \\ & \text{not gfa}(g, a). \end{aligned}$$

For all these wffs there corresponds true sentences about the world. These predicates and their interpretation can be regarded as a database about a part of the Frank family, containing answers to questions like, “Who is the father of Henri?” or “Whose father is Andrew?” Together with the rule about grandfathers being the fathers of fathers, we can even deduce more complex relations. In a short while, a system will be presented that stores and retrieves facts in essentially the form given

above. It can be used to build small databases and prototypes to try out larger ones. This system will clearly show how databases are related to formal systems.

We will make some assumptions as we write predicates in our database examples. One of them is that all stated predicates are implicitly true, i.e., we will write predicates without explicitly mentioning that they have the value true assigned to every one of them. In addition, the universal quantifier “for-all” is always in effect wherever a variable is used and is, by convention, never explicitly written. The existential quantifier “there-exists” is simply not used. Other assumptions will be mentioned later. It should be obvious that we will write our database using the style of the streamlined clausal forms (Horn and facts) mentioned earlier.

There are many different ways to express parts of reality in predicates, just as there are many different ways to select the parts of reality included in a model. Primarily it depends on the application as to what is an appropriate selection and representation (a “good” model), but there are some guidelines for the design. They will be discussed throughout the next section.

## 2.4 Organization of Information

To form a theory we could combine any information we have about the model (providing that there are no contradictions, of course). Theoretically, an inference engine could find all valid conclusions. This is the approach taken by many expert systems, where a relatively small number of facts and rules are used to produce some very impressive output. Some information systems contain large collections of facts and rules in their knowledge bases and the number of conclusions possible to deduce from them is astronomical.

Even with small collections of axioms, however, it becomes extremely difficult for humans to track the interactions between the various components. The recursively convoluted nature of the inferencing mechanism can easily impede a designer’s understanding of the effect of adding, deleting, or modifying an axiom in the theory.

Fortunately, however, many facts (and even some rules) have a structure of their own which can be exploited to ease the problem. We have assumed all along that it was possible to organize information about some piece of our world (e.g., land or family structures); it should come as no surprise to find that the same organization

methods can be applied directly to information in general.

We now introduce some fundamental concepts (taken from the field of artificial intelligence) which can be used as tools to organize and describe any model of a part of reality. The ensemble of these concepts is called a data model. In selecting them we must first be completely general; these concepts must be applicable for any use, not just for the specific examples given here. Moreover, we must be careful that they are sufficiently different from each other and cannot replace one another (this is technically called orthogonality).

From the very beginning of computer programming, we can observe the separation between data and programs (Figure 2.2). The idea of a stored program is fundamental to computers as we know them today and is often credited to John von Neumann.

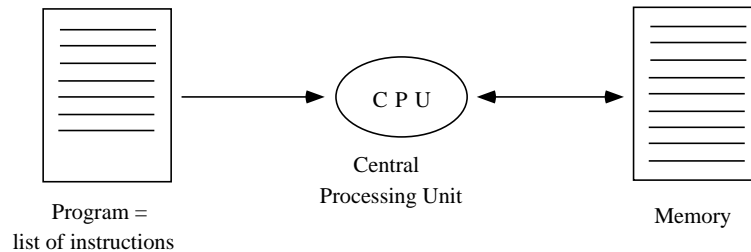


Figure 2.2: Separation of data and computer program.

Other kinds of machines operate using a physical design which determines their operation; computers can acquire and store a “program” that governs their action. The computer then operates on data according to the directives in the program. With most modern computers, programs and data reside in the same type of storage cells and have a very similar format. In fact, it is possible to have computers (under program control) change the programs themselves, even the program that they are currently executing; this is called self-modifying code. In the days of assembly language programming, it became apparent that self-modifying code could be extremely difficult to comprehend, to debug or later modify. Therefore, self-modifying code was strictly advised against and programming languages of the

FORTRAN, COBOL, Algol, Pascal, etc., type included no provisions that would allow the writing of self-modifying code. There is, however a considerable tradition in languages designed for artificial intelligence research (primarily LISP and PROLOG), which permit self-modifying code. This is not an inherently dangerous process, but it should not be used without appropriate design provisions and support from the language. APL can be considered a language which permits self-modifying code. It can otherwise be used for programming in the style of FORTRAN or Pascal; APL code is nearly impossible to decipher.

A program includes (in the “old” languages in an implicit form) a description of the data. The program defines what structure the input data must have and what output data is afterwards produced. Pascal, with the record data type, allows this data description to be more explicitly defined. This is only a gradual change from earlier languages, however, as the data description continues to be embodied in the program. The first step of the database concept is to separate the description of the data structure from the program, to put it into a central data description facility (often called a data dictionary) and then include it into all programs that use the same data. This guarantees that all programs use the same description.

When separate descriptions of the data structure are used they are called “meta-data” (from the greek meta—above/about). Meta-data describe the structure of data, but of course, meta-data is data itself (it could be a text file in a language similar to Pascal’s data definition syntax).

The data/meta-data separation is a very useful one as long as we use traditional programming languages, but it is not inherent in the problem; in fact, quite the contrary is true, as we will see shortly. The data/meta-data separation often introduces peculiar problems of its own, which are surprising at first until one understands where the problems are originating. (Further discussion will follow when we introduce traditional data models).

It is probably fair to note that the data/meta-data separation can result in faster and easier processing of data. It is usually necessary for the compilation of programs from a human-oriented high-level language into a executable machine code. Efficient, high-level languages that do not include this separation are not yet widely available.

## 2.5 Uniform Representation of Data and Meta-Data

Using first-order predicate calculus as a programming language makes it superfluous to separate data from meta-data, or, for that matter, data from programs.

Is the clause: `fa (andrew, stella)` data or program? How about: `gfa (A, C)` if `fa (A, B)` and `fa (B, C)`? If you thought that the latter, being a rule, is more similar to a program, then what is `gfa (henri, stella)`?

In PROLOG, there is no need to separate the two (except, however, in that their order of listing may affect the interpreter's search strategy. It may be also advisable to group data and meta-data-like clauses to achieve organizational clarity and a more easily understood design). We can gain considerable advantages by being able to treat the data and meta-data in the same expressions. Humans typically use data from all levels (data and meta-data) in their reasoning and most artificial intelligence work needs to move smoothly between data and meta-data and use whatever is required in deduction.

Generally, deduction involving meta-data is more efficient a process than working with a large collection of factual data. In a search for all grandfather's, for example, half the problem space can be eliminated at the meta-data level if there is a rule stating that grandfathers must be males (no need to look at the female population).

We will here take advantage of this unification as it seems easier to understand what a database is doing if no artificial barriers are put between data and meta-data. It is then possible, to completely explain the logical behavior of a database in a uniform setting and with a very small amount of code. Later we will discuss what limitations a relational and a standard network database management systems impose, as well as their effects, still in terms of logic. To use an efficiently implemented database management system should then be only a minor switch from one language to another.

## 2.6 Using the Same Data Model for Data and Meta-Data

A data model provides the means to describe the data structure. As the meta-data, the description of the data using the data model is again data, it must have a structure as well. There is obviously nothing more simple than to use the same data model again (this time to describe itself). This recursive situation is similar to our use

2.6. *USING THE SAME DATA MODEL FOR DATA AND META-DATA* 15

of the Backus-Naur form for description of the Backus-Naur form for production rules.



## Chapter 3

# Structural Components of a Logic-Based Data Model

A data model can be considered on many different levels. Our logic based model will be founded on basic abstraction methods and the following fundamental concepts:

- entities,
- relations,
- properties, and
- values.

The concepts used for organizing data in this chapter are fundamental and generally applicable. Unfortunately, most of the available database management systems do not incorporate them completely nor do they fully use the semantics presented here. In order to improve performance and to lower storage utilization, only reduced and less general constructs are included.

We will use predicate logic to build a small database describing the facts modeling reality. It is extremely important to understand these concepts as these form part of the rationale on which the database is founded.

Although the formalisms in this and subsequent chapters are expressed using PROLOG, they are independent of any language or specific database management system. PROLOG provides a convenient notation for a first-order logic with the additional benefit that the formulae can be executed and tried as a small database.

Remember that if a query contains all constant arguments, the result is simply a confirmation of the presence of that fact in the database; PROLOG returns success or failure. If there are variables in the query, the result is a list of values located in the database that satisfy the query.

Later we may employ the extensions provided by PROLOG, but for now, except where noted, the formulas are exactly “first-order” and do not make use of any built-in “extra-logical” predicates, (e.g., the “cut”, which is only used to provide a “not” operator unavailable in standard PROLOG). Readers are encouraged to type in these examples and try them out. Playing with the concepts can certainly help one to grasp the simple, but far reaching concepts.

### 3.1 Entities

For modeling purposes we have to identify things we want to model and separate them from their environment. Those things are called entities. Generally, an entity will be represented by an entity symbol (i.e., the entity’s “name”).

An *entity* is anything, real or imaginary, that is thought of as having an independent existence.

Entities can be “John Alexander”, “the marriage of John Doe”, or “the tree in front of my home.”

The concept of entity should not be too narrowly restricted; keep a very open mind about what entities can be. We will often refer to entities as concrete objects in our examples, e.g., people and puppies, each of which may be qualified somehow by the existence of a property with various values, or by its participation in some relationships with other entities. These properties, values, and relations, however, can also be considered as entities themselves in certain contexts, and the interrelated nature of these concepts may result in some very profound confusion.

What an entity is, therefore, depends on the application; however, regardless of the application, it must have a clear and distinct existence and be fully determined. We would not consider things described by selection whose result may vary as entities (e.g., “the best SVE student,” but “the best SVE student in 1984” is ok).

In a formal system we need a mechanism to identify uniquely each entity. We employ an identifying constant (e.g., an alpha-numeric constant) for each entity, and we assume in our model that there is a mapping between the real thing and the constant. Do not confuse the real world name of an entity with the identifier that we will often use to represent the entity. A name is simply a property of some entities and any number of individuals can have the same name (as they share the color of their hair or an academic major).

To simplify exposition in this text, we will assume that the identifier name (the representative constant symbol) is unique for that entity (corresponding to the *Unique Name Assumption*) in our database. If we want to use the actual names as constant symbols we have to be careful that they are unique—otherwise we have to add to them till they are unique (e.g., peter1, peter2, peter3). We will then add a property, “name,” which yields the name of the individual (e.g., Peter for all three peter1, peter2, peter3). It becomes meaningful to ask who has the same name, but it is clearly not meaningful to ask which individuals are the same.

Although, to be formally correct, entities can neither be read nor printed, PROLOG contains a facility that uses the printable form of the entity symbol for user reference in input and output. This is clearly a very useful extension, and should be understood as an additional property every entity has, which yields its symbol as a printable name.

## 3.2 Relations

Entities by themselves would be mere collections of the things of concern. Even though the existence or non-existence of something is occasionally of interest, it is far more interesting to record and analyze how entities are related to each other. This will be expressed through relations.

*A binary relation consists of a relation name and two entities that are linked by this relation.*

A relation is not necessarily a function, but it is a mapping from one set of entities onto another. For a binary relation, we will write a predicate of the form:

$$p(\text{Entity1}, \text{RelationName}, \text{Entity2}).$$

Relation operations are often continuable in that the result of the relation may be another entity which in turn can be used as input for another relation. It is possible to link relations together and it becomes interesting to consider what successive application of a relation means, comparable to the successive application of a function in regular algebra:

$$\text{SQRT}(\text{SQR}(25)). \text{ or}$$

$$\text{ADD}(3, \text{ADD}(4, 5)).$$

Although nested functions are formally permitted in first-order logic, (as terms, i.e., arguments of a predicate) they are not in PROLOG. Predicates are special cases of functions that only return Boolean values and generally we need non-Boolean values for the predicate arguments. We can achieve the same results, however, by chaining relation predicates together (the third argument of the first is equal to the first argument of the second, etc.).

Suppose we want a predicate like:

$$p(X, w, f(Z)).$$

where  $f(Z)$  is a function that returns some value that would act as the third argument of  $p$ . The predicate (relation)  $p$  is trying to state that  $X$  has the relation  $w$  with the result of function  $f(Z)$ . Using an example from our family relations:  $f(Z)$  could be a function that returns the father of  $Z$ . For example, the result of  $f(\text{stella})$  would be Andrew. Then  $p$  could mean that Irja,  $X$ , would have the relation wife,  $w$ , with the father of Stella,  $Z$ , which is Andrew or  $f(Z)$ . PROLOG does not allow this formulation, so we must rewrite  $p$  as:

$$p(X, wf, Z) \text{ if } p(X, w, Y), p(Y, f, Z).$$

$X$  is the wife-of-the-father of  $Z$  if  $X$  is the wife of  $Y$  and  $Y$  is the father of  $Z$ . This is straightforward interpretation of the previous (illegal) form.

### 3.3 Properties

Entities have properties describing them. For ordinary properties (single-valued properties), there is always exactly one property value associated with one property, for example, my haircolor (property name) is brown (property value).

There is a functional dependency from the entity to the value of a property, i.e., given the entity there is only one value for this property. We can see a property as a function which maps from the domain of entities onto the range of property values.

propertyName (Entity) = value.

For instance:

firstname (max) = Max.

haircolor (max) = brown.

If the type of values is not restricted (and eventually functions with multiple results will be included) then this is an extremely powerful concept.

*Properties* are a pair, consisting of the *property name* and a *property value*, which describe an entity. For an entity, a value (of whatever type) is associated with the property name. A property name can be considered a function to be applied to an entity which yields a value.

In PROLOG we can accommodate properties for entities as a predicate (e.g., “p”) with three arguments:

p (Entity-Id, Property-Name, Property-Value).

For instance,

p (maria, haircolor, brown).

p (david, haircolor, blonde).

Questions like, “Who has brown hair,” can be asked.

p (X, haircolor, brown).

Remember that the constants david and maria in the above example stand for entity identifiers and not the names of people. It is proper to add the properties “firstName” and “familyName” as follows:

p (david, firstName, ‘david’).

```
p (david, familyName, ``bowie``).
```

There are no formal reasons why a property must map to a single value. It is certainly possible to develop a model in which the result of a property mapping could be an array, a Pascal-like record structure, a LISP-like list structure, or any other complex object. Implementing such a model is difficult, however, since additional provisions must be made for operations on the complex value types. Recent versions of PROLOG have introduced some complex value handling facilities.

### 3.4 Values

*Values* are the primitive symbols which characterize the quantity or quality of a particular property of an entity. They respond to basic operations.

There is no firm guideline to separate entities and values; it all depends on the purpose of the model you design. What is a value in one model may be an entity in another. The functional difference is in the operations that can be applied to one or the other. Entities are objects which can have properties which are grouped into classes and which can participate in relations. Values, on the other hand, respond to different operations (e.g., input/output or arithmetic, etc.; operations not found in first-order logic, but which are extensions provided by PROLOG).

We will, for practical reasons, assume that an entity can generally change a property value over time without significantly changing the identity of the entity. There is considerable research into databases that can handle a temporal aspect. It is, moreover, a philosophical problem beyond the scope of this text to argue questions such as: are you the same or a different person from the one you were ten years ago, one year ago, one second ago, etc.?

## Chapter 4

# Advanced Features of a Logic-Based Data Model

The logic database model, i.e., the proof-theoretic data model, allows for a more powerful construction than the relational model. In this chapter we will explore some of the methods for knowledge representation which will demonstrate this. We will also discuss some properties of relations.

It is important to realize that describing our model in formal terms is not only crucial to building the mechanical aspects of an information system, but also to allow users the opportunity to check the database designer's understanding of the model. Recall that the same word may have a different (even if only slightly different) meaning for different people and that there are no secure ways to check these differences or to communicate them. It is generally assumed in linguistics that the meaning of words is conveyed by their image and by their relationship to other words. If we model formally the exact connection between words it will be possible for others to see what meaning we have assigned to a given word.

For instance, if we want to know what an uncle is in a particular proof-theoretic database implementation, all information about uncles can be obtained in the facts and rules stored in that database.

`uncle (X, Y) if brother (X, Z), parent (Z, Y).`

If the preceding is all that can be found on the subject of uncles, it can easily be determined that this database does not consider the husband of the sister of a

parent as an uncle nor does it allow the term to be used as an honorific title for an old family friend.

In general the techniques mentioned here are designed to describe the model we build and each technique can be described using the ordinary tools of first-order logic. We will show how certain situations can be generalized and organized in a way which should help us clarify and reason about properties.

## 4.1 Relations between Relations

Often one relation is closely related to another such that we can deduce the existence of one from the other. Such situations are well-known in mathematics and we shall use terminology from mathematics to describe what follows. It may be surprising that similar concepts apply to very common every-day situations.

Many of the relations described here are a potential source of redundancy, especially if both of the related relations are stored. To get the appropriate response from a proof-theoretic database which has axioms describing relations such as these requires deep analysis of their rather extensive effects.

In mathematics a relation is called the *converse* of another one, if the existence of a particular relation mapping from  $a$  onto  $b$  (written  $a R b$ ) implies that there is another relation (the converse) that will map  $b$  onto  $a$  ( $a R^{-1} b$ ).

If  $R$  is a relation, the converse (reversed) relation of  $R$ , written  $R^{-1}$ , is a relation such that  $y R^{-1} x$  if and only if  $x R y$ .

A number of examples for relations with converses:

- parent / child
- supervises / is supervised by
- owns / is owned by

Typically the English language uses the active/passive voice and the same verb to express a relation and its converse.

A binary relation is called *reflexive* if for any  $x$ , the relation  $x$  to  $x$  is true: i.e.,  $\forall x : x R x$ .

Relations with equals in them (e.g.,  $=$ ,  $\leq$ ,  $\geq$ ) are reflexive. An entity is always equal to itself. If it can be assumed that when you talk at least you are listening to what you say then the relation *ListensTo* would be reflexive.

A binary relation is called *irreflexive* if it is not reflexive, i.e., not  $\forall x : x R x$ .

Relations without equals (e.g.,  $<$ ,  $>$ ,  $\neq$ ) are irreflexive. The relation *BrotherOf* is irreflexive, because no one is his or her own brother.

A relation is called *symmetric* if,  $\forall x, y : x R y \Rightarrow y R x$ .

This states automatically that a symmetric relation is its own converse. For instance, a partnership is symmetric, because  $x$  is a partner of  $y$ , thus  $y$  is a partner of  $x$ . Other examples of symmetric relations include spouses and true friends.

A relation is called *asymmetric* if we can conclude from  $x R y$  that  $y R x$  does not hold, i.e.,  $\forall x, y : x R y \Rightarrow \text{not}(y R x)$ .

Examples of asymmetric relations include supervises and father.

A relation is called *antisymmetric* if  $\forall x, y : x R y \wedge y R x \Rightarrow x = y$ .

Examples of antisymmetric relations include less than or equal to and not older.

The difference between symmetry, asymmetry, and antisymmetry is that with symmetry the related entities can always be reversed; with asymmetry the entities can never be reversed; with antisymmetry the entities may be reversed, but only if they are equal.

A relation is called *linear* (or connex) if  $\forall x, y : x R y \vee y R x$ .

Examples include, less than, greater than, and older.

A relation is called *transitive* if  $\forall x, z \exists y : x R y \wedge y R z \rightarrow x R z$ .

A typical transitive relation is equal or less, but there are many others like ancestor, supervises, and prerequisite of.

## 4.2 Combinations of Relations

A relation can have more than one of the above properties. The extremely important class of *equivalence relations* are defined as being reflexive, symmetric, and transitive. Relations which are antisymmetric and transitive introduce an *order* into the entities.

We assume that describing relations like those above will help clarify the database schema. Mathematics considers equality and order to be fundamental to its existence; it is most likely that they will be just as necessary for any non-trivial database application.

## 4.3 Implementations of Relations

To better utilize the ideas presented, keep in mind that a relation can be considered at the same time on many different levels of abstraction; as a relation between two entities; as an entity itself with a property/value pair; or as an entity related to another entity. We can then add facts to the database that not only describe simple entities, e.g., that tables are brown or that Andrew is Stella's father, etc., but we can also add facts about the relations, i.e., that they can have properties like transitivity or reflexivity.

For instance, *supervises* has the property of being a transitive relation; the relation between *supervises* and *supervised* is that they are converses.

```
p (bob, supervises, ted).
p (supervises, relationProp, transitive).
p (supervises, converse, supervised).
```

General rules that implement the properties of relations follow:

```
p (X, ConverseRelation, Y) if
    p (Relation, converse, ConverseRelation),
    p (Y, Relation, X).
p (X, Relation, X) if
    p (Relation, relationProperty, reflexive).
```

```

p (X, Relation, Y) if
    p (Relation, relationProperty, symmetric),
    p (Y, Relation, X).
p (X, Relation, Y) if
    p (Relation, relationProperty, transitive),
    p (X, Relation, XY), p (XY, Relation, Y).
p (X, ConverseRelation, Y) if
    p (Relation, converse, ConverseRelation),
    p (Relation, relationProperty, transitive),
    p (Y, Relation, YX), p (YX, Relation, X).

```

In a very abstract sense this takes care of many of the deduction rules which would otherwise be necessary in a database. There is no specific need to explain individually that “supervises” and “is supervised by” are the converse of each other, nor the fact that my boss’ boss is also my boss (transitivity).

Having these tools available we can proceed to apply them to the description of our data model.

## 4.4 Caveats

The above clauses assume that for a relation and its converse all the facts are stored with the relation and none with the converse. With some additional rules this limitation can be lifted and facts stored with both, i.e.,

```

p (X, ConverseRelation, Y) if
    p (ConverseRelation, converse, Relation),
    p (Y, Relation, X).
p (X, ConverseRelation, Y) if
    p (ConverseRelation, converse, Relation),
    p (Relation, relationProperty, transitive),
    p (Y, Relation, YX), p (YX, Relation, X).

```

It is not advisable to do so, however; it makes the comprehension of the database more difficult.

In addition many of the above rules result in infinite recursions (the equivalent of an infinite loop) if attempts are made to implement them directly in PROLOG. Remember that a predicate will be recursive if the same predicate appears in both the consequent and the antecedent side of a clause.

Consider the problem of stating a rule for a symmetric relation, where we are tempted to write:

$$a(X, Y) \text{ if } a(Y, X).$$

The use of this rule will obviously cause infinite recursion, because attempts to satisfy the antecedent cause a new invocation of the consequent. The same predicate is used on both sides and there is no limiting mechanism employed. Such expressions are called directly recursive; there are indirect forms as well.

This limitation is a problem of the inference engine employed. It is possible to add a check to the inference algorithm such that infinite recursion results in failure. Only finite sequences can reach positive goals. Thus, this technique does not change the theoretical power of the inference. This so-called “occurs check” is somewhat expensive to perform automatically during execution and most PROLOG systems do not do it, requiring the programmer to take precautions against infinite recursion in his selection of axioms.

## Chapter 5

# Abstraction Methods

As discussed before, humans perceive reality in a selective way, influenced by their need to know, their experience, and their education. They form mental models which may again be simplified more than the original perception. All these steps condense billions of irrelevant details into a few meaningful traits, an operation to which we refer as *abstraction*.

Obviously humans organize their mental models of reality in ways that permit them to reason using different levels of abstraction at once, not just one specific level. Information systems that properly model reality must be able to do so at different levels of abstraction as well.

If you reason about a friend's economic situation, you use pieces of knowledge about his or her special situation (e.g., he has a car, she has a work-study job). You also use knowledge about abstract things that may also pertain to your friend, e.g., if your friend is a student, you may consider the cost of tuition or the salary range of work-study positions.

The term abstraction means that we leave away some traits that are inconsequential for the problem at hand and concentrate on the properties of importance. This is clearly dependent on the task at hand and can be done in different ways. One of the major problems in building an integrated information system is to isolate the important concerns and include them in the formal system.

Experientialists have observed that the base concepts humans use are established early in their life as abstractions from the bodily experience. This experience is the same for all humans as they depend primarily on the physiology of the human body, and establishes a foundation on which to base communication and to avoid subjectivism. The meta model is the generic description of a situation (e.g., “a person owns a building” as opposed to the specific model “Doe Smith owns the building at 56 Park Street”). The abstraction mechanisms available in the data model determine the meta models and specific models that can be used, and thus, the expressive power of a GIS (Figure 5.1). If the abstraction mechanisms are insufficient, the meta model of reality is inadequate and the mapping from the user concept of the object behavior onto the GIS model will be strenuous and difficult to understand. This makes the GIS hard to use.

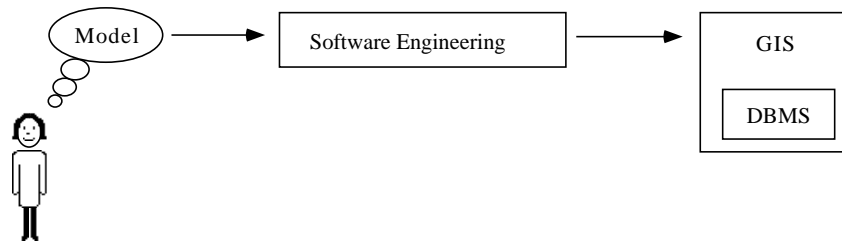


Figure 5.1: Model, Software Engineering, and DBMS for GIS.

Abstraction can be applied on all levels of model building. We can especially consider the abstract concepts produced by a previous step of abstraction as individuals on which further abstraction steps can be applied (i.e., the continuability of abstraction operations). As an example, consider something that is the result of an abstraction process (e.g., the concept “dog” resulting from the abstraction of many details in individual dogs) as an individual subjected to another abstraction (e.g., the concepts “dog”, “table”, and “chair” can be considered to be “things with legs” and that these can be further considered as “classes of things”, etc.). We will show how the data model used in this chapter can be described in terms of itself. This step is comparable to our use of the Backus-Naur form to describe the Backus-Naur form of production rules.

In this chapter we will describe abstraction methods as the most important aspect of the organization of knowledge. We will, therefore, primarily deal with abstraction in reference to meta-data. Several branches in computer science (artificial intelligence, software engineering, database management systems, human-computer interaction) have recently promoted an *object-oriented* approach to overcome the inherent problems of using traditional methods for these so-called non-standard applications. Object-oriented data models have been developed to capture more semantics than the relational model; interfaces make systems appear more natural and easier to use; object-oriented database management systems have been investigated to provide the corresponding features for storage and retrieval of complex objects; and object-oriented software engineering techniques and programming languages have been developed to support the implementation of software systems which were designed following an object-oriented approach and to allow for immediate implementations of object concepts rather than simulating them with traditional programming languages.

The focus will be on conceptual achievements which will help to improve the modeling power of spatial information systems so that the often complex spatial phenomena may be expressed in terms closer to the humans' thinking. Attaching the adjective *object-oriented* to a system just because it was implemented in an object-oriented programming language is misleading. Such a detail should not be visible to the user and thus irrelevant.

## 5.1 Object-Orientation

This section introduces the notation of *objects* and the *abstraction tools* available to deal with them, following Dittrich's synthesis. A definition of object-orientation is that

- any entity, independent of whatever complexity and structure, may be represented by exactly one object.

No artificial decomposition into simpler parts should be necessary due to technical restrictions. This is referred to as *structural object-orientation*.

Complex data types *per se*, modeling large objects such as entire cities (with all details about streets, buildings, etc.), do not overcome the problem of data structur-

ing, and only the combination of complex object types and operations upon such instances provides the necessary view of objects. This second component of object-orientation is called *operational object-orientation* and requires that

- operations on complex objects are possible without having to decompose the objects into a number of simple objects.

The third notion, *behavioral object-orientation*, states that

- a system must allow its objects to be accessed and modified only through a set of operations specific to an object type.

This chapter will focus on the aspects related to structural object-orientation—behavioral aspects are covered in the following chapter and the operational aspects, which are rather implementation driven, will be considered in Part 3.

To describe structural object-orientation, we will use four fundamental abstraction mechanisms and discuss their properties and interactions:

- **classification:** a concern with the type of the individual, rather than the individual itself. It is the recognition of common traits among individuals.
- **generalization:** the recognition of commonalities among classes.
- **aggregation:** the construction of complex objects from more fundamental ones.

These three abstraction methods are independent from one another as can be seen from the domain and range of each.

## 5.2 Classification

*Classification* is the mapping of several objects (instances) onto a common class. The word *object* is used for a single occurrence (instantiation) of data describing something that has some individuality and some observable behavior. The terms *object type*, *sort*, *type*, *abstract data type*, or *module* refer to types of objects, depending on the context. In the object-oriented approach, for every object, there

exists at least one corresponding class, i.e., every object is an instance of a class; therefore, classification is often referred to as the *instance of* relationship.

A type characterizes the behavior of its instances by describing the common *operators* that can manipulate those objects. These operations are the only means to manipulate objects. All objects that belong to the same class are described by the same properties and have the same operations. For example, the model for a TOWN may include the classes RESIDENCE, COMMERCIALBUILDING, STREET, and LANDPARCEL. A single instance, such as the building with the address “26 Grove Street,” is an object of the corresponding object type, that is, the particular object is an instance of the class RESIDENCE. Operations and properties are assigned to object types, for instance, the class RESIDENCE may have the property NUMBEROFBEDROOMS which is specific for all residences. Likewise, the class STREET may have an operation to determine all ADJACENTPARCELS (Figure 5.2).

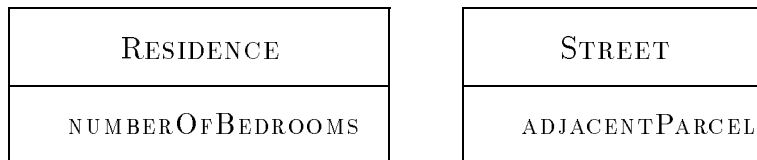


Figure 5.2: Graphical representations for the two *classes* RESIDENCE and STREET.

Differences between objects of the same class are based upon their property values. Property values describe the individual characteristic of each object. For example, two LANDPARCELS may be distinguished by their ADDRESSES, different values of their AREAS, or specific LANDUSETYPES.

In a predicate calculus environment, the conventional axiom (ground axiom) to indicate classification is:

```
r (john, instanceOf, student).
```

or alternatively:

```
instanceOf (john, student).
```

The equivalence of the long and short form is established by the rule:

```
r (object, instanceOf, class) if
    instanceOf (Object, Class).
```

The converse relation to classification is *instantiation*, where a particular entity (i.e., an instance of the class) is derived from the generic description.

```
r (Class, classOf, Object) if
    r (Object, instanceOf, Class).
```

or, shorter, using a more general method:

```
converse (instanceOf, classOf).
```

It is very important not to confuse an individual (a concrete object) with the generic, abstract form into which it may be classified. A class of objects merely describes a potential for particular characteristics; the individual members of the class embody those characteristics. A class can be thought of as the generic or typical thing, e.g., the class “dog” and the generic dog are essentially the same concept. For instance, it is typical for a dog to have a name. Each individual dog has its own name, whereas the generic dog, the class of dog, only has the property of having a name. The difference has a profound effect on the semantics of the rules for relations among entities and their classes.

When an entity has a property that concerns the entity directly, it is referred to as a factual or extensional property; when a class (or type) has a property that defines what it is to be of that class (i.e., it is a property concerning its members) it is referred to as definitional or intensional.

```
EXTENSIONAL = Describing the instantiated form.
INTENSIONAL = Describing the generic form.
```

Having brown hair coloring is an intensional property of the class of Brunettes. That Mary has brown hair is an extensional property of Mary.

An individual can be classified into two or more different generic forms simultaneously. Stella’s toy truck belongs, concurrently, in the classes “toy” and “truck” and it has, at the same time, properties typical for trucks and for toys.

To help clarify the idea of a class, it may be beneficial to compare a class to a Pascal TYPE statement. The TYPE statement serves as a description of the structure of some important program entity; the actual entity must, however, wait for a VAR statement to be instantiated (i.e., to have memory space allotted). A type declaration does not declare a variable, but only describes what it is to be one of that type (class) of which none, one or more may exist.

There is, of course, more to the idea of a class than a description of the internal structure of a member entity. A class is more appropriately compared to the structured programming notion of an abstract data type (ADT) in that it provides not only a description of the structure of an entity, but of the operations that are permitted on that entity as well. It still does not, however, actually represent a specific instance of a member entity.

### 5.3 Generalization

*Generalization*<sup>1</sup> groups several classes of objects with common operations into a more general superclass. The term *superclass* characterizes this grouping and refers to object types which are related by an *is\_a* relation. The converse relation of superclass, the *subclass*, describes a specialization of the superclass. Frequently, the terms *parent* and *child* are also used for superclass and subclass, respectively. Though this terminology is helpful to clarify the dependency of subclasses from superclasses, it is not accurate with respect to the abstraction, because the relationship between parent and child is not *is\_a*. Subclass and superclass are abstractions for the same object and do not describe two different objects. For example, each RESIDENCE is a BUILDING; RESIDENCE is a subclass of BUILDING, while BUILDING is its superclass (Figure 5.3). The residence with the address “26 Grove Street,” for example, is simultaneously an instance of the classes RESIDENCE and its superclass BUILDING.

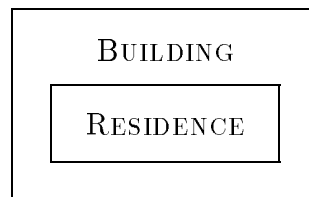


Figure 5.3: A generalization hierarchy with the more general class (BUILDING) depicted around the more specialized class (RESIDENCE).

---

<sup>1</sup>This is not to be confused with the same term used in cartography.

Two properties of generalization should be mentioned in more detail:

- A superclass may encompass multiple subclasses. For example, besides RESIDENCES, there may be other building types such as HOSPITALS and COMMERCIALBUILDINGS (Figure 5.4).

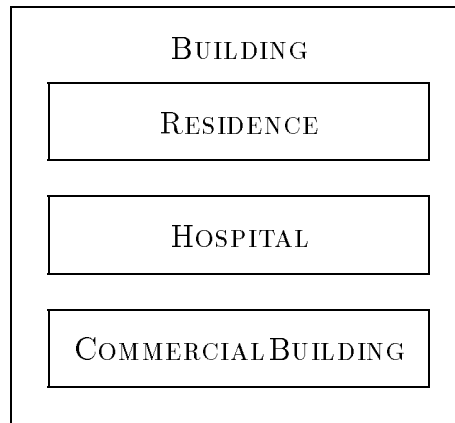


Figure 5.4: A generalization hierarchy with multiple subclasses of a superclass.

- Generalization may have an arbitrary number of levels in which a subclass has the role of a superclass for another, more specific class. For example, the specialization from BUILDINGS to RESIDENCES can be extended with the classes RURALRESIDENCE and CITYRESIDENCE, both being subclasses of RESIDENCE. While RESIDENCE is a subclass of BUILDING, it is at the same time a superclass for RURALRESIDENCE and CITYRESIDENCE (Figure 5.5).

We add to our database (theory) that: all students are persons; all persons are mammals; all mammals are animals; all animals are living things.

```
r (student, superclass, person).
r (person, superclass, mammal).
r (mammal, superclass, animal).
r (animal, superclass, livingThing).
```



## 5.4 Aggregation

An abstraction mechanism, similar to association, is *aggregation* which models composed objects, i.e., objects which consist of other objects. Several objects can be combined to form a semantically higher-level object, called *aggregate* or *composite object*, where each part has its own functionality. The terms *subpart* or *component* refer to the parts of the composite object. Operations of aggregates are not compatible with operations on parts, and vice-versa. When considering the aggregate, details of the constituent objects are suppressed. Every instance of an aggregate object can be decomposed into the instances of the corresponding component objects.

The relation established by aggregation is often called the *part\_of*-relation since aggregated instances are parts of the aggregate and the relationship converse to *part\_of* is called *consists\_of*. For example, a `CITY` may be modeled as an aggregate of all `HOUSELOTS`, `STREETS`, and `PARKS`—they are *part\_of* a `CITY` or, conversely, a `CITY` *consists\_of* them (Figure 5.6).

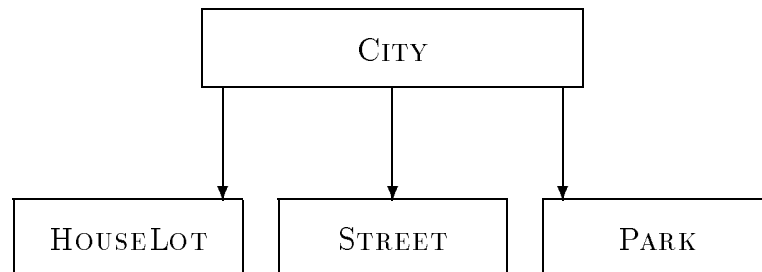


Figure 5.6: A `CITY` modelled as the *aggregate* of `HOUSELOTS`, `STREETS`, and `PARKS`.

Aggregation applied to objects (components) produces an aggregate (or record) type data structure. An operation over an aggregate consists of a fixed number of different operations in sequence or in parallel, one for each component. Hence, aggregation relates to sequence or parallel control structures.

The `partOf` relation in each case is a special one, different from any other. If one

considers the range and domain of these *partOf* relations, one can see that they are all different. It may be necessary to indicate these differences by using specialized relation names, e.g., *partOfFamily*, *partOfHouse*, etc.

## 5.5 Concurrent Abstraction Processes

A warning: the three abstraction processes just described differ fundamentally in the domain and range of the relations involved: entity-entity (aggregation and association), entity-class (classification), and class-class (generalization). Maintaining an appropriate separation, however, is not easy.

For instance, although we have previously defined “dog” as a class of objects, a particular group of dogs may have a collective property/value pair, e.g., an average height, that is not a common property or property/value pair of the individuals of the group. In this case, the collection to which the term *dog* refers is not considered as a generic representation, but as a particular (aggregate) entity; the term *dog* no longer refers to what is essential in being a dog, but to a special, well-defined set of dogs.

Recognizing the distinctions in the use of the term is difficult sometimes, but always critical; otherwise, inappropriate operations may be applied such that incorrect results are produced.

Aggregation always induces classification and vice versa. Entities described as being instances of a class are also, at the same time, described as being part of a group, i.e., the group of instances of the class. Entities defined as being part of something are also instances of a class which represents what it is to be a part of that thing. For example, members of a family aggregate to form a distinct collective, i.e., the family, but the members all have, by virtue of their family membership, the common property of being of that family.

Stella is *partOf* the Frank family, but she can also belong to that class of people who are Frank family members.

Dogs have four legs. “Dogs” is a class and class is an abstract concept; it cannot have legs. The term “Dogs,” however, can also stand for that entity collectively representing all of the current members of the class who have 4 legs.

This dualism may not always be explicitly stated, but it always exists. Humans constantly use it to switch from one version of abstraction to another, placing an in-

dividual into a class, which is then used as an entity for aggregation into some other entity, which may be then further classified, etc. Humans often suspend formalities of interpretation when convenience and efficiency dictate. A good database (theory) should be able to handle similar translations, but it is obvious that its construction will not be easy.

## Chapter 6

# Modeling Behavior

Initially, the abstraction mechanisms were targeted only to model *complexly structured objects*, such as molecules in chemistry, geographic data, or solids in CAD/CAM. Over the last decade, these discussions have progressed to detailed descriptions of the tools to model the *behavior* of objects. These ideas of formalizing behavior are closely linked to notions in software engineering such as *abstract data types* or *algebraic specifications*, which will be discussed in detail in Part 3.

This chapter focuses on the the crucial concepts of *inheritance* and *propagation* to describe the derivation of properties in generalization hierarchies and values in aggregation hierarchies, respectively. Sometimes propagation is also called *upward inheritance*, but it should become clear from this paper that these are two different concepts which must be strictly separated.

### 6.1 Inheritance

In a generalization hierarchy, the properties and methods of the subclasses depend upon the structure and properties of the superclass or superclasses. *Inheritance* is a method to define a class in terms of one or more other, more general classes. Properties which are common to a class and its subclasses are defined only once (with the superclass), and inherited to all objects of the subclass. Subclasses may have additional, specific properties and operations which are not shared with the superclass, but they strictly have all operations and properties of the superclass.

Operations of the superclass are compatible among objects of the superclass and all its subclasses. Every operation on an object of a superclass can be carried out on the subclass as well; however, operations specifically defined for the subclass are not compatible with superclass objects.

The implementation of an operation common to several classes may be different for each class, but must obey the rules set forth by the superclass. For example, two-dimensional and three-dimensional coordinates are both representations for POINTS. From this superclass POINT both subclasses, 2D-POINT and 3D-POINT, inherit operations such as calculating the DISTANCE and DIRECTION between two points. The implementation of these operations is different; however, from the outside they behave the same in terms of the definition of distance and direction.

In terms of the algebra, a superclass is the intersection of all the axioms of all its subclasses, or, the reverse, a subclass contains all the axioms of the superclass plus some additional ones. In order to maintain such simplicity, one has to exclude that a subclass may only inherit parts of the operations prescribed by the superclass. Otherwise, complex exception rules apply.

The concept of inheritance can be concisely represented in FOL. We start with the links between classes and its properties. Each property of a class is expressed as a predicate of the form  $p(\text{Class}, \text{Property})$ , e.g.,

```
p (building, address).
p (building, owner).
p (residence, resident).
```

Generalization is described as the *is\_a*-predicate of the form  $\text{is\_a}(\text{Subclass}, \text{Superclass})$ , e.g.,

```
is_a (ruralResidence, residence).
is_a (urbanResidence, residence).
is_a (residence, building).
```

Inheritance is then defined by the predicate *properties* which recursively derives the properties associated with a class and all its superclasses.

```
properties (Class, Property) IF
    p (Class, Property).
```

```

properties (Class, Property) IF
    is_a (Class, Superclass),
    properties (Superclass, Property).

```

All properties of the class *urbanResidence* can then be determined with the predicate

```
properties (urbanResidence, Prop).
```

which the following values for the variable `prop` fulfill:

```

Prop = resident
Prop = address
Prop = owner

```

Inheritance is transitive, i.e., the properties are passed along from a superclass to all related subclasses, and to their subclasses, etc. This concept is very powerful, because it reduces information redundancy and maintains integrity. Modularity and consistency are supported since essential properties of an object are defined only once and inherited in all relationships in which it takes part.

### 6.1.1 Single Inheritance

Inheritance can be strictly hierarchical and is then often referred to as *single inheritance*. Single inheritance requires that any class has at most one single immediate superclass. This restriction implies that each subclass belongs only to a single hierarchy group and one class cannot be part of several distinct hierarchies.

The following example shows inheritance along a generalization hierarchy (Figure 6.1). `RESIDENCE` is the general superclass and `CITYRESIDENCE` and `RURALRESIDENCE` are the specific subclasses. All properties and operations of the class `RESIDENCE` are inherited to its two subclasses. For example, `RESIDENT` and `MOVINGIN` are associated with the class `RESIDENCE` and inherited to all `CITYRESIDENCES` and `RURALRESIDENCES`. They are compatible with with `CITYRESIDENCES` and `RURALRESIDENCES`. On the other hand, the operations defined specifically for a subclass are not applicable to objects of the superclasses. For instance, `NEXTSUBWAYSTOP` is a property which applies only to `CITYRESIDENCES`.

The transitive property of inheritance implies that any property is passed not only from the superclass to the immediate subclasses, but also to their sub-subclasses,

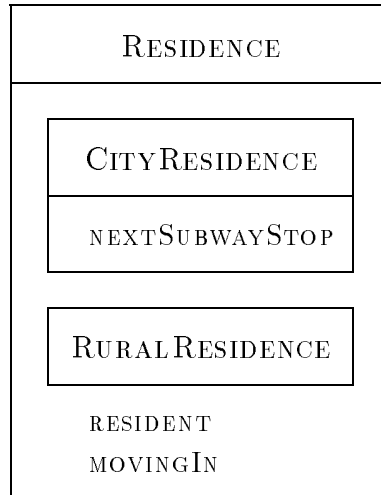


Figure 6.1: Inheritance of the property `RESIDENT` along the generalization hierarchy.

etc. For example, the properties of a `BUILDING`, such as `ADDRESS` and `OWNER`, are inherited to the subclass `RESIDENCE`, and also transitively to the sub-subclasses `RURALRESIDENCE` and `CITYRESIDENCE` (Figure 6.2).

### 6.1.2 Multiple Inheritance

The structure of a strict hierarchy is an idealized model and fails most often when applied to real world data. Most “hierarchies” have a few non-hierarchical exceptions in which one subclass has more than a single, direct superclass. Thus, pure hierarchies are not always the adequate structure for inheritance. Instead, the concept of *multiple inheritance* permits one to pass properties from several higher-level classes to another class. This structure is not hierarchical, because—in terms of the parent-child relation—one child can have several parents. In the simplest case of multiple inheritance, a subclass inherits properties from two distinct superclasses. For example, the different roles of a `LANDPARCEL` as a `TAXABLEITEM` and a `REALESTATEOBJECT` can be effectively modeled by multiple inheritance (Figure 6.3).

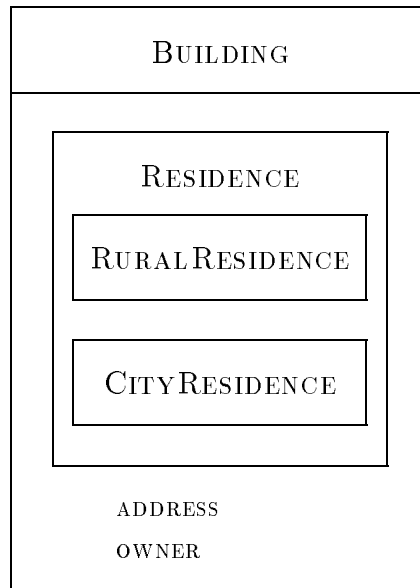


Figure 6.2: Transitively inheriting the properties ADDRESS and OWNER to all subclasses of BUILDING.

A more complex GIS example shows how multiple inheritance combines two distinct hierarchies (Figure 6.4). The first hierarchy is determined by the separation of TRANSPORTATIONLINKS into ARTIFICIALLINKS and NATURALLINKS. HIGHWAYS and CHANNELS are ARTIFICIALTRANSPORTATIONWAYS, and NAVIGABLERIVERS are NATURALTRANSPORTATIONLINKS. WATERBODIES with PONDS, CHANNELS, and RIVERS form a second hierarchy, in which two types of rivers are distinguished: NAVIGABLERIVERS and NON-NAVIGABLERIVERS. Classes with properties from both hierarchies are CHANNELS that are ARTIFICIALTRANSPORTATIONLINKS and WATERBODIES, and NAVIGABLERIVERS that are RIVERS and NATURALTRANSPORTATIONLINKS. These hierarchies cannot be compared with each other, because a WATERBODY is not necessarily a TRANSPORTATIONLINK and, vice-versa, not every TRANSPORTATIONLINK is a WATERBODY either; however, the hierarchies share common subclasses, because

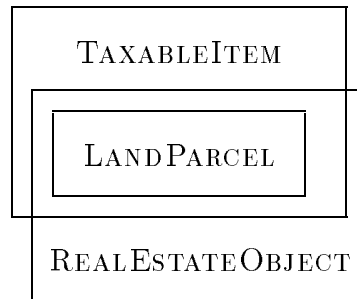


Figure 6.3: A `LANDPARCEL` modeled with multiple inheritance as a `TAXABLEITEM` and a `REALESTATEOBJECT`.

`CHANNELS` are both `WATERBODIES` and `ARTIFICIALTRANSPORTATIONLINKS`, and `NAVIGABLERIVERS` are `RIVERS` and `NATURALTRANSPORTATIONLINKS`. Other classes, such as `HIGHWAY` or `POND`, belong only to a single inheritance hierarchy in this schema.

An argument that received much attention is the problem of *name clashes* or *inheritance conflicts*. If a class has several superclasses, it may inherit distinct operations with the same name, but different meanings. For instance, a `LANDPARCEL` has a `VALUE` as a `REALESTATEOBJECT` and as a `TAXABLEITEM`. Both values are based on different assessments and used for different purposes. Single inheritance has a simple rule to resolve such name conflicts by giving preference to the most specific method (i.e., the one associated with the most detailed superclass). This selection may not necessarily be what was intended with the model, but it is at least a simple and consistent rule. For multiple inheritance, there are no such simple conceptual rules which could capture the intended meaning. Frequently, the conflict is resolved by giving preference to the methods in the order they are listed in the data definition; however, this would not be a valid solution for the value of the `LANDPARCEL`. Since the two names actually describe two different properties, it is necessary to distinguish between them by tagging the property names with their class names, e.g., `REALESTATEOBJECT.VALUE` and `TAXABLEITEM.VALUE`.

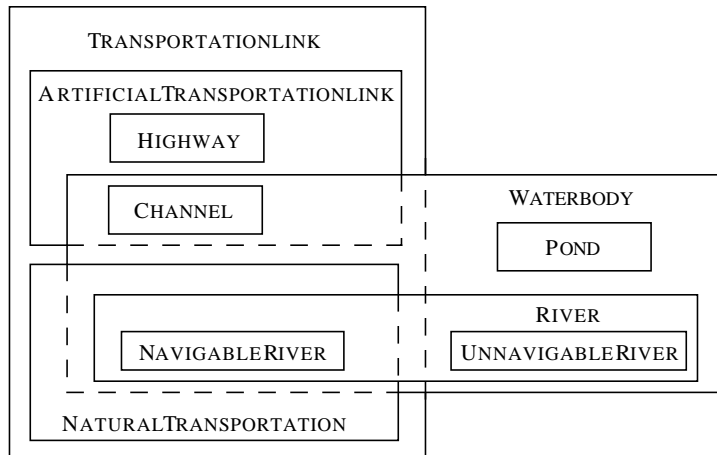


Figure 6.4: A GIS example of the use of multiple inheritance.

## 6.2 Propagation

Frequently, complex objects are not independent and have some property values which rely upon values of other objects. In aggregation hierarchies, for example, some values of a composite object depend on values of the properties of its components. These dependencies are of interest and in order to guarantee consistency and integrity, their correct modeling is crucial. Of course, a composite object may have property values which it owns specifically and which are independent from those of their components. In contrast to less powerful models which require redundant storage of such values, the object-oriented model allows objects to have properties with values which rely on values of other objects and models these dependencies consistently. This model is superior, because it enforces integrity by constraints. These derived values frequently describe geometric or statistical properties. Particularly in GISs, a large number of attribute values at one level of abstraction depends upon values from another level and must be derived from them. When combining local and regional data, this concept of modeling data at different levels of resolution must be used to furnish consistency among dependent values. The population

of a county, for example, depends on the populations of all related settlements; therefore, the value for the property population of a county must be derived from all values of the property population owned by the settlements (Figure 6.5).

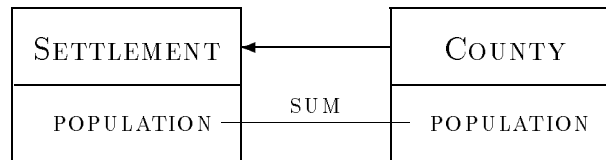


Figure 6.5: The value of the COUNTY POPULATION is propagated as the SUM of the POPULATION of the aggregated SETTLEMENTS.

Again, FOL will be employed to formally describe these concepts. We use the following (simplified) facts which describe the county Penobscot as an aggregate of two settlements Bangor and Orono—and some more in the rural areas—with the property settlementPopulation.

```
p (orono, settlementPopulation, 10000).
p (bangor, settlementPopulation, 50000).
p (orono, partOf, penobscot).
p (bangor, partOf, penobscot).
```

The population of the largest settlement in a county is derived from the settlements as the maximum of their populations. This dependency is expressed by the following rule, stating that the population of a specific county is the maximum of the population of all settlements which are part of it.

```
propagates (partOf, settlementPopulation,
            populationOfLargestSettlement, maximum).
```

The generic rule for propagation is the following predicate. It describes the value of the property of an aggregate in terms of the values of the components using a specific aggregation function.

```

p (AggregateClass, AggregateProperty, AggregateValue)
  IF propagates (Relation, ComponentProperty,
                 AggregateProperty, Operation),
p (ComponentClass, Relation, AggregateClass),
p (ComponentClass, ComponentProperty, ComponentValue),
p (Operation, ComponentValue, AggregateValue).

```

For example, the value of the property `countyPopulation` is then evaluated with

```
p (county, populationOfLargestSettlement, X).
```

and results in

```
X = 50000
```

While inheritance describes properties of subclasses (types and operations), *propagation* describes how a *value* of a property of one class is derived from values of properties of another class. The notion of propagation is sometimes also used for modeling the behavior of operations, such as copy, destroy, print, and save, upon composite objects and how these operations propagate to their components, and consistency of actions. Here, propagation describes dependencies in the reverse direction—from the components to the composite object. Formal definitions of propagation, also demonstrating the differences between inheritance and propagation, have been given in terms of first-order predicate calculus and are also part of more comprehensive algebras for complex objects.

Propagation becomes trivial if the complex object happens to be composed of a single part and the value of the aggregate refers to a single value of the part; however, in most cases propagation involves values of multiple components. If more than a single value contributes to the derived value, the combination of the values must be described by an aggregate function. Aggregate functions combine the values of one or several properties of the components to a single value. This value reduces the amount of detail available for a complex object. It may determine the sum or union of values of the components, or define a specific, outstanding part such as the greatest, heaviest, or conversely, the smallest or lightest one. On the other hand, it may be representative, such as the average or weighted average

of the values of a specific property. Common aggregate operations are minimum, maximum, sum, average, and weighted average. For example, the population of the biggest city in a county is the maximum of the populations of all its cities; the area of a state is the sum of the areas of all its counties; the population density of the state is the average of the population density of its counties weighted by the county areas.

The concept of propagation guarantees consistency, because data is only stored once and the dependent values of the aggregate are derived; therefore, derived aggregate values need not be updated every time the components are changed. Of course, updates underlie the common rules for updates of views, i.e., no derived properties can be updated explicitly, but only the fundamental properties. For example, modifying the population of Penobscot county by assigning the value 65,000 to the county population if the town population of Orono grows by 5,000 is not allowed. Instead, the population of the settlements must be modified which implicitly updates the county population.

Two characteristics of propagation are observed: (1) the propagation of an aggregate value may involve several values from different classes, and (2) propagation may be transitive, i.e., propagated values may be used to derive further aggregated values. The following two examples clarify these characteristics. The `AREA` of a `COUNTY` depends on the `AREAS` of its `LANDPARCELS`, `ROADS`, `LAKES`, and `RIVERS` (Figure 6.6) and must be derived as the sum of all `AREAS` of these components.

An example for the transitivity of propagation is the population of the largest county (`POPULATIONOFLARGESTCOUNTY`) in a `STATE`, which depends on the `POPULATION` of the `COUNTIES`, which in turn is the sum of the `POPULATIONS` of their `SETTLEMENTS`. Implementation considerations recommend that these computationally expensive aggregate operations are reduced to a minimum to improve query performance, sometimes by introducing redundant storage of aggregated values.

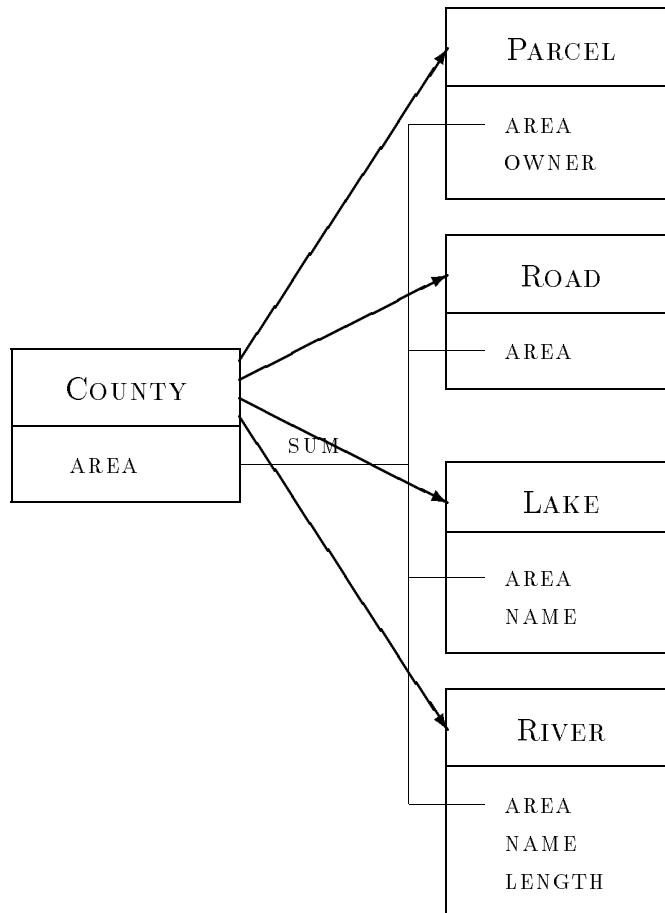


Figure 6.6: The value of the COUNTY AREA is propagated as the SUM of the AREAS of the aggregated PARCELS, ROADS, LAKES, and RIVERS.



## Chapter 7

# The Relational Data Model

The previous chapters introduced a theoretical foundation for an information system which is based on first order predicate calculus and the understanding that an information system is a theory which can be interpreted as a model for some part of reality. That model relies heavily on the concept of data retrieval by logical deduction and for this reason is called the proof-theoretic data model. In this chapter we discuss the more traditional notions of the relational data model. There is a strong theoretical base to the relational model as well, which is referenced by some authors as the model-theoretic view of a database.

The relational model has had considerable influence on database theory and practice in recent times and is the base for a large number of commercially available systems, but we will point out how it is more limited than the latter concept.

### 7.1 The Relational Model Concept

The relational data model was originally proposed by E.F. Codd in 1970. It is based on a firm mathematical foundation and can be (relatively) easily implemented. Thus it became an attractive concept for both theoretical and practical reasons. Most of the database literature produced in the last ten years discusses database problems in terms of the relational data model.

Central to the relation concept is that we can arrange data in form of *tables*, each row describing a distinct fact.

## STUDENT

Name	Student-Id	Major	Level	Street	No.	Town
peter	545	cs	so	main	3	oldTown
max	346	sv	gd	birch	5	orono

The rows in the table are called *tuples*, a name derived by generalizing the terminology describing ordinal groupings such as quintuple, sextuple, septuple, etc. For any individual relation, the tuple component count is always the same (i.e., there is always the same number of items in each row).

The individual tuple components are called *attributes* the values of which are always positionally fixed so that the same kinds of values fall into the same column. The contents of the columns are described by attribute headers (or field names).

This tabular arrangement can easily be translated into a corresponding number of atomic formulae (ground axioms) of the predicate form:

student (peter, 545, cs, so, main, 3, oldTown).  
student (max, 346, sv, gd, birch, 5, orono).

Tuples of the same relation are collected as rows in a table to be stored in the database. It is imperative, therefore, to have something that can identify a given tuple and permit its subsequent retrieval. Such a device is called a *key* and usually one of the attributes is chosen (or designed) to be a key. The relational data model demands that the attribute value for each tuple be distinct from the value of that attribute in every other tuple of that relation.

KEY = An attribute (or combination of attributes) the values of which can uniquely identify each tuple of a relation.

A relation may have more than one key, e.g., the name and the student ID number can serve independently as keys to a student tuple. In this case each independent key is said to be a “candidate” key.

A relation can have a *composite key*, where two or more attributes in combination are required to uniquely identify a tuple, e.g., if the student relation had first and last name as attributes, both together may make a suitable key whereas separately they may not uniquely determine a student.

There are “natural” keys that are values included in the model which identify an object (e.g., a social security number), but most often a natural, certainly unique, key is not available (e.g., a person’s name is not a unique key in a realistic setting, and the use of a social security number may be unlawful in some contexts). In many cases, keys must be artificially created, usually by numbering the tuples.

## 7.2 Relational Operators

The traditional relational data model does not allow the use of rule predicates such as the grandfather ( $X, Y$ ) rule mentioned above. It functions through the application of *relational operators*. These manipulate existing data relations stored in the database to form new relations.

Relational operators always have relations as input and their results are also always relations. Whenever a system always produces a result that is of the same type object as the input to that system, then the domain of the system is said to be *closed*. In the *relational algebra* system, the domain of relations is closed under relational operations. The significance of this is that the operators can be combined in arbitrarily complex ways providing us with great expressive power. A relationally complete set of relational operators consists of projection, selection, Cartesian product, set union, and set difference. Other operations, such as the common *join*, can be expressed as combinations of these five operators.

Relational databases with operations that insure closure are said to be relationally complete. This level of “completeness” is generally taken as the yardstick against which database retrieval languages are measured. Many popular database query languages are, in one way or another, built around the use of these operators and understanding them is, thus, not only of a theoretical interest, but also of practical use.

We will show later that this “completeness” is quite relative and the term is not used in the formal logic sense of “complete and sound” theories. While any single piece of data in the collection can be retrieved, not all possible queries can be satisfied. The relational data model ignores an important class of queries that cannot be formulated with the standard relational operators.

We will now describe some of the more common relational operators. They are defined for relational tables, but some are very similar to operations defined for

sets. Redundant tuples are ignored the way redundant elements of a set are. This requirement that each tuple be unique is implied in the definition of a relational table.

The descriptions below are presented in predicate form, instead of the more typical (but still equivalent) tabular form, in order to accomplish comparisons with the logic based model more easily.

### 7.2.1 Union

Given two relational tables,  $A$  and  $B$ , with the same number of arguments (and, if we consider types, with the corresponding arguments being of the same type), we can form a new relational table as the *union* of the two, denoted by  $A \cup B$ :

Relation  $A$  (e.g., a father relation):

```
father (andrew, stella).
father (henri, andrew).
father (george, henri).
```

Relation  $B$  (e.g., a parent relation):

```
parent (andrew, sella).
parent (irja, stella).
parent (andrew, astrid).
```

Relation  $C$  from  $A \cup B$ :

```
enhanced-parent (andrew, stella).
enhanced-parent (irja, stella).
enhanced-parent (andrew, astrid).
enhanced-parent (henri, andrew).
enhanced-parent (george, henri).
```

An equivalent operation in first-order logic for the union is:

$$c(x_1, x_2, \dots, x_n) \text{ if } a(x_1, x_2, \dots, x_n) \text{ or } b(x_1, x_2, \dots, x_n).$$

For example:

$$\text{enhanced-parent}(A, B) \text{ if } \text{parent}(A, B) \text{ or } \text{father}(A, B).$$

### 7.2.2 Set Difference and Set Intersection

Given two tables,  $A$  and  $B$ , with the same number and types of arguments, the set difference  $A \setminus B$  is a new relation:

Relation  $A$ :

```
parent (andrew, sella).
parent (irja, stella).
parent (andrew, astrid).
parent (irja, astrid).
```

Relation  $B$ :

```
father (andrew, stella).
father (henri, andrew).
father (andrew, astrid).
```

Relation  $C$  from  $A \setminus B$ :

```
mother (irja, stella).
mother (irja, astrid).
```

The set difference in FOL:

$$c(x_1, x_2, \dots, x_n) \text{ if } a(x_1, x_2, \dots, x_n) \text{ and not } b(x_1, x_2, \dots, x_n).$$

Given two relations,  $A$  and  $B$ , with the same number and types of arguments, the set intersection  $A \cap B$  is a new relation:

Relation  $A$ :

```
father (andrew, stella).
father (henri, andrew).
father (george, henri).
```

Relation  $B$ :

```
parent (andrew, stella).
parent (irja, stella).
parent (andrew, astrid).
```

Relation  $C$  from  $A \cap B$ :

father-parent (andrew, stella).

Intersection in FOL:

$$c(x_1, x_2, \dots, x_n) \text{ if } a(x_1, x_2, \dots, x_n) \text{ and } b(x_1, x_2, \dots, x_n).$$

The intersection of two relations is an abbreviation for  $A \setminus (A \setminus B)$ .

### 7.2.3 Cartesian Product

We can construct a new relation from the combination of two relations,  $A$  and  $B$ , (with no restriction on the number of arguments and types) called the Cartesian product ( $A \times B$ ).

Relation  $A$ :

male (andrew).  
 male (henri).  
 male (george).

Relation  $B$ :

female (irja).  
 female (stella).  
 female (astrid).

Relation  $C$  from  $A \times B$ :

dancing-partners (andrew, stella).  
 dancing-partners (henri, stella).  
 dancing-partners (george, stella).  
 dancing-partners (andrew, irja).  
 dancing-partners (henri, irja).  
 dancing-partners (george, irja).  
 dancing-partners (andrew, astrid).  
 dancing-partners (henri, astrid).  
 dancing-partners (george, astrid).

The number of tuples in the resulting new relation can be determined by multiplying the sizes of the originals; this is often a significantly large amount of data to handle. The Cartesian product is an operation most DBMS try to avoid when actually executing a query.

Cartesian product in FOL:

$$c(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n) \quad \text{if} \quad a(x_1, x_2, \dots, x_n) \text{ and } b(y_1, y_2, \dots, y_n).$$

### 7.2.4 Selection

From a relation  $B$ , we can construct a new relation by demanding that certain attributes,  $a$ , have specific constant values,  $v$  ( $\sigma_{a=v}(B)$ ).

Relation  $B$ :

```
parent (andrew, stella).
parent (irja, stella).
parent (andrew, astrid).
```

Relation  $C$  from  $\sigma_{2\text{nd attribute}=\text{stella}}(B)$ :

```
stella-parent (andrew, stella).
stella-parent (irja, stella).
```

If selection is applied over a set of relations, the result is usually a smaller number of new relations, i.e., those that meet the selection criteria.

Selection in FOL:

$$c(x, y) \quad \text{if} \quad (\text{equal}(y, \text{stella}) \text{ and } b(x, y)).$$

### 7.2.5 Projection

From a relation,  $B$ , (with more than one attribute) we can construct a new relation by keeping some of the attributes,  $a$ , which is denoted by ( $\pi_a(B)$ ).

Relation  $B$ :

```
parent (andrew, stella).
parent (irja, stella).
parent (andrew, astrid).
```

Relation  $C$  from  $\pi_{2\text{nd attribute}}(B)$ :

child (stella).  
child (astrid).

The projection operation not only produces shorter tuples, but it often produces fewer tuples as well. The attributes not included in the projection may have been those that differentiated the originals and, without them, the results may contain many redundancies; remember, however, that only one copy of a tuple is retained—at least in relational theory. Some implementations of relational query languages retain all tuple versions regardless of whether or not they are still unique.

Projection in FOL:

$$c(x) \quad \text{if} \quad b(y, x).$$

### 7.2.6 Join

As with intersection, this is not a base operator; it can be composed from others, in this case a Cartesian product and a selection. The join is an operation to produce a new relation from two existing ones, such that tuples are combined when specified attributes,  $i$  and  $j$ , have some relationship (theta) with one another, called a *theta-join* ( $\bowtie_{i\theta j}$ ). When the relationship ( $\theta$ ) is “equals” the operation is called an equi-join ( $\bowtie_{i=j}$ ); when it is “equals” and the attributes have the same name, it is called a natural join ( $\bowtie$ ).

Relation  $A$ :

father (andrew, stella).  
father (henri, andrew).  
father (george, henri).  
father (andrew, astrid).

Relation  $B$ :

mother (Eve, Able).  
mother (irja, stella).  
mother (Eve, Cain).  
mother (irja, astrid).

Relation  $C$  from  $A \bowtie_{2nd\ attribute=2nd\ attribute} B$ —if the 2nd attribute of each relation were named “Child” then the join would have been a natural join:

```
parents (andrew, irja, stella).
parents (andrew, irja, astrid).
```

An equi-join in FOL:

$$c(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n) \text{ if } a(x_1, x_2, \dots, x_n) \text{ and } b(y_1, y_2, \dots, y_n) \\ \text{and equal } (x_i, y_j).$$

### 7.2.7 The Composition of Relational Operators

It is common that actual queries consist of combinations of operators. This is made possible, because the relational algebra is closed under the relations described. The result of one operation is another relation which can participate immediately in another relational operation. The composition can be arbitrarily complex.

As a simple example of a composition suppose we were given the relations father and mother and we wished to obtain a relation of grandparents. First a union of father and mother would give us a relation parent. Then a Cartesian product of parent with parent followed by a selection on those tuples in which the 2nd argument of the first component matched the 1st argument of the second (this is really an equi-join). The result now is a relation describing all 3 generation chains: parent, child of parent, and child of child of parent. To obtain the grandparent relation, simply perform a projection on the 1st and 3rd argument of the latest result. The order of the application of operators is obviously important.

Relation father:

```
father (andrew, stella).
father (henri, andrew).
father (george, henri).
father (andrew, astrid).
```

Relation mother:

```

mother (eve, able).
mother (irja, stella).
mother (eve, cain).
mother (irja, astrid).

```

parent = father  $\cup$  mother :

```

parent (andrew, stella).
parent (henri, andrew).
parent (george, henri).
parent (andrew, astrid).
parent (eve, able).
parent (irja, stella).
parent (eve, cain).
parent (irja, astrid).

```

threeGen = parent  $\bowtie$  1st attribute=2nd attribute parent:

```

threeGen (henri, andrew, stella).
threeGen (henri, andrew, astrid).
threeGen (george, henri, andrew).

```

grandParent =  $\pi$  1st attribute,3rd attribute (threegen):

```

grandParent (henri, stella).
grandParent (henri, astrid).
grandParent (george, andrew).

```

### 7.3 Functional Dependency

A dependency (or functional dependency) exists if one value is dependent on another value in the way of a function. For instance,

$$\text{sqr}(2) = 4$$

If the two is given, the square is always four.

The concept of a functional dependency is crucial to the formal study and practical implementation of a relational database and a dependency exists if one attribute of a tuple is somehow a function of another of the same tuple.

Dependencies are actually assertions about the real world; they cannot be proved, but we might expect them to be enforced by a DBMS. There is nothing inherent in the data that indicates a functional dependency. The recognition of the existence of one is accomplished by the database designer through an analysis of the meanings of the relations and their attributes.

A (functional) dependency is equivalent to a rule added to the data which demands that if a predicate has a specific argument in one position it must also have a specific argument in a corresponding position to be true.

$$r(x,y) \text{ and } r(z,y) \text{ implies } x = z$$

If `father (andrew, stella)` is true then `father (mike, stella)` cannot be true if we have a functional dependency from child to father that each child has only one father:

$$\text{father}(x,y) \text{ and } \text{father}(z,y) \text{ implies } x = z.$$

The existence of a dependency is part of the semantics (the meaning) of the data. Including a dependency in a relational schema is a statement about how we define the relations in terms of their attributes (e.g., the meaning of “father” is here defined as the natural father and does not include stepfathers, guardians, etc.).

If the value of any attribute,  $x$ , depends on the value of some other,  $y$ , then  $y$  is said to *determine*  $x$ , or that  $y$  is a *determinant* of  $x$ . Conversely,  $x$  is said to be *functionally dependent* on  $y$ .

Formally in FOL: An argument  $i$  is functionally dependent on another argument  $j$  if:

$$f(a_1, a_2, a_3, \dots, a_n) \text{ and } f(b_1, b_2, b_3, \dots, b_n) \text{ and } a_i = b_i \text{ implies } a_j = b_j$$

address	(andrew,	30 Grove St.,	orono, ME,	04473)
address	(doug,	14 Island Ave.,	orono, ME,	04473)
address	(george,	207 Bramber Dr.,	broomall, PA,	19008)
address	(eunice,	6434 England St.,	chicago, IL,	60631)

In the above address example, the name, Orono, ME is functionally dependent on the zip code 04473; once the zip code is given, the name of the town is given as well (at least in a simplified zip code system).

We want to avoid storing the same information more than once, in this case that the name of the town for 04473 is Orono, ME. It wastes space, but more importantly, it could lead to inconsistencies if one version is modified and another is not. Avoiding this situation is simple: break up address into two relations, one for simple address, containing name, street, and zip code, and another one for the zip code containing only the zip code and the town name.

Relation address:

```
address (andrew, 30 Grove St., 04473)
address (doug, 14 Island Ave., 04473)
```

Relation ziptown:

```
ziptown (orono, ME, 04473)
```

In the real world, dependencies are a problem, because they are most often “nearly” true, but there are always some exceptions found that contradict the rule: there are some towns that share a zip code (e.g., Bangor and Veazie) and there are towns that use more than one zip code. Reality does not abide by rules as simple as those found in mathematics and it is the information system designer’s primary task to decide which simplifications can be made without interfering with the task.

A dependency can be detected by building the database and then checking if it holds up (i.e., is consistent). This is called an extensional test, and does not really help us in the design, as there is no guarantee that tomorrow some facts will not be stored that produce some contradiction. On the other hand, dependencies are part of the definition of the model, purposely added to the formal system, and cannot, in the model, be ignored or violated. This is a hard decision the designer cannot shy away from.

A functional dependency requires that the implementation be less general than otherwise because of the constraints the dependency imposes upon the structure of the database; however, compensation is realized in that the implementation may become more operationally efficient.

We will often abbreviate functional dependencies by writing  $a_i \rightarrow a_j$ .

### 7.3.1 Transitive Dependencies

Functional dependencies are generally transitive: if  $b$  depends on  $a$  and  $c$  depends on  $b$  then we can conclude that  $c$  depends also on  $a$ ; this is called a *transitive dependency*.

(Zip  $\rightarrow$  Town  $\rightarrow$  Post Office) implies (Zip  $\rightarrow$  Post Office)

### 7.3.2 Partial Dependencies

An argument  $z$  is said to be *partially dependent* on the combination argument  $x, y$  if  $z$  is fully dependent on a part of  $x, y$ , (i.e.,  $z$  is functionally dependent on  $x$  in  $f(x, y, z)$ ), but not functionally dependent on  $x, y$  together.

The importance of recognizing a partial dependency lies in the possibility of  $x, y$  being a composite key.  $z$  would then be dependent on only part of the key, which has ramifications we will explain below.

### 7.3.3 Multi-Valued Dependency

A *multi-valued dependency* (written  $a \Rightarrow b$ ) holds if, for each value of  $a$ , some particular set of values for  $b$  are present.

( $f(a_1, b_1)$  and  $f(a_2, b_2)$ ) implies ( $f(a_2, b_1)$  and  $f(a_1, b_2)$ ).

A multi-valued dependency can be demonstrated in an example with teachers, subjects and texts, where for each subject the same texts are always used, independent of the teacher: subject  $\Rightarrow$  texts.

Earl uses two books in his legal course: EarlsLegalCourse  $\Rightarrow$  Smith, Black.

Multi-valued dependencies are no longer functions in mathematical terminology, since a function is formally a mapping with only one possible result for each input value (i.e., a 1:n relation). Multi-valued dependencies are easy to define, however, they make an interesting formalism of their own (i.e., an n:m relation), though in practical situations they are not particularly common.

## 7.4 Normalization Rules

Previously, we had presented some suggestions on how a database schema should be designed in order to avoid redundancy and other problems in the relational data

model. There exist a set of rules which will avoid some of these problems. They are known as the normalization rules to reach normal forms: first normal form, second normal form, third, fourth, and so on. We will treat the rules for normalization within the context of logic, which is somewhat different from the traditional treatment based on the relational data model.

### 7.4.1 First Normal Form

A relation is in *first normal form (1NF)* if and only if all underlying domains contain atomic values only.

Predicates (and their equivalent tabular forms) essentially already fulfill the requirements for first normal form. Remember that a predicate has a defined number of arguments, called the arity of the predicate; it is not acceptable practice, under our restrictions, to have predicates with different arities share the same name. The arguments for a predicate, in regular relational theory, must be single (atomic) values, i.e., they cannot be multi-valued (arrays, etc.) or refer to other predicates. (A predicate should be constrained to arguments which are of a certain type, but the concept of “type” will be discussed later). The number of arguments of every predicate is fixed and every atomic formula (i.e., a predicate with specific values for its arguments) must have the same number of them.

The requirement that the attribute values be atomic is somewhat restrictive, as the acceptable types for atomic values are usually limited to string, integer, real and do not include arrays, records etc. In our view this is an unnecessary and undesirable restriction, at least for engineering applications (so-called “non-standard” database applications).

### 7.4.2 Second Normal Form

A relation is in *second normal form (2NF)* if it is in 1NF and every non-key attribute is fully dependent on the primary key. To achieve 2NF it may be necessary to break up a predicate several times to eliminate partial dependencies. A predicate that is in 1NF, but not in 2NF, must, therefore, have a composite key. If we introduce artificial, single-value keys, our predicates are automatically in 2NF.

price	(shop'n Save,	old Town,	apples,	.59).
price	(shop'n Save,	old Town,	carrots,	.39).
price	(shop'n Save,	old Town,	onions,	.25).
price	(shaw's,	bangor,	apples,	.59).
price	(shaw's,	bangor,	onions,	.25).
price	(l & A,	orono,	apples,	.59).

The key is the combination of store and item. The price of each item, however, is dependent only on the item and not on the store in which it is sold (e.g., apples are 59 cents everywhere). This example violates 2NF and, therefore, has to be broken down into the following 3 (normalized) relations:

Relation 1:

groceries	(shop'n Save,	apples)
groceries	(shop'n Save,	carrots)
groceries	(shop'n Save,	onions)
groceries	(shaw's,	apples)
groceries	(shaw's,	onions)
groceries	(l & A,	apples)

Relation 2:

location	(shop'n Save,	oldTown)
location	(shaw's,	bangor)
location	(l & A,	orono)

Relation 3:

price	(apples,	.59)
price	(carrots,	.39)
price	(onions,	.59)

If the initial relation was like the following, then it would have been normalized and need no further decompositions:

price	(shop'n Save,	oldTown,	apples,	.59).
price	(shop'n Save,	oldTown,	carrots,	.39).
price	(shop'n Save,	oldTown,	onions,	.25).
price	(shaw's,	bangor,	apples,	.69).
price	(shaw's,	bangor,	onions,	.30).
price	(l & A,	orono,	apples,	.45).

There are no partial dependencies in this relation. The price of each item depends on both the item and where it is sold. This qualifies this example for 2NF.

### 7.4.3 Third Normal Form

The third normal form requires the elimination of transitive dependencies in predicates (usually accomplished by splitting the relation in some way). A predicate is in *third normal form (3NF)* if it is in 2NF and no non-key attribute is transitively dependent on the primary key.

### 7.4.4 Other Normal Forms

It has been determined that multi-valued dependencies are only a specific case of a more general set of dependencies, e.g., join-dependencies, Tableau's dependencies, etc. As a consequence additional normal forms have been defined. Their consideration is beyond the scope of this text.

## 7.5 Practical Consideration of the Relational Database

If the design of a database is not appropriate, we see counter-intuitive behavior of the database during updates. Simple changes have side effects that were clearly not intended. We see difficulties in including some facts that should be possible to add.

Consider the relation:

student-takes (Student, Subject, Teacher, Text)

containing information on which students are taught which subject by which teacher using what text. It is assumed that each student is taught a subject by only one teacher, and that for a given subject all teachers use the same text.

The extension of student-takes (i.e., the listed relations) is:

## 7.5. PRACTICAL CONSIDERATION OF THE RELATIONAL DATABASE 69

student-takes	(max,	law,	earl,	Black)
student-takes	(max,	law,	earl,	Smith)
student-takes	(peter,	geodesy,	werner,	vaniceck)
student-takes	(paul,	geodesy,	alfred,	vaniceck)
student-takes	(john,	english,	berta,	maClan)

Users may request projections of this relation to build lists of all subjects offered by a given teacher, to find all texts used for a given subject, etc.

- If we delete the entry for the student Peter, we also lose the data informing us that Werner offers to teach geodesy—even if no student is actually enrolled; this is called an anomaly in deletion.
- Can we add a new subject with Andrew teaching a course in cartography using a text by Date? Not unless we have a student who enrolls in this course (but how should he or she, if he or she relies on the projection of student-takes to tell him what subjects are offered?)
- In another update example, assume that Earl’s courses are now to be taught by Jim Clapp? We have to modify at least two entries to record this change. Unless we are sure that all entries have been located and modified, we must worry about the possibility of an inconsistent database.
- If a new student enrolls in Earl’s law course, we have to add multiple records to represent the fact that this student will also use both the book by Black and the one by Smith as texts.

This should clearly demonstrate that if the relation/predicate contains several pieces of information that are somewhat dependent on one another, problems may arise.

To reduce the effect of these anomalies in updates, all relations need to be normalized to the highest degree possible by splitting the relations until the only dependencies remaining are those based fully on key values. Unfortunately, even if this is done, problems still exist.

Taken together with the whole design discussion based on the relational data model, concern with normalizing relations is somewhat misdirected; the design

questions are focusing on the wrong end of the problem. Remember that complex situations in reality are intended to be mapped into simple models and, thus, supposedly reduced in complexity. Strange behaviors may then be detected and remedies sought. There are many strange behaviors in the relational model that normalization cannot repair. The relational data model is too simple to adequately map reality and a way must be found to have more real world meaning preserved in the data model.

In reality dependencies exist, but are very seldom clearly defined. Many cases include exceptions for one reason or another, (e.g., zip code and name of town are (nearly) functionally dependent, but exceptions include Veazie and Bangor having the same zip code, and, Orono having two zip codes: one for the town, one for the university). Similar examples can be found almost everywhere. The theoretical discussions of the regular relational data model cannot cope with such “dirty” reality, and it can earnestly be asked that if this is so, of what use are the normalization rules.

## Chapter 8

# Extending the Relational Data Model with Logic

In a data model that has more semantic content can provide help in normalization or, if these problems do not appear at all, this could be taken as evidence that the relational model has too little semantic content (which is perhaps responsible for the popularity it enjoys by theoreticians; there is an enormous literature base about relational theory).

Predicates can be used for representing arbitrary true statements about the world (given an appropriate, fixed interpretation). Their selection would pose few significant problems if users only needed to store facts and retrieve them afterwards; we would only have to guarantee that nothing gets lost—in fact, a lower level of the database software will fulfill exactly this requirement. In an information system, however, users may not only wish to retrieve facts in the form in which they had originally been stored, but they may wish to add some rules to the collection of facts and then inquire about the truth of some predicates that can be deduced from the stored ones.

In our family database with the rule for the deduction of the grandfather relation, users may question (in the usual way):

Is grandfather (stella, peter) a true statement? (the answer is, in this case, no).

We will also permit questions of the form:

Retrieve all X, such that grandfather (henri, X) is true.

This can lead to a definition of a database system which can store facts and rules, provide the means for the retrieval of those facts, and permit the deduction of other facts from them. This is a substantial extension of a simple “storage and retrieval” system which only stores facts and permits retrieval of the same facts, and it is even an extension of the relational data model.

As mentioned in the chapter on formal systems, we may in many ways consider a database system as a theory and the users’ queries as well-formed formulae to be deduced from the theory. We then can say a wff (presented as a query) follows from the theory when data is successfully retrieved from the database. This is called the *proof-theoretic* understanding of a database. It is different from the *model-theoretic understanding* that is prevalent in the current literature, and to which we have been referring as the relational data model. The most significant operational difference between the model- and proof-theoretic views on databases is that, in the relational or model-theoretic database, any deductive retrieval procedures must be specified by the user through queries composed of named relations and relational operators, (or incorporated into the DBMS program code) and in the logic or proof-theoretic database the deductive procedures are part of the database. The proof-theoretic understanding is more powerful and helps us deal with a number of problems the model-theoretic cannot handle easily; furthermore, we believe that it is not more complicated and, given the tools of automatic theorem proving (e.g. as found in PROLOG), easily accessible for execution.

In order to distinguish more easily facts and rules added to the database from queries to the database, we will preface the facts and rules with an asterisk (“\*”). We will also, hereafter, end all complete clauses (from the DB or the query) with a period. Thus, the following predicates indicate facts and rules to entered into the database:

```
father (andrew, stella).  
father (henri, andrew).  
grandfather (X, Z) if father (X, Y), father (Y, Z).
```

while the next two predicates indicate queries to be presented to the database:

```
father (X, stella).
```

grandfather (x, stella).

If we consider the database in the theoretical view we have to add a number of axioms to our theory to have a system with the behavior we expect from a database system. One of the major advantages of the proof-theoretical view is that these axioms can be explicitly stated and are not just implied in the programming code that processes database queries.

Reiter has developed a first-order theory which is at least equivalent to the relational database theory (with its model-theoretic interpretation). He describes a few basic axioms that need to be added to the ground axioms to get exactly the power of a relational database. He then showed that this theory can easily be extended to handle incomplete knowledge, null values, etc., which are all known to be hard problems to deal with in a relational database.

The proof-theoretical view is a way to understand a database; it must not be confused with the implementation of the database program. Using one or another way of looking at a database does not necessarily influence the method of coding it (and does not influence its performance). This is exemplified in the actual PROLOG inference mechanism which includes these axioms in the interpreter's execution and does not require their explicit statement in clausal form.

## 8.1 Domain Closure Axiom

It is necessary to incorporate an axiom into the theory stating that the constants used in the ground axioms are exactly all the constants of the theory. In the view of our model, the individuals explicitly named are the only individuals that exist and that there are no others.

$\forall X : X = \text{stella} \text{ or } X = \text{andrew} \text{ or } X = \text{henri} \text{ or } X = \text{george}.$

There are no other persons (at least not in this mini-world) and  $X$  cannot take on any value except the four mentioned above.

## 8.2 Unique Name Assumption

We must also insist that each constant is different from any other. In other words, two different things always have two different names and any one thing has one

and only one name. Any other assumption causes enormous problems for any data processing installation. Can you imagine a university administration where each student may choose as many aliases as he or she wishes and may register for courses with any of their names?

for all distinct constants  $C, C'$ :  $\text{not } (C = C')$ .

or:

$\text{not } (A=B), \text{not } (A=C), \text{not } (A=D), \text{not } (B=C), \text{etc.}$

### 8.3 Equality Relation

An equality relation must exist and it must have the regular properties of an equality operation and the rule that equal terms may be substituted without effect.

REFLEXIVITY:  $X = X$

COMMUTATIVITY:  $X = Y \Rightarrow Y = X$

TRANSITIVITY:  $X = Y \text{ and } Y = Z \Rightarrow X = Z$

### 8.4 Closed World Assumption

In order to be able to treat negation reasonably it is necessary to add a completion axiom for each predicate. This states that the only true facts that can be obtained from the database are the facts explicitly entered (as ground axioms) or the true facts that are formally deduced from them. Databases are built on the assumption that they contain all the facts that are true for the model and that the absence of a positive fact must be interpreted as its negation.

- The student database for the University of Maine assumes complete knowledge of all students at this university. A person not contained in the database is considered not to be a student at the University of Maine.

This assumption is not only reasonable and encompasses much of the human style of inference from knowledge of the world, but it is clearly economically necessary: to explicitly state all the relations which are not the case, even for a small domain, would be prohibitively expensive!

- If the student database at the University of Maine would not make the closed world assumption, it would need to state for each living human: whether he or she is a student at this university, or is not a student at this university or if the enrollment status is unknown. This is clearly not practical.

Consider the small family database and enumerate all the negative facts concerning fatherhood: for four individuals there are  $4^2$  possible relations of which only three are true:

```
father( andrew, stella ).  
father( henri, andrew ).  
father( george, henri ).
```

The other 13 are false. (For larger domains, the proportions are usually far worse!)

```
father( andrew, henri ).  
father( andrew, andrew ).  
father( andrew, george ).  
father( henri, stella ).  
father( henri, henri ).  
father( henri, george ).  
father( george, george ).  
father( george, stella ).  
father( george, andrew ).  
father( stella, stella ).  
father( stella, andrew ).  
father( stella, henri ).  
father( stella, george ).
```

Humans, in their reasoning, may take into account whether or not they have reasonably complete information and thus feel entitled to use the closed world assumption or not as the occasion demands. Databases typically contain the closed world assumption “hardwired in” and use it always. It is important for users to be aware of this and use all information which is deduced relying on the closed

world assumption with some reservation. Conclusions are typically dependent on the closed world assumption if a negation is included somewhere in the query (example: is Mike not a student at the University of Maine?) or if the answer is negative (example: is George a student at the University of Maine? no).

All PROLOG interpreters rely on the closed world assumption and conclude from the absence of a positive statement, the correctness of its negation (also referred to as “negation as failure” or “the convention for negative information”). As long as a database contains only clauses with positive literals (i.e., atoms without “not”) and we receive a positive answer, the closed world assumption is not used; however, if a negative literal is used during the inference, PROLOG interprets “not  $p(X)$ ” as “from all the individuals in the system, those for which the predicate  $p$  is not stated.” There you find the underlying assumption that the (system’s) world is closed and all individuals are known to it.

Formally, the closed world assumption is stated for each predicate  $p(X, Y)$  as:  
 $\forall X_1, \dots, X_n : p(X_1, \dots, X_n) \Rightarrow p(A_1, \dots, A_n) \text{ or } p(B_1, \dots, B_n) \text{ or } \dots$

The PROLOG implementation for the not operation is:

```
notP (E, P, V) if p (E, P, V), cut, fail.
notP (E, P, V).
```

This is interpreted as “if the positive fact (i.e.,  $p(E, P, V)$ ) is found or deduced, stop the process and return failure for the negation; otherwise return success.” The construction of the second rule, a consequence with no antecedents, always returns true (success).

The domain closure axiom, the unique name axiom, and the closed world assumption together are sometimes referred to as the *particularization axioms*. Remember, all those axioms (assumptions) are not really stored in a working database, but the behavior of the database is as if they were.

## Chapter 9

# Consistency of a Database

A database must respond to a number of demands in order to be useful; among other things, users will expect that the answers they receive do not contradict one another. A contradiction in the information received is a sign that some of the information is not correct. Humans frequently use this situation to critically review information they receive from others. Typically we expect the stories somebody tells us to be consistent and contradictions in some details reduce credibility for the total information received. During cross-examinations lawyers try to find some contradictions in the testimony of a witness; if any are discovered, they can then argue that other points also may not be true. We must expect human users to apply similar standards to an information system.

Consistency may be linked to the very basic concept of a two-valued logic. The definition of “not” in a truth table leads to three different formulations of the rule of “the exclusion of the third” (by this is meant a third truth value: classically “tertium non datur”). The clauses are:

A or (not A)

not (A and not A)

not (not A) equivalent A

It is possible to construct logic systems with three truth values (there is more than one possibility), but none of the proposed systems agrees completely with our intuition. For instance, a “maybe” value can be inserted between “true” and “false” and negation defined as follows:

A	not A
true	false
maybe	maybe
false	true

This is an interesting topic that needs further research, but it is not of consequence at the moment. PROLOG is clearly based on a two-valued logic, and, moreover, it can be shown that all multi-valued logics can be mapped to a two-valued logic (with some additional mechanisms).

## 9.1 Terminology

We will use the term consistency to describe the absence of contradictions in the information received from a database. This not only excludes direct contradiction among the facts reported, e.g., deducing both `father (andrew, stella)` and `not father (andrew, stella)`, but also contradictions between single facts reported and general rules accepted within the model of reality. An example is the rule that a person has only one father, thus an information system that reports `father (andrew, stella)` and `father (mike, stella)` is inconsistent.

Many authors use the terms integrity interchangeably with consistency. Some differentiate between the two terms, consistency describing the absence of contradiction within the database, and integrity asking for non-contradiction with general rules; however, if the general rules are incorporated into the theory, the two concepts unite.

We will tentatively use the term integrity for the completely separate concept that data is preserved and not lost, that only authorized persons can access or update the database, etc.

## 9.2 Formal Definition for Consistency

Formally consistency is a property of the theory and not the model. In order to understand the regular definition, we have to explain what a logician understands

by an interpretation of a theory (this is very close to explanations we have given for a model).

A variable free clause is true in an interpretation  $I$  if and only if, whenever all of its conditions are true in  $I$ , at least one of its conclusions is true in  $I$ . Equivalently, the clause is true in  $I$  if and only if at least one of its conditions is false in  $I$  or at least one of its conclusions is true in  $I$ . Otherwise the clause is false in  $I$ .

This is essentially a definition of the truth for the implication.

A clause is true in an interpretation of a set of clauses  $S$  if and only if every variable-free instance of the clause, obtained by replacing variables by terms from the universe of discourse of  $S$ , is true in the interpretation. Otherwise the clause is false in the interpretation.

A set of clauses  $S$  is inconsistent if and only if it is not consistent. It is consistent if and only if all of its clauses are true in some interpretation of  $S$ .

A theory (a set of clauses) is thus inconsistent if there exists no interpretation for it. This means that there is no way to replace all variables with constants so that all clauses are true. The PROLOG interpreter is based on showing the inconsistency of the theory with the negation of your query and then lists the substitution for the variables, which results in the inconsistency. This process is similar to proving a result by proving the contrapositive.

At least for me, it is somehow surprising that the task of data retrieval and the decision on whether or not something can be deduced from a theory is essentially the same.

Informally we can say that a theory is inconsistent if it contains a contradiction.

There exists  $x$ , such that “ $p(X)$ ” and “not  $p(X)$ ” are both in the theory. (This definition is equivalent to the previously cited one).

### 9.3 Definition of Consistency of a Database

In order to keep a database consistent, we have to first formally define what we understand by consistency. A database does not become inconsistent if we add

```
p ( david, hairColor, black) .,
```

even if we all know that this is not correct. Nor does it do so simply by the fact that it also contains,

```
p (david, hairColor, blonde) .,
```

which already records “blonde” as the color of david’s hair. It becomes inconsistent only when we also add a rule saying that no one can have more than one value for the hair color property. If we agree informally that such a rule is implied by our definition of value, then it can be argued that the database is inconsistent, however, technically it becomes so only after we have added the rule requiring the values to be unique.

It is important not to mix *factual correctness* with *consistency*. Factual correctness is related to the theory modeling certain aspects of the world and the homomorphism between reality and information system. It is not a topic which can be treated logically and within the theory alone. The database cannot walk out into the real world and check if something entered is correct.

Adding a second color for david’s hair, however, is different as it violates our understanding that every person has only one haircolor. A rule that enforces this understanding, can easily be written:

```
equal (C1, C2) if p (Person, hairColor, C1),
                p (Person, hairColor, C2).
```

Similar formulae can be found for other properties which model our understanding of the world and allow us to incorporate them into the theory. An important one is certainly to state that each individual that has a property must belong to some class.

```
There exists C: classExists (I, C) if p (I, P, V),
                                instanceOf (I, C).
```

These consistency rules could be written in different ways and included and used in a database system at various times.

If we add such rules to our theory, the consistency of the database becomes defined as a topic for treatment within the theory. A decision whether or not a database is consistent no longer needs to rely on any additional world knowledge. It becomes a purely formal issue which can be decided using formal reasoning as embodied in a computer program.

In a theory consisting of the database from previous chapters and the new “unique

haircolor” rule, adding “( david, hairColor, black)” makes the theory provably inconsistent. (however, adding “p (andrew, hairColor, blonde)” does not. Why?).

Before we discuss further how to formulate and use such rules, we will discuss the need for consistency from another point of view.

## 9.4 Preconditions of Programs

Ordinary computer programs (e.g., a Pascal or Fortran program) work on some data and produce some output. It is necessary that the input data fulfills some condition, ordinarily called the precondition of the program, and that the result fulfills some other condition, called the postcondition.

A program is only applicable to input that which fulfills the precondition established for it; its behavior is not defined for other inputs. While designing programs, one must be attentive to clearly state what input a program (or program part like a procedure or function) accepts, i.e., what are the preconditions. It is often reasonable, especially if the input is produced by a human user, to make the preconditions as weak as possible. The most general situation is to have an empty precondition, stating that any input is acceptable. This, however, requires much effort for testing within the program for inappropriate input and for producing reasonable error messages when necessary.

Programs with weaker preconditions are often more versatile to use, but in general are more difficult to write. Their code is usually substantially longer. Tests for the preconditions for even simple programs need some processing time. For some complex conditions this time may be considerable.

Assume a program uses a directed graph data structure and that the algorithm employed requires that there are no loops in it. A test for the absence of loops in a network is quite expensive.

Looking from this perspective, we see that consistency constraints become very important in order to allow us to program efficiently; they guarantee certain qualities of the data on which the programs may rely.

If the input data is initially stored in a database, it is better to maintain consistency there. Then programs using the data are guaranteed by the database that the data fulfills their preconditions for input without additional testing. This has the further benefit that inconsistencies in the data are discovered during or shortly after

the input into the database. Generally, errors are cheaper to correct the earlier they are detected.

## 9.5 Consistency Constraints as Conditions on a Database

Recall the two consistency constraints that we have formulated previously:

```
equal (C1, C2) if p (Person, hairColor, C1),
                p (Person, hairColor, C2).
```

means that there can be no person with two different haircolors.

```
classExists(I, C) if p (I, P, V), instanceOf (I, C).
```

means that there must be a class for all entities with properties.

We can use rules like these to check an existing database for consistency or to prevent the input of clauses that would make the database inconsistent.

### 9.5.1 Checking a Database

We can move predicates from the consequent to the antecedent side in a Horn clause by negating it. The first of the two rules stated just above can be written:

```
inconsistentHairColor if p (Person, hairColor, C1),
                       p (Person, hairColor, C2), notEqual (C1, C2).
```

If “inconsistentHairColor” is used as an immediate command and can be satisfied, we know that the database is inconsistent. Thus the desired answer from a PROLOG system should be FALSE (i.e., there is no inconsistency). The second rule modified similarly yields:

```
inconsistentClassExists if p (I, P, V),
                          notClassExists (I).
```

and we describe a predicate notClassExists as:

```
notClassExists (I) if p (I, instanceOf, C), cut, fail.
notClassExists (I).
```

For each consistency constraint we can formulate a rule which cannot be satisfied in a consistent database. This rule states: if the database is consistent, it is not the case that “p” is true. Such rules can then be used to check consistency whenever this seems necessary, e.g., after some data is input.

### 9.5.2 Checking An Input

It is somewhat wasteful to check the complete database after each input, especially if the database is large. It seems much more economical to exploit the fact that, if the database was consistent prior to input, we have only to check the new input for consistency violations.

It is of course not sufficient to check the new input with the rules used to check a complete database as this would amount to forming a new theory with the new input and checking this theory for consistency independently. This is insufficient because an inconsistency can emerge from the combination of the new input with the previous database. This idea becomes obvious in the example with the predicate “hairColor”: the new hairColor, by itself, does not violate a consistency constraint, but merging this new input into the existing database may do so.

In lieu of testing the database after adding a new clause, we have to test if the new clause would make it inconsistent if it were added.

How do we find rules to check new input? Starting with:

```
inconsistentHairColor if p (P, hairColor, C1),
                        p (P, hairColor, C2), notequal (C1, C2).
```

We move the “p (P, hairColor, C2)” to the consequent side:

```
notp (P, hairColor, C2) if p (P, hairColor, C1),
                           notequal (C1, C2).
```

Now we need to change the notp in the consequent into a single predicate:

```
checkHairColor (P, C2) if p (P, hairColor, C1),
                           notequal (C1, C2).
```

If for a given person p and a color C2, checkHairColor is true, we can conclude that adding “p (P, hairColor, C2)” would make the theory inconsistent and should, therefore, not be done. If the result is WRONG, that says that we can safely add the new fact.

For the other consistency constraint, we proceed similarly:

```
inconsistentClassExists if p (I, P, V),
                           notClassExists (I).
```

Moving “p (I, P, V)” to the consequent side:

```
notp (I, P, V) if notClassExists (I).
```

and renaming the notp

```
checkP (I, P, V) if notClassExists (I).
```

This can be read as: do not add a “p (I, P, V)” fact if the individual is not yet defined into a class.

### 9.5.3 More General Considerations for Keeping a Theory Consistent

For the maintenance of larger proof-theoretic databases it seems important to avoid inconsistency in a more general approach. Before we add any new clause we have to check if the resulting theory will be consistent; otherwise the new clause must not be added.

Assume we have a consistent theory  $T$  which is the current database. We wish to add a fact  $C$  and show that  $T \text{ AND } C$  is still consistent. (Technically, we show that  $T \text{ AND } C$  is not inconsistent.) The empty clause (which stands for inconsistent) cannot be deduced from  $T \text{ AND } C$  if  $\text{NOT } C$  can be deduced from  $T$  alone.

Within the framework of the proof methods used in a PROLOG interpreter it may not be obvious how to do this testing as it involves a proof for a negated literal ( $\text{not } C$ ). In PROLOG the absence of a fact is interpreted as its negation—technically called “finite failure interpreted as negation” or “negation by failure.” Thus we cannot differentiate between “provably  $\text{NOT } C$ ,” and “ $\text{NOT}$  provably  $C$ ,” which we would need.

This is a topic which merits some future research as it would provide a most general setting to handle inconsistency. We could install all of the general rules into the database first and then, before each new factual clause was added, it would be automatically checked to see if this would make the theory inconsistent.