tanulmányok

187/1986

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

Magyar Tudomanyos Akadémia
Számítástechnikai és Automatizálási Kutató Intézete
Computer and Automation Institute, Hungarian Academy of Sciences

PROCEEDINGS OF THE WINTER SCHOOL ON

CONCEPTUAL MODELLING

VISEGRÁD

27-30 January, 1986.

Előd KNUTH
András MÁRKUS

(editors)

Tanulmányok 187/1986
Studies 187/1986

Felelős kiadó:

KEVICZKY LÁSZLÓ

ISBN 963 311 216 8

ISSN 0324-2951

C O N T E N T S

LIST OF PARTICIPANTS ... Page

Tomiyama, T.: *Integrated data description scheme* ... 11

Knůth, E., Hannák, L., Hernádi, A.: *Foundations of conceptual representations* 43

Brückler, H., Fritz, W., Haase, V., Kalcher, R.: *Intelligent databases* 65

Siklóssy, L.: *Active collaborative systems* 81

Hernádi, Á.: *Abstraction and data structuring* 89

Davis, M., Mitchell, R.: *Semantic data models: A software technologist's perspective* 111

WINTER SCHOOL ON CONCEPTUAL MODELLING

Visegrád, 27-30 January, 1986

LIST OF PARTICIPANTS

AUSTRIA:

Brueckler, Helmut

Institut für Maschinelle Dokumentation
Steyrergasse 25 a
A- 8010 Graz

Geymayer, Barbara

Institut für Digitale Bildverarbeitung und Graphik
Graz, Wastiangasse 6.

Haase, Volkmar H.

Institut für Maschinelle Dokumentation
Steyrergasse 25 a
A- 8010 Graz

Kalcher, Robert

Institut für Maschinelle Dokumentation
Steyrergasse 25 a
A- 8010 Graz

CZECHOSLOVAKIA:

Kelemen, Jozef

Department of Theoretical Cybernetics
Faculty of Mathematics and Physics
Comenius University
842 15 Bratislava

Tóth Attila

INORGA
Moyzesova ul. 24.
Kassa, 04001

FRANCE:

Ganascia, Jean-Gabriel

Laboratoire de Recherche en Informatique
Bât. 490, Université Paris-Sud
91405 Orsay

Vrain, Christel

Laboratoire de Recherche en Informatique
Bât. 490, Université Paris-Sud
91405 Orsay

HUNGARY:

CHINOIN Gyógyszer és Vegyészeti Termékek Gyára Rt.
Budapest, 1045. Tó u. 1-5.

Greguss Pál

Gödöllői Agrártudományi Egyetem
Gödöllő, Pf.303., 2103.

Hernádi Ágnes

Központi Statisztikai Hivatal Számitóközpontja
Budapest, 1023. Budai L.u. 1-3,

Farkas György
Majtényi Edit
Somogyi Péter

Magyar Tudományos Akadémia Automataelméleti TKCS.
Szeged, 6720. Somogyi Béla u.7.

Simon Endre
Toczki János

Magyar Tudományos Akadémia Központi Fizikai Kutató Intézet
Budapest, 1525, Pf.49.

Krauth Péter
Molnár Bálint
Nicholson Dávid
Papp Mikós

Magyar Tudományos Akadémia Számitástechnikai és Automatizálási Kutató Intézet
Budapest, Pf.63., 1502.

Bach Iván
Bajza János
Bernus Péter
Éltető László
Farkas Ernő
Kelen Miklós
Knuth Előd
Krammer Gergely
Lakatos Péter
Létray Zoltán
Márkusz Zsuzsanna
Máté Levente
Muzsik Gyula
Naszódi Mátyás
Réti Zoltán
Ruttkay Zsófia
Váncza József

Marx Károly Közgazdaságtudományi Egyetem
Matematikai és Számitástudományi Intézet
Informatikai Osztály, Budapest, 1092. Kinizsi u.1-7.

Barna Gyula

Országos Tervhivatal Számitástechnikai Központja
Budapest, 1149. Angol u.27.

Alcziebler Veronika
Asztalos Domonkos
Kiss Zoltán
Krekó Béla
Ulbrich Péter

Országos Vezetőképző Központ
Budapest, 1087. Könyves Kálmán krt.48-52.

Breznay Péter Tamás
Kiss László Nándor

Számitástechnika Alkalmazási Vállalat
Budapest, 1119. Szakasits Árpád ut 68.

Aszalós János
Eiben Ágoston
Kakas Károlyné
Koch Péter
Kovács Kálmánné
Laczay István
Sebestyén Ferenc
Szily Márta
Sztanev Ivánné
Völker-Müllner Ildikó

Számitástechnikai Koordinációs Intézet
Budapest, 1015. Donáti u.35-45.

Balogh Kálmán
Bedő Árpád
Bogdánfy Géza
Domán András
Domokos Mária
Farkas Zsuzsa
Garami Péter
Harányi Annamária
Losonczi Ilona
Móri Judit
Nagy Zsolt
Sándor Gábor
Sántáné-Tóth Edit
Solti Gabriella
Sziray József

ITALY:

Guarini, Nicola

LABSEB
CNR - Padova

JAPAN:

Tomiyama, Tetsuo

Centre for Mathematics and Computer Science
Interactive Systems
Kruislaan 413, 1098 SJ Amsterdam /NETHERLANDS/

NETHERLANDS:

Silóssy, Laurent

Vrije Universiteit Informatica
Postbus 7161
NL- 1007 MC Amsterdam

RUMANIA:

Barbuceanu, Mihai

Institute of Computer Technique and Informatics
Bucharest

UNION OF SOVIET SOCIALIST REPUBLICS:

Salikov, Leonid

Moscow University

Trishina, Elena

Laboratory for Computer Science
Siberian Division of the USSR Academy of Sciences
Lavrent'eva av. 6.
Computing Centre
630090 Novosibirsk - 90

UNITED KINGDOM:

Davis, Megan

The Hatfield Polytechnic
School of Information Sciences
POB 109, College Lane
Hatfield, Herts.
AL10 9AB

Seel, Nigel

STL
London Road
Harlow, Essex
CM17 9NA

UNITED STATES OF AMERICA:

Whinston, Andrew

Krannert Graduate School of Management
Purdue University
West Lafayette
IN 47907

* * * * * *

INTEGRATED DATA DESCRIPTION SCHEMA

- Issues on Representation of Knowledge for CAD Systems -

Winter School on Conceptual Modelling

January 27-30, 1986, Visegrád

Tetsuo Tomiyama

Interactive Systems

Centre for Mathematics and Computer Science

Kruislaan 413, 1098 SJ Amsterdam, The Netherlands,
Telephone +31-20-592 9333, Telex 12571 mactr nl.

ABSTRACT

In this memo we discuss a knowledge representation schema for future CAD systems namely for mechanical design. First, we present an idea of IIICAD (Intelligent Integrated Interactive CAD) systems. These systems will be realized by using techniques of knowledge engineering which are necessary to make the system intelligent and integrated. Among the elements of a IIICAD system, IDDS (Integrated Data Description Schema) is most important, because this is the mechanism to allow free and smooth information flow between system elements and because using IDDL, the language of IDDS, we describe the design objects and the design knowledge for IIICAD. In this memo, we try to derive the specifications for IDDS and IDDL from various discussions, such as formalization of design process, representation of machine, etc., in the context of the conceptual modeling of the design objects and their knowledge representation problem.

1. INTRODUCTION

CAD systems are now not optional but necessary for most of industries. In this paper, we try to clarify many problems and troubles around conventional CAD systems. However, it is difficult to do so in general, because designing activities and philosophy for CAD systems are completely dependent on the target area. Thus, those problems should be discussed in a particular field. We consider machine design as the subject area in this paper.

A typical problem of conventional CAD systems is, for example, that CAD systems do not check errors or mistakes of the designer. Usually, the final drawings are so impressive that no one can detect those errors.

Another problem is integration of different models; this problem is especially significant in case of mechanical design which must deal with complicated structures. A design object must be viewed from many *points of view*, which means in the whole design process one object is represented in many models, such as a geometric model, a kinematic model, a dynamic model, etc., each of which is allowed to have different attributes.

An idea of *product modelling* was proposed [KSH83] to integrate information for all through CAD/CAM activities and it seems very powerful. However, we have not yet seen a final solution which unifies all those models and guarantees the integrity and consistency of the information. We may call such a model virtually *metamodel* and it is a great concern for researchers in this field.

In geometric modeling, the problem of separation of the topology and the geometry of objects have been often discussed. Most of geometric modeling systems have a feature for this separation, and quite often the dimensions can be also separated. This issue would be generalized to the distinction between structure and value. Sometimes, we want to separate the structure of an object from the values of attributes, because what we must first decide is the structure, for instance. On the contrary, quite often structural constraints influence values. In this case, separation of structure from value is not necessary. Therefore, we have two contradicting propositions which are causing a big problem in CAD fundamental research:

- *How can we get a language which distinguishes structure from value?*
- *On the other hand, how can we unify them for expressing the constraints?*

In order to solve these problems of conventional CAD systems, we must avoid *ad hoc* approaches, because more or less designing is an intellectual process of human which may require a deep investigation. Therefore, we need a **theory of CAD** which would indispensably consist of the following three elements to make our direction of investigation firm.

(1) First of all, we need to know what are designing and design processes; i.e., we need a **theory of design**.

(2) We need something for expressing the design objects; i.e., we need a theory of design objects which, in our case, is translated into **theory of machine**.

(3) How to implement representation of objects is really a matter of investigation recently in the field of knowledge engineering, pattern recognition, and computer graphics, for example. So, we need **knowledge engineering** as an implementation technology. Moreover, because a future CAD system must be implemented as a tool for an intellectual process, we need **theories of knowledge, action, learning**, etc., that are more directly related to human thinking process; or, we need philosophy, cognitive science, psychology, etc., something epistemological which may contribute to establishing a theory of design or a theory of machine. Traditionally in the database engineering field, this aspect has been also called **conceptual modeling** [BMS84].

In this paper, we first describe in *Chapter 2* the direction of future CAD systems that are supposed to solve the problems of conventional CAD systems. Secondly, we will point out requirements for future CAD systems coming from discussions on design theory, philosophical consideration, and consideration about design objects, i.e., machinery in *Chapter 3, 4,* and *5*, respectively.

Among the elements of our future CAD system configuration, the mechanism to describe objects in an unified way is most important, because we need to realize ability to describe metamodels, for example. In this paper, we call this mechanism **Integrated Data Description Schema** (**IDDS**) and its language **Integrated Data Description Language** (**IDDL**). In those chapters we gradually clarify the requirements for *IDDS* and *IDDL*. In *Chapter 6*, those requirements result in *IDDL* specifications, and we will show an example of its highly experimental version. The language itself is still under development.

2. DIRECTION OF FUTURE CAD SYSTEMS

The direction of future CAD systems is deduced from problems of conventional CAD systems [ToY85a, ToY85b] The followings are the requirements for future CAD systems; the system should be

- **intelligent**;
- **integrated**;
- **interactive**.

Due to these requirements, our future CAD systems will be called **IIICAD** (pronounced as *three-CAD*) standing for **Intelligent Integrated Interactive CAD systems**. To realize a *IIICAD* system, knowledge engineering is regarded as one of the key technologies, because those requirements are deeply concerned with the nature of human intellectual activity such as designing. This means that future CAD systems will be realized to be knowledge based CAD systems and that they have considerably different configuration from the conventional ones.

Figure 1 shows our configuration of a *IIICAD* system[1]. It has several important components.

(1) **Supervisor** (*SPV*): First of all there must be an intelligent supervisor which watches all the operations in the system, for example, user behavior, information flow, status of the system, etc. By doing this, it at least tries to understand the intension of the user. And, for instance, when the user made an obvious mistake, the supervisor should detect it by

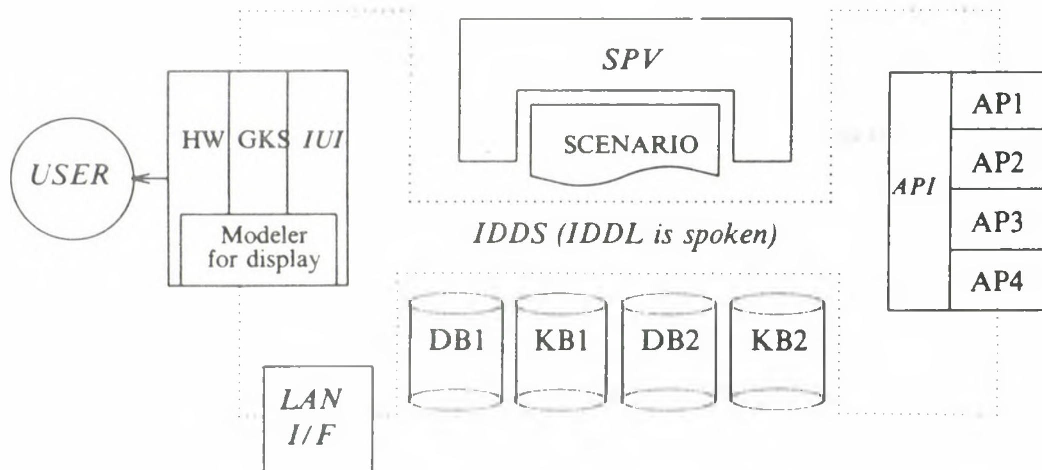


Figure 1. Configuration of IIICAD

[1] The author is now preparing a material about the *IIICAD* system concepts.

comparing the user's actions with a **scenario** which describes a standard designing procedure. This would give intelligence to the system, although the supervisor itself does not have the initiative for the whole design process because the final responsibility for the design should be held by the designer.

(2) **Integrated Data Description Schema** (*IDDS*): Another important element is *IDDS*. One of them is to provide an integrated and unified method for describing models, i.e., a *metamodel*, in the system. *IDDS* is also a gateway for the **databases** and **knowledge bases** (*DB/KB*). Normally, the user does not have to pay attention to where and how to store and retrieve information. All the information, therefore, comes in and out through *IDDS*, which means databases and knowledge bases are transparent to the user. *IDDS* has a language called **Integrated Data Description Language** (*IDDL*), and the features of *IDDS* will be described as those of *IDDL*. *SPV* is drived in *IDDL*.

(3) **Intelligent User Interface** (*IUI*): The interface between the system and the user is controlled by several interfacing systems controlling from very low level input/output to very high abstracted level. The highest level interfacing system is called *IUI* which accepts messages from the supervisor or other application programs written in *IDDL* and sends them to lower level input/output systems like *GKS* which controls the hardware (*HW*) or physical devices. It also accepts user's inputs from, for instance, *GKS* and translates them to descriptions in *IDDL* which in turn will be sent to the supervisor.

(4) **Application Program Interface** (*API*): There must be an interface between *IDDS* and application programs. Following a scenario, *SPV* may invoke necessary and proper application programs for the situation. All the information which will be required by the application should be supplied by somebody; it can be the user, one of the databases, or another application programs. That information will be fed to the application program from *IDDS* in *IDDL*, and *API* should translate it from *IDDL* to proper data format.

(5) **Scenario**: A scenario is given beforehand to describe a chunk of *procedures* necessary to complete the design task, including from abstracted descriptions to very low and simple ones. It may refer to other scenarios to perform a set of prodedures. According to the progress of the design, most suitable scenario will be selected by *SPV* dynamically. The whole set of scenarios selected during a design process may become a record of the design and may be used next time. A set of scenario is normally provided by the system designer and stored in the *knowledge bases*.

As you see, a *IIICAD* system is controled in coorperation of *SPV* using *scenarios* and the user. Its basic actions are described as follows.

(1) Each system component, such as *IUI*, *API*, etc., reports its own status or requests to *SPV*.

(2) *SPV* asks *KB* whether there is a proper scenario for the situation created by the status reports from the subsystems.

(3) Having a proper scenario, *SPV* then passes requests from one subsystem to the most proper one. For example, data request from the application program might be passed to *IUI*, i.e., the user.

(4) The response will be returned to the subsytem which originally asked it. If the subsystem is satisfied with the provided data, it reports satisfaction to *SPV* and *SPV* will proceed according to the scenario. If not, *SPV* must take next procedure which might or might not be described in the scenario.

(5) If the scenario does not have description what to do next, *SPV* tries to find another scenario which looks valid in that situation. If it fails, *SPV* returns the system control to the user, reporting the system status, etc., and asks the user what to do.

Now, we have following two design policies for *IDDL*. These design policies (abbriviated **DP**) will be turned into the specifications later in *Chapter 6*.

DP 1: It must be possible that *IDDL* describe status information of the system.

DP 2: It must be possible that *IDDL* describe control information of the system, i's origin and destination and the time stamp.

As *Figure 1* suggests, there exist a couple of layers or boundaries in the system. According to the kind of information flowing around, there is a boundary called **semantics/syntax boundary** between the supervisor and the rest which separates the semantics layer and the syntax layer. The meaning of these two layers or of this boundary will be described later in *Chapter 4*.

Another important boundary is the **intension/extension boundary** [ToY86] which will be described more precisely also in *Chapter 4*. The extensional layer is a domain where all the information is described in relationships with other entities. Therefore, each entity may have no extra meaning other than being a symbol. On the other hand, in the intensional layer an entity is represented as a set of attributes; thus, an entity itself can be decomposable. This boundary exists between systems which care the information about entities themselves and systems which care the information about relationship among entities.

These two boundaries influences the implementation, because in a particular layer a particular type of information is favored or required and because there exists a certain type of implementation techniques suitable for that particular layer. Thus, we have also several **implementation layers** in our *IIICAD* system.

As a result of having these layers, descriptions (or in conventional terminology, programs) for the supervisor only contain semantical and extensional notations, while descriptions for *IUI* and *API* are mainly occupied by transformation rules from semantical description to syntactical one and from extensional description to intensional one, respectively. This means there is a complete separation between higher level description of the user's intent and lower level description for the system control.

As we already pointed out, in order to build a *IIICAD* system, we need to introduce a technique like knowledge engineering which has several important issues, such as knowledge representation, knowledge acquisition, and inference, in appling it to real problems. When we consider a design problem, for example, we can find several knowledge representational problems; such as knowledge representation of machine, of design process, and design knowledge. Then, a very simple question may arise:

- *Is it really possible to represent a machine in any computer language?*
- *If possible, how do we do that?*

This is a very philosophical question; in order to answer them, we need a theoretically sound basis to handle the knowledge. This means we need a design theory [ToY86].

3. ISSUES COMING FROM DESIGN THEORY

A good design theory is necessary from the following reasons.

(1) In order to build a CAD system that can support and help the designer in all the design process, we need to formalize a design process.

(2) In order to implement a *IIICAD*, we need knowledge engineering to handle our knowledge about design. This requires a theory to formalize our design knowledge.

General design theory proposed by Yoshikawa [ToY86, Yos81] is a theory that can fulfill these two requirements. It is based on axiomatic set theory and assumes three axioms; and then we can derive theorems which are supposed to explain real design processes. From the result of general design theory, we can also deduce several important issues for the designing of a *IIICAD* system. In this chapter we attempt to interpret important results of general design theory from a viewpoint of building a *IIICAD* system.

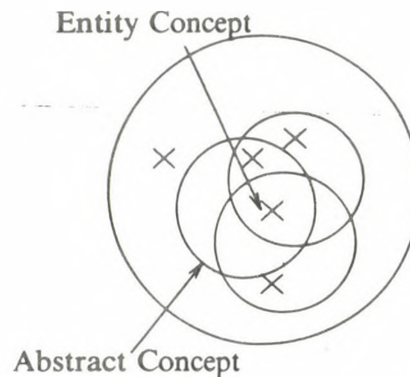


Figure 2. Extensional Definition of Entities

3.1. Interpretation of Axioms of General Design Theory

Now, we try to show important issues coming from the axioms of general design theory. These issues are not necessarily relevant to the theory itself but to the usage of the theory.

AXIOM 1: (Axiom of recognition) *Any entity can be recognized or described by attributes and/or other abstract concepts.*

In general design theory, an entity itself is not defined; thus, it is a rather symbolic existence and anything can be an entity. Instead of defining directly what an entity is, we just guarantee the observability of entities, which made this axiom look rather trivial. However, in fact, it yields a further philosophically big issue, because it says that the representation of an entity is given by its attributes and that this is the only way for describing an entity (*Figure 2*). Actually, it says the description method of an entity concept must be **extensional** (or **denotative**), but not **intensional** (nor **connotative**). In other words, an entity is classified into some category according to its relative position to another entity, such that its relationships with other entities should be described. Here, classification is carried out by counting up of entities that belong to the same attribute concept, although it might be done in an either subjective or objective way.

Suppose we have a watch. This watch can be described as an integration of all the parts, from a metal case to a quartz oscillator. Then, it is possible to regard a list of those parts as the whole watch. But this is strange, because the list itself does not construct a watch. A watch is a watch because it is not a list of parts. Therefore, extensional representation means something holistic and all the attributes, including so-called structure, do not define the entity. Because attributes are generated by observation followed by abstraction, they do have nothing to do with the real structure or whatever an entity has as its properties. Thus, we have:

DP 3: The expression of *IDDL* should allow an extensional description.

Before proceeding to the next axiom, we define two terms, entity set and concept of entity.

DEFINITION 1: *The entity set is a set which includes all entities in it as elements. By all entities, we mean entities which existed in the past, are existing presently, and will exist in the future. This set is denoted by S'.*

DEFINITION 2: *A concept of entity is a concept which one has formed according to actual experiences with an entity. This concept is different from an abstract concept, i.e., a concept of attribute or function, which is abstracted from the entity.*

AXIOM 2: (Axiom of correspondence) *The entity set S' and the set of entity concept (ideal) S have one-to-one correspondence.*

This axiom indicates that it is enough only to think about the set of entity concept S instead of the entity set S', because between these two sets there is a perfect one-to-one mapping. However, by definition, S may include even entities which will exist in the future, and this ideal knowledge is far from our real knowledge. We have to consider the *relationship* between the logical world and the real physical world. In this context, this axiom guarantees the existence of a superman who knows everything; in other words, it just shows an ideal and ultimate state of our knowledge, and that we have only imperfect design knowledge. This forces us to check the *feasibility* or *compatibility* of the knowledge with the realities besides the completeness, soundness, and inconsistency checks in our realistic world.

DP 4: *IDDL* should provide facilities or a mechanism to check the completeness, soundness, and feasibility of the knowledge.

Now, we have the third axiom.

AXIOM 3: (Axiom of operation) *The set of abstract concept is a topology of the set of entity concept.*

This axiom signifies that it is possible to operate abstract concepts logically, as if they were just ordinary mathematical sets. Accordingly, we get set operations, such as intersection, union, negation, etc. From a mathematical point of view, because axiomatic set theory is based on predicate logic which is normally associated with traditional two-valued logic or *natural deduction*, general design theory must also follow natural deduction. In natural deduction, a proposition

$$P \vee \neg P,$$

generated from any logical term, P, is always true by *the law of the excluded middle*.

But, it is sometimes happens that we cannot decide between true and false in designing, unless we have another information or proposition Q to decide it. In fact, in everyday inference at least distinguishing *known* from *unknown* is necessary, which means introduction of intuitionism.

DP 5: It is necessary to introduce three valued logic (or intuitionism) to *IDDL*.

3.2. Useful Results of General Design Theory

Two important and useful results of general design theory [ToY86] are described here. One is introduction of distance into the attribute space; the other is formalization of design processes with the metamodel concept.

3.2.1. Distance in the Attribute Space

Two theorems tell the following:

THEOREM: *In the real knowledge there exists a distance between two different entities.*

THEOREM: *In the real knowledge an attribute has a value.*

When you read these two, you might feel that these two sound trivial. However, they have significant meanings and should be interpreted as follows:

(1) Given a certain metric, different entities, s_1 and s_2, can be measured differently; here, the metric will be given by a function

$$f: S \rightarrow [0, 1], \ (S\text{: Real knowledge}),$$

and attributes have values, if we have a proper distance function, d, such that

$$0 \leqslant d(s_1, s_2) = |f(s_1) - f(s_2)| \leqslant 1$$

(2) We may use attributes as the metric as far as attributes are second countable. That means all the attributes *cannot be always* measured. In order to clarify what kind of attributes can be measured, we probably need a study on representation methods of physical phenomena. Only from this study, we will obtain a guide line about how to represent things rationally.

(3) When we have two candidate design solutions, A and B, it is possible to judge whether A is nearer to the specifications than B or not.

(4) It is also possible to measure the convergence speed of the design solution.

(5) There exists a *mule* between a *horse* and a *donkey*. However, it never guarantees the existence of a donkey nor gives how to *creat* a donkey, but it simply explains its existence.

In addition to these issues, the theorems indicate another important aspect about how to describe a value of an attribute. Usually, a triple of

$$[attribute_name,\ type,\ value]$$

is used for describing an attribute and its value. However, this notation is vague because it is not clear whether this notation specifies the structure of an attribute or the value itself.

The theorem tells that there exists only a distance between two different entities and that the value of a particular attribute is generated from the function which gives this distance and a particular *standard* entity which we can naturally set to 0. This means that the value, v, of an attribute, A, of a particular entity, e, is given by a function (or a predicate),

$$v = A(e).$$

We also distinguish two cases; one is when e has an attribute a, and we write this as

$$A(e).$$

The other is when we want to say that A of e has value v. In this case, from now on we use a notation,

$$v = A(e).$$

Note that the first notation suggests only that e has an attribute A of which value is e but not that e has a structure which is represented by A. As we discussed in *Section 3.1*, simply gathering attributes together does not constitute the structure.

DP 6: *IDDL* should have a distinction between the fact that an entity has an attribute and the fact that an attribute has a value.

3.2.2. Formalization of Design Processes

Another important result of general design theory is that it can give good explanations of a design process together with a reasonable definition of *metamodel*. In fact, in general design theory we can derive a design process model called *evolution model* which indicates a design process is an evolution process of the metamodel [ToY86].

A *metamodel* can be defined as a finite set of attributes and the metamodel set is defined as an intermediate space between the function space where specifications are written and the attribute space where solutions are obtained. A design process, in the evolution model, is explained as follows.

Given the specifications functionally described, we may imagine a rough description of an entity concept as a candidate. The attributive description of this candidate will be detailed according to the progress of the designing. The whole design process will be explained as successive single steps. Each step corresponds to a single design procedure such as finite element analysis, motion analysis, data analysis of a specific experiment, etc. In this way the designing proceeds and accordingly the amount of information of the metamodel increases.

This evolution model suggests many important issues.

(1) A design process can be decomposed into small basic design procedures. The whole design process will be realized and simulated in a CAD system by a set of those basic procedures, which in turn means the whole process will be described by a *scenario*.

(2) The metamodel produces models for evaluation of functions [ToY85b]. At each design step, the metamodel contains attributive information which has been decided so far. The design procedure for that step extracts some interesting attributes from the metamodel and creates a model for an evaluation. This process will be formulated as follows. Let M_i be the metamodel at a design step i and d be a procedure or function to extract information for the creation of a model m_{ij} under a specific circumstance or field f_j. Here, functional evaluation of a design solution is equivalent to evaluate the model

$$m_{ij} = d(M_i, f_j).$$

Therefore,

$$e(m_{ij}) = e(d(M_i, f_j))$$

would give the design decision, where e is an evaluation function.

(3) By using the metamodel concept, we can integrate all the models which appear in a designing process. This means that in a design process the metamodel is the central database from where we can get the necessary information for creating models. And, as shown in the definition, a metamodel should be described by a finite number of attributes expressed in a way described in the previous section.

DP 7: *IDDL* should be used for describing also for scenarios.

DP 8: *IDDL* should have an ability to describe metamodels, so that we get a model for an evaluation in a particular condition.

4. ISSUES COMING FROM PHILOSOPHICAL (METAPHYSICAL) CONSIDERATION

In this chapter we discuss issues about our philosophical view, because it is necessary to decide our standpoint for describing things. In whatever way the user may want to describe a thing, *IDDS* should provide a framework and a mechanism to complete it. In other words, *IDDS* must be flexible enough to make the user free to describe anything he wants.

We employ here first order predicate logic for notation. Although *IDDL* does not necessarily have to employ logic programming, in this chapter we just use it for convenience.

4.1. Syntax and Semantics

A *IIICAD* system should be *intelligent* by definition. Then, what is an intelligent workstation? Because there is no consensus about *intelligence*, we can use our own definition[2].

- *An* **intelligent** *system is one that can handle user's semantics.*
- **Semantics** *in a particular domain is what gives or defines (meaningful) relationships, names, etc. to the symbols used in that domain.*
- **Syntax** *is defined to be equivalent to the topology on the set of the symbols.*

These definitions are more precisely denoted as follows. Let S_b and O_p be the set of symbols and of operators, respectively. The set of formulae F can be defined as the set of lists of symbols and operators; i.e.,

$$F \equiv \{(l_1, l_2, \cdots) \mid l_i \in S_b \vee l_i \in O_p, \text{ and grammatically valid}\}.$$

Relationships among symbols will be defined as predicate logic formulae as follows; this is the definition of *syntax*.

$$S_y \equiv \{(r, l) \mid r \in R,\ l \in F\},$$

where R is the set of relationship. Of course, R cannot be arbitrary; if we want to have first order

[2] This also indicates that the terminology is valid only here. Any definition is possible.

Table 1. Various Layers in a IIICAD System

	Description	Implementation	Example
Intensional	Notational function	Object oriented	GKS, AP, IDDS
Extensional	Syntax S_y	Predicate oriented	IDDS, IUI
	Semantics S_m	Predicate oriented	IDDS, SPV

predicate logic, we need to restrict it such that

$$R \cap S_b = \emptyset.$$

Otherwise, we will have higher order predicate logic (see *Section 4.4*).

Then, *semantics* S_m should be defined as a subset of syntax S_y **by the user**, i.e.,

$$S_m \subset S_y,$$

so that it must have a specific meaning for the user under a specific circumstance. This is necessary, because a predicate cannot make sense unless the term (or subject) belongs to a particular category. For instance, a fact,

$$weight(x, 50),$$

has no meaning, if x is not an object in the three dimensional world. If it is a point or a line, this fact is totally nonsense.

This distinction of syntax from semantics results in the following distinction of input checking, for instance. Suppose we have an input x, and there may exist three types of checking.

(1) $x \in S_b$, $x \in O_p$, or $x \in F$ is a **lexical** check.

(2) $x \in S_y$ is a **syntactical** check.

(3) $x_1, x_2, \cdots \in S_m$ is a **semantical** check, because usually we must take the sequence of inputs into consideration.

This means that input checking and error recovery should not executed by one subsystem. For example, lexical checks should be done by a lower level subsystem like *GKS* or *IUI*, whereas syntactical checks should be done by *IUI*, and semantical checks by *SPV* (see *Table 1*).

DP 9: In *IDDL*, the semantics should be defined by the system designer; i.e., the system designer should be allowed to implement application dependent semantics.

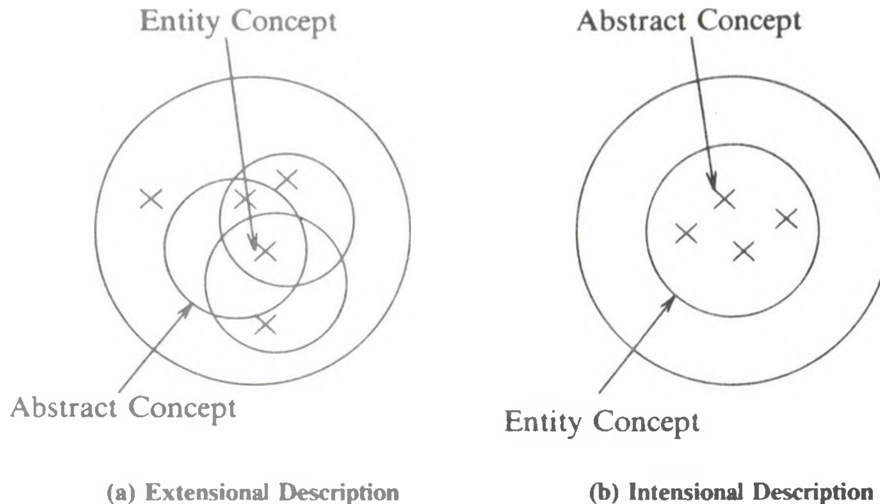
4.2 Extension and Intension

As we have already pointed out in *Chapter 2*, there is a boundary called the intension/extension boundary in a *IIICAD* system. In *Section 3.1*, we have discussed that the world view of general design theory is extensional which at the same time means somehow holistic view. Here, we think about this issue for further details, especially, from a viewpoint of CAD application. *Figure 3* shows a comparison of the extensional description and the intensional one in a set notation.

4.2.1. Extensional Description

An *extensional* description is defined as a situation where an entity concept is an element of the entity concept set and an attribute concept is its topology (see *Figure 3* (a)). This description method has the following properties.

(1) Mathematically, the situation in *Figure 3* (a) is defined as follows. Let e_i and A_j be an entity concept and an attribute concept, respectively. Then, A_j is defined extensionally,



(a) Extensional Description (b) Intensional Description

Figure 3. Extensional/Intensional Descriptions

using e_i, as

$$A_j \equiv \{e_1, e_2, \cdots\}.$$

(2) As the definition above suggests, an attribute is a description found common to all those entities; this means first a common property is found by an observation and then this property is named an attribute, A_j. In other words, an attribute, A_j, is telling a relationship between entities, $e_1, e_2, \cdots$.

(3) In this context, an attribute concept is a product of abstraction following observation and recognition of entities. Moreover, it is influenced by the *sense of value* which was dominant at the time of abstraction and also by the entire entity set. Therefore, an attribute concept is relative, but not absolute.

(4) On the other hand, an entity concept is philosophically a symbolic existence. It is nothing else but a symbol e_i.

(5) An entity concept is only identified and described by attribute concepts defined to be a topology of the set of entity concepts. Thus, an entity concept is also relative (to other entity concepts), which means it will be only described in the relationship between that entity concept and others.

(6) An entity concept cannot be decomposed, because it is a symbolic existence and because it is not described by small parts. Therefore, the extensional description is holistic, which implies everything in the domain world will be denoted only by a collection of facts about relationships among (symbolic) entities. (From this, the description method can be called also *fact oriented* description.) Actually, in this *metaphysics*, existence of entities is not observed; there exist only relationships.

4.2.2. Intensional Description

On the contrary, an intensional description is defined to be a situation where an attribute concept is an element of the attribute concept set and an entity concept is its topology (see *Figure 3* (b)). This description method has the following properties.

(1) The situation in *Figure 3* (b) is defined as follows. Let a_i and E_j be an attribute concept and an entity concept, respectively. Then, using a_i, E_j is defined intensionally by

$$E_j \equiv \{a_1, a_2, \cdots\}.$$

(2) The definition is based on a metaphysics that an entity should be described by its attributes. Sometimes, attributes may represent so-called structure of an entity, which may further suggest an entity is decomposable into small particles.

(3) An entity concept in the intensional description is generated from attributes which will be given somewhat existing beforehand and hence absolute. Therefore, an entity concept can be often described by a fixed number of attributes as a Cartesian product set, like

$$E_j \equiv \{(a_1, a_2, \cdots) \mid \Sigma(a_1, a_2, \cdots)\},$$

where Σ is an additional set of constraints.

(4) Philosophically, in an intensionally description, an entity concept is equivalent to a collection of attributes, which will be regarded as an object. This is one of the properties of famous *object oriented* programming paradigm. In this paradigm, there exit no abstract [illegible] relationships among entities but concrete objects.

4.2.3. Further Notes on Extensional/Intensional Descriptions

The reader may have been confused by the terminology. The author admits that it is not so easy to understand the difference between an extensional description and an intensional description. To solve this problem, let us consider a concrete example.

The implementational advantages and disadvantages of these two description methods will be further discussed in *Section 4.2.4* and in *Chapter 5*, especially, when they are employed to CAD applications.

4.2.3.1. Hierarchical Example

Suppose you have a car which consists of lots of parts, from an engine to a radio, and the engine itself can be decomposed into thousands of small parts. We traditionally call these part-assembly relationships *hierarchy* shown in *Figure 4* as a result of abstraction which is a typical human ability. Note that all the names appearing in this figure refer to concrete entities or instances; they are by no means generic names. *Figure 4* is showing more or less something specific to your car.

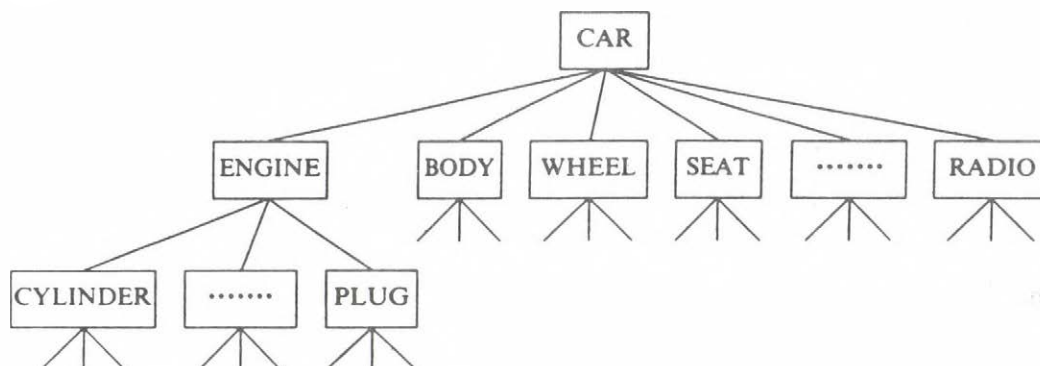


Figure 4. An Abstraction of a Car (Some Part of Hierarchy)

In an extensional description method, probably the most natural interpretation of *Figure 4* is as follows:

(1) First, we regard a symbol, such as *CAR*, as an atom which cannot be decomposed any more; thus, *CAR* is an identifier and can be replaced by another meaningless string such as @*AB*5306.

(2) Then, an arrow that connects two symbols is regarded as the relationship between them. Maybe, we can use *consists_of* relationship for the moment, although its meaning cannot be directly defined right now.

(3) For example, the relationship between *CAR* and *ENGINE* will be interpreted such that *CAR* and *ENGINE* make a subset in the entity set as a topology;

$$consists_of = \{CAR, ENGINE\}.$$

(4) Because the relationship *consists_of* here is specific to *CAR* and *ENGINE*, we need to introduce different names as many as the relationships. But, this is not necessary and we can use the same name for all other relationships, as far as we understand implicitly the meaning of these relationships are basically the same.

(5) Therefore, this hierarchy will be denoted in a following way, using a predicate *consists_of*.

consists_of(*CAR*, *ENGINE*),
consists_of(*CAR*, *BODY*),
consists_of(*CAR*, *WHEEL*),
consists_of(*CAR*, *SEAT*),
consists_of(*CAR*, *RADIO*),
consists_of(*ENGINE*, *CYLINDER*),
consists_of(*ENGINE*, *PLUG*).

On the other hand, in an intensional description method, *Figure 4* will be interpreted in a totally different way.

(1) In this description method, there is a strong belief that a *CAR* is decomposable into fragmental parts such as *ENGINE*, *BODY*, etc. A *CAR* is built from combination of those parts.

(2) A symbol, such as *CAR*, is not a mere identifier, because it has an inner structure which is strongly related to a string "*CAR*."

(3) Each part can be further decomposed, like *ENGINE* is decomposed into *CYLINDER*, *PLUG*, etc. Therefore, an intensional description method may lead us to ideas such as typing (or class, subclass, etc., any equivalent concept), object and its instance, etc.

(4) The relationship, *consists_of*, we have discussed is hiding in the structure of entities.

(5) The following is an intensional notation of this hierarchy.

$CAR = (ENGINE, BODY, WHEEL, SEAT, \cdots, RADIO)$
$ENGINE = (CYLINDER, \cdots, PLUG)$

4.2.3.2. Meanings

It must be noted that concepts of syntax/semantics and extension/intension are very much alike but completely different (see *Table 1*). It is possible that both of intensional and extensional descriptions have meanings.

- **Intensional meaning** is defined to be *meaning of a symbol*.
- On the other hand, **extensional meaning** is defined as *semantics* which was defined in *Section 4.1*

The *meaning of a symbol* is further explained in *Section, 4.4*, and it is in fact defined by the concept of object oriented programming paradigm.

4.2.3.3. Comparison of Extensional/Intensional Descriptions

It is interesting to compare the extensional description method with the intensional one. We can compare these two from various points of view.

Figure 5 shows one of the disadvantages of the intensional description method. *Figure 5* (a-E) shows that two *different* abstract concepts are denoting an entity. This situation can be also *identically* described in the intensional description as in *Figure 5* (a-I). However, as in *Figure 5* (b-E), if two *similar* or *hierarchical* abstract concepts are denoting an entity, the similarity or the hierarchy cannot be expressed so well in the intensional description (see *Figure 5* (b-I)); they are just represented in the same way as in *Figure 5* (a-I).

The whole story tells that, in case of the intensional description method, slight differences in the meaning would be lost or ignored; or quite similar concepts would be recognized differently.

This type of loss of data would also happen in an exchange of information between an extensional description method and an intensional one. This can be pointed out mathematically. Suppose $\downarrow x$ be an intension of x, and $\uparrow x$ be an extension of x. Mathematically,

$$\uparrow\downarrow x = x$$

always holds; but, the other way around

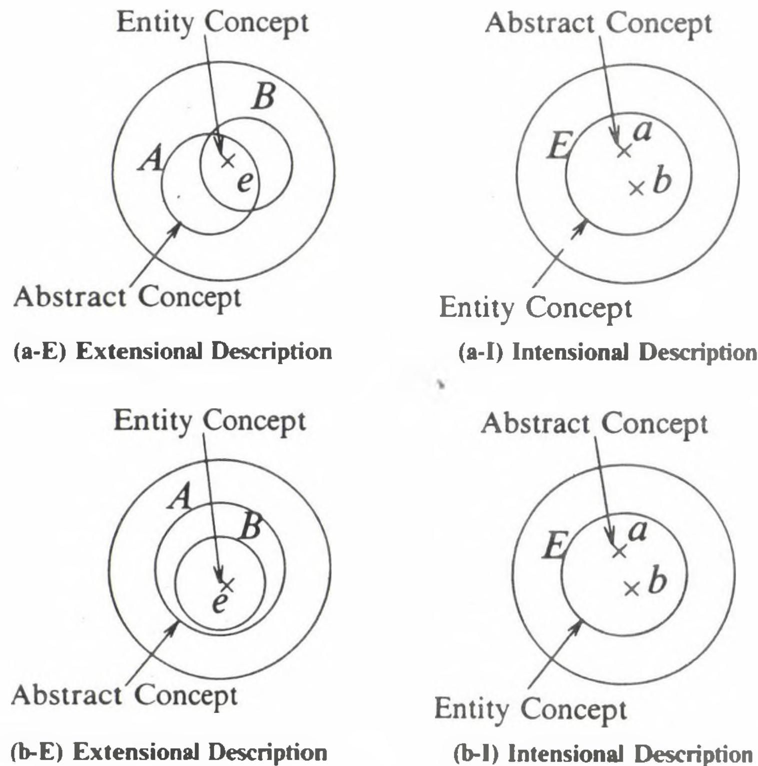


Figure 5. Comparison of Extensional/Intensional Descriptions

$$\downarrow\uparrow x = x$$

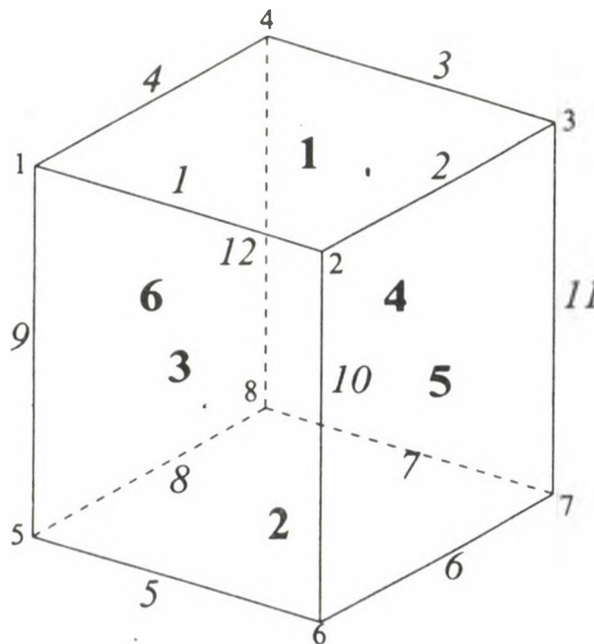
does not always hold.

A famous example is *the morning star* and *the evening star*. The morning star has an extension, *Venus*. The evening star is an intension of Venus. Thus, an intension of an extension of an entity is not always identical to the entity. Therefore, when these two data description methods are necessary and when we need to exchange information between these two, we must be careful for loss or twist of information caused by the exchange.

4.2.4. Implementational Notes on Extensional/Intensional Descriptions

Although this chapter is for discussions from a philosophical or metaphysical point of view, it might be useful to compare implementations of extensional and intensional description methods from a practical point of view. Let us examine it problem with a concrete example.

Figure 6 shows a cube, and the following facts denoted by predicates are its extensional representations, because in an extensional representation system the subject is an entity and the predicates are its attributes.



NB: Roman numbers indicate vertices. *Italic* numbers indicate edges. **Bold** numbers indicate surfaces.

Figure 6. A Cube

vertex(1).
...
vertex(8).
line(*1*).
...
line(*12*).
surface(**1**).
...
surface(**6**).
consists_of(1, *9*).
consists_of(5, *9*).
...
consists_of(*1*, **1**).
consists_of(*2*, **1**).
...
consists_of(**1**, *cube*).
...
consists_of(**6**, *cube*).

On the other hand, the intensional representation of this cube would be

$$cube\{(\mathbf{1}, \mathbf{2}, \cdots, \mathbf{6}, \mathit{1}, \mathit{2}, \cdots, \mathit{12}, 1, 2, \cdots, 8) \mid \Sigma(\mathbf{1}, \mathbf{2}, \cdots, \mathbf{6}, \mathit{1}, \mathit{2}, \cdots, \mathit{12}, 1, 2, \cdots, 8)\},$$

where Σ implies the necessary conditions for this object to exist as a cube. Here, the entity, *cube*, is regarded as the predicate and its attributes are the subjects. Usually, this data structure can be realized in CAD systems as a set of data connected by pointers illustrated in *Figure 7*, or sometimes as a tuple of a relational database [Lor82] (*Table 2*).

In an extensional representation system, the data would be described by a set of facts (e.g., by predicate logic formulae) independent from each other. Even the constraints will be expressed generally by a set of predicates. Generally speaking, to change or to add new facts in this data representation scheme is not difficult. To modify a particular data is just a matter of checking the

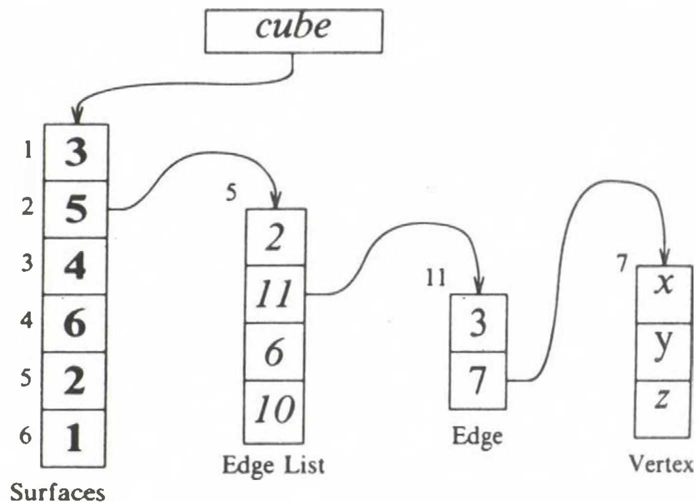


Figure 7. Example of Data Structure of a Cube

Table 2. Relation Type Data Structure of a Cube

Surface	line	line	line	line
1	1	2	3	4
2	5	8	7	6
3	1	9	5	10
4	3	11	7	12
5	2	10	6	11
6	4	12	8	9

data integrity using constraints, because all facts presented by predicates are independent.

Values of attributes are implemented as follows. For example, we regard a predicate,

$$point(P),$$

as a function, and the value of this function becomes the coordinates, such as

$$[X, Y, Z] = point(P).$$

This means we can treat an entity, its attributes (i.e. predicates), and their values individually.

On the contrary, in an intensional representation system, the data would be totally described in a chunk of data strongly and mutually connected by pointers together with the constraints Σ (see *Figure 7*). In this case, dependencies between the data become so strong that it is difficult to change or to modify the data schema. But, the meaning of a symbol can be easily decided by its relative position in the data structure.

Moreover, there is no separation of an entity, its attributes, and their values. For example, in a relational database system, we can separate relations and tuples like in *Table 2*. But, this separation of the relation and the tuple can be so complete that there will be inevitable mutual dependencies such as the order among the data in the relation.

To sum up, an extensional representation system has the following implementational advantages.

EA1) It is easy to add new facts about entities, i.e., subjects.

EA2) It is also easy to modify facts and predicates.

EA3) Assertion of a proposition can be done by a simple search with pattern matching.

Disadvantages of an extensional representation system can be pointed out as follows.

ED1) It is rather difficult to grasp the entire meaning of what the logical formulae as a whole are saying, because we need to interpret logically all of the descriptions and because fairly large amount of computation would be required.

ED2) Predicates may loose their meanings; they only can have meanings defined by each other. This is one of the typical disadvantages of formal logic and not particular to the extensional representation.

ED3) As far as implementation is concerned, it is not so easy to realize an efficient processing.

On the other hand, an intensional representation system has the following implementational advantages.

IA1) Normally the meanings of each predicate can be easily understood, because the constraints, Σ, define them clearly.

IA2) What a piece of data says can be easily understood from its position in the data structure, i.e., by its address.

IA3) As far as implementation is concerned, it is not so difficult to realize an efficient processing.

Disadvantages of an intensional representation system can be summarized as follows.

ID1) It is difficult to modify the data schema totally due to its strong mutual dependencies. Adding new facts is easy.

ID2) Modification of the structure of propositions requires changing the constraints, Σ. This is also difficult.

ID3) Assertion of a proposition may contain considerably complicated calculation of the constraints, Σ.

4.2.5. Summary

DP 10: *IDDL* should basically use an extensional data description method.

DP 11: From an implementational point of view, *IDDL* may be obliged to have also an intensional data description method because of performance, for example.

DP 12: When in *IDDL* both an extensional data description method and an intensional data description method are combined we need to carefully design it, because it is possible to have unexpected loss or twist of information in a data exchange between those two methods..

4.3. About Names

In this section, we would like to get rid of misunderstandings concerning *names* and *attributes*. Generally speaking, names are considered to be attributes; or there is a saying that the simplest attribute is a name. However, this idea is doubtful.

Since *IDDS* is more or less a database system, it must have *identifiers* to distinguish entities (or entry records). An identifier, or any similar concept such as a *key*, are desired to be unique in one system [Lor82]; therefore, it should be created automatically by the system.

An entity concept must be identified in some way, although we cannot use attributes to identify it because attributes will be created after entity concepts. Therefore, an entity concept will have an identifier which should be unique in the system and which is different from either a name or an attribute. However, a *name* system containing nouns in a natural language is normally used as an identifier, because under a certain circumstance nouns can be unique[3].

On the other hand, an attribute can be regarded as a *qualifier* which describes properties of a thing. Sometimes, it happens that a concrete value of a name is filled with a qualifier which is not at all an identifier. For instance, you can call your dog like *blacky* because he is black. But, it is easy to find thousands of black dogs and, therefore, in this case his name *blacky* is less a name but more a qualifier. *Nouns* in a natural language is a monster, because they imply being qualifiers.

DP 13: Generally speaking, names are not attributes; nouns can be attributes, although they must be distinguished from identifiers.

DP 14: Names are categorized into two groups. *System name* is internal and should be unique. *User name* is external and can be modified.

4.4. Predicates, Objects, and Functions

As we discussed heavily in *Section 4.2*, an extensional description method looks better than an intensional description method. We will later try to compare these two from a view point of

[3] Of course, this is not guaranteed for all the situations. People may have the same name!

machine design or CAD application in the next chapter. To do so, in this section, we elaborate our idea along this distinction from a rather practical point of view.

An extensional description method is concentrated only in an aspect of relationships among entities, while an intensional one is focused on the structures of entities. Unfortunately, we cannot use exclusively either of them because of practical reasons discussed in the next chapter. Thus, we need to combine these two methods which look totally contradicting (see *Design Policy 11*).

As we have suggested in the previous discussions, an extensional description method is *fact oriented* which will be further transformed to **predicate oriented**, because it is heavily concentrated on the relationships of entities. On the other hand, an intensional description method is **object oriented**, since it forms closure.

Then, a question may arise:

- *Is it possible to integrate these two (programming) paradigms?*

A famous work to answer this question is conducted by ICOT, Japan, as an implementation of *Concurrent Prolog* [ShT83]. Similar work is also carried out in University of Edinburgh [Bij86].

Figure 8 shows our paradigm combined the object oriented programming paradigm with the logic programming paradigm (in our terminology, *predicate oriented*), which will be discussed in *Chapter 6* as the language *IDDL*.

Suppose small circles in this figure correspond to crosses in *Figure 3* (a) which indicate entities, and arrows to topology which indicates relationships between entities. An object may have internal memory spaces to store information just like slots in Minsky's *frame theory* [Min75], but the internal structure of an object really does not matter because the user is interested not in how to store the information but in how to use it. We also introduce concepts such as *message*, *class*, *inheritance*, etc., but in different terminology from the frame theory or from object oriented languages [GoR83]. For instance, note that here the *message* mechanism is realized only to get perfect information enclosure.

This paradigm is also similar to *entity-relationship model* in database theory [Che76], but there is a big difference that in our paradigm relationships can be created, modified, and deleted all the time during the execution. In conventional database systems once we fixed the data schema, it may never changed.

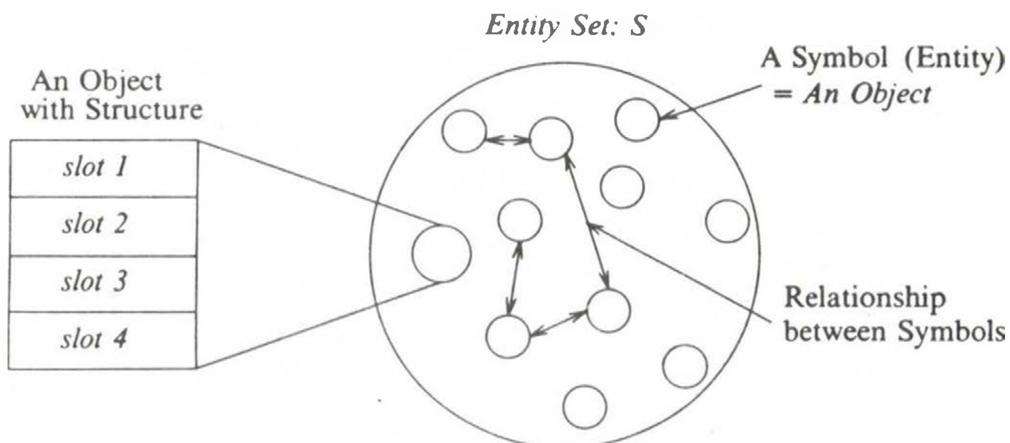


Figure 8. Objects and Relationships

Each object, which actually semantically represents an entity and is represented syntactically by a symbol, is treated as *Prolog's* [ClM81] term or variable. Therefore, for instance, a fact that an object A is an automobile is denoted by

$$automobile(A).$$

We can naturally introduce the concept of class into our syntax; for example, a fact that all the automobiles are vehicles can be denoted by

$$vehicle(X) :- automobile(X),$$

where a symbol ":−" is supposed to have the same meaning as in *Prolog*[4]. This must be interpreted as

If X belongs to automobile-class, then it belongs to vehicle-class.

It also naturally corresponds to an idea that the set of vehicles is a superset of the set of automobiles. It is easy to understand that this notation provides natural inheritance mechanism of properties together with exception handling.

An object has its internal structure and it is possible to store information which might be not only simple data but also complex procedures. An access to that information is done by invoking a *function*, such as

$$function(object) \rightarrow value$$

for inquiry, or

$$function(object) \leftarrow (any\ procedural\ definition)$$

for definition. In fact, here functions are used exactly in the same meaning as messages. Note that in *IDDS* the value obtained by issuing an inquiry to a function should be consumed immediately by other process, i.e., it flows out of *IDDS* to application programs, or should be used by program control, i.e., it is absorbed into the predicate logic world. In any case, they never remain because they are not objects.

Now, we have two different things around objects;

$$function(object)$$

and

$$predicate(object).$$

In the predicate world, we can generate complex clauses by combining simple clauses. In the object world, we can define a complex function by combining simple functions. During the execution, a complex function should be decomposed into primitive predicates or functions so that they are executed by *SPV*.

In this context, we may have the following distinctions:

(1) A *function* is an intensional description of an object. It will be defined **functionally** (or sometimes, **procedurally**) by using primitive functions. Those primitive functions should be built-in functions so that they are **executable** finally. Thus, a function, $f_0(o)$, for an object, o, is expressing the value of an item f_0. It is also possible to express constraints among items, such as

$$f_1(o) \equiv f_2(o) + f_3(o) / 2.$$

(2) A *predicate* is an extensional description of objects. It will be defined **declaratively** by using primitive predicates. Thus, any predicate should be **deductible** (or **decomposable**)

[4] It does not mean that we use also this symbol in an actual implementation.

into primitive predicates which are defined by single facts, system predicates, or functions. For example, a predicate *greater_than* can be defined by

$$greater_than(x,y) \equiv \textbf{if } val(x) > val(y) \textbf{ then } \text{true } \textbf{else} \text{ false},$$

using a function, *val*[6].

An *instance* of an object is created, when a variable denoting an object is instantiated by assertion, or when a function is invoked and it accessed to the object. Also, an instance is removed from the scope when it is clearly deleted by a system predicate. It is important that changes in the internal information of an object by an application program or by the user creates a new object; an object with the old information should remain in the scope until they are declared to vanish. However, it must be also possible to have an object which is varying according to the changes of the environment. An example of objects of this type is one which is linked to an application program and which has a possibility to be changed by an application program. This means that information of an instance is rigid; otherwise, it is very difficult to keep the integrity or the consistency of objects with the constraints.

Anyhow, it should be noted that system functions or system primitives must be well designed so that *IDDL* would meet requirements of applications.

DP 15: In *IDDL*, there must be **objects** denoting entities, **predicates** which represent relationships among entities, and **functions** to bridge objects and predicates.

DP 16: An object is created by assertion of a predicate or by a function. Whenever a new assertion is carried out, a new object is created without changing the old one. Therefore, we need time stamps for objects and predicates.

5. ISSUES COMING FROM CONSIDERATION ON DESIGN OBJECTS

In this chapter, we discuss issues for *IDDL* coming from consideration on design objects. Since we are now talking about machine as the design target, we discuss specifically how to represent machinery, machine design, CAD applications in machine design, etc.

5.1. Attributive Representation of Machine

What are the characteristics of *machinery* which differentiate it from other entities in the world? That is *structure* which is regarded as a part of attributes. First, we discuss the issues relevant to structure and then those relevant to attributes in general.

5.1.1. Representation of Structure of Machine

Usually, because we can observe the following things in machinery, it is said it has so-called structure. (See *Figure 4* for an example.)

(1) Existence of parts.

(2) Existence of relationships among the parts.

(3) Existence of sub-structures, i.e., hierarchy.

In most of cases, those relationships change dynamically due to motion of the mechanism. Moreover, relationships are *multidisciplinary* such as geometrical, kinematic, hydraulic, magnetic, electric, etc.

Therefore, a hierarchical structure as in *Figure 4* is not the only one in an automobile. The figure merely shows the part-assembly hierarchy of an automobile. It has another network, e.g., in an electrical aspect, which may require another system of parts and categorization for giving relationship. For instance, a part *WHEEL* may never appear in the electrical aspect. Sometimes, an

[6] This shows only a possibility. It never means this should be done.

identical entity may have different names depending upon the aspect; a connector used for electric wiring could be called a bolt, if you looked at it from mechanical engineer's eyes. What we need is probably a kind of aliases.

DP 17: *IDDL* should represent multidisciplinary nature of structure of machinery.

We can point out another important issue [ToY85a] relevant to the structure problem. In the situation of *Figure 9*, let us define a binary relationship

$$on(X, Y)$$

which should read

X is on Y.

The following four relationships describe the state of *Figure 9 (a)*;

$$on(B, A),\ on(C, A),\ on(D, A),\ on(D, C).$$

Now, to examine the relationship between A and D more precisely, we need to have another relationship such as

$$above(X, Y) \equiv on(X, Z) \wedge on(Z, Y).$$

Although this is a more precise and general expression because it is not restricted to a relationship meaning touching on the surface, it is doubtful whether these two relationships, *on* and *above*, are practically useful projections of this world to the mathematical world. A much more natural expression will be to introduce a unique expression like

$$upper(X, Y)$$

which would read

X is somewhere in the upper space above Y ,

and which would be defined by

$$upper(X, Y) \equiv on(X, Y) \vee (on(X, Z) \wedge on(Z, Y)).$$

This story tells that we need to pay good attention to interpretation or semantical definition of predicates. Because there is no concrete definition for a predicate, *on*, (such as one by the position of objects) other than simple four facts here, a predicate system is heavily dependent on interpretation. Thus, it can easily acquire ambiguity which is recommended to avoid.

DP 18: *IDDL* must maintain the compatibility of predicates with the application world.

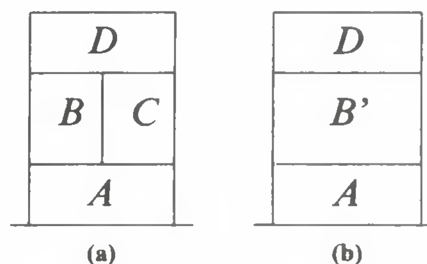


Figure 9. Relationship in Parts

Figure 9 tells another important fact. In *Figure 9* (b), the role of B' is identical to the role of B and C in *Figure 9* (a), because they are both together supporting D. We sometimes want to treat a couple of parts as one chunk. This requires us to do either of following two things.

(1) We extend n-ary relationship formulae to m-ary relationship formulae in a natural way.

(2) We have a way to produce a chunk created from several things, which is by no means a kind of part-assembly relationship.

DP 19: In *IDDL*, an object can be regarded as a chunk of other objects (kind of part-assembly relationship).

5.1.2. Representation of Attributes of Machine

In the previous discussion, we are concentrated too much on so-called part-assembly relationship. However, that is not the only thing we have to think about machinery. From a practical point of view, we can count up those attributes such as roughness of a surface, tolerance in dimension, weight of a part, material description of a part, etc., which are called *technical information*. In fact, geometrical information is merely one of those attributes.

A machine has many properties other than structure, and structure should be ultimately expressed by attributes in terms of geometrical information. This means that structure can be by no means the center of attributive expression of a machine, although it does play an important role in machine design. Therefore, the metamodel concept (see *Section 3.2.2*) must not be constructed on top of so-called geometrical models.

DP 20: *IDDL* should not be designed as a geometric modeling system.

This issue is also supported by following facts. For example, in the traditional drawings of mechanical parts, a *dimension* is defined usually by a relative distance between two parallel surfaces or, in case of a cylindrical surface, between two diameters (*Figure 10*). It would never be defined by the length of an edge. On the other hand, a geometrical model might have data describing such length. In most of geometrical modeling systems, consequently, the information about dimensions is added and separated from information about both the geometry and the topology. And, most of technical information is relevant to surface information, because what we can create with machine tools is a surface. Accordingly, even for representation of structural information, geometrical

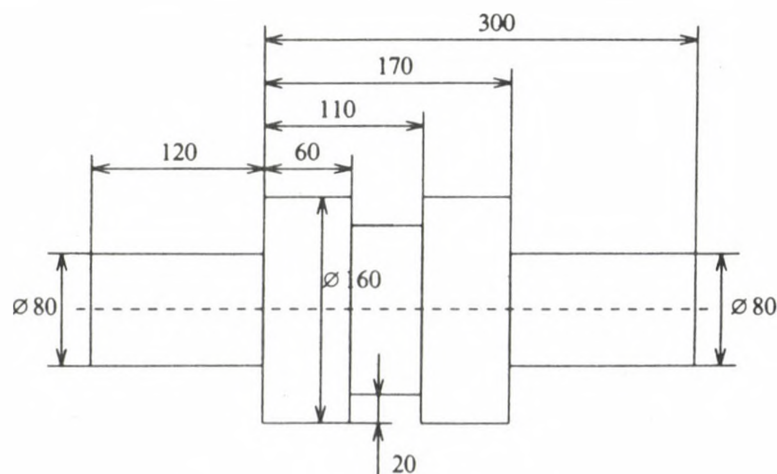


Figure 10. A Definition of Dimensions

models need additional attributes.

5.2. Functional Representation of Machine

Not only representation of attributes but also representation of functions is a big problem in machine design. Generally speaking, there are many unsolved problems in this issue, because we have not yet obtained any satisfactory definition of *function* itself.

First of all, we can define or explain what a machine is [Rod71] (see *Figure 11*). It seems that many researchers have accepted this definition [HuP85].

- *A machine, receiving information (I), energy (E), and material (M), transforms them into a new state.*

This is a definition of a machine, but it also defines the function of machine. (Probably, we can call it a functional definition of a machine.) However, although it might be theoretically perfect, it leaves many questions.

For example, it is difficult to say that a pair of a bolt and a nut transmits something, although obviously it has a function of fixing or binding. In fact, because it transmits nothing, it h[illegible] function. It is possible to say it transmits a resisting power or force against the external force. However, we cannot say this is an acceptable explanation. Probably, definition of functions can vary from application domain to domain, and even in one domain it is difficult to get unified agreement on what is a function. Function can be very subjective.

This tells us that we must give up to describe a function in a unified and systematic way and that we need to introduce a very flexible framework to describe any type of functions. Furthermore, because of this lack of uniformity, we will be obliged to describe our knowledge about function fragmentary. (Thus, the use of production rule based systems might be justified where we have to deal with functional expressions [ToY85a].)

DP 21: To allow descriptions about functions, *IDDL* should be flexible and user-definable.

5.3. CAD Applications in Machine Design

Problems concerning modeling for CAD applications in machine design have been long discussed (for instance, see a literature [EnK82]). There are several issues specific to design and/or machine design. Followings are characteristics which should be taken into consideration in designing CAD systems.

(1) **Diversity**: As we have pointed out in *Section 3.2.2*, there are many ways of representing machinery, i.e., models. In other words, most of design works lack uniformity; this is why we need so many models. Nevertheless, they must keep integrity and consistency of the information.

(2) **Dynamic changeability**: Models are changing dynamically during the design process from very vague initial one to detailed final drawings.

Figure 11. Definition of a Machine

(3) **Bulkiness**: Usually, amount of information used in a CAD system is enormous.

Another thing quite specific to machine design is that there exists a traditional description level or one chunk for knowledge representation of machinery. Traditionally, mechanical engineers have developed the concept of *machine elements*, such as bolt, nut, screw, spring, key, shaft, etc. They are standarized in most cases and are dealt with as untouchable things, so to speak. Probably, this also applies to designing in other fields, such as VLSI, aircraft, architecture, etc.

This means that those elements should be treated as a kind of chunk and their structure, for example, should never be changed. Therefore, the information about those elements should be stored and retrieved in chunk, because their internal structures have already fixed as industrial standards. Actually, traditional database systems are suitable for this type of information. In this context, we can also treat much higher level machine parts as chunks, such as motor, gear box, etc., which are usually bought from specialized manufacturers. Generally speaking, once information is catalogued, we can use any type of conventional database systems.

In *Section 4.4*, we have discussed a lot about *objects* which will describe entities intensionally and *relations* which will describe entities extensionally. Now, we can compare these two from a viewpoint of the abovementioned machine design.

First, we have found the following issue in a discussion of *Section 4.2*.

- An intensional description method is suitable for applications where information does not change its structure, while an extensional description method is good where information changes dynamically.

Therefore, because of *dynamic changeability* of models, we shall use an extensional description method to describe models appearing in a design process. However, for machine elements and existing parts we will use an intensional description method, because these things have prefixed information and because we never change them. And also, the problem of *diversity* will be solved by employing an extensional description method, because it provides descriptions that allows a multidisciplinary point of view (*Section 5.1.1*).

There still remains a problem of *bulkness* which is essential. However, because this will be only solved by implementation and/or by hardware development, we are not going to discuss it here.

DP 22: In *IDDL*, both intensional and extensional descritpion methods should be employed and they are realized as objects and predicates, respectively.

6. IDDS AND ITS LANGUAGE IDDL

Here, we describe the functional specifications for *IDDS* and *IDDL*. Firstly, we summarize functions required to *IDDS*. Secondly, we clarify necessary properties of *IDDL*. summarizing the design policies for *IDDL* which have been pointed out so far. Next, we describe temporary specifications for *IDDL*. Although its minute syntactical specifications are not fixed, we will find an example of a block manipulating program written in an experimental version of *IDDL* in the *APPENDIX*.

6.1. Functions of IDDS

In *Chapter 2* we have discussed the concepts of a *IIICAD* system. Followings are the functions required to *IDDS*.

(1) By speaking the common language *IDDL*, *IDDS* should provide a mechanism to exchange and utilize information smoothly. This means *IDDS* not only passes information through from one subsystem to another but also translates the format and syntax together with *IUI* and *API*. Let us call it **transparent mechanism for data transfer and storage**.

(2) *IDDS* itself is a database system (or a knowledge base system). In fact, *IDDS* may have some databases and knowledge bases for real data operations; it works as a gateway or an entrance to those systems. And, it will be given rules and facts in *IDDL*, and based on them it will **answer to queries** using those database systems. This indicates *IDDS* is a **database as theory**, but not a *database as implementation*.

(3) Although system components directly communicate with *IDDS* by speaking *IDDL*, they do not necessarily speak the same terminology. *SPV* may say, "give me a shaft," which must be passed to *IUI* as "create a cylinder with diameter X, height H, and name @A321001." This means *IDDS* must provide a **transformation system** from user-oriented semantical expression to system expressions. This can be done by rules that are given as the domain knowledge and working in the background of information flow.

(4) In a *IIICAD* system, it is important to distinguish extensional descriptions from intensional ones. They must be diligently and completely separated, but there should remain linkage between those two. Therefore, *IDDS* must provide a mechanism for **intension/extension linking**.

(5) *IDDS* must also know about the system itself; in other words, it should have **metaknowledge**. For example, it must know who knows what. Besides this purpose, it should record the history of system activities, automatically. This can be used for undoing, replaying, making new scenarios, etc., as well.

6.2. Necessary Properties of IDDL

As the basis of discussion, we employ (first order) predicate logic to describe *IDDL*. Although it is not necessary that the syntax of its real implementation is based on first order predicate logic, we use it for the purpose of discussion. In fact, as pointed out in *Design Policy 20*, *IDDL* or *IDDS* will not be designed as a geometric modeling system; it is a schema to describe data in an integrated way.

From the discussion of the previous section and design policies clarified so far, we will discuss necessary properties of *IDDL* in this section. We have obtained in all twenty two design policies some of which are identical. Removeing those doubles, we may get the following eight design guides for *IDDL*.

(1) *IDDL* should be possible to describe status information of the system, control information of the system, its origin and destination, and the time stamp. It should be used for describing also for scenarios. This means we need to design *IDDL* to have a uniform syntax to express those meta-information besides facts and rules to define the information schema, to answer queries, etc.

(2) *IDDL* should have objects to represent entities intensionally, predicates to represent relationships among entities extensionally, and functions to bridge objects and predicates. This is necessary for implementation and performance.

(3) Predicate logic should be based on three-valued logic; i.e., there should be (at least) **UNKNOWN** or **UNDEFINED** besides **TRUE** and **FALSE**. For the moment, very classic (and primitive) three-valued logic seems sufficient.

(4) An object should express class-subclass hierarchy as well as part-assembly relationship. Every time either an assertion or an access by a function is done, a new object is created. There is a distinction between the fact that an object has an attribute and the fact that an attribute has a value. The object world and the predicate world are completely separated, so that unexpected loss or twist of information in a data exchange may not happen.

(5) An object has a unique system name which cannot be modified and a user name which can be modified freely. Names are not attributes but identifiers.

(6) The system designer can implement predicates depending on the target application to describe a multidisciplinary world where metamodels are described.

(7) *IDDL* should provide facilities or a mechanism to check the completeness, soundness, and feasibility of the knowledge. This is possible, only if we have a formal (mathematical) logic system.

(8) Finally, it is important to introduce user-friendlyness, readability, etc., to *IDDS* for the sake of programming productivity, bug-free-programming, etc.

6.3. Temporary Design of IDDL

According to eight design guides in the previous section, we can proceed the design of *IDDL*.

(1) A string beginning with a lower case character means a constant, whereas one beginning with an upper case character means a variable.

(2) A predicate begins with an alphabet. An object is a string beginning with "#". A function is a string beginning with "%". A query is indicated by "?" placed at the end. Therefore, a sentence

`shaft(#s₁).`

should read as

IDDL should define that an object `#s₁` *is a shaft.*

If a sentence

`shaft(#s₁)?`

is given, it must be interpreted as a query. In the same way, a rule

```
IF shaft(#X) THEN cylinder(#X),
                  transmit_power(#X),
                  supported_by(#X, #Y).
```

means that

if `#X` *is a shaft, then* `#X` *is a cylinder, transmits power, and is supported by* `#Y` *(something else).*

This means that *IDDL* is designed as a *database as theory*.

(3) In order that *SPV* should accept a user input or a report from a subsystem in the same way as the scenario, *IDDS* must allow dynamic definition, deletion, and modification of predicates. By allowing it, we have a problem of priority. Here, we simply say that any last modification has the priority.

(4) If a predicate is sent to *IDDS*, it will look at rules whether it is possible to apply one of them or not. For example, if a fact

`shaft(#s₁)`

is sent to *IDDS*, then a rule

```
IF shaft(#X) THEN cylinder(#X),
                  transmit_power(#X),
                  supported_by(#X, #Y).
```

is applied and new three facts

```
cylinder(#s₁)
transmit_power(#s₁)
supported_by(#s₁, #s₂)
```

are added. Here, if *IDDS* does not know `#s₂`, it will ask the user.

(5) There are some predicates that have special meaning to the system (**built-in predicates**).

(6) Every object has its own system name and user name. The system name is usually not known to the user and cannot be modified. In case of a constant object, like `#s₁`, the system name is something like *@34AEC*, and the user name is `#s₁`. In case of a variable

object, like #X, the system name is something like *@270FF*, and the user name is #X. System names are maintained by the system automatically, so that they are always unique. If a variable object is instantiated, it may be treated as a constant object without specific constant name.

(7) We do not restrict *IDDL* to Horn logic which can contain only one negative clause at most, because we would like to express a rule like **if** P_1 and P_2 **then** Q and R. This would be denoted in *IDDL* by

```
IF P1, P2 THEN Q, R.
```

Disjunction, i.e., "$\vee$," is expressed by "|." Anyway, the mechanism of *IDDS* is more or less similar to that of production rule systems [New73].

(8) The reasoning method of *IDDL* is a simple deductive reasoning without backtracking, which implies *width first search*. Unfortunately, it also implies that it is space-consuming and that even to get the first simple answer we need to wait for ends of other searches.

(9) *IDDL* is based on so-called *open world assumption* which implies the introduction of intuitionistic logic or three valued logic. There remains a matter of discussion about the system behavior when it meets a negative result or an unknown result.

(10) To control the information flow, in *IDDL*, we can describe the origin, destination, and time stamp of the event. Therefore its basic syntax is as follows;

```
predicate ::= "string beginning with an alphabet"
object ::= #"string beginning with an alphabet"
object_list ::= object [, object_list]
unit_clause ::= predicate ([object_list])
clause ::= unit_clause [{,|"|"} clause]
sentence ::= [IF clause THEN] clause[?];
                   FROM origin TO destination AT time_stamp.
```

(11) The intensional description is supported as follows. There is a declaration part to declare use of objects. This is done by declaration of functions to get information about objects. The syntax is something like this.

```
Object definition: object_class_name;
        %f1: real,
        %f2: real,
        %f3 = %f1 + %f2: real.
```

These are the definition of an object clause and its internal structure. The value of each items is assigned somewhere like

```
%f1(#X) := 3.0.
```

REFERENCE

[Bij86] BIJL, A., An Approach to Design Theory, in *Design Theory for CAD, Proceedings of IFIP W.G. 5.2 Working Conference 1985 (Tokyo)*, H. YOSHIKAWA (ed.), to be published from North-Holland, Amsterdam, 1986. (in English).

[BMS84] BRODIE, M. L., J. MYLOPOULOS and J. W. SCHMIDT (eds.), *On Conceptual Modelling Perspectives from Artificial Intelligence, Databases, and Programming Languages*, Springer, New York, Berlin, Heidelberg, Tokyo, 1984.

[Che76] CHEN, P. P., The Entity-Relationship Model - Toward a Unified View of Data, *ACM Transactions on Database Systems 1*, 1 (March 1976), 9-36, ACM.

[ClM81] CLOCKSIN, W. F. and C. S. MELLISH, *Programming in Prolog*, Springer, Berin, Heidelberg, New York, 1981.

[EnK82] ENCARNACAO, J. and F.-L. KRAUSE (eds.), *File Structures and Data Bases for CAD, Proceedings of IFIP WG5.2 Working Conference in 1981 (Seeheim)*, North-Holland,

Amsterdam, 1982.

[GoR83] GOLDBERG, A. and D. ROBSON, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.

[HuP85] HUBKA, V. and PROGRAMME COMMITTEE (eds.), *WDK 12, Proceedings of ICED 85 (Hamburg) -Theory and Practice of Engineering Design in International Comparison*, Heurista, Zurich, 1985.

[KSH83] KIMURA, F., T. SATA and M. HOSAKA, Integration of Design and Manufacturing Activities Based on Object Modelling, in *Advances in CAD/CAM*, T. M. R. ELLIS and O. I. SEMENKOV (ed.), North-Holland, Amsterdam, 1983, 375.

[Lor82] LORIE, R. A., Issues in Database for Design Applications, in *File Structures and Data Bases for CAD, Proceedings of IFIP WG5.2 Working Conference in 1981 (Seeheim)*, J. ENCARNACAO and F.-L. KRAUSE (ed.), North-Holland, Amsterdam, 1982, 213.

[Min75] MINSKY, M., A Framework for Representing Knowledge, in *The Psychology of Computer Vision*, P. H. WINSTON (ed.), McGraw-Hill, New York, 1975, 285.

[New73] NEWELL, A., Production Systems; Models of Control Structure, in *Visual Information Processing*, W. C. CHASE (ed.), Academic Press, New York, 1973, 463-526.

[Rod71] RODENACKER, W., *Methodisches Konstruieren*, Springer, Berin, Heidelberg, New York, 1971.

[ShT83] SHAPIRO, E. and A. TAKEUCHI, Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, 1983, 25.

[ToY85a] TOMIYAMA, T. and H. YOSHIKAWA, Requirements and Principles for Intelligent CAD Systems, in *Knowledge Engineering in Computer-Aided Design, Proceedings of IFIP W.G. 5.2 Working Conference 1984 (Budapest)*, J. S. GERO (ed.), North-Holland, Amsterdam, 1985, 1-23. (in English).

[ToY85b] TOMIYAMA, T. and H. YOSHIKAWA, Knowledge Engineering and CAD, *FGCS 1*, 4 (June 1985), 237-243, North-Holland. (in English).

[ToY86] TOMIYAMA, T. and H. YOSHIKAWA, Extended General Design Theory, in *Design Theory for CAD, Proceedings of IFIP W.G. 5.2 Working Conference 1985 (Tokyo)*, H. YOSHIKAWA (ed.), to be published from North-Holland, Amsterdam, 1986. (in English).

[Yos81] YOSHIKAWA, H., General Design Theory and a CAD System, in *Man-Machine Communication in CAD/CAM: Proceedings of IFIP WG5.2/5.3 Working Conference in 1980 (Tokyo)*, T. SATA and E. WARMAN (ed.), North-Holland, Amsterdam, 1981, 35.

APPENDIX

```
== DESCRIPTION OF THE BLOCK WORLD ===========================

       +-----+          +-----+          +-----+
       | Red |          |Green|          |Blue |
       +-----+          +-----+          +-----+

Suppose you have a world of blocks something like a tower of Hanoi.
You can move these three boxes to a certain place by issuing a
command like
       Red should come (on top of| under) Green.
There can exist certain restrictions like
       Blue must not come under Red,
but for the moment we don't think about it.
```

```
== DESCRIPTION OF APPLICATION ======================================

Function      INIT(POS): Boolean;
                    { Gives an initial position to boxes.          }
                    { Returns TRUE if success, otherwise FALSE.     }

              Out: POS = array [(red, green, blue), (x, y, z)] of real;
                    { ex. POS[red, y] => the Y coordinate of the    }
                    { red box.                              }

Function      MOVE(IND, X, Y, Z, POS): Boolean;
                    { Moves a box IND to a new position (x, y, z),  }
                    { and reset the position data POS.          }
                    { In this function, constraints can be written. }
                    { Returns TRUE if success, otherwise FALSE.     }

              In: IND = (red, green, blue);
              In: X, Y, Z = real;
              Out: POS = array [(red, green, blue), (x, y, z)] of real;

Function      GETPOS(IND, X, Y, Z, POS): Boolean;
                    { Lets the user know the position of a box IND. }
                    { Returns TRUE if success, otherwise FALSE.     }

              In: IND = (red, green, blue);
              Out: X, Y, Z = real;
              In: POS = array [(red, green, blue), (x, y, z)] of real;


== DESCRIPTION OF USER I/O =========================================

Output Function     DRWCUB(X, Y, Z, CL): Boolean;
                    { Draws a cube at position (X, Y, Z) in a color }
                    { CL.  Returns TRUE if success, otherwise FALSE.}

              In: X, Y, Z = real;
              In: CL = (red, green, blue);

Input Function      GETCOM(COM, ARG1, ARG2): Boolean;
                    { Gets a command from the user.  COM is a }
                    { verb, ARG1 and ARG2 are its arguments.  }
                    { Returns TRUE if success, otherwise FALSE.     }

              Out: COM = (on, under, display, end);
              Out: CL = (red, green, blue);


== OBJECT DESCRIPTION FOR IDDS =====================================

Object definition:  box;
      %Color: (red, green, blue),
      %Coordinate: point.
```

```
Object definition:  point;
      1: %X = %R * cos(%Theta): real,
         %Y = %R * sin(%Theta): real,
         %Z = %[2]Z: real;
      2: %R = sqrt(%X**2 + %Y**2): real,
         %Theta = {if %X = 0 then
                  { if %Y > 0 then
                        PI/2
                    else if %Y = 0 then
                        Undefined
                    else
                        -PI/2 }
                 else if %X > 0 then
                  arctan(%Y/%X)
                 else if %X > 0 then
                  PI + arctan(%Y/%X)}: real,
         %Z = %[1]Z: real.

Instance:
      box(#redbox),
      box(#greenbox),
      box(#bluebox),
      %Color(#redbox) := red,
      %Color(#greenbox) := green,
      %Color(#bluebox) := blue.

Equivalent:
      %X(%Coordinate(#A))   =  POS(%Color(#A), x),
      %Y(%Coordinate(#A))   =  POS(%Color(#A), y),
      %Z(%Coordinate(#A))   =  POS(%Color(#A), z).


== INTERFACE DESCRIPTION FOR IDDS ====================================

AP/IF initialize  = [ INIT(POS) ].

      on(#A, #B)? = [ GETPOS(%Color(#A), X1, Y1, Z1, POS);
                      GETPOS(%Color(#B), X2, Y2, Z2, POS) ];
                    if Z1 > Z2 then return(TRUE)
                    else return(FALSE).

      on(#A, #B)? & on(#B, #A)
                  = [ GETPOS(%Color(#A), X1, Y1, Z1, POS);
                      GETPOS(%Color(#B), X2, Y2, Z2, POS) ];
                    swap((X1, Y1, Z1), (X2, Y2, Z2));
                    [ MOVE(%Color(#A), X1, Y1, Z1, POS);
                      MOVE(%Color(#B), X2, Y2, Z2, POS) ].

UIF   display(#A) = [ DRWCUB(%X(%Coordinate(#A)), %Y(%Coordinate(#A)),
                             %Z(%Coordinate(#A)), %Color(#A)) ].

      display(#A) = [ GETCOM(display, %Color(#A), DUMMY) ].
```

```
  end        = [ GETCOM(end, DUMMY, DUMMY) ].

       move(#A, #B, LOC) = [ GETCOM(LOC, #A, #B) ].


== SCENARIO FOR SUPERVISOR ============================================

FLOW: initialize() ; FROM SPV TO AP ONLY_ONCE.
      on(#X, #Y) ; FROM SPV TO AP.
      display(#A) ; FROM UI TO SPV.
      display(#A) ; FROM SPV TO UI.
      move(#X, #Y, Loc) ; FROM UI TO SPV.
      end() ; FROM UI TO SPV.

RULES:initialize().
      IF under(#X, #Y) THEN on(#Y, #X).
      IF move(#X, #Y, on), box(#X), box(#Y), ~on(#X, #Y)
           THEN on(#X, #Y).
      IF move(#X, #Y, under), box(#X), box(#Y), ~under(#X, #Y)
           THEN under(#X, #Y).
      IF end() THEN END.
```

FOUNDATIONS OF CONCEPTUAL REPRESENTATIONS

Elöd Knuth, Laszlo Hannák, Agnes Hernádi

Computer & Automation Institute
Hungarian Academy of Sciences
Budapest 112, POB 63, H 1502 Hungary

a b s t r a c t

A number of excellent design methodologies have been proposed for data and knowledge intensive applications. Most techniques, however, focus on the middle third of the design process only (i.e. conceptual and logical schema design).

Our paper examines the extendibility of such modelling techniques to the design stages not yet fully covered. We distinguish acquisition-, conceptual-, infological-, and data structure oriented layers of modelling. A coherent hierarchy of modelling concepts is introduced in accordance with the above layering phylosophy.

Based on these concepts a computer aided technique is purposed to support the mental processes of acquisition, conceptualization, and knowledge transfer in close association with the design of data and knowledge intensive application systems.

1. Introductory remarks

Conceptualization tends to become the central headache of modern computer science. Not surprisingly. Computer programming, the everyday's abstraction, is now open for housewives. At the other end, large sophisticated distributed software systems demand higher and higher levels of abstractions, reference models, i.e. adequate concepts as keys to manage complexity.

At the same time, computer manipulation of conceptual level information is becoming increasingly common in several fields of computer science. Its story began at the mid sixties covering then the birth of abstract data structures, knowledge representation techniques, and database conceptual schemas, as captured by the slogan of conceptual modelling recently [BMS 84].

This paper attempts to outline a top-down scheme of concepts necessary for the conceptualization process itself. The reader is supposed to be familiar with abstraction mechanisms, knowledge representation schemes, conceptual languages, and data abstractions, see e.g. respectively [G 85]; [ML 84]; [MW 80]; [SFL 81]; [KM 82]; [SH 84].

1.1 The right interrelation of conceptual modelling areas

"Conceptual modelling" recognized the need for higher level abstract concepts, tools, techniques. The book [BMS 84] surveys new achievements of AI, databases, and programming languages in this respect. We suggest to refine this comparison in the following way.

All the three fields mentioned employ concepts which are, in a degree, specific to the particular goal addressed. In fact, however, there are far more general concepts, which are common for all kinds of conceptual representations, and hence it is important to deal with them distinctively and explicitely. Conceptual modelling could therefore be built up as shown:

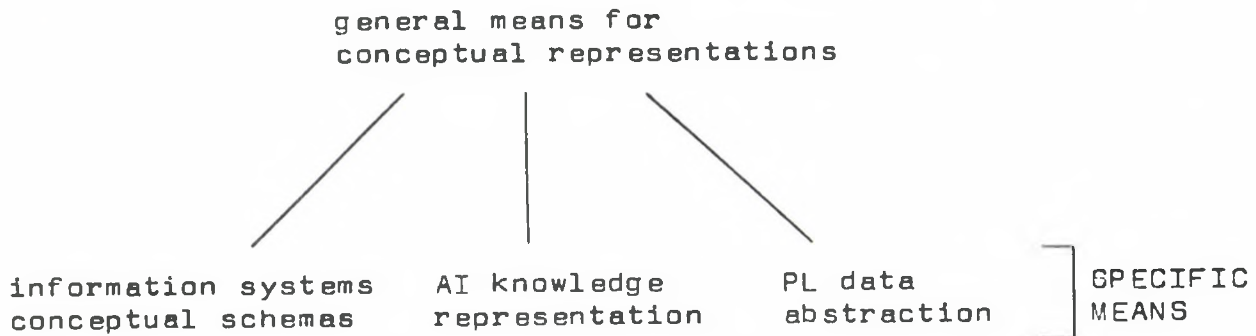


Figure 1.

Partition of conceptual modelling areas.

1.2 Conceptual modelling and information systems design

A version of the commonly accepted reference framework for information system's design consists of the stages shown below:

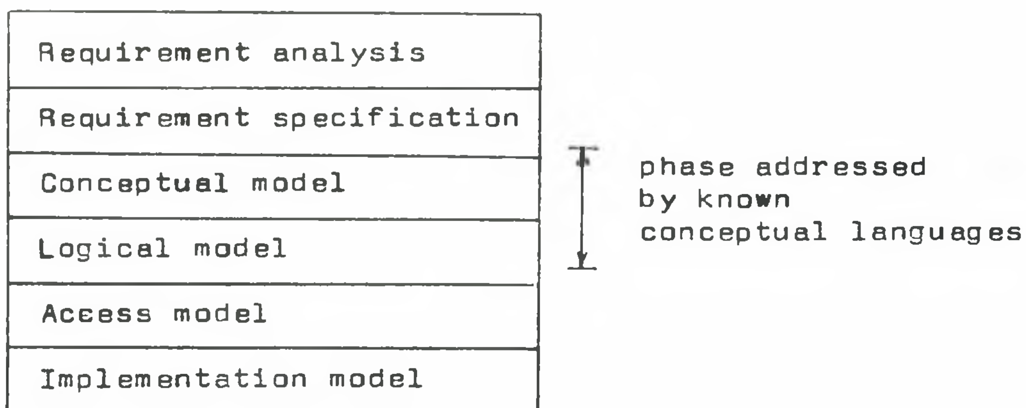


Figure 2.

ISDM modelling stages.

As indicated, most conceptual design methodologies (models, languages) e.g. TAXIS [MW 80], ADAPLEX [SFL 81], Event Model [KM 84], etc. address roughly the "middle third", that is they do not suit for the work to be done at the initial phase where everything is vague, uncertain, and changing from day to day. In our belief, conceptual means applicable at the initial stage should be

- far **more general** than those offered by present days conceptual languages;
- **flexible enough** ensuring the dynamics of the modelling process;
- **compatible with** and well connected to the techniques proposed and adequate at the middle level (see references above).

2. Concepts for top level modelling

2.1 General version

A major practical difficulty of modelling and conceptual descriptions is avoiding going into details irrevelant on certain given levels. It is therefore essential, that modelling tools applied on top levels be stimulating for the concentration on relevant issues only. Possibly, the vaguest form of a conceptual model is the semantic network, see e.g. [QUIL 68], [BRAC 79]. We think however, that in order to start with, even weaker concepts are needed.

Hence, at the top level we suggest to use three concepts only, namely

- **"concept"** itself (at this level still not distinguishing between abstract and concrete ones!);
- **"properties"** in general (at this level replacing "attributes", "constraints" and a lot of other possible constructs);
- a relation over concepts called **"case-of"** (at this level replacing both the usual "is-a" and "instance-of" relations).

Explanation:

(i) A "concept" is given by an identification constraint (e.g. a name) and a (possibly empty) set of properties associated.

(ii) A "property" - at this level - is anything (e.g. a natural language sentence) what we consider necessary to mention to define a concept.

(iii) The relation "case-of" is used to connect pairs of concepts whenever one of them has (in some sense) all the properties of the other.

Example-1.

When thinking about e.g. an "enterprise", at the vaguest stage we might only say things like these:

```
concept enterprise;
   properties:
      Has a management board:
      Has a scope;
      Has departments;
      Obeys tax laws;
      Must be rentable;
      ... etc

concept steel work;
   case-of enterprise;
```

Note that from a certain viewpoint "steel work" might be an "instance" of "enterprise" (in the traditional sense) and at the same time from another, it might only be a special version of it (i.e. an "is-a" one). What we think most important, however, is that such a refining decision must not be made until reaching an appropriate stage within the modelling process itself!

Example-2.

Of course, our modelling concepts are themselves "concepts" in the very sense introduced here. So, we may even write our specifications as:

```
concept concept;
   properties:
      Identifiable;
      Has a set of properties;

concept property;
   case-of concept;
   properties:
      Has a content;

concept case-of;
   case-of concept;
   properties:
      Has  an  object and a subject  which
                          together identify it;
      Constitutes  a loop  free  directed
                                         graph;
```

Thus we have

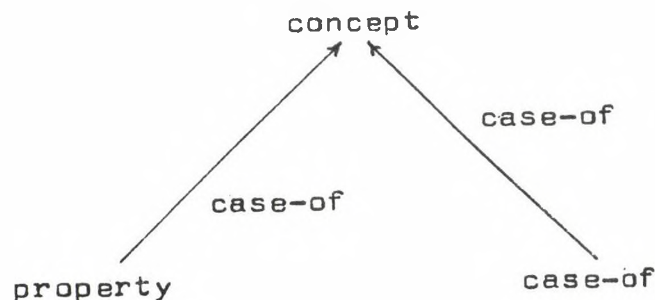


Figure 3.

Interconnection of initial concepts.

2.2 Revised version

At this point we could slightly revise and refine our initial model:

(a) We might explicitely distinguish between the two main types if identification (e.g. by a name versus a key). That is, we can early distinguish between entities and relationships in the sense of [CHEN 76] i.e. defining

```
concept entity;
   case-of CONCEPT;
   properties:
      Has a name;

concept relationship;
   case-of CONCEPT;
   properties:
      Has  an  object and a  subject  which
                                identifies it;
```

and then rearranging the hierarchy as shown below:

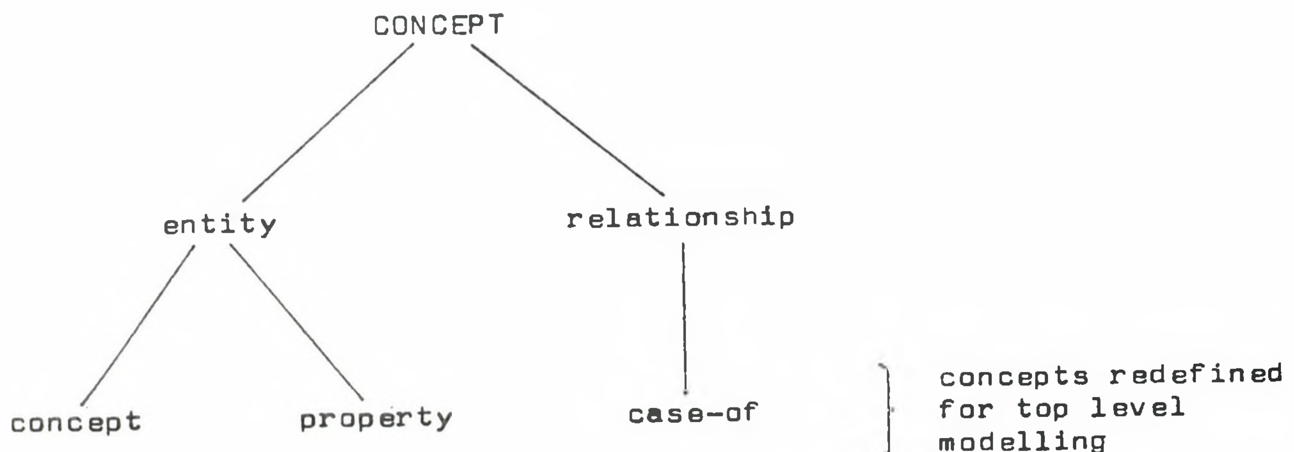


Figure 4.

Initial concepts, revised version.

(We remark that transformations of existing conceptual schemes like the one leading from fig.3. to fig.4. are frequent in the practice of top level modelling. Therefore it is important that a computer aid should support such model-transformations.)

(b) We can distinguish between two main types of properties namely those possessed and of those fulfilled. It is usual to call the former ones "attributes" and the latter "constraints":

```
concept attribute;
   case-of property;
```

```
    properties:
        Its content is "value";


concept constraint;
    case-of property;
    properties:
        Its content is a "predicate";
```

Thus we get

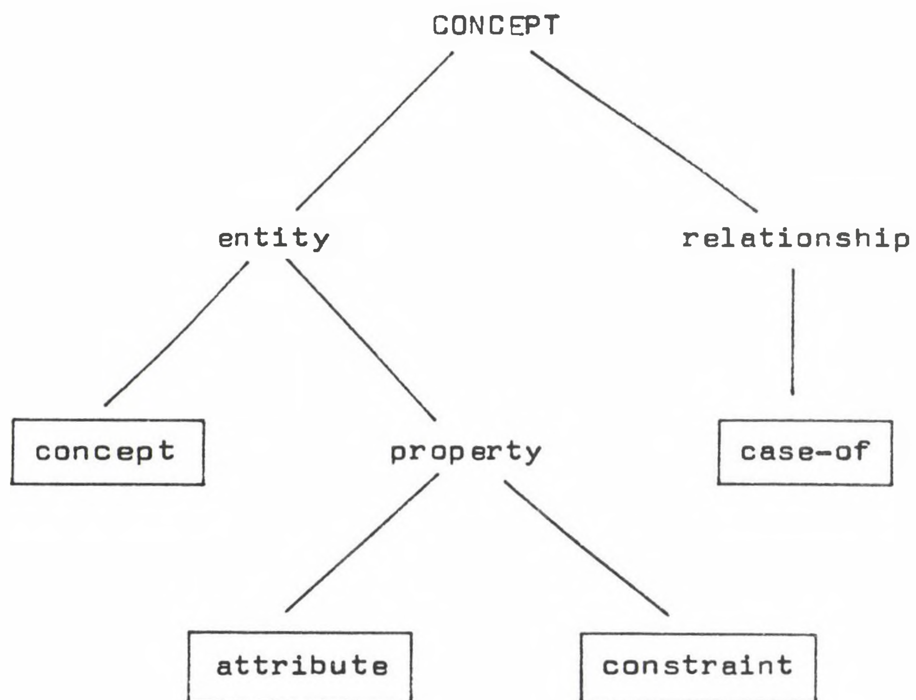


Figure 5.

Concepts proposed for top level modelling.
Final version.

Now we can reformulate Example-1 as

```
concept enterprise;
   properties:
      (has) Managemenet board;
      (has) Scope;
      (has) Departments;
      (fulfils) Tax requirements;
      (fulfils) Rentability;
```

or by using a more aesthetic formalism:

```
concept enterprise;
   attributes:
      Management board,
      Scope,
      set of Departments;
   constraints:
      Tax laws,
      Rentability;
```

More exactly, attributes are introduced in the following way:

(a) Names of attributes locally identify them with respect to the concept they belong.

(b) Each attribute has a value which is supposed to be a concept (allowing "pregiven" ones too, e.g. "number", "text", etc.)

(c) An attribute might also have a "mode" (which can e.g. be "individual", "set", "list", "array", "tuple", etc.)

Formally:

```
concept attribute;
   case-of property;
   attributes:
      name: identifier;
      value: concept;
      mode: (enumeration);
   constraints:
      "Name" locally identifies the attribute.
```

3. Level of information system's traditional conceptual schemes

Having a top-level model of a phenomenon (system) we are then to refine it gradually. These transformations are, however, far from mechanic since abstraction levels differ not only in their degree of details but, in nature, in their conceptual bases too, see e.g. [LUD 84].

Hence refinement means not only making design decisions, but gathering and adding new information. Nevertheless, it is important to ensure "smooth" transitions between modelling levels by the compatibility of conceptual models belonging to different layers.

The top-level model outlined previously - because of its openess - can now be enriched and refined into several particular directions. One way of refinement is obtaining one of the well known conceptual schema description models e.g. TAXIS [MW 80], DAPLEX [SH 81], Galileo [AC 83]. Below we outline this way.

3.1 Instance

Classification the historically first postulated and most important abstraction mechanism is widely used since the birth of SIMULA 67 [D 70] and in some form it is adopted by all conceptual languages. (We do not quote the definitions here, these can be found elsewhere, e.g. in any of our references.)

In this sense we can say that one concept can be an **instance** of another, the former being an "abstract" one in comparison with the latter. Hence, "instance-of" can be considered as a special case of our "case-of" relationship with the following important property (called the "homomorphism rule" of attribute selection):

If P is an attribute of concept C, C.P denotes is value, and C' is an isntance of C, then C'.P must be an instance of C.P:

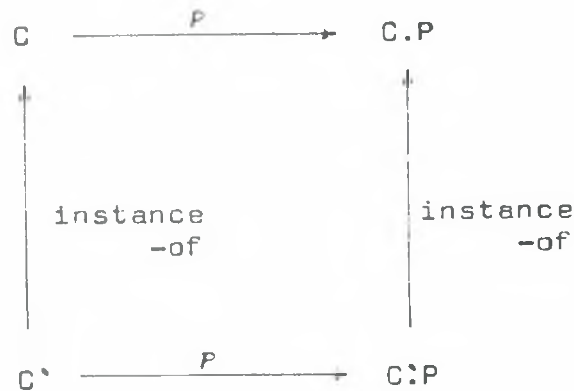


Figure 6.

Homomorphism rule of instanciation

(Note that from the definition of the relationship "case-of" it follows that C' must also possess the attribute P.) In our notations:

```
concept instance-of;
   case-of case-of;
   constraint:
      Attributes of its "object" are instances
      of the corresponding attributes of its
      "subject" (homomorphism);
```

(Note that the concept "instance-of" has two attributes - namely an "object" and a "subject" not written above explicitely - which were already defined in case of "case-of".)

3.2 Is-a

The second most important abstraction mechanism is the **generalization** (implemented also early in SIMULA 67) usually denoted by "is-a". For two concepts A and B "A is-a B" if a subset of A's properties are identical with B's entire set of properties. Hence

```
concept is-a;
   case-of case-of;;
```

and therefore

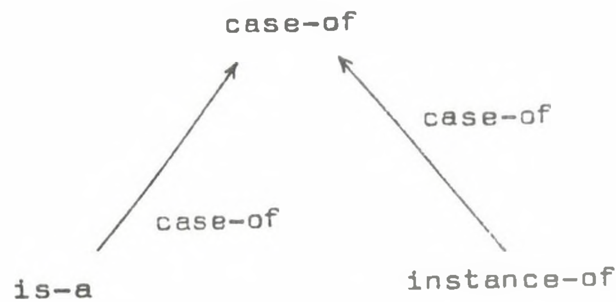


Figure 7.

Cases of "case-of".

Let us denote the transitive closure of "is-a" by "is-a*", and define the relation "instance-of$^\circ$" by

$$\text{instance-of}^\circ = \text{instance-of} \odot \text{is-a}^*.$$

where denotes the "circle product" (natural join) of relations.

Obviously

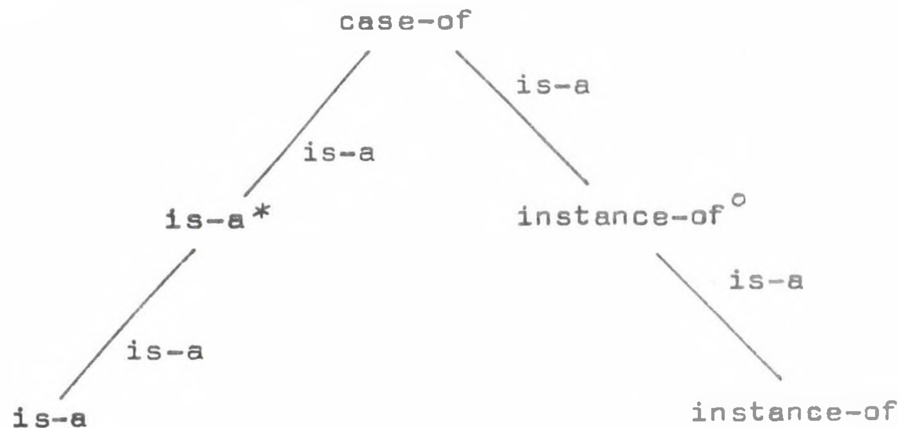


Figure 8.

Multiple "case-of".

Hence, when speaking about instances in general (i.e. by "instance-of$^\circ$") if "A is-a*" then the set of A's instances is necessarily a subset of B's ones. This fact may lead to wrong conclusions. One may define "is-a" rela-

tionships by "subsetting criteria" applied on the set of instances. We do not accept this and insist upon that definitions of "is-a" relationships among concepts _must not refer to instances_.

(Remark: In case of is-a* and instance-of$^\circ$ relations the attribute homomorphism rule should be weakened using the formalism offered by [H 81], [AGNS 80] and can be stated in a new form called the "convexity rule". We do not detail this here, see. e.g. [KR 85], [DKR 86].)

3.3 Universes

We have not yet made any restrictions about the use of relationships "is-a" and "instance-of" (disregarding the general constraint of loopfreeness declared at the level of "case-of"). There are several possible ways for such restrictions in order to obtain a meaningful and disciplined model.

a) Uniqueness assumption for instance subjects:

A instance of$^\circ$ B		B is-a C
and	$\Rightarrow$	_or_
A instance of$^\circ$ C		C is-a B

(including C = A, i.e. the relation "instance of" generates classes. This assumption is adopted by all known conceptual models.)

b) Layering assumption:

First we define "abstraction levels" in the following way. Let L_1 be the set of concepts which are not instances of other ones. Let L_k the set of instances of concept belonging to L_{k-1}. Assumption:

(i) L_1 is a classification over the entire set of concepts.

(ii) If A is-a B then A and B belong to the same abstraction level.

(iii) Attributes values of a concept must belong to the same level the concept belong.

(This assumption is also adopted by all conceptual

methodologies. It is a common belief that at most 3 levels are sufficient for all kinds of modelling. In fact, some methodologies employ three - e.g. TAXIS [MW 80], while others only two.

c) Uniqueness assumption for is-a subjects:

A is-a B and A is-a C ⇒ B = C.

(Some methodologies do not apply this restriction. In this paper, we do not suggest its acceptance or refusal. It is a possibility. Refusing the assumption has a cost, of course.)

d) Restriction to two levels:

Finally we may restrict the number of abstraction levels to two, thus obtaining a level of "abstract" concepts (classes, - the meta level), and "concrete" objects. Hence **objects** are instances of **classes**, and we make the following supplementary assumption:

> Relation "is-a" is applicable at the level of classes only.

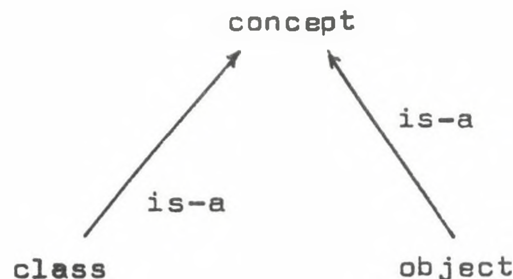


Figure 9.
Division into "abstract" and "concrete".

(In fact, this division is essentially adopted by known methodologies. So called "meta-meta" level introduced in some models serves only a "subsidiary" role. We remark also that SIMULA 67 obeys exactly a), b), c), d).)

At this point we have the following full set of concepts:

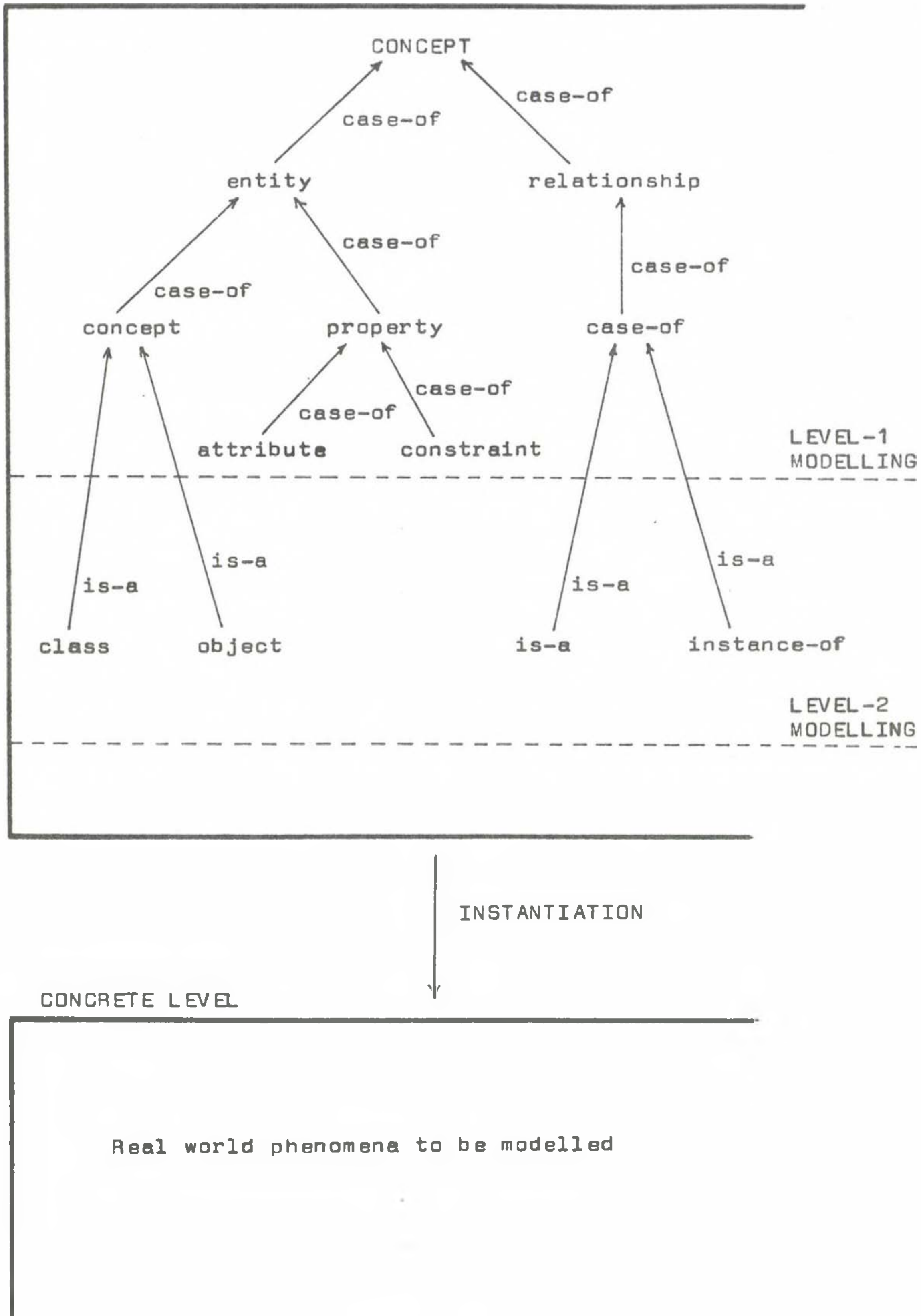


Figure 10.

Summary of modelling concepts: level 1. and 2.

4. Level of information system's traditional logical models

We can go further into directions of any of the known, more detailed models. In case of information systems it is most important to distinguish between "information" structures and "transactions" and to provide specific means for handling them both. Hence we can refine the "class" concept as:

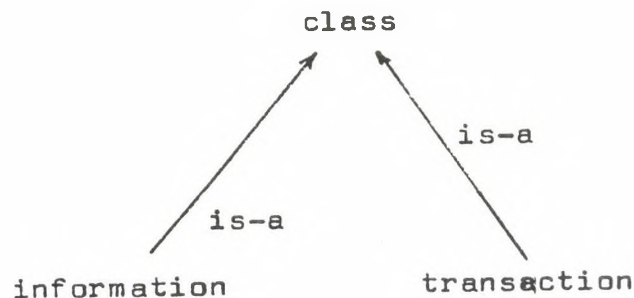


Figure 11.

Information system's basic modelling classes.

4.1 Information structures and transactions

To describe information structures at the logical level of modelling specific attributes are used. For instance, TAXIS introduces "keys", "constants", "variables" (names changed here). Therefore, we can e.g. define the concept "information" as:

```
class information;
   is-a class;
   attributes:
      key: information;
      constant: attribute;
      variable: attribute;
```

Remarks:

a) Attributes in general (without further restriction) are multi valued. This applies to the above definition too, i.e. an "information" structure may have any number of keys, constants, variables.

b) The "key" can be an arbitrary compound information structure according to the above definition. (See also 4.3 later.)

c) One might propose to represent "keys" "constants", variables" as is-a versions of the concept "attribute". We intentionally did not choose that way.

Similarly we can write e.g.:

```
class transaction;
   is-a class;
   attributes:
      parameter: information;
      local: attribute;
      function: action;
      returns: information;
    constraints:
      prereqs: predicate;
      results: predicate;
```

Here we used two undefined concepts namely "action" and "predicate". In fact, algorithmic and logical expressions do not really belong to logical or conceptual levels of modelling. At this stage, for the convenience of the real user, these should better be expressed verbally, or aided by a few well chosen specification formalisms, but certainly not fully formally (i.e. not by presently available formal mathematical specification tools.)The right selection of formalisms really needed at this level is a research to be done.

4.2 Attribute properties

For information systems design purposes it is convenient to define a number of a properties of attributes. A particularly useful selection is the one proposed in [KM 84] including

- multi valued
- single valued
- unique
- exhausting
- non null
- reverse

etc.

ones. These can be modelled either by adding new attributes and constraints to the definition of "attribute"

itself or by defining attribute-subtypes. We do not go into details here.

4.3 Attribute modes

As mentioned earlier, to each attribute a mode can be associated (e.g. "set-of", "list-of", etc.). These can serve to express secondary abstraction mechanisms namely "aggregation" and "association" see e.g. [BMS 84], [G 85].

One might wish to apply "aggregation" and "association" as selfcontained separate tools to create new concepts and use concept definition constructors like "is a set of", e.g.:

directorial board **is a set of** persons;

We do not accept this way however. Instead, we say that "being a member" is a property and should be represented this way, i.e.:

```
concept directorial board
   attributes:
      members: (set of) person;
      (further possible properties);
```

A number of other "modes" can be suggested ("array-of", "tuple", etc.). We do not fix them here. There is a question however, whether the modelling person is allowed to define his own new modes (as concepts) himself? This question is not answered in the present paper.

A final remark concerns "set properties". Some models introduce separate mechanisms to handle properties associated to sets of object instances (with variable membership, of course. This is for instance, one reason of introducing meta-classes in TAXIS.) We have different opinion. Merely a set of objects is not a conceptual level construct. If it _is_, then it must be selfcontained concept _with a name_ and defined in the normal way (e.g. as "directional board" above. And then we may well declare:)

```
concept directional board;
   attributes:
      members: (set of) person;
      average-salary: numeric;
      etc.
```

5. Open questions: model transformations

For practical use of computer aided conceptual modelling the key is the dynamics with respect to the modelling processing itself in two ways:

5.1 Reformulation an existing model

The knowledge process acquisition is iterative in nature:

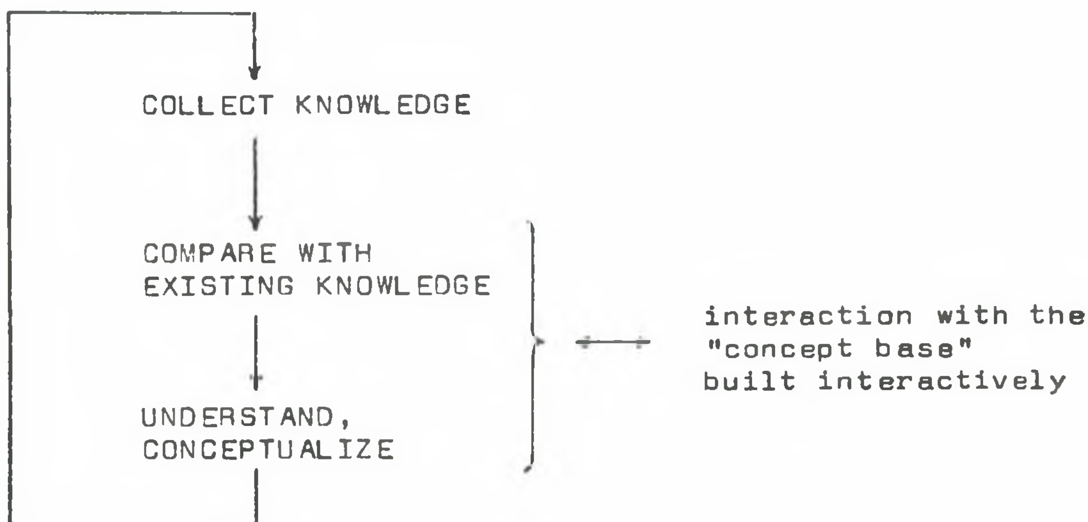


Figure 12.

Knowledge acquisition process.

It is typical that new knowledge sometimes leads to the need of radical changes in the knowledge base not allowed by presently used models (e.g. changing or even cancelling classes having instances). Of course, depending on the exact model used there is always a well definable set of transformations preserving the integrity of the knowledge base in a way. To give precisely these transformations is an important work to be done.

5.2 Mapping between levels.

Using a multi-layered technique of top-down modelling we have selfcontained models in each level (each one should be "complete" according to appropriate criteria which are characteristic at the given level). Hence we must frequently handle **same concepts** on different levels (in different degree of abstraction).

Therefore an adequate mapping mechanism is needed to connect levels guaranting the compatibility among interdependent concepts. This is also an area of further work (in addition to the lot of others mentioned in the sequel).

REFERENCES

[AGNS 80]
Andréka,H., Gergely,T., Németi,I., Sain,I.: Theory morphisms, stepwise refinement of program specifications, representation of knowledge, and cylindric algebras. Preprint 1980.

[AC 83]
Albano, A., Cardielli,L., Orsini,R.: Galileo: A Strongly-Typed Interactive Conceptual Language. ACM TODS, Vol 10, No.2, pp.230-260, June 1985.

[BRAC 79]
Brachman,R.J.: On the Epistemological Status of Semantic Networks. In: Findler,N.V.: Associative Networks: Representation and Use of Knowledge by Computer, pp.3-50. Academic Press, 1979.

[BMS 84]
Brodie,M.L., Mylopoulos,J., Schmidt,J.W.(Eds.): On Conceptual Modelling. Springer Verlag, 1984.

[CHEN 76]
Chen,P.P.S.: The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems. Vol.1, No.1, March 1976.

[D 70]
Dahl,O.J., Myrhaug,B., Nygaard,K.: SIMULA 67 common base language. Norwegian Computer Center, Oslo, 1970.

[DKR 86]
Demetrovics,J., Knuth,E., Radó,P.: Computer aided specification techniques. World Scientific, Series in Computer Science Vol.1, pp.1-114, Singapore, 1986.

[G 85]
Gibbs, S.J.: Conceptual Modelling and Office Information Systems. In: Tzichritzis, D. (Ed.): Office Automation pp.194-224, Springer Verlag, 1985.

[H 81]
Henkin,L., Monk,J.D., Tarski,A., Andréka,H., Németi,I.: Cylindric Set Algebras. Lecture Notes in Mathematics 883, Springer Verlag, 1981.

[KM 82]
King,R., McLeod,D.: Semantic Database Models. In: Yao,S.B.(ed.) Principles of Database Design. Prentice-Hall, Englewood Cliffs,N.J.(to appear).

[KM 84]
King,R., McLeod,D.: A Unified Model and Methodology for Conceptual Database Design. In: Brodie,M.L., Mylopoulos,J., Schmidt,J.W. (Eds.): On Conceptual Modelling, pp.313-327, Springer Verlag, 1984.

[KR 85]
Knuth,E., Rónyai,L.: Closed Convex Reference Schemes. In: Teichroew, D., David, G. (Eds.): System Description Methodologies, pp.435-453, North Holland, 1985.

[LUD 84]
Ludewig,J., Mitchell,R.: Incompleteness and Abstraction in Program Descriptions. International Workshop on Models and the Languages for Software Specifications and Design. Orlando, Florida, 1984.

[MW 80]
Mylopoulos,J., Wong,H.: Some Features of the TAXIS Data Model. Proc. 6th International Conference on Very Large Databases. Montreal, Canada, October 1980.

[QUIL 68]
Quillian,M.R.: Semantic Memory. In: Minsky,M. (Ed.):Semantic Information Processing. MIT Press, 1968.

[SH 81]
Shipman,D.W.: The Functional Data Model and the Data Language DAPLEX. ACM TODS Vol.6, No.1, pp.140-173, March 1981.

Intelligent Databases

H. Brückler, W. Fritz, V. Haase, R. Kalcher
Institut für Maschinelle Dokumentation
Forschungsgesellschaft Joanneum
Graz/Austria

How to use existing databases to build intelligent question-answering systems (Requirements and an outlook)

Most existing and planned expert systems use a knowledge base together with inference mechanisms especially designed for the specific task. This implies that the basic facts which constitute the knowledge base have to be entered into the system during the building period. Not much is known how to use existing machine readable data and databases to build knowledge bases. In this paper we want to discuss mechanisms how to use existing textual relational databases as knowledge bases for expert systems, and report on some experiences in the intelligent use of literature databases.

This field is especially interesting as enormous amounts of referral data have been and are continuously stored in internationally accessible databases. CUADRA ASSOC. lists approx. 2700 databases, several dozens of different query languages are used, almost none of them allows a really intelligent dialogue with the user. On the other hand to reenter all these data into an expert system manually is not possible. Typically a database (e.g. INSPEC /1/) holds more than 2 500 000 records, each of them consisting of up to approx 1000 characters grouped into 10 to 20 fields, classified by 10 to 50 relevant keywords. We need an intelligent interface to existing databases, an interface that gives the user as much help as possible.

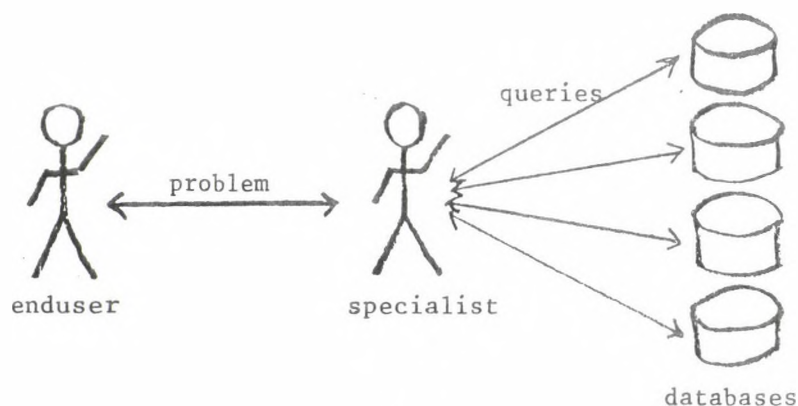
Up to now investigations and limited tests have been performed to find out how AI principles could help to make searching in databases - especially literature databases - easier. We can identify two methods to follow:

a) An "Expert-System-Black-Box" as interface to one or more online databases.

b) A two step method which first transforms conventional databases into a knowledge base which then can be accessed via an expert system.

I. An "Expert-System-Black-Box"

In many cases a person who is interested to retrieve information from a database - especially from a literature database - is not able to use this database without any help. Therefore specialists must be involved to satisfy the wishes and requests of the user. These specialists represent an interface between the enduser - typically not a computer or database specialist - and the database management system.



Pic. 1

But this present method has disadvantages. On the one hand there is a first group of disadvantages for the enduser:

1) To get the information the user must meet the expert physically. That means that the user has to go to the office and this implies the next disadvantage:
2) The enduser must pay attention to the office hours especially at weekends, holidays,... no information retrieval is possible.
3) An human expert can only serve one customer at a time. This fact causes high costs to the enduser.

On the other hand also the expert is troubled with this method of information retrieval.

4) The expert must be familiar with different query languages because each database must be served in a different manner, as well as
5) with the structure of many databases covering various different scientific disciplines.

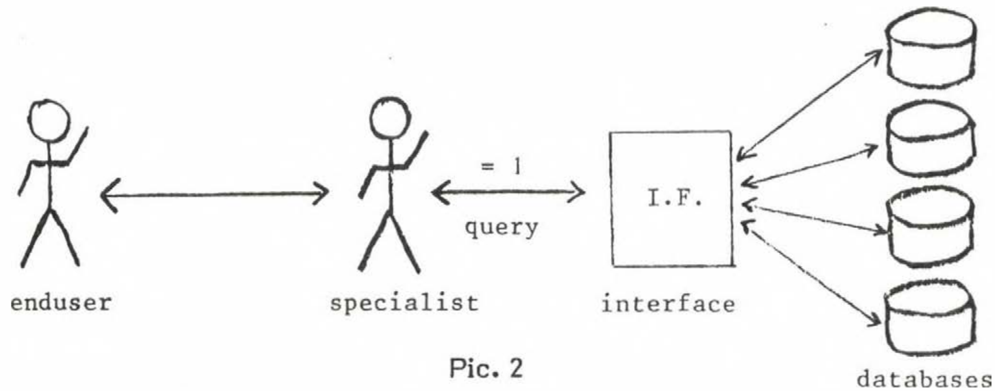
As long as you are working only with one database there are no great problems for an expert to learn the usage of the system. But as soon as you are working with more than one database management system there is a great danger in mixing up different queries.

One example: To find documents with a special keyword you have to use for instance **f "KW"** (for Find) in one language, or **s "KW"** (for Search) in another, or **l "KW"** (for Locate) in the third one. And if you use a dozen of different systems almost each character of the alphabet is used for this function. What is even more difficult is to change to a language that uses context sensitive queries (different meanings in various operation modes (e.g. Search mode, Print mode, etc.)).

These problems arise for an information retrieval expert who uses more than one database. On the other hand if an enduser wants to use the database himself (e.g. he is a scientist and he has the opportunity to use a terminal e.g. for his daily mathematical work, and now wants to use this equipment for information retrieval without help of a specialist) he may have these problems already with just one database. Therefore it makes no sense for a casual user to learn a special query language only for a few retrievals.

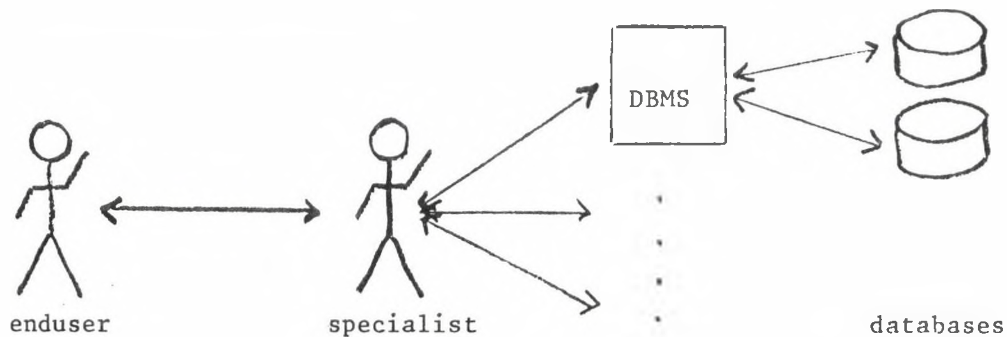
But now a little remark to the upper topic: Query languages do not differ very much in their principal functions but more in small - but nevertheless important - syntactic features.

Therefore the idea comes up that these syntactic differences could be filtered out by building an interface between the operator and the different database systems. The updated version of our first picture may look as follows:



Pic. 2

The difference to picture 1 is that the operator has to learn only one query mechanism to work with different databases. Such systems do already exist. E.g. the DIALOG database consists of about 300 databases which are combined and use only one query language. Using this system we get the following picture.

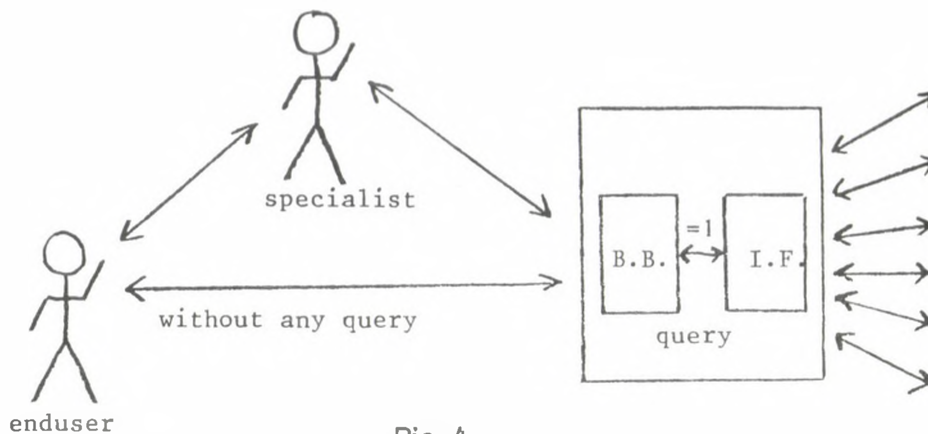


Pic. 3

The improvement to picture 1 is that the operator has not to serve each database with a different query language but can use a group of databases with one unique manipulation mechanism. This means that about 2700 existing databases worldwide can be accessed with some dozens of query systems.

But the situation today is not optimal because only two different query languages can be troublesome for an human operator. Even the idealistic idea of picture 2 does not solve the problem for the enduser if he wants to use the system by himself, because - as said before - one query formulated in a special syntax is one query too much for a casual user.

Therefore the idea in picture 2 must be changed in such a way that the enduser should be able to serve the information retrieval system without learning any special query language. We should alter picture 2 to picture 4 by adding a new function unit: the black box in the following diagram.



Pic. 4

Now we want to point out the ways the enduser will have to communicate with this database without using special query mechanisms.

1) A very fine realisation of this communication would be the unrestricted use of spoken natural language just the way you would speak to an human expert. But you will certainly agree with us that this is a little bit utopic today.
2) Just the same as above but query input is performed using a normal keyboard. In this case we can assume that a person who has the technical equipment (e.g. a terminal, a teletype or a videotex computer) can make his/her own retrievals if he/she is familiar with the handling of a keyboard.
3) Another method to realise such a communication is by the use of computer graphics. As an example we may look at the new type of operating systems such as on Apple MacIntosh or PAM (abbreviation for Personal Applications Manager) /2/ on Hewlett-Packard computers or GEM (Graphics Environment Manager) from Digital Research which is available for CP/M and MS-DOS and therefore for nearly all types of personal computers. Using this method the user sees ideograms on the screen and can formulate his requests with the help of a mouse or a touch screen.

   An additional advantage of this third method is its independance of any natural language. You can build one unique system for different countries and different languages, because there is no difference between e.g. german, english, or hungarian systems because pictures are understood everywhere. Up to now we have done no

investigations until now whether a communication based only on ideograms is possible or not.

Certainly other concepts are also thinkable ...

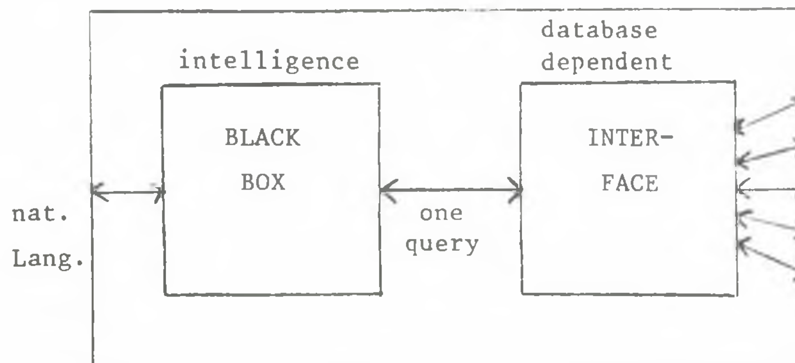
Let us now have a look on the advantages and disadvantages of these three principal possibilities:

	1) Speech	2) Alphanumerics	3) Graphics
terminal	expensive; hardware not available commercially; software expensive;	very cheap; teletypes and typewriters possible	expensive hardware (bit map display)
bandwidth for data transfer from/to terminal	only few characters 300 baud will be enough (both directions)		either: much data and no terminal-software or: few data and a great deal of software (the host sends a coded character which causes the terminal to draw a whole diagram)
intelligence of terminal	+++++ very essential	- not necessary	+ not forced to be intelligent
today available	- will come several years from now	++ in use terminal, videotex, teletype	+ existing but limited commercial usage until now

Pic. 5

The most practicable way of the realisation of such a black box seems to be method 2. We will discuss the further problems of implementation on this model because it seems that with this method the user will get a practicable system which satisfies our first requirements, and it can be realised without a great deal of hardware because all the necessary hardware is available and not too expensive.

As there are no bigger problems with the hardware, all the difficulties arise in the software of this realisation. Let us now have a look inside the black box and the interface.



Pic. 6

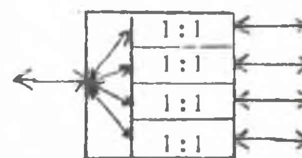
First we start with the less complex part of this black box interface system: the interface box.

There are two possible solutions for the realisation of such an interface:

1) 1 1:N - Convertion
2) n 1:1 - Convertions

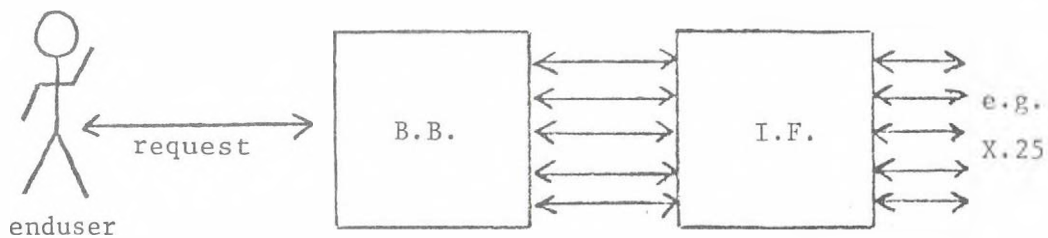
To illustrate these facts some pictures:

Pic. 7.1



Pic. 7.2

Currently when using widespread hardware techniques only one connection to a database may be active at the same time. If there are requests to more than one database the system must spool these requests and must serve them sequentially one database after the other. For future requirements one user request may be split up into several queries (semantically identically, syntactically different) to different databases. By support of powerful communication hardware these queries can be processed parallelly and handling with the user can be done comfortable (speed, costs,...).



Pic. 8

The first implementation would be an overall translation program which can transform the standard query into any of the special database queries. A great disadvantage of this version lies in the wish to add a new database to the system because it could be that the whole program must be changed in this situation, that means that already existing transformation algorithms must also be tested for correctness once again.

On the other hand version 2 has dedicated modules for each database management system, and therefore some syntactic features are listed in each module (problem of redundancy). But when you want to communicate with a new database you have to add only a new module to your interface. A welcome side effect of this second version is that the enduser can configurate a system for his own special personal requirements. This means that the modules for those databases he is not interested in are not added to the interface.

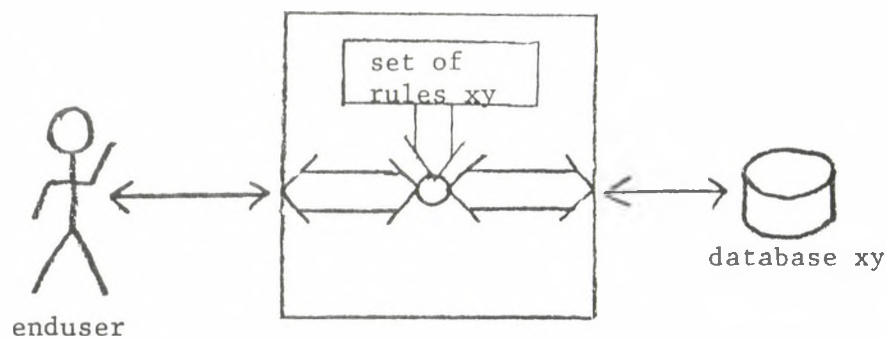
This shows that the second method seems to be preferable to the first method. At this point we should investigate how these modules are to be realised. Two practical ways are thinkable:

1) Each module is implemented as a whole program.
2) It is also thinkable that these modules only contain the rules for the transformation which are used by the black box to translate the standard query to the special database dependent queries.

The second way is very realistic in so far as you will certainly have to use an expert system to realise the black box and in this occasion this expert system can also manage the rules of the modules.

This has the consequence that our interface modules are no more independent programs but only a collection of data and facts which are used to control the translation of the standard query.

We can explain our results until now in the following picture:



Pic. 9

The rules for these procedures can be stored in the databases themselves and are downloaded if they are needed.

After we have seen the function of the interface modules we now want to change the black box into a "white box".

Let us resume the features of this unit:

1) It must handle the communication with the enduser in or near natural language (e.g. english, german, hungarian,...).
2) The requests formulated in natural language must be translated into the standard query language.
3) And as just said before this standard query syntax is to be transformed into the special database dependent query format with the use of the rules in the interface modules (see also picture 9).

The third aspect seems to be not so problematic as the other two because you have only to change one fixed syntax to another well defined syntax. The rules for these grammars are stored in the above discussed interface modules.

The real problem are the points 1 and 2 where it is to decide whether these two points are really two independent functions or must be seen as one global component.

A beginning in this direction is already done by some existing systems like HAM-ANS /3/ which is an intelligent hotel manager system for booking hotel rooms by natural language. Another system would be INTELLECT /4/ which is a natural language interface to a database. Another example for a natural language interface is implemented in the geographic database CHAT /5/ which can answer questions formulated in natural English.

Systems such as SHRDLU /6/ and ELIZA /7/ are certainly a good help for investigations on this topic too. Let us just list some of the most important features of such a communication system.

1) First of all the system must be able to explain all the functions and possibilities of the system itself. It must offer also information about all administrative functions to the user such as accounting, general costs,...
2) The whole man-machine dialogue must be an iterative process because in most cases the answers of the system will not match with the aims and the requirements of the user and vice versa.

3) Another important aspect is that the system should ask only as much as necessary to reach the next user's aim to help him to shorten his work for input. An example of this way would be MYCIN /5, 8/.
4) Moreover the communication system must test the user input for some semantic correctness as for example questions formulated too weak would lead to an enormous mass of data. For example when you are searching for all books written in English.
5) Another point is an explanation function. E.g. if you use boolean operators the system should show how the endresult is constructed out of the subresults.

With this small table of desired functions we will finish this aspect of intelligent retrieval systems. In the second part we want to discuss a completely different way to make databases more intelligent.

II. A two step method which first transforms conventional databases into a knowledge base which then can be accessed via an expert system

Today a great number of databases exist all over the world. They offer enormous amounts of data in various fields. We may also get the knowledge of experts to use these traditional databases, but not in such a manner as we need such knowledge of an expert for building a knowledge based system.

It seems there are two main reasons why we cannot use existing data immediately for setting up a knowledge base.

1) Traditional databases are in most cases not well structured as far as semantic relationship of the content is concerned. Stored data are more or less a loose collection of facts in fact databases or of referral data in literature databases.
2) Knowledge bases (intelligent databases) as needed for setting up and working with expert systems are structured in different ways. As commonly known there are several ways to represent knowledge in a knowledge base. Some ways of representation are frames, semantic networks and ruled based production systems.

In the next years we can expect a great evolution in different fields of artificial intelligence in general and in special fields of expert systems - or more general - knowledge based question answering systems.

On the other hand usual searches in traditional databases often do not satisfy customers due to different reasons as already mentioned above. The main reason seems that there are many different and inflexible query languages which are inefficent even if you are able to describe your questions precisely in natural language.

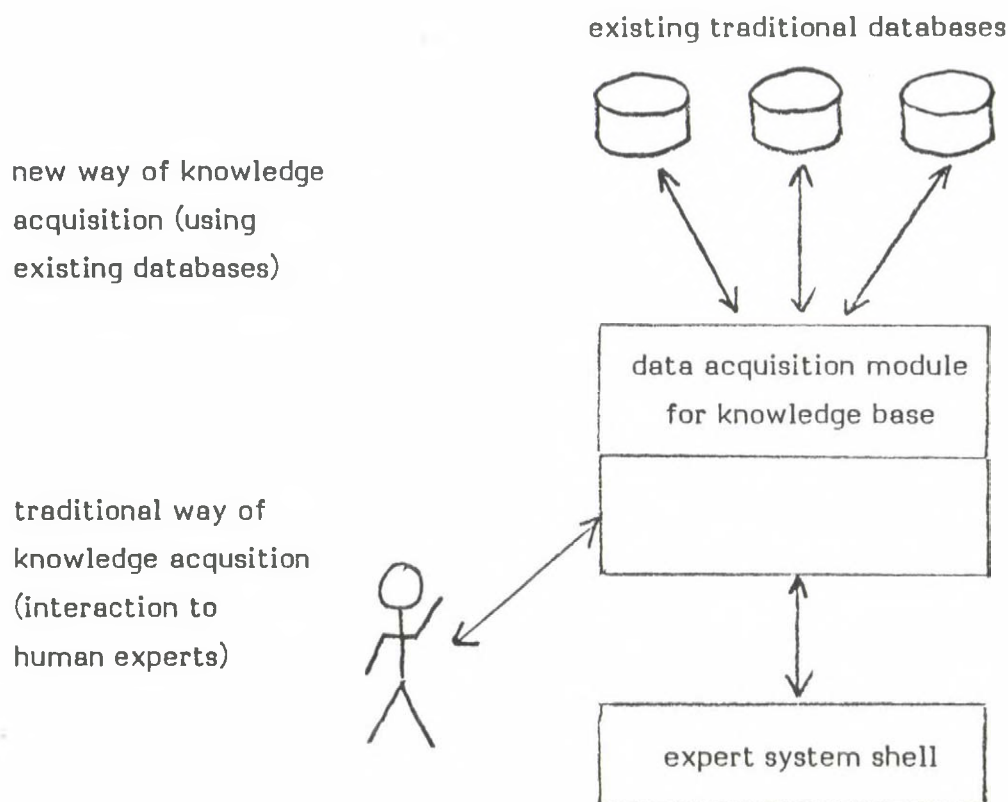
One possible way to build an expert system for literature searching nevertheless is to use existing databases and not to start from scratch.

For such an approach two main components of an expert system are of great interest. First the knowledge base and second the knowledge acquisition module.

For the representation of knowledge in the knowledge base which is to be built we may start using rules of the common known kind: "If condition (and condition ..) then action". As inference engine we may use a production system working independently of the contents of the knowledge base.

The kernel of our considerations is the construction of a knowledge acquistion module.
Its aim is the acquisition of knowledge from various sources including existing databases. We want to find a way to support the knowledge acquisition process using an intelligent program.

One important aspect is that work should be done automatically or at least half automatically with as little as possible interaction by human experts. Essentially knowledge should be extracted and filtered out of existing databases.



Pic. 10

At present the use of data of a literature database for building a knowledge base seems to be difficult. The main question is how knowledge of an expert can be extracted from record fields as title, author, classification codes, index terms, abstract, etc. /9/

As in fact databases a knowledge base does not exist only of facts resp. citations but also of knowledge "how to use this information".

As an example we may use chemical substances. Each substance has characteristics such as molecular formula, synonyms, extraction procedures, colour reactions, gas and thin layer chromatography, infra red and mass spectrum, etc. /10/

These facts are stored in some structured way in an existing database. This structure is known to the transformation program or at least to an human expert working with the system.

So one way to get data out of a conventional database into a knowledge base is the following half automatic process:

1) An human expert sets up a set of rules how incoming data is to be manipulated if the transformation program does not know the "special proceeding" for a particular database so far.
   The structure of incoming data and the restructuring mechanism are defined and stored in a library. In the case that the transformation program knows how to handle incoming data nothing has to be done by the expert. /11/
2) The transformation program restructures the incoming data according to the rules set up and fills the knowledge base.

Conclusion

This is a very rough model as you see but it is only a first theoretical investigation and we hope that when we will meet next we can show you runable preversions of the today's discussed black box or the transformation program.

Literature:

/1/ DIALOG Information Services, Inc.,
DATABASE CATALOG 1985

/2/ COM - Das österreichische Magazin für Computer Anwender
Jänner 1986, S 18ff

/3/ Hahn, W. von, et al.
The Anatomy of the Natural Language Dialogue System HAM - RPM,
In: Natural Language Based Computer Systems
Carl HANSER Verlag, München 1980

/4/ Simons, G. L.
Introducing Artificial Intelligence
NCC Publications, Manchester 1984

/5/ O'Shea, T. and Eisenstadt, M.
Artificial Intelligence
Harper a. Row, New York 1984

/6/ Winograd, T.
Understanding Natural Language
Academic Press, New York 1972

/7/ Weizenbaum, J.
ELIZA - A Computer Program for the Study of Natural Language
Communication Between Man and Machine
CACM 9 (1966), pp 36

/8/ Shortliffe, E. H.
Computer - Based Medical Consultations: MYCIN
Elsevier, New York 1976

/9/ Yakubowitz, Z.
Linkage Retrieval System
University Campus-Beth Hatefutsoth, Israel

/10/ Battista, H. J. et al.
Ein modernes Informationssystem für die toxikologische Analytik auf der Basis des (IV+V) - Systems
Zeitschrift für Rechtsmedizin
Springer 1985, S 235ff

/11/ Institut für Maschinelle Dokumentation
Aufbau und Einsatz medizinisch - bibliographischer Datenbanken
Graz 1986

ACTIVE COLLABORATIVE SYSTEMS

Prof. dr. L. Siklóssy
Vrije Universiteit
Subfac. Wiskunde en Informatica
Postbus 7161. 1007 MC Amsterdam

ABSTRACT

Active collaborative systems (ACS) not only answer questions from the user, but will give additional information not directly requested by the user, and might suggest other questions that may be more pertinent to the user's goals. In addition, active collaborative systems include a component which broadcasts information that may interest the user, even when the user did not request any information or ask any questions. ACS incorporate a model of the user to guide the transfer of information. Metric relationships among data play an important role in the strategy of an ACS.

1. INTRODUCTION

Data bases are becoming essential components of modern organizations. They can be accessed by specialized programs, by queries written in a query language, or (within some limitations) by queries expressed in a natural language. Several investigators have pointed out that systems which only answer a query are unfriendly, unsatisfactory and much less helpful than they could be.

A goal of our research is the development and mastery of techniques which will permit the rapid and efficient implementation of certain types of knowledgeable, friendly and helpful assistants to decision makers. These assistants will make use of data bases in an active, collaborative way: they are Active Collaborative Systems (ACS). By contrast, a data base system which only answers queries can be viewed as passive: it waits for a question, and has finished its task after providing the answer.

In section 2, we review three different ways in which passive question-answering systems have been judged unsatisfactory. The solutions that have been suggested to render the systems more satisfactory have lacked generality. In section 3, we propose a general framework which may remove all the inadequacies mentioned. This framework is based on a descriptive topology, both quantitative and qualitative, of the space of topics that is stored in the database. Section 4 outlines an extension of ACS to what we have called Active Data Bases. An Active Data Base provides a user with the sort of information that he would have requested, had he taken the time to request it. (But in fact, for a variety of reasons -lack of time, lack of computer-related skills, etc.- the user did not formulate the queries.) Thus, we see that an active data base broadcasts information to the user. It can be seen as responding to a set of latent questions that describe a user's interests. (Sometimes, we include active data bases under the term ACS, since the two types of systems have the same goal: inform and help the user.)

User models are needed by both of the above active systems: the systems' reactions must be adapted to the user's goals. Section 5 discusses user models. Section 6 is centered around the need for automatic restructuring of the storage structures of existing data bases to increase the efficiency with which the topological neighborhoods can be accessed from one particular topic.

2. INADEQUACIES OF PASSIVE QUESTION-ANSWERING SYSTEMS

We call passive those question-answering systems which simply answer a

question (see Siklossy [10].) Wahlster [12] discusses the inadequacies of such systems, namely:

a) the system may fail to inform the user that a modified question, closely related to his initial question, has a "significantly better" answer, which would help the user better to achieve his goals.

b) the system may fail to inform the user that his question includes presuppositions which are not actually valid, so that the answer given may be misleading or meaningless.

c) the system may fail to provide the user with additional information which the user would normally expect, and which he must now request explicitly.

To these categories, we add a fourth one:

d) the system may fail to inform the user of the additional, related topics that the system is ready to pursue at the user's request.

We shall now give short, illustrative examples for each of these inadequacies.

a) A company asks its decision support system about the cost of receiving a shipment of 800 widgets on Monday. The system answers, but fails to point out that an interesting quantity discount is available for orders of 1000 or more widgets, and that delivery on Wednesday or later would avoid the heavy "rush" surcharge.

Here is another similar example:
A company asks its decision support system about the cost of manufacturing 10000 widgets in its plant at location A. The system answers, but fails to point out that plant A has a considerable backlog and therefore the order may be much delayed. Another possibility (in addition, or separately from the above) is the failure of the system to mention that plant B could have the order ready sooner and/or at less cost, or that the order could be subcontracted to Company XYB at advantageous terms.

Siklossy [8,9,11] was the first to discuss the above type of inadequacy. He developed a theory, based on discontinuities, which tells a question-answering system when it should provide similar additional information. The theory has been implemented in such diverse areas as: air travel and airline tariffs, shipment of goods by truck or air, manpower assignment to projects, selection of a restaurant, selection of commercial real-estate for investment, distribution of tools and machines to various jobs and sites in a large corporation, vacationing, ordering spare parts for machines, etc.

b) Kaplan [4] and Webber [13] give examples of user presuppositions and misconceptions which are not usually detected by question-answering systems. Let us illustrate their points by considering the two short question/answer sequences:

-How many women teach in the Department of Information?
-Zero.

-Is John at the meeting?
-No.

The first query assumes that there is a Department of Information, and that it has teachers. If there is no Department, or if there is one but no teaching within it, then the answer "zero" is correct but misleading. The second query assumes that there is a John, that there is a particular meeting and that John could come to the meeting. If any of these suppositions is false, the response is again misleading.

Here is another small dialogue [illegible] first answer is somewhat misleading, while the second one is not.

-How many boxes of computer cards do we have in stock?
-Zero.
-Zero. We no longer use computer cards here.

c) Wahlster [12] describes a system which gives answers that are typical of those expected by human users when the minimal answer would be yes or no. To the question:

-Is there someone in room S5.23?

the minimal answer would be : -Yes. or -No. A more acceptable answer would be:

-Yes, George and Martha both entered S5.23 at 10.32 am. or -No. George and Martha left the room at 6.45 am.

A similar example is:
-Have you started on Project XBY?
-No.
-No. I dont even know anything about Project XBY.

We could expect a good ACS to be even more helpful, as for instance in the following:

-Have you started on Project XBY?
-No.
-No. Our department does not handle project XBY. The project is handled by the Archetypes department.
or:
-No. Our department does not handle project XBY. We handle design problems. Project XBY is a security project; these projects are usually handled by the Confidentiality Department. Shall I check whether they are indeed working on this project?

d) One problem in giving additional, not specifically requested information is: how much information should be given? A user model will help, but sometimes it is important to let the user decide so as not to annoy or bore her. Also, it is important to bring to the user's attention additional information, pertaining to the user's question, which the user may wish to explore. This information would not always interest the user, and therefore it should not be broadcast every time. Instead, it makes the user aware of other directions that she may wish to explore.

We shall not try to provide a complete bibliography of work touching on this area. Additional works in natural language processing which can diagnose presuppositions, and respond accordingly, include Lesmo [5,6]. The references mentioned in this section contain pointers to other relevant works. Of particular relevance are the developments of plan based theories of dialogue, which try to understand a user's goals, and which will be taken up briefly in section 5, User Models.

3. A TOPOLOGY OF TOPICS FOR ACTIVE QUESTION-ANSWERING SYSTEMS.

The four types of inadequacies in responses that were mentioned in the preceding section can all be removed by the introduction of a topology of the topics that are known to the system. The topology describes the multidimensional structure of the topics and their neighborhoods. We shall outline the form of the topology for each of the examples of the preceding section.

a) The topic here is a shipment of goods. The description of a shipment includes such dimensions as: what is shipped, quantity, dates for pick-up and delivery, route followed, carrier, itemized costs, likelihood of de-

lays, etc. Given a particul .pic, i.e. a particular shipment, the system searches for discontinuities that are favorable to the user, i.e. where a small change in the topic along some dimensions can result in large, favorable changes along some other dimensions. Thus, we notice that the topology includes metric information. As mentioned, this technique of discontinuities has been implemented in a variety of domains (see section 2.a.)

b) In the first example of section 2.b, the topic concerns departments (of a university, say). Information about departments includes their name, composition, leadership, location, function, budget, etc. Certain aspects are dependent, and therefore may presuppose others. For example, an "empty" department, i.e. one with a name but no budget or staff, has no leader. If a department has no teaching function, there are no teachers teaching in it, etc. Neighboring topics include departments with similar names (Department of Information Science), similar functions, some or considerable overlap of staff, etc. Therefore, a possible answer to the question:

-How many women teach in the Department of Information?

could have been:

-There is no such Department. The Department of Information Science has 3 women teachers out of 11 teachers. or:

-There is no teaching within that Department. Many of the Department members teach in the Department of Information Science.

In the second example, the topic is a meeting. The description of a meeting includes place, time, expected and invited attendees, whether the meeting is open and to whom, etc. Neighborhoods include other relevant meetings (say over the same topic, both in the past and future), meetings scheduled for the expected attendees, etc. Therefore, possible answers to the question: -Is John at the meeting? could be:

-No, it is a closed meeting, and he was not invited.
-No, John is away. He is expected at the following meeting of the "Widget advertisement group" on July 8th.

c) in the example of the query: "-Is there someone in room S5.23?," the topic is rooms, or perhaps, more generally, locations. (Actually, there is a second topic: the activities of John.) The dimensions of the topic include identification of the room, uses (with dynamic values), contents, etc. Neighboring topics include similar rooms (with similar functions, or occupied by similar occupants -persons or objects-), rooms which are geographically close by, etc. Descriptions of the dimensions can be bundled (preferably in a hierarchical way, which would be user dependent) so that responses which touch upon one dimension within a bundle will trigger further explanations along the other dimensions of the bundle. In our example, a request about occupancy of a room would search automatically (if the occupancy is not zero) for number and identification of the occupants, and for some information about times of arrival. So an answer could have been:
-The room is occupied by members of the "Widget advertisement group." They have occupied the room since 10 am.

Here is another example in a similar vein:
-Do you have Smid's book on Computer Networks?
-No.
A helpful ACS might say:
-No. In fact, we have no books by Smid.
And an even more helpful ACS would elaborate:
-No. We have no books by Smid on Computers. We have books on Computer Networks by A. S. Smit, and one book on Computer Communication by B. T. Smith. Here are the references:...

d) The mechanisms required for informing the user of additional information that the system may provide, if the user wishes to have it, are similar to

those required for part c) ai . ror example, one of the previous answers could have been:

-The room is occupied by members of the "Widget advertisement group." I can further tell you about:

A. when they started occupying the room.
B. until what time they have reserved the room.
C. which members of the group are present.
D. which members of the group are absent. etc.

Here is another example where the ACS indicates where it could be helpful, if the user so chose:
-Did Smid finish his program for the Widget Project?
-No.
An ACS could do better:
-No. Smid left the company ten months ago, before the beginning of the Widget Project. Would you like to know where he now is?
-Yes.
-He works for Outstanding Software Inc. I can tell you more about that company if you wish.

Therefore, it suffices to indicate, in the description of the topic, which dimensions or related neighborhoods will be specifically searched, with their values printed, and which dimensions or related neighborhoods will be simply mentioned as areas that the system could further explore. Again, these indications can be user dependent.

The topology provides a methodology for describing active question-answering systems and active data bases (see section 4). The topology provides a systematic approach to the design of these systems. We envision a simple man-machine interface which will allow the user to describe the topics of his domain, their dimensions, metrics, neighborhoods, etc. We hope that, with the use of this methodology, it will be possible to build, relatively rapidly and easily, high quality active systems.

4. ACTIVE DATA BASES: INFORMATION WITHOUT QUESTIONING.

Some persons find that they do not have the time or the desire to access a question-answering system. They may not wish to learn about the use of computers and terminals, or they find that their time is better spent in other pursuits. Such potential users of question- answering systems, for example high level managers, medical doctors, etc., could profit greatly from information in data bases. It may even appear that those who would profit most from the use of databases, due to the importance of their decisions, are the ones least likely to use them!

Active Data Bases may be the solution. An Active Data Base (Siklóssy [10]) broadcasts information to a user without requiring the user to actually ask for information. In practice, from a user model (section 5) we generate a set of temporary or permanent latent questions that are used as queries to the data base. Ideally, these questions would be precisely those that the user would have generated. They could be implemented as questions, filters or demons (equivalently, triggers.) Active Data Bases would inform the user of new, relevant developments on a timely basis, in the areas of interest to the user, and at the level of detail that the user would want. (Active Data Bases could have other uses too, such as occasional reviews of activities in a field of interest.)

5. USER MODELS AND GOALS.

The inadequacies of question-answering systems that we have discussed can be understood as follows: users have goals and expectations. Their goals are often not well formed. They ask questions to be able to gather information to achieve their goals. They have expectations about the level and amount of detail of an answer, the inclusion of relevant information, etc. They want to be informed of relevant new developments. The active systems

that we have described attempt to make it easier for the user to achieve his goals.

Active systems will require user models. We intend to develop frameworks which will permit the description of user models. Much work has been done recently in the area of models of user plans, in particular to understand dialogue: Schmidt [7], Cohen [1], Hayes-Roth [2], Jackson [3], etc. For us, it is particularly important to have user models which can interact smoothly with the topics topology that was described in section 3.

6. AUTOMATIC RESTRUCTURING OF DATA TO SUPPORT THE TOPOLOGY.

Active systems must not only find answers to a particular query, but must also search the neighborhoods of the query. The techniques developed in data base research have emphasized the development of techniques of storing data and subsequently retrieving such data given specific, single queries. To support efficiently the search within neighborhoods, we must look into novel techniques for data storage. Present storage techniques, left unchanged, cannot provide an adequate support to the development of active systems.

The topology that we have described can be viewed as describing certain types of relations among data items. We can view this topology as an extension of the conceptual schema of a data base. The topology introduces additional considerations not found in current conceptual schema: relationships among neighboring elements in a data base. Some of these relationships are quantitative and are based on metrics; others are qualitative. Data base technology does not provide facilities for easy and efficient passage from one datum to a neighboring datum, especially when the neighborhoods can be along several dimensions. (There are some exceptions: for example, neighborhoods defined by 'next' or 'close by' keys in relational data bases.)

7. CONCLUSIONS.

The Active Systems that we have described fall into two categories: Active Collaborative Systems and Active Data Bases. These two types of systems are complementary. Active Collaborative Systems will provide pertinent and, in a certain sense, practically complete answers to questions. Active Data Bases will keep users informed of developments in a changing environment. These systems will be built with the help of a single, uniform tool: a descriptive topology of the topics known to the database. The use of a single tool is a far cry from the diverse, often ad-hoc techniques that have been used to remove the inadequacies of passive data base systems. The topology is user dependent, and user models play an important role. One of the goals of this proposal is the development of practical, efficient framework for the description of users. Finally, the practicality of active systems requires new developments in data storage techniques to support the descriptive topology.

8. REFERENCES

[1] Cohen, P.R. and C. R. Perrault: Elements of a Plan-Based Theory of Speech Acts (Cognitive Science, 3, 1979, pp.177-212).

[2] Hayes-Roth, B. and Hayes-Roth, F.: A Cognitive Model of Planning (Cognitive Science 3, 1979, pp.275-310).

[3] Jackson, P. and Lefrere, P.: On the Application of Rule-Based Techniques to the Design of Advice-Giving Systems (Int. J. Man-Machine Studies, 20, 1984, pp.63-86).

[4] Kaplan, S. J.: On the Difference between Natural Language and High Level Query Languages (Proc. Association for Computing Machinery 1978 Annual Conference, 1978, pp.27-38).

[5] Lesmo, L., Siklóssy, L. and Torasso, P.: A Two-Level Net for Integrating Selectional Restrictions and Semantic Knowledge. (Proc. IEEE Int. Conf. on System, Man and Cybernetics, India, 1983, pp. 14-18).

[6] Lesmo, L., Siklóssy, L. and Torasso, P.: Semantic and Pragmatic Processing in FIDO; A Flexible Interface for Database Operations. (Information systems, 1984, has appeared).

[7] Schmidt, C. F., Sridharan, N. S. and Goodson, J. L.: The Plan Recognition Problem (Artificial Intelligence, 11, 1978, pp.45-83).

[8] Siklóssy, L.: Question-Asking - Question-Answering Systems (Proc. Int. Seminar on Intelligent Question-Answering and Data-base Systems, I.R.I.A., Rocquencourt, France, 1977).

[9] Siklóssy, L.: Impertinent Question-Answering Systems: Justification and Theory (Proc. Association for Computing Machinery 1978 Annual Conference, 1978, pp. 39-44).

[10] Siklóssy, L.: Passive vs. Active Question-Answering (Proc. First Int. Symposium on Policy Analysis and Information Systems, Durham, NC, USA, 1979, pp.271-276).

[11] Siklóssy, L.: Experiments with a Query Adjusting Knowledge Based System (Proc. Fifth Int. Congress on Cybernetics and Systems, World Organization of General Systems and Cybernetics, Mexico City, 1981).

[12] Wahlster, W., Marburger, H., Jameson, A. and Busemann, S.: Over-Answering Yes-No Questions: Extended Responses in a NL Interface to a Vision System (Proc. 1983 Int. Joint Conference on Artificial Intelligence, Karlsruhe, 1983, pp.643-646).

[13] Webber, B. L. and Mays, E.: Varieties of User Misconceptions: Detection and Correction (Proc. 1983. Int. Joint Conference on Artificial Intelligence, Karlsruhe, 1983, pp.650-652).

ABSTRACTION AND DATA STRUCTURING

Ágnes Hernádi

Abstract

Abstraction mechanisms such as classification/instantiation, aggregation/decomposition, lambda abstraction (procedure formation), specification/implementation, specialization/generalization, association/membership etc. have gradually developed with our increasing knowledge about the matter of programming. Four of them, namely classification, aggregation, generalization and association, provide organizational axes to structure a data space and are fundamental to most semantic data models.
Regardless of the specific structure of domains of database applications the above organizational principles seem to be widely applicable.

1. Introduction

Recent research in data modelling is confronted with the paradox:

- at the application level one generally perceives unrepresentable details, and
- at the representation level one generally represents unperceivable details.

The abstraction mechanisms cope with this problem by suppressing unnecessary details, and by structuring and formalizing relevant information.

1.1 Abstraction and modelling

What we mean by abstraction, is called modelling in the areas of natural and social sciences. Both of them require a decision of the important features in the system, the variability

involved, the formal specification, the way of validation and so on to be applied.

Modelling comprises three intellectual tasks:
- the perception of reality from a certain aspect,
- the choice of the appropriate model, and
- the representation of the perceived reality according to the given model.

A representation is obviously some kind of reality, and as such it is a target for another representation according to another model.

Owing to the incompleteness of human observation and our limited ability to represent reality completely, any representation is an abstraction of reality.

1.2 Structure and semantics

There are a number of different abstraction mechanisms supported in various models, although none of them is supported in each model. These models have developed gradually with our increasing knowledge about the matter of programming. Some of the most frequently applied abstraction machanisms are:
- classification/instantiation,
- aggregation/decomposition,
- generalization/specialization,
- association/membership,
- lambda abstraction,
- specification/implementation, and
- mapping or multiple views.

We use the notion of relationship in the sense of combining several units into a large single unit. So we can think of a model as consisting of units of some sort related by relationships of various kinds, and an abstraction mechanism aiding data structuring as a kind of relationship.

Plenty of open problems conserning structure are addressed both in database and in programming language areas, such as

- features to be checked statically and the ones that have to be checked dynamically;
- the features that cannot be represented structurally; and
- the problem of defining and implementing complex constraints by means of structure.

Due to the large number of complex constraints and the amount of data values in databases, these problems may be more severe for database applications than for programming language applications. In recent databases persistent structural properties of data are emphasized instead of operations changing in the course of time. A structural description can be more simple and terse than its behavioural variant. It is easier to understand and to analyse a structural description than a behavioural one, accordingly it is usually more convenient to verify structural properties than to prove that programs are correct. Finally we already have well-known tools to describe, analyse and implement structural properties. Therefore it is important to make a study of how far these existing structural tools can express semantics.

2. Abstraction mechanisms aiding data structuring

2.1 Classification/instantiation

The application of classification in programming can be traced back to the early autocodes where the concept of type traditionally referred to the internal structure of data structures implementing language symbols instead of referring to relationships between things represented by those symbols.

Programming languages, databases and artificial intelligence systems all presume that entities or objects can be classified into "types". As one might expect however, "type" has not got the same meaning neither in different nor within individual areas.

Types have got a powerful mathematical foundation [GO'75], [GO'77], [GO'77a], [EH'78], [TH'78] and there is no doubt of them being useful for many purposes. Types are primarily used in classifying objects or entities. The manner of classification however varies considerably. Neither fixed concept nor standard way of usage has evolved for types as yet. It seems as though types are tools in solving various problems in the above mentioned areas.

On designing a type system or a set of types at will, we have to choose a particular meaning for the undefined notions in the type definition such as objects, properties and so on.

For example objects in the type HORSE (Fig. 1) have properties such as name, sex, owner, date of birth, measurements, species, colour, marks and utility.

```
data type HORSE with
 attributes
   name: HORSE_NAME;
   sex: {Mare,Stallion,Gelding};
   owner: OWNER;
   date-of-birth: DATE;
   measurements: set of MEASUREMENT;
   species: SPECIES;
   colour: COLOUR;
   marks: set of MARK;
   used-for: set of {Racing,Sport,Breeding,Slaughter,Others};
end HORSE;
```

Figure 1.

The "objects" should not be limited to data. Type systems can be defined for procedures and for relationships too. There are different ways of building a type system for procedures or transactions. For example on the pure functionality of procedures or on any other abstract information belonging to the procedures. We can think of a sorting procedure as a type of procedures, independently of types of its arguments and outputs. One can organize procedure specifications into types based on the relationships of the pre- and postconditions.

Then again instances of these types can be considered as actual applications of the procedure. The operations applicable to the instances of a type of procedures present the parameters and local variables for procedures.

For example in describing REGISTER, the action of registering a foal (Fig. 2) , we may have two parameters for the foal and its dam respectively, with constraints stating, that for each instantiation of REGISTER

- the two parameters have to be of type HORSE;
- moreover a mare with her foal at foot;
- there has to be a registration of covering this mare at a stud-farm 11 month before foaling; and
- finally there must be a birth-certificate issued for the foal when an instantiation terminates.

```
transaction REGISTER with
  parameters
    f: HORSE;
    m: HORSE;
  prerequisites
    Mare-with-foal-at-foot?;
    Sire-known?;
  actions
    a: register f;
  postrequisites
    Birth-certificate-issued-for f;
end REGISTER;
```

Figure 2.

In the presence of a lot of constraints it could be sensible to group these into types of their own. A type of assertions would have free variables and one of its instances would replace these free variables by constants.

Our example (Fig. 3) could be instantiated by replacing Z by any type for which "name" is a one-to-one mapping; so Z can

be replaced by the type THOROUGHBRED.

$$\forall x,y \in Z (name(x)=name(y) \Rightarrow x=y)$$

Figure 3.

A type can also be considered as an object, and as such can itself be classified.

Considering types as classes of objects together with collections of operations on objects of the corresponding class, seems to be suitable to support modelling. This concept of type involves a notion of the composition of types from the more primitive ones and as a consequence the notion of object composition and that of operation composition. Using this notion of type aggregation and generalization may be represented as a type composition. The operation and the applicability constraints associated with a type may be considered the representation of the dynamic properties of objects.

In practical applications there are a plenty of problems addressed and not referred to by all definitions of type. Such problems are among others:

- the issue of subtype,
- the matter of nominal versus structural equality of types,
- the possibility of multiply typed objects.

Types got their full power by the concept of parameterization [TH'78] which has lead to the concept of "parameterized types" or "type constructors". This allows to specify classes of abstract data types with different objects and even partly different operation semantics.

There are at least as many purposes of applying types as many users exist. From the middle of 1970's in programming languages there is a strong emphasize on binding operations up

with abstract data types and protecting integrity of representation. So types are used to check whether the use of an operation is valid or not and to select a particular instance of a generic operation required by a given application.
In database systems types are used also for checking the validity of operations - Codd has used such a concept of type to justify that join operations are meaningful in databases - and for describing information about objects or entities.
In the areas of artificial intelligence one can find all the uses of types applied in programming languages and databases. In those systems types give incremental descriptions about the objects of the real world and types are used to control the search space. Types are introduced in order to remove certain properties of objects from the general purpose inference machinery, and to utilize them more efficiently.

It is not only the definition of type that matters, but the structure of type system in particular. The basic purpose of a type system is to allow us to partition the universe objects to reflect distinctions relevant to the user, the designer or both. When required partitions may be allowed to overlap.

Let me mention some basic kinds of type systems.

- In the simplest case every object has a single type and all types are mutually exclusive. Most present-day type systems in programming languages - with no abstract or user-defined data types - belong to this category. If a number is of type INTEGER it cannot also be of type REAL.
- Next we can think of a tree hierarchy. In such a case a data structure D can be of type QUEUE, but that does not preclude it from being of type BUFFER with the same first-in-first-out properties.
- The included hierarchy may be a directed acyclic graph instead of a simple tree. In this case each object can belong to more than one type, and there is no mutual exclusion enforced among types. This kind of type system is used in se-

mantic networks. For instance the object representing a horse named "KEMÁL" can belong to the types STALLION, THOROUGHBRED and STUD_HORSE.

- Finally we could permit a general graph - with cycles and potentially high connectivity - rather than a strict hierarchy.

2.2 Aggregation/decomposition

Aggregation has long been used in programming languages to express the structure of data types. Aggregation is a fundamental composition rule used to define structured types such as records and arrays, from simpler constituent types. The methodology of step-wise decomposition is a basic abstraction mechanism in software engineering.

Aggregation - on the basis of PART_OF relationships - enables a relationship to be considered a higher level aggregate object. At the level of the aggregate object specific details of its parts are suppressed. For example the aggregate "horse" as a physical object consists of parts such as head, legs etc. From an other point of view a horse could be thought of as an aggregation of its brand, date of birth and sex. Considering the aggregate object however suppresses the details of its parts.

Each instance of an aggregate object can be decomposed into instances of the component objects:

Let A be an aggregate with components $P_1, \ldots, P_n$

$$A(x) \Leftrightarrow \exists y_1, \ldots, y_n \left(P_1(x, y_1) \wedge \ldots \wedge P_n(x, y_n)\right)$$

$$x \in \{z \mid A(z)\} \Longleftrightarrow x \in \{z \mid \exists y_1 P_1(z, y_1)\} \wedge \ldots \wedge x \in \{z \mid \exists y_n P_n(z, y_n)\}$$

In the Limited Generic Database Model [BRO'80] aggregation is expressed via Cartesian product operator. This operator produces an aggregate of attributes from two or more attributes. Instances of the resulting type are tuples instead of relations resulting from the extended Cartesian product in the

relational algebra.

Let us consider an example for the application of aggregation. We have got two types, the type of mares and the type of stallions, and an operation or act involving an object of each, the operation of covering a mare (Fig. 4).

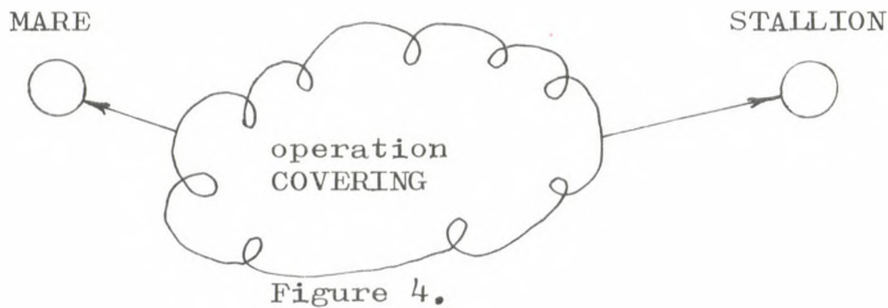


Figure 4.

For almost no database model supports considering transactions themselves as pieces of data, it is the result and not the action what is modelled. There are two ways of modelling this same thing:

- It could be considered as an aggregation relationship between the types MARE and STALLION. So another type, say REGISTRATION_OF_COVERING should be introduced, the instances of which are records of covering events (Fig. 5).
- Alternatively a datamodel with the concepts of object and attribute should be used. STALLION would be defined as an attribute of the objects of type MARE, or MARE would be defined as an attribute of objects of the type STALLION, or both (Fig. 6).

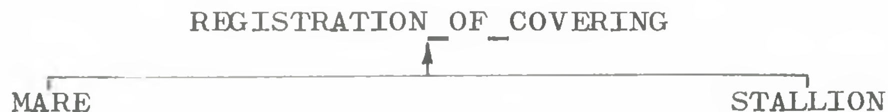


Figure 5.

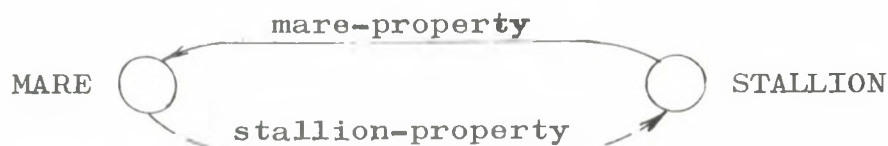


Figure 6.

In the first case individual coverings can be distinguished, but in the second case it cannot. The first solution models a fact as a data object and the second one as an operation. There are many considerations which would prefer one of our solutions. One prefers to invoke an operation and give it a couple of parameters, someone else can ask for all the instances of covering events that happened over the year and so may want to view them as entities.

On the other hand step-wise decomposition can be adapted to describe the components of a transaction too. In this view some operations applicable to the instances of a transaction have actions as their value, and the constraints specify

- the type of each of these,
- the order of invocation of components, and
- the relationship between the parameters of the transaction and its components.

For example, REGISTER could have parts (Fig. 7) generate-registration-number, record-parents and fill-in-birth-certificate. For each instantiation of REGISTER record-parents may even be wanted to mark the covering certificate as resulting a foal. This could be stated easily.

```
transaction REGISTER with
  parameters
    f: HORSE;
    m: HORSE;
  prerequisites
    Mare-with-foal-at-foot?;
    Sire-known?;
  actions
    a1: generate-registration-number;
    a2: record-parents;
    a3: fill-in-birth-certificate;
  postrequisites
    Birth-certificate-issued-for f;
end REGISTER;
```

Figure 7.

The recursive application of aggregation results in an aggregation hierarchy which has an upward inheritance property. Each property of a constituent becomes a constituent property of the aggregate. Decomposition, the inverse process produces a hierarchial breakdown. Therefore aggregation defines an organizational principle for objects.

2.3 Generalization/specification

In programming languages generalization has scarcely been used. Some similarity can be found between the concept of union types in programming languages and the concept of generalization, as both embed types into other types. Unions consist of a finite collection of explicitly identified types, which may share no operations.

Generalization relates a type to more generic ones. Generalization - on the basis of IS_A relationships - enables similar types to be thought of as a single generic type. At the level of the generic type the similarities of the constituent types are emphasized and their differences are ignored.

A generic or super type can be decomposed into its constituent types, considered as specializations or subtypes of the generic:

Let G be a generic type with subtypes $C_1, \ldots, C_n$

$$C_i(x) \Rightarrow G(x) \quad i=1,\ldots,n$$

$$\{z \mid C_i(z)\} \subset \{z \mid G(z)\} \quad i=1,\ldots,n$$

The strict interpretation of specialization only requires the introduction of additional or new operations in defining a subtype. For example a MARE is a HORSE. Objects in the type MARE (Fig. 8) will have an additional constraint stating that their sex is Mare, and a new property stating the date of last foaling or possibly their being barren.

```
data type MARE is a HORSE with
 attributes
    sex: {Mare};
    last-foaling: DATE ∪ {No,Barren} ;
end MARE;
```

Figure 8.

Similarly we can define the subtypes STALLION and GELDING.

The same applies on specializing a transaction. Additional constraints can be asserted on existing parameters and local variables and/or new parameters and local variables can be introduced with their related constraints. Additional constraints involve strengthenin too. For example registering a thoroughbred is a REGISTER for any instance of which the foal cannot be more than two-days-old, and each parent has to be of type THOROUGHBRED, the postrequisite states that the foal should be branded and there would be also an additional component, recording the brand mark (Fig. 9).

```
specialize REGISTER (f:HORSE, m:THOROUGHBRED)
  add
    prerequisites
      Not-too-old?;
      Sire-thoroughbred?;
    actions
      a4: record-brand-marks;
    postrequisites
      f  should-be-branded;
end;
```

Figure 9.

Since instances of a type must also be instances of all its supertypes, consistency in description requires a specialized transaction to cause at least the same "changes" to any state as its general counterpart. This requirement is explicitly presented in our example for pre- and post-requisites, but is harder to describe and to implement for components.

The IS_A hierarchy for constraints would be based on implication.

It is this strong sense of specialization that is present in TAXIS. However this simple intuitive notion of inheritance is insufficient to represent a substantial fraction of all the information we would like to represent. For instance, let us consider a classical problem, exemplified by the task of representing the following information in a type hierarchy:

Birds fly, penguins are birds, penguins do not fly.

This exemplifies the simplest form of non-monotonic inference in the inheritance process. Clearly, it is useful to store typical information with the type and note the few exceptions on the instances. Most birds fly, and if we did not know that penguins specially do not fly, we would have concluded otherwise. Non-monotonicity means that the addition of new information can invalidate previously valid inferences. Here the inheritance algorithm chooses the information stored lowest in the type hierarchy when conflicting information is found.

A generic type is defined by choosing a subset of the operations applicable to the type being generalized. In this context an operation may simply return a property of or a piece of the object, or the operations may change the state of the object. So the generic type HORSE may have operations to obtain an owner, a species etc. These are a subset of the operations applicable to STALLION (Fig. 10) which would also have operations to obtain and modify fertilizing capacity and other stallion properties. Obviously multiple generalizations on the same type are possible by choosing different subsets of the operations to be preserved.

Generalization as defined above is openended; if a new type having the operations required by a generalization is defined, then it automatically becomes a subtypes of the generic type.

In the Limited Generic Database Model generalization is expressed via union, intersection, difference and division set

operators and by restriction and projection as long as one key is projected [BRO'80].

The union, intersection and difference operators each produces a type with a value set being respectively the union, intersection and set difference of the value sets of the two argument object types. For example, the generic type HORSE is the union of the common properties of subtypes MARE, STALLION and GELDING; the generic type STUD_HORSE is the intersection of STALLION and USED_FOR_BREEDING (Fig. 10). The generic type HEAVY_CROSSED_BY_LIGHT is the difference of HORSE and the union of WARM_BLOODED_HORSE and COLD_BLOODED_HORSE (Fig. 10). The division operator produces a type with a value set computed by applying the relational division to the arguments based on some specified, compatible attributes.
The type restriction operator corresponds to the relational restriction.
The projection operator corresponds to the relational projection.

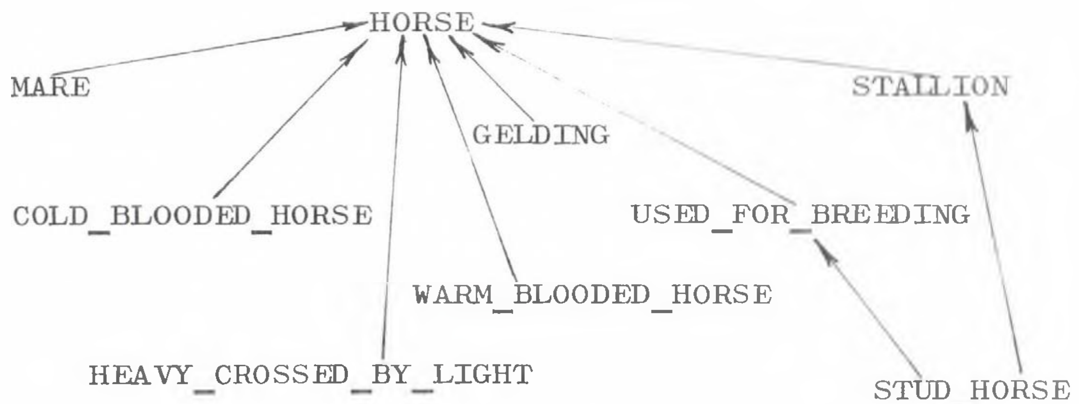


Figure 10.

Generalization can be applied repeatedly to types resulting in a partial ordering. Accordingly the generalization relationship organizes types into a generalization or IS_A hierarchy, which has a downward inheritance property. Each prop-

erty of a generic type is inherited by all its subtypes, however, a subtype may have properties distinguishing it from the other subtypes. Due to this inheritance property IS_A hierarchies have a common use in semantic networks to minimize storage requirements.

Not all approaches treat generalization in the same way. Some of them consider classification as a form of generalization saying: A class of objects or tokens can be abstracted using generalization over instances, to form a type representing the class [BRO'80].

2.4 Association/membership

Association has not been widely used in programming languages.

Association - on the basis of MEMBER_OF relationships - enables a set of similar objects to be considered a higher level set object. At the level of the set object the details of a member object are suppressed, and the properties of the set object are emphasized.

A set object type is a powerset of the member object type. Each instance of a set object can be decomposed into a set of instances of the member objects:

Let S be a set type with member type M

$$S(x) \Leftrightarrow \forall y (y \in x \Rightarrow M(y))$$

$x \in \{z \mid S(z)\} \Leftrightarrow x \in P(M)$ where $P(M)$ is the powerset over M

Association is used to group or partition a variable number of objects of a given type, for example a set of stud-horses forms a stud-farm (Fig. 11). A stud-horse might have as attributes fertilizing capacity, breeding ability and so on, whereas the set stud-farm would not have these attributes, but would have an income from stud-fees and other properties.

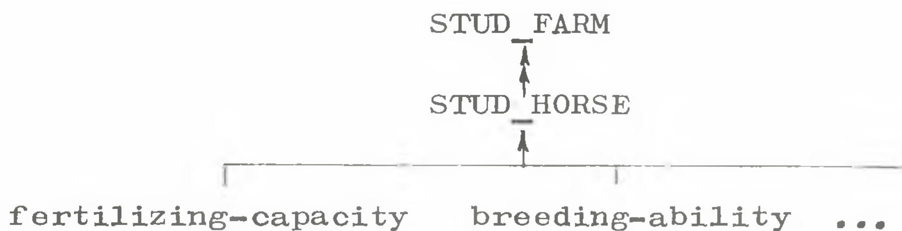


Figure 11.

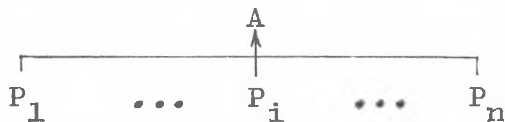
The recursive application of association results in an association hierarchy, so that if set object A is below set object B, then everything visible or present in B is also visible in A unless other wise specified. So association emphasizes set-oriented design as a special case of composition/decomposition.

3. Design tools

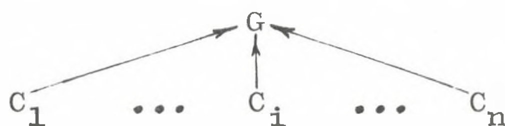
3.1 A graphical tool

Most models provide some kind of graphical design tools to represent the objects and structural relationships of a database application, like a data structure diagram or an E-R diagram, and to aid rough design of structure. SHM+ provides objects schemes for this purpose. An object scheme is a directed graph, in which the nodes are strings denoting objects and the edges identify aggregation, generalization and association relationships between objects (Fig. 12).

A is an aggregate of components $P_1,...,P_n$



G is a generic of $C_1,...,C_n$



S is a set of members from the object type M

Figure 12.

Due to the application of object schemes the rough design of structure makes use of abstraction and modularity. On designing an object detailed information about constituents can be suppressed and structural properties can be assumed through property inheritance.

Object schemes opposed to data structure diagrams, E-R diagrams and relational schemes support semantic relativism and the principles of abstraction and localization, and distinguish aggregation and association. These features are only implicitly represented in relations and in one-to-many relationships of data structure and E-R diagrams. In addition even higher order relationships can be expressed by object schemes.

Although object schemes have originally been introduced for aiding SHM+ databases, they can be used with most other data models too. SHM+ object schemes can be effectively mapped to all "classical" - hierarchic, network and relational - data models, and can be used with semantic data models supporting semantic ralativism - SHM, TAXIS, SDM, RM/T - [BRO'84].

3.2 The principle of localization

To manage the complexity of database design there is one more principle used in SHM+ in addition to the principle of abstraction. It is the principle of localization. A designer should model each property of an application object independently - localized - and then to integrate the properties to produce a complete design.

Due to the principle of localization the result of applying the three forms of abstraction to an object can be repre-

sented in one or more object schemes (Fig. 13). Additional properties of objects can be modelled by extending available or by designing independent object schemes. Combining the object schemes of all independent objects produces an object scheme containing each object and relationship.

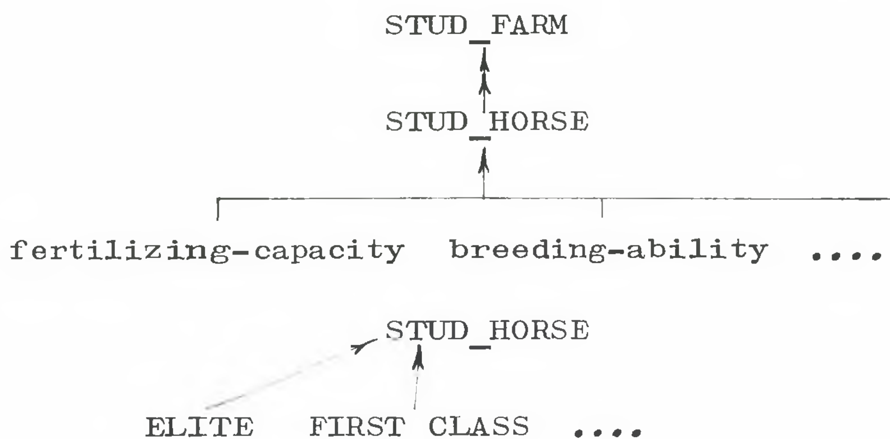


Figure 13.

4. Summary

Classification, aggregation, generalization and association are fundamental tools for most semantic data models. Classification, aggregation and generalization are being used in various forms - SHM, TAXIS, RM/T, SDM/Event Model - while on the other hand association has only currently been formalized in SHM+.

Due to the orthogonal nature of aggregation, generalization and association an object can simultaneously take part in all three kinds of hierarchies.

We have argued that data, transactions and constraints can all be structured by aggregation, classification and generalization, furthermore they can be interrelated. This can be applied among others to requirements specification and verification [BOR'81].

One of the most important advantages of abstraction mechanisms is however the fact, that there is a reasonably small set of uniform tools suitable in describing both the static and dynamic aspects of a system.

REFERENCES

BOR'81 Borgida,A.T.

Data and Activities: Exploiting Hierarchies of Classes

Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling;
Pingree Park, Colorado, June 23-26, 1980;
SIGPLAN Notices, Vol. 16, No. 1, January 1981, pp. 98-100

BRA'83 Brachman,R.J.

What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks

Computer, Vol. 16, No. 10, October 1983, pp. 30-36

BRO'80 Brodie,M.L.

The Application of Data Types to Database Semantic Integrity

Information Systems, Vol. 5, No. 4, 1980, pp. 287-296

BRO'81 Brodie,M.L.

Data Abstraction for Designing Database-Intensive Applications

Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling;
Pigree Park, Colorado, June 23-26, 1980;
SIGPLAN Notices, Vol. 16, No. 1, January 1981, pp. 101-103

BRO'82 Brodie,M.L.

Axiomatic Definitions for Data Model Semantics

Information Systems. Vol. 7, No. 2, 1982, pp. 183-197

BRO'84 Brodie,M.L.,Mylopoulos,J.,Schmidt,J.W.(eds.)

Conceptual Modelling

Springer Verlag, 1984

CAR'81 Carbonell,J.G.

Default Reasoning and Inheritance Mechanisms on Type Hierarchies

Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling;
Pingree Park, Colorado, June 23-26, 1980;
SIGPLAN Notices, Vol. 16, No. 1, January 1981, pp. 107-109

CO'79 Codd,E.F.

Extending the Database Relational Model to Capture More Meaning

ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979, pp. 397-434

EH'78 Ehring,H.,Krekowski,H.J.,Weber,H.
Algebraic Specification Shemes for Database Systems
Proceedings of the Fourth International Conference on Very Large Databases, 1978 West Berlin, September 13-15, pp. 427-440

GO'75 Goguen,J.A.,Thatcher,J.W.,Wagner,E.G.,Wright,J.B.
Abstract Data Types as Initial Algebras and Correctness of Data Representations
Proceedings of the Conference on Computer Graphics and Pattern Recognition and Data Structures, pp. 89-93

GO'77 Goguen,J.A.
Abstract Errors for Abstract Data Types
Semantics and Theory of Computation Report 6, UCLA

GO'77a Goguen,J.A.
Algebraic Specification
Semantics and Theory of Computation Report 9, UCLA

GU'80 Guttag,J.V.
Notes on Type Abstraction /Version 2/.
IEEE Transactions on Software Engineering, Vol. SE-6, January 1980, pp. 13-23

HAM'78 Hammer,M.,McLeod,D.
The Semantic Data Model: A Modelling Mechanism for Database Applications
Proceedings of ACM SIGMOD International Conference on the Management of Data, 1978 Austin TX, 31 May-2 June 1978, pp. 26-35

HAM'81 Hammer.M.,McLeod,D.
Database Description with SDM: A Semantic Database Model
ACM Transactions on Database Systems, Vol. 6, No. 3, September 1981

LI'74 Liskov,B.,Zilles,S.
Programming with Abstract Data Types
SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-59

MA'81 Mayr,H.C.
Make More of Data Types
Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling;
Pingree Park, Colorado, June 23-26, 1980;
SIGPLAN Notices, Vol. 16, No. 1, January 1981, pp. 158-160

ML'81 McLeod,D.,Smith,J.M.
Abstraction in Databases
Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling: Pingree Park, Colorado, June 23-26, 1980; SIGPLAN Notices, Vol. 16. No. 1. January 1981, pp. 19-25

MY'80 Mylopoulos,J.,Bernstein,P.A.,Wong,H.K.
A Language Facility for Designing Database-Intensive Applications
ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980

MY'83 Mylopoulos,J.,Shibahara,R.,Tsotsos,J.K.
Building Knowledge-Based Systems: The PSN Experience
Computer, Vol. 16, No. 10, October 1983, pp. 83-89

SHA'81 Shaw,M.
Abstraction, Data Types and Models for Software
Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling; Pingree Park, Colorado, June 23-26, 1980; SIGPLAN Notices, Vol. 16, No. 1, January 1981, pp. 189-191

SHI'81 Shipman,D.W.
The Functional Data Model and the Data Language DAPLEX
ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981

SM'77a Smith,J.M.,Smith,D.C.P.
Database Abstraction: Aggregation
Communications of the ACM, Vol. 20, No. 6, June 1977, pp. 405-413

SM'77b Smith,J.M.,Smith,D.C.P.
Database Abstractions: Aggregation and Generalization
ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133

TH'78 Thatcher,J.W.,Wagner,E.G.,Wright,J.B.
Data Type Specification: Parameterization and the Power of Specification Techniques
Proceedings of the SIGACT Tenth Annual Symposium on Theory of Computing, May 1978, pp. 119-132

University of Agricultural Sciences,
GÖDÖLLŐ Pf. 303,
H-2103, Hungary

Semantic Data Models: A Software Technologist's Perspective

Megan Davis and Richard Mitchell

Division of Computer Science
The Hatfield Polytechnic
PO Box 109, Hatfield, Herts AL10 9AB, UK

Abstract

The terminologies associated with programming languages and data model languages often hide similarities between the two classes of language. A number of concepts associated with semantic data model languages are described using the terminology normally associated with programming languages. Developments in programming languages are examined as a source of inspiration for the development of data models.

Introduction

Software technologists are concerned with the specification, design and implementation of software systems, and must therefore concern themselves with modelling aspects of the real world by data and associated algorithms. Database designers are concerned with the specification, design and implementation of database systems and they, too, model aspects of the real world by data and associated algorithms. The universes of discourse of the software technologist and the database designer overlap considerably, yet they have evolved different languages of discourse. This paper aims to show that the different languages frequently describe the same ideas, and is motivated by the belief that mutual understanding between workers in the two areas will benefit both.

We concentrate particularly on describing concepts associated with data models of the database world using the terminology of programming languages. Taking the software technologist's point of view, we begin by looking at a well known data model, the relational model. Then we consider some features of semantic extentions proposed for the relational model. Lastly, we consider recent develoments in programming languages, and the implications for semantic data models.

The term *semantic data model* is potentially misleading: all data models capture some of the semantics of the world being modelled. We can only properly say that one data model captures more or less semantics than another, or possibly just different semantics from another. We follow the typical usage of the term semantic data model, to denote a data model designed to capture the concept of an object, or type of object, as opposed to a data model that clearly has the concept of a record in its parentage.

The Relational Model

There is no universally agreed definition of the term relational model. There are many implementations of database systems which have relational features. There are many implementations which claim to be relational. There is the ANSI draft SQL standard (ANSI 1985). The discussion in this paper will be based on the model outlined in (Codd 1980), which lists the following three components of a data model:

objects

operators or inferencing rules which may be applied to the objects

general integrity rules which constrain the valid database states or changes of state. These are distinct from any application dependent integrity rules (such as salary cannot decrease, or no-one earns more than their manager).

We shall illustrate these components of the relational model by considering a small example, concerning the parts of a door (Figure 1). We can start to model this world of doors by the following relations:

Parts

partId	partName
p37	doorway
p4	frame
p1	door
p3	hinge
p2	mounting
p107	pivot
p9	handle
p123	screw

Assembly

majorPart	minorPart	quantity
p37	p4	1
p37	p1	1
p1	p3	2
p1	p9	1
p3	p2	2
p3	p107	1
p3	p123	8

The more general world of which the door is an example can be described by the following relational schema:

```
SCHEMA Components

    DOMAIN    PartId      STRING
    DOMAIN    PartName    STRING
    DOMAIN    Count       INTEGER


    RELATION Parts
        ATTRIBUTE    partId       DOMAIN    PartId
        ATTRIBUTE    partName     DOMAIN    PartName

    RELATION Assembly
        ATTRIBUTE    majorPart    DOMAIN    PartId
        ATTRIBUTE    minorPart    DOMAIN    PartId
        ATTRIBUTE    quantity     DOMAIN    Count

END SCHEMA Components.
```

There is a number of questions concerning such schemas that we wish to answer from a software technologist's point of view:

what does it mean when we use words such as INTEGER and STRING ?
what does it mean when we declare a DOMAIN ?
what does the identifier *PartId* identify ?
what does it mean when we define a RELATION ?

INTEGER, STRING and so on are the the pre-defined types of the language. These are typical of types in current database languages.

To answer the question of what it means to declare a DOMAIN we shall look briefly at another example which makes the point more clearly.

Compare the data model declaration

```
DOMAIN      ShoeSize     INTEGER
DOMAIN      ChildsAge    INTEGER
```

with the programming language declaration

```
TYPE
      ShoeSize     =     1 .. 13
      ChildsAge    =     1 .. 13
```

In the programming language declaration both *ShoeSize* and *ChildsAge* rename the type 1..13. In programming terms then, declaring a domain renames a pre-defined type. One question this raises is whether or not *ShoeSize* and *ChildsAge* are the same type. In the data model which we are considering, they are not; typing is by name rather than by structure. If in reality it makes little sense to compare a childs age with a shoe size, this can be reflected in the model by making it illegal to compare a value of type *ChildsAge* with a value of type *ShoeSize*.

There is a small difference between type renaming in a general purpose programming language and type renaming in the relational model; in the relational model this renaming must be done.

We now turn our attention to the declaration of relations. Compare the declaration of the relation called *Parts* with that of the following array of records :

```
RELATION Parts
      ATTRIBUTE    partId      DOMAIN   PartId
      ATTRIBUTE    partName    DOMAIN   PartName


TYPE
      Part = RECORD
                  partId      :    Partid;
                  partName    :    PartName
             END
      Parts = ARRAY [ 1 . . n ] OF Part
```

RECORD is not itself a type, unlike, say, INTEGER, but it allows types to be constructed. In this example a type called *Part* is constructed. So, RECORD is a type constructor. The operations associated with any type constructed using the RECORD type constructor are field referencing operations. Similarly ARRAY is a type constructor, constructing, in this example, a type called *Parts.*

Just as RECORD and ARRAY are type constructors in programming languages, so RELATION is a type constructor in the data model language, constructing, in this example, a type called *Parts*. The word ATTRIBUTE is simply part of the syntax of a relation declaration.

There is, in fact, more to a relation declaration than we have uncovered so far. Consider the following program declarations:

```
VAR
      w   :   Parts
      x   :   Parts
      y   :   PartId
      z   :   Integer
```

If we look for something equivalent in the data model, we see nothing which obviously corresponds, because variables are not explicitly declared in the relational model. There are two reason for this. First, there are no variables which are not relations, so there is no analogue of:

```
      y   :   PartId
      z   :   Integer
```

Secondly, there is no concept of two variables of the same type and so no analogue of:

```
      w   :   Parts
      x   :   Parts
```

In the relational model there is exactly one variable of each type. Thus declaring a relation declares both a type and a variable of that type, with one name for both:

relation declaration = type declaration + variable declaration.

The operations associated with relations constructed using the RELATION type constructor are largely based on the relational calculus or relational algebra, and vary from one data model language to another. They may include operations on relations of different types, such as the union of two relations. Operations associated with the base types, such as INTEGER, from which a relation is constructed, and type transfer operations that map between values of a relation type and values of a non-relation type are external to the relational calculus or algebra. So also are operations that acknowlege that relations are variables. Consider the following relational algebra expression:

```
JOIN
    Parts
WITH
    Assembly
OVER
    Parts.partId,  Assembly.majorPart
```

This is an expression, and the relations *Parts* and *Assembly* in it may be thought of as values, just as *a* and *b* may be thought of as values in the following program fragment:

```
PROCEDURE sum ( a, b : INTEGER ) : INTEGER ;
BEGIN
    RETURN a+b
END sum ;
```

To understand such a sub-set of a programming language we do not need to know that *a* and *b* are the names of variables, just that they denote values of type INTEGER. Similarly, in order to define queries or views in a data model language, we do not need to know that identifiers such as *Parts* and *Assembly* are names of variables. However, we also need the ability to update a working database, that is, to be able to store, modify, or delete information. In this context we are clearly dealing with variables. A statement such as

```
APPEND TO Parts
    ( partId = p11,
      partName = window )
```

in a data model language has properties in common with an assignment statement in a programming

language such as:

average := sum (a, b) / 2 ;

The relational model provides integrity rules (Date 1986) that constrain the values of such variables in the presence of updates. As applied to the *Components* example, these rules are:

Rule 1: Entity integrity: The attributes *partId*, *majorPart* and *minorPart* may not take null values.

Rule 2: Referential integrity: every value which occurs for the attribute *minorPart* and every value which occurs for the attribute *majorPart* must also occur as a value of the attribute *partId*. That is, an assembly must consist of known parts and must constitute a known part, where by known we mean known to the model. We see that in the case of referential integrity, the values of the variables are constrained with respect to each other.

In other words, the relation types of a schema are not independent; we can consider the whole schema *Components* as a single type (constructed using the pre-defined type constructor SCHEMA and consisting of the two relation types *Parts* and *Assembly*). Referential integrity can then be considered as an invariant on this type, in the same way as a type *stack* may consist of an array type and an integer cursor with an invariant constraining the position of the cursor.

To summarise, the relational model gives :

base type renaming
pre-defined type constructors, RELATION and SCHEMA
pre-defined operations on constructed types
implicit declarations of variables (one variable for each relation)
an expression sub-language based on relational calculus or algebra
a means to assign values of expressions to relation variables.

Semantic Models

In what way do semantic models differ from the relational model just described? A semantic model captures more of the meaning of the data by, in some way, allowing more meaningful abstractions to be made. We take as an example one aspect of the semantic model RM/T (Codd 1979) which builds on the relational model to provide a richer set of modelling constructs.

Consider a world of employees, some of whom are doctors, some of whom are typists, some of whom are both, and some of whom are neither. Some common information will be recorded about typists and doctors. For example, we may wish to store the name and address of each employee, or pay each employee. We may also wish to handle some different information about typists and doctors. For example, we may wish to record a doctor's specialism but a typist's speed. Doctors and typists have some attributes in common and some which are peculiar. We need the ability to define that typists and doctors are sub-sets of employee, and to define whether or not these sub-sets overlap (Codd 1979).

Informally, we want to construct tables with irregular shapes:

Name Address Specialism Speed

Such a structure could be described in an RM/T-like language as follows:

```
RELATION employee
        ATTRIBUTE       name            DOMAIN      String
        ATTRIBUTE       address         DOMAIN      String

RELATION doctor IS employee WITH
        ATTRIBUTE       specialism      DOMAIN      String

RELATION typist IS employee WITH
        ATTRIBUTE       speed           DOMAIN      Integer
```

An analogue in programming language terms could be a set of records of the following type:

```
employee =  RECORD
                name      :    STRING
                address   :    STRING

                CASE
                    doctor
                OF
                    true.  specialisation  :  STRING
                END

                CASE
                    typist
                OF
                    true.  speed           :  INTEGER
                END
            END
```

in which some form of variant or discriminated union mechanism is used. We see that again we are dealing with a type constructor.

Thus RM/T builds on the relational model and provides a richer set of modelling constructs by providing a more complex pre-defined type constructor. An analogous development in

programming languages is, for example, that Fortran provided the type constructor ARRAY whereas Pascal provided the type constructors ARRAY and RECORD.

Developments in programming

We have described the relational model in terms of a Pascal-like language. In the nature of analogies, it is not exact (for example we appreciate that a relation is more abstract than a record or an array). However, let us persue the analogy a little by considering the development of programming languages. The nature of a database world is that it is concerned with regular, persistent, shared, yet changing data. Can semantic modelling of such a world incorporate ideas from the programming community? In the development of programming languages the move has been away from defining a data structure in terms of a representation and towards defining it in terms of the the operations that can be performed on it, and implementing it so that its representation is hidden from those parts of the program that use the data structure.

We take as an example a stack of characters. The following specification defines the type stack of characters by defining the operations that may be performed on values of the type and by defining the properties of these operations. The specification in no way indicates an implementation.

```
CharStack =
        Char + Boolean +

        sorts        CharStack

        ops          nullCharStack : -> CharStack
                     push : CharStack , Char -> CharStack
                     top   : CharStack ->Char
                     empty : CharStack -> Boolean
                     delete : CharStack -> CharStack

        eqns    s : CharStack , ch  : Char :-
                     empty ( nullCharStack ) = TRUE
                     empty ( push ( s , ch ) ) = FALSE
                     top ( push ( s, ch ) ) = ch
                     delete ( push ( s , ch ) ) = s
```

The table below gives a very informal picture of developments in the world of programming, in the more mathematical world of specifications, and in the database world.

programming	specification	database
subroutines (e.g. Fortran)		
subroutines and user-defined types (e.g. Pascal)	model based specifications (e.g. VDM)	relational model and extensions (e.g. RM/T)
subroutines and modules (e.g. Modula 2)	property oriented specifications of functions and abstract data types (e.g. OBJ)	?

The question mark in the table stands for "what significance is there for the database world of the move in the programming and specification worlds towards separating abstract and concrete views of data?". We do not attempt a full answer to this question, but we propose one partial answer based on the following argument. A schema in, for example, RM/T is expressed in terms of concrete data types (constructed using type constructors such as RELATION). Similarly, the internal workings of, for example, a Modula 2 module are expressed in terms of concrete data types such as array and record types. But an abstract view of the module can be devised before the concrete details are worked out. This abstract view can be expressed using specification languages for defining functions and abstract data types. In the database world, an abstract view of a schema could be devised in terms of abstract data types. The database designer could define what sorts of

data values the database is to manipulate (for example, name and typing speed), and what operations are to be performed on these values (for example, enter new speed against given name), and later map this abstract view to a more concrete one expressed in terms of the types provided in a data model language.

Summary

Comparisons of relational database schema with programming language declarations show that the database designer is working with a small number of what a programmer would call built-in types and type constructors, with implicit declaration of variables of the defined types (one variable per relation type). A relational database language provides an expression sub-language based on relational calculus or relational algebra, together with imperative constructs to express changes of state. Semantic models such as RM/T differ from the basic relational model in that they offer a slightly wider range of type constructors.

The trend in programming towards separating the definition of the permissible operations on a data structure from the actual representation details of the structure has no immediate parallel in the relational model and its semantic extensions. It is proposed here that techniques for specifying abstract data types developed in the world of programming can find use in the database world in formulating the requirements on a database prior to devising a schema to satisfy those requirements.

References

(ANSI 1985)

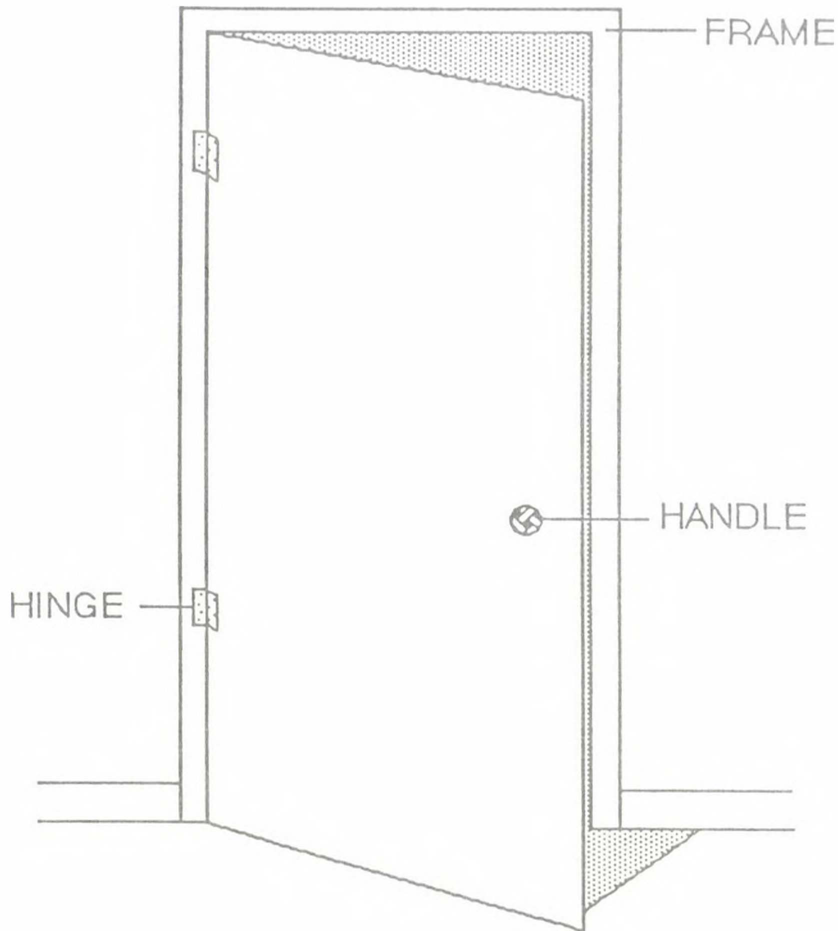
X3H2 (American National Standards Database Committee)
American National Standard Database Language SQL
Working Draft Doc X3H2-85-1, ANSI, January 1985

(Codd 1979)

E F Codd
Extending the database relational model to capture more meaning
ACM *Trans. Database Syst* . 4, December 1979, pp. 397-434.

(Codd 1980)
E F Codd
Data models in database management
Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling
ACM, June 1980, pp112-114

(Date 1986)
C J Date
An Introduction to Database Systems, Vol. 1
Addison Wesley, 4th Edition, 1986



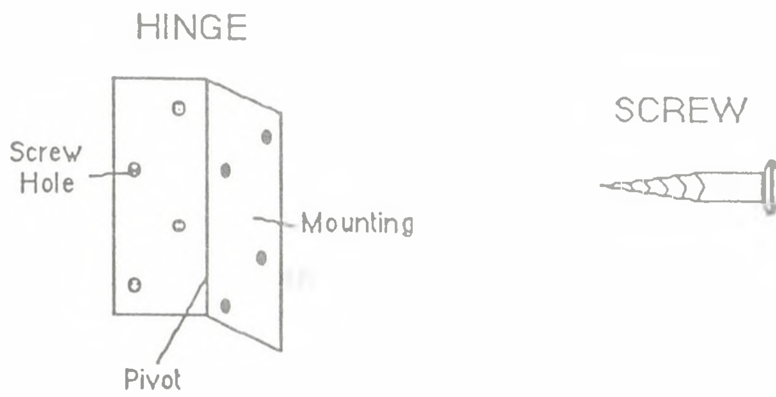


Figure 1. A door and its parts

1985-BEN MEGJELENTEK:

166/1985 Radó Péter: Információs rendszerek számitógépes tervezése

167/1985 Studies in Applied Stochastic Programming I. Szerkesztette: Prékopa András /utánnyomás/

168/1985 Böszörményi László - Kovács László - Martos Balázs - Szabó Miklós: LILIPUTH

169/1985 Horváth Mátyás: Alkatrészgyártási folyamatok automatizált tervezése

170/1985 Márkus Gábor: Algoritmus mátrix alapu logaritmus kiszámitására kriptográfiai alkalmazásokkal

171/1985 Tamás Várady: Integration of free-form surfaces into a volumetric modeller

172/1985 Reviczky János: A számitógépes grafika terület-kitöltő algoritmusai

173/1985 Kacsukné Bruckner Livia: Mozgáspálya generálás bonyolult geometriáju felületek 2 1/2D-s NC meg-munkálásához

174/1985 Bolla Marianna: Mátrixok spektrálfelbontásának és szinguláris felbontásának módszerei

175/1985 Hannák László, Radó Péter: Adatmodellek, adatbázis-filozófiák

176/1985 Számitógépes képfeldolgozási és alakfelismerési kutatók találkozója.
Szerkesztette: Csetverikov Dmitirj,
Főglein János és Solt Péter

177/1985 Gyárfás András: Problems from the world surrounding perfect graphs

178/1985 PUBLIKÁCIÓK'84
Szerkesztette: Petróczy Judit

1986-BAN EDDIG MEGJELENTEK:

179/1986 Terlaky Tamás: Egy véges criss-cross módszer és alkalmazásai

180/1986 K.N. Čimev: Separable sets of arguments of functions

181/1986 Renner Gábor: Kör approximációja a számitógépes geometriai tervezésben

182/1986 Proceedings of the Joint Bulgarian-Hungarian Workshop on "Mathematical Cybernetics and Data Processing" Scientific Station of Sofia University, Giulecica /Bulgaria/, May 6-10, 1985 /Editors: J. Denev, B. Uhrin/ Vol I

183/1986 Proceedings of the Joint Bulgarian-Hungarian Workshop on "Mathematical Cybernetics and Data Processing" Scientific Station of Sofia University, Giulečica /Bulgaria/, May 6-10, 1985 /Editors: J. Denev, B. Uhrin/ Vol II

184/1986 HO THUAN: Contribution to the theory of relational databases

185/1986 Proceedings of the 4th International Meeting of Young Computer Scientists IMICS'86 /Smolenice, 1986/ /Editors: J. Demetrovics, J. Kelemen/

186/1986 PUBLIKÁCIÓK - PUBLICATIONS 1985
Szerkesztette: Petróczy Judit

Készült az Országos Széchényi Könyvtár Sokszorosító
üzemében,Budapest.Felelős vezető: Rosta Lajosné
Példányszám: 320
Terjedelem: 16 A/5 iv
Munkaszám: 86 301