International Institute for Applied Systems Analysis
IIASA www.iiasa.ac.at

The Role of Logical Domain Models in Decision Support Systems

Lee, R.M.

IIASA Working Paper

WP-81-155

December 1981

THE ROLE OF LOGICAL DOMAIN MODELS IN DECISION SUPPORT SYSTEMS

Ronald M. Lee

December 1981
WP-81-155

ABSTRACT

Principal "content" resources of a DSS are regarded as databases, organized according to data models, and algorithms, representing decision models.

The use of *logical domain models* is here proposed as an intermediate, integrating between data models and decision models. The function is to provide a framework of *qualitative* inference providing higher level interpretations on databases, and provide a qualitative context for interpreting quantitative decision models.

CONTENTS

THE ROLE OF LOGICAL DOMAIN MODELS IN DECISION SUPPORT SYSTEMS

Ronald M. Lee

I. INTRODUCTION

A generally accepted function of a decision support system (DSS) is to provide flexible linkage of a variety of computer based resources—e.g., databases, decision algorithms, user dialogue interfaces, graphic displays, etc.--bringing these to bear on the user's problem of the moment. (See e.g. Sprague, 1980). Two commonly cited characteristics of the problems suited to a DSS rather than other technologies are irregularity and semi-structuredness. Irregular problems are ones that arise infrequently. Semi-structured problems are ones not covered by a single decision model (nor, presumably, by a single database).

In neither type of situation is the usual flow-chart approach to systems building appropriate. This approach gives to much emphasis to efficient operation and too little to rapid construction for infrequent problems. Likewise it offers too little flexibility for semi-structured problems which may require experimentation with a variety of combinations of system resources.

The development of effective decision support systems presents a variety of problems.

First there are technical, software system engineering problems in designing the various computer-based resources in such a way that they can be fitted together in arbitrary combinations, passing data and control from one to the other.

At another level, DSS design presents certain human engineering problems.These involve determining what characteristics make a DSS interface "friendly" to various user groups, and subsequently, how to design such interfaces. (see e.g., Melichar 1981 for a survey).

However, there remains still another, what may be regarded as a more theoretical, level of problems posed by decision support systems. Within the view of a DSS presented thus far, it is the user that specifies the selection and combination of the available resources for the problem at hand.

Thus, from a conventional application system where computer resources are applied in a determined combination by the program (or chain of programs), in a DSS these resources are left uncoupled.

Yet, while the DSS will not be used only for problems of a single type, it will generally be dedicated to a certain problem *domain* (e.g., production scheduling, financial planning). Within this problem domain, there are certain combinations of resources (e.g., certain sets of data fed to certain programs) that are sensible and useful, while others are not. Further, there will be certain resource combinations that are used often enough that one would like to "chunk" them into a higher level concept. Lastly and most importantly, if we are aiming at building a *support* system, rather than simply a collection of computer resources, then we need a way of re-orienting these resources into the language and concepts of the user's problem domain.

The mechanism to accomplish this integration and translation of computational resources and user problem orientation is what we call a *domain model*. This paper discusses the use of formal logic as the framework for representing such domain models. The potential value of such an approach is examined both from the practical standpoint of DSS operation as well as from the theoretical perspective of a DSS as a bridge between computational theories and applications disciplines.

II. APPROACHES IN REPRESENTING PROBLEM DOMAINS

The notion of a domain model is implicit in many computational resources. For instance, in a database system, names of files and fields of data are chosen which reflect their external interpretation. For instance a file called EMPLOYEE has fields NAME, AGE, SALARY, etc. Of course, as regards database retrieval and update, any arbitrary label would suffice. These may thus be viewed as memory aids for the programmers and other users.

However, the labeling conveys somewhat more, namely that important characteristics of each employee are his or her name, age, and salary. The file design implicitly reveals an abstraction (modeling) of the environment regarded as useful for various organizational tasks.

Obviously a program which interacts with a user will print messages and prompts which the user can understand. However, even if the program doesn't do input-output, names of variables and other data structures are often chosen which reflect their applications oriented interpretation.

These aspects are all, in a sense, domain models. That is, implicit in data file/data base structures and application program code there is a vocabulary of terminology that has generally understood references in the environment, and by these computational mechanisms, a certain interdependence between this terminology. The choice of terminology and this inter-dependence are the domain model.

However, this mode of modeling the environment has several problems from the DSS standpoint.

First, the method of describing environmental characteristics with programs and data files tends to be *ad hoc*. That is, terminology is invented and used within these computational resources to solve specific applications problems without a general coordinating view of how this fits in a larger, integrated view of the environment.*

A second problem is that this form of domain representation is mixed, and therefore confounded by, other computational considerations.

In an applications programs, the central task is generally some transformation of data. This is organized as a procedure consisting of smaller computational steps. Domain representation is therefore mixed with considerations of data types and structures, computational operations and procedural control. These are aspects specific to the program's operation which do not necessarily model aspects in the environment.

Somewhat similarly, the design of data files and databases is motivated by various data storage and retrieval factors which do not model environmental aspects. In the case of data files, these factors include access methods, physical data storage arrangements, etc. The data controlled by a data management system is presented in a more abstract form, but non-modeling considerations are still present. Even in the mathematically abstracted "relational model" of Codd (1970), the concept of data structure introduces factors in addition to that of environmental modeling.

A third problem, related to the other two, is the *accessibility* of the domain model information, by other computational resources: For the modeling aspects embedded in program code this is especially difficult. It is very hard, for instance, for a human to learn about the structure of the external environment by reading a COBOL or FORTRAN application program. It is much harder still to create other programs which can do this.

Accessibility of modeling information is somewhat less of a problem in data files and data bases. With a database, there is usually a separate specification (the database schema) which indicates the inter-relationship of data structures and hence, implicitly some modeling aspects of the environment. Also data dictionaries are sometimes provided for data file and databases. These however are usually informal explanations of how the data is defined, an not (easily) interpretable by other software.

* This is also a point of major concern in the database management literature on conceptual schemas. See for instance the survey (van Griethuysen *et al.* 1981).

In decision support systems, where resources are uncoupled, frequently modified or replaced, and combined in a variety of ways, a form of domain representation which avoided these problems would be especially valuable.

Several attractive candidates for such representation are emerging from artificial intelligence research in knowledge representation and expert systems (Elam *et al.* 1980, presents a good survey and discussion from a DSS standpoint).

An important point of debate are the advantages of so-called declarative vs. procedural representations of domain knowledge. A procedural representation is the strategy implicit in most programming languages--i.e., that knowledge is conveyed as a series of ordered steps in performing some task. In a non-procedural or declarative representation this sequencing information is left out. For instance, to take a mundane example, we could convey the concept of a chocolate layer cake by a recipe giving the procedure for how to make one. Alternatively, we could describe the concept of a cake declaratively by indicating the properties of a finished cake.

If all you want to do is make a cake, the procedure is probably the more useful representation. But if for instance the cake figures in other types of problems--e.g., how it will taste in relation to other items in the meal, how it should be stored and shipped, how it will look on the table, its dietary characteristics--a declarative description has more flexibility.

In computer based representations a similar choice exists. The procedural representations have the advantages that they are easier to formulate computationally, and are as well more efficient in computer time and storage space. Declarative representations have the corresponding disadvantage that domain modeling is typically much more difficult and, because use of these representations often involves non-deterministic search, they can be quite inefficient.

On the other hand, they have several key advantages, especially for use in DSS domain modeling. The foremost of these is that because the knowledge does not depend on a particular sequence of interpretation, they offer a great deal more flexibility in usage. In a procedure one particular path of deduction is determined. However, in a declarative representation this is left open and may thus be used for inferences in a variety of directions.

In providing flexibility of usage, this also addresses the problem of accessibility mentioned above. Further, since these are abstract formalisms, they also have the advantage of providing a robust framework for a consistent, non ad-hoc modeling of the environment while at the same time avoiding extraneous computational aspects.

For a good discussion of declarative vs. procedural representations see Winograd (1975). A survey of ongoing research in such representations is provided in Brachman *et al.* (1980). One popular type of declarative representation scheme are the so-called "semantic nets," first introduced by Quillian (1968). For a current evaluation and proposal for semantic networks, see Brachman (1978). Another popular formalism is that of "frames," introduced by Minsky, (1975).

One additional class of declarative representation frameworks are those based on formal logic. Unlike the others, these representations do not originate in artificial intelligence/computer science but nonetheless are researched there from a computational standpoint. At the level of first order predicate calculi, there now exists fairly solid computational approaches (see Nilsson (1980) for a survey), and in fact a programming language, PROLOG (= programming in logic) now exists in several implementations which is based on the first order predicate calculus. (See Coelho *et al.* 1980 for a more tutorial presentation and a bibliography as well as a survey of applications and implementations.)

It is a matter of continuing debate which of these representations is most advantageous from a computational standpoint.

However, from the standpoint of modeling, representation based on formal logics have one clear advantage: since they originate outside the field of computing, their scope of application tends to be much broader, not being biased by the limitations of current technologies. For instance, model logics, temporal logics, deontic logics, multi-valued and fuzzy logics, intensional logics, etc. do not as yet have a well developed computational interpretation.

Thus, through the use of logical representations we may concentrate on formal modeling of application domains in advance of the development of computational frameworks to support these models.

This leads us to consider what we may call DSS *theory* as opposed to DSS state of the art. As expressed in a recent DSS conference (Fick and Sprgue, Jr. 1981), the goals of DSS research are specifically application oriented, i.e., placing a priority on aiding important areas of decision making rather than on technical problems. DSS research thus focuses on decision applications for which there do not yet exist complete computational solutions, and some sort of cooperative problem solving between system and user is therefore required.

Thus, theoretically, DSS research involves a balanced contribution between computer science capabilities and application discipline requirements.

In actual fact, however, most work in decision support systems has a distinct technical bias. The research is largely case study and tool oriented and typically presents a package of computational functions and shows how these can be applied to assist in a given area of semi-structured decision making.

This is still "technology push." If we are going to succeed in arriving at a "application pull" orientation in DSS, we need ways of imagining and analyzing decision support applications that are not bounded by currently available technology.

One way of doing this is to conceive of a DSS abstractly, as a logical system whose criteria for feasibility is logical tractability rather than computability.

The role of a logical representation of a problem domain area in these cases is thus to serve as an intermediate specification between the theories of applied disciplines (such as production, finance, accounting) and computation. Through this approach, computational issues can be prioritized from an applications standpoint.

In summary, logical domain models have now been introduced from two motivating perspectives.

Insofar as the logic used is implemented computationally, the logical domain model provides a qualitative framework for integrating and orienting various computational resources (e.g., databases, algorithms) to a particular problem domain. As a declarative representation, a logical domain model provides a flexible framework that may be used for a wide variety of inferencing purposes.

However, insofar as we make use of logics not yet computational in designing domain models, this allows us an application oriented specification of a potential DSS, encouraging an "application pull" computer science research.

One other distinction is implicit here, namely, the scope of the domain in question. In the first case, where we are considering operational systems, the problem domain we have in mind is also fairly specific--i.e., confined to a functional area of a specific firm.

In the second case, where we consider logical domain modeling as a theoretical tool for describing abstract DSS's in advance of current technology, the domain we generally associate to this is likewise much more broadly defined--e.g., logical representation of production control processes, financial transactions, accounting theories, etc.

However, one additional merit of a logical representation scheme is, by the addition of increasingly more specific axioms, it allows a smooth transition from broad based theories to specific application contexts. The approach thus has the potential to serve not only as a bridge between application and technology but also as a bridge from theory to actual practice.

III. LOGICAL DOMAIN MODELING

A. Brief Tutorial

So far we have discussed the concept of logical domain modeling only in broad terms, without mentioning specifics. In this section we present a brief and somewhat superficial sketch of logical domain representation for decision support systems. The presentation here is tutorial, and does not presume background in formal logic. However, it is not meant as an introduction to logic in general, but rather highlights certain aspects of logic particularly relevant to DSS domain modeling.

The concepts and syntax presented here are part of a larger logical modeling formalism called CANDID, originally developed in Lee (1980).

1. *Propositional Logic*

The simplest level of logics is the *propositional logic*. Here a vocabulary is introduced, called the *universe of discourse*, consisting of complete statement's (propositions), corresponding to declarative sentences in English or some other natural language sentences. We denote these as single capital letters.

For instance,

A = the world is flat

B = two plus two is four

C = three plus three is six

D = Ronald Reagan is bald.

We assume, without defense, that A and D are false and B and C are true.

Compound propositions can be formed from these elementary ones using the logical connectives:

~	not	(negation)
&	and	(conjunction)
V	or	(disjunction)
→	implies	(implication)
↔	if and only if	(bi-conditional)

Here we have used an English gloss (interpretation) of the logical symbol. This is useful to convey a general intuition as to how these symbols are used, but it is important to emphasize that the correspondence is not exact. For instance, using the previous example propositions, (B V C) is true though its gloss "two plus two is four or three plus three is six" sounds false in natural language. This is because the logical symbol is an inclusive or while the English "or" is usually used exclusively. Likewise (A → D) is logically true though its gloss "the world is flat implies Ronald Reagan is bald" would be considered false by most people.

However, the logical connectives have the advantage that they offer a precise interpretation of the truth of compound propositions given the truth values of its elements. This is given by the so-called "truth tables" illustrated in Exhibit 1. For two arbitrary propositions, P and Q, the table shows the truth value of their various compounds depending on the truth values of P and Q.

P	Q	~P	P & Q	P V Q	P → Q	P ↔ Q
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Exhibit 1.

It should be noted that compound propositions, composed of elementary ones, can be used in forming still higher level compounds. In doing this, it is often convenient to add further proposition symbols to represent these compounds. To indicate this, we introduce another symbol, "$\Longleftrightarrow$," as a *defining* bi-conditional. This acts just as the ordinary bi-conditional ("$\leftrightarrow$"), but conveys the additional notion of definition--i.e., that its left hand term is derived from the more primitive propositions in the right hand expression. For instance, we may define a new proposition, E, as

$$E \Longleftrightarrow A \vee B \vee C$$

which has the logical force of

$$E \leftrightarrow A \vee B \vee C$$

plus that of definition. Note however that this notion of definition is imposed on and not explicitly controlled by the logical syntax.

2. Predicate Logic

The next level of logical sophistication is what is called a *predicate calculus*. Here, propositions are decomposed to distinguish individual objects which the statements are about and the properties and relationships that are asserted of these objects. To distinguish the individual objects under discussion, we adopt an internal naming scheme consisting of the character "@" followed by one or more lower case letters--e.g., @a, @ron, @bill. In a computer implementation these would be internally generated identifiers. It is assumed that each such logical name corresponds to one unique object (though an object may have multiple logical names).

The concept of a universe of discourse is somewhat different at this predicate calculus level. It amounts to specifying the set of individuals considered in the logical discussion; for instance the set of living people, the set of people at this conference, the set of integer numbers, etc. This universe is designated informally, i.e., in English or some other natural language, and it is presumed that all parties involved in using the logical application agree about what is contained in the universe.

Once this universe has been defined and a naming scheme adapted for the individuals in it, further properties and relationships on these individuals are ascribed by means of *predicates*. A predicate will be denoted as one or more capital letters followed by an argument list.

For illustration, suppose a universe consisting of a small group of people named as follows:

U = { @bill, @george, @sue, @mary }

A useful property to identify within this universe might be the predicate FEMALE, e.g.,

FEMALE(@mary)

FEMALE(@sue)

A multi-place predicate is used to indicate a relationship between more than one individual, e.g., that two are married:

MARRIED(@bill, @mary)
MARRIED(@george, @sue)

The pairs, or more generally, the tuples satisfying a given multi-place predicate describe a *relation* (in the mathematical sense). A restricted case of this is a *function*, where one of the arguments is uniquely determined when the others are specified. An alternative notation is used to indicate this, which uses the notation as for an individual, but followed by an argument list. For instance, suppose we have the predicate FATHER and

FATHER(@bill,@sue)

then we might refer to bill indirectly as the father of sue as follows:

@father(@sue)

In addition, let us adopt a new predicate, "=", for equality. Unlike the other predicates which have an argument list, it is more common to use "=" in an infix notation. Thus, to indicate that bill is sue's father; we write

@father(@sue) = @bill.

This is equivalent to the previous predicate,

@father(@sue) = @bill $\leftrightarrow$ FATHER(@bill, @sue).

However, since the result of this function is a reference to a unique individual, such functions can appear as arguments to other predicates, e.g.,

MARRIED(@father(@sue),@mary).

Likewise, we may sometimes use one function in the definition of others. For this we use the notation ":=" to indicate a "defining equality." Like the defining bi-conditional used earlier, this carries the logical connotation of equality, plus the extra logical notion of definition. For instance, the function "grandather on father's side" can be defined as father of father:

grandfather-on-father-side(x):= father(father(x)).

Predicates applied to individuals constitute propositions in the sense explained earlier. We may therefore use the same truth connectives to construct more complex statements. For instance,

FEMALE(@mary) & ~ FEMALE(@bill) & MARRIED(@bill, @mary).

We need however, one more construct to obtain a reasonable level of descriptive power in this calculus: that of a logical *variable*. We will denote logical variables as single lower case letters optionally followed by an integer subscript, e.g., x, y, z_1, z_2, z_3, etc.

Syntactically, these may appear in any context where a logical constant appears. However, one must in addition indicate the range of these variables, i.e., how many individuals in the range of these variables, i.e., how many individuals in the universe may potentially be represented by the variable.

The typical way of doing this is by using the symbols, $\forall$and $\exists$, called respectively the universal and existential quantifiers. To illustrate their use, suppose P is an arbitrary one place predicate. Then

$$(\forall x)P(x)$$

asserts P is true of all individuals in the universe, whereas

$$(\exists x)P(x)$$

asserts P to be true of at least one (but possibly more) individuals in the universe.

Recall from the earlier discussion of propositional logic that we introduced the symbol "$\Leftrightarrow$"to indicate the definition of higher level propositions. With the notion of a logical variable and quantifier, we are now able to correspondingly define higher level predicates. For instance, suppose we adopt the following three predicates as primative:

$$\begin{gathered}\mathrm{FEMALE}(x)\\ \mathrm{MARRIED}(s,y)\\ \mathrm{PARENT}(x,y)\end{gathered}$$

We may then for instance define a male as a non-female:

$$\forall x\ \mathrm{MALE}(x) \Leftrightarrow {\sim}\mathrm{FEMALE}(x)$$

a father as a male parent

$$(\forall x)(\forall y)\ \mathrm{FATHER}(x,y) \Leftrightarrow \mathrm{PARENT}(x,y)\ \&\ \mathrm{MALE}(x)$$

a child as the converse relationship to parent:

$$(\forall x)(\forall y)\ \mathrm{CHILD}(x,y) \Leftrightarrow \mathrm{PARENT}(y,x)$$

a husband as a male individual married to someone

$$(\forall x)\ \mathrm{HUSBAND}(x) \Leftrightarrow \mathrm{MALE}(x)\ \&\ (\exists y)\ \mathrm{MARRIED}(x,y)$$

A sibling relationship as having common parents:

$$\begin{gathered}(\forall x\ \forall y)\ \mathrm{SIBLING}(x,y) \Leftrightarrow \exists u\ \exists v\ \mathrm{PARENT}(u,x)\ \&\ \mathrm{PARENT}(u,y)\ \&\\ \mathrm{PARENT}(v,x)\ \&\ \mathrm{PARENT}(v,y)\end{gathered}$$

A grandparent relationship:

$$(\forall x)(\forall y)\ \mathrm{GRANDPARENT}(x,y) \Longleftrightarrow (\exists z)\ \mathrm{PARENT}(x,z)\ \&\ \mathrm{PARENT}(z,y)$$

In similar fashion such other familial relationships as son, daughter, brother, sister, aunt, uncle, niece, grandson, etc. etc. can be defined. We thus see that a very rich vocabulary can be defined from a very restricted set of primitive qualities.

B. Logical Representation of Data Objects

In the predicate calculus illustrated thus far, the individual objects represented by the logical variables and constants were individuals *in the environment*. By contrast, the basic objects in a computer system are *data objects*. This distinction is fundamental to the role of a logical data model: to describe how data objects correspond to external objects and their properties.

To elaborate this, we need to extend the predicate calculus to recognize not one universe of discourse, but two. This involves what is called a multi-sorted logic. The first universe, as before, consists of objects in the problem domain environment. The second consists of data objects as they are generally regarded in programming languages and file and database management systems. Following the usual views, we assume elementary data objects to be of two basic types: character strings and numbers. Numbers are often further distinguished between reals and integers, but we can ignore that for our purposes here.

To represent these in the calculus, we need some additional notation. Character string constants are denoted as a list of letters, digits or other punctuation between double quotes—e.g.,

"this is a character string"

Numeric constants are denoted, as usual as Arabic digits, with or without a decimal point—e.g., 1, 2.0, 3.5, .315.

Variables for character strings and numbers are denoted as for external individuals, i.e., as a lower case letter followed by zero or more lower case letters or digits.

A numeric function is one which *results* in a number, though its arguments may or may not be numeric. Similarly, a character string function is one which results in a character string.

Functions once again will be denoted by a lower case name followed by an argument list.

However as we are now discussing several universes at once we will introduce an (extra logical) convention in this naming scheme as a visual reminder: numeric variable and function names will have a "#" appended, whereas character variable and function names have a "$" appended.

For notational convenience we will represent the standard arithmetic functions by their usual infix notation: +, -, *, /.

Data objects, however, are often collected into structured collections, called data structures. We therefore need to introduce some simple structuring devices in our calculus. While used mainly to describe collections of data, these apply as well to collections of external objects.

A *set* is an unordered collection of objects. A set constant can be defined in two ways--extensionally, as a list of individuals between brackets, e.g.,

{ @a, @b, @c }

or intensionally, by indicating a predicate which defines the criterion of membership, e.g.,

{x | P(x) }

read the set of all x (individuals in either universe) satisfying the predicate P.

Here P is a single place predicate. A set defined intensionally by means of a multi-place predicate is called a *relation*. Here, the set has an internal structure corresponding to the place positions in the predicate. Each member of a relation is called a tuple and is indicated within angle brackets: e.g., for a two place predicate Q, its corresponding relation is denoted:

{<x,y> | Q(x,y)}.

Relations can as well be defined extensionally by listing the tuples--e.g.,

{<@a,@b>, <@c,@d>, <@e,@f>}.

As proposed by Codd (1970), a relation is a useful way of abstracting the structure of a database. We will therefore adopt this as a simplified view of data files and databases. Codd's notation for a relational database is the relation name followed by a parenthetical list of "attributes" corresponding to tuple positions. For instance, for a file of employee data

EMP(ID, NAME, SALARY).

In the notation here, this becomes:

@emp = {<id#, name$, salary#>}

Here we adapt the convention of identifying the relation itself as a logical individual.

It should be observed that this is a relation of *data values*, not of logical individuals. Furthermore, it is a relation defined extensionally, by the listing of data tuples. For instance, @emp might appear as

@emp = {<12, "SMITH", 30000><25, "JONES", 25000><18, "ABLE", 40000>}

The expression <id#, name$, salary#> is thus a tuple of variables that range over the tuple constants in the database.

Elements of a database relation are therefore data tuples. To refer to a data element within a tuple we use a dot notation analogous to that in Codds (1971) relational calculus, e.g.,

$$\exists x\ (x \in @emp)\ \&\ (x.salary\# = 30000)$$

While this gives a fairly adequate logical characterization of the database resources in a DSS, we need also to characterize the data structure commonly used in various types of analysis programs.

Here, two faily common structures are vectors and matrices. (Other structures are also used but these are at least representative.)

These are basically mathematical concepts, though we will here use these terms in a more general sense, applying to any type of individual.

A *vector* is, logically speaking, a linear ordering of individuals. This ordering is indicated by an integer index. We may thus describe a vector as a two place relation associating some set of individuals with the positive integers. Denoting the integers by the predicate I, and the vector association as V, a vector therefore has the form:

$$\{<i,x> \mid I(i)\ \&\ V(i,x)\}$$

Similarly, a matrix is a two dimensional ordering of individuals, i.e., each individual is associated with two integers. This is therefore described logically as a three place relation, e.g., for a matrix predicate M,

$$\{<i,j,x> \mid I(i)\ \&\ I(j)\ \&\ M(i,j,x)\}$$

By a similar method, higher order structures can be defined, e.g., three or four dimensional arrays.

For convenience, let us abbreviate a vector, v, of length n as

$$v[n]$$

and a matrix, w, of order m by n as

$$w[mxn]$$

Using these logical structuring devices as applied to the universe of data objects, most types of quantitative analysis programs can be described as functional transformations from one or more (possibly elementary) structures to another.

For instance the function, average (avg#) maps from a set of numbers to a scalar:

$$avg\#(\{x\}) = y$$

A correlation routine (corr#) maps a two place numeric relation to a scalar correlation coefficient:

$$corr\#(\{<x,y>\}) = z$$

A multiple regression routine (mv#) maps a dependent variable vector and a matrix (independent variables) to another vector of beta coefficients:

mr#(dep[mx1],ind[mxn]) = beta[nx1]

A linear programming (lp#) routine maps two vectors, and a matrix to another vector

lp#(c[1xn],b[mx1],a[mxn]) = x[1xn]

C. From Data Objects to Domain Description

Earlier, it was indicated how the properties and relationships of objects in the DSS problem domain are represented in the predicate calculus. Next, by including a universe of data objects and adding certain structuring devices to the calculus, we indicated how data structures and transformations thereon could be represented logically.

In this section we indicate how these two universes are related.

As discussed so far, we might classify the problem oriented resources of a DSS as *databases*, which provide data or facts about the environment, and *analytic routines*, which provide specific inferences on certain, usually quantitative data. The purpose of this paper has been to propose a third class of resource, a *logical domain model*. From a modeling perspective the functions of these three types of these three types of resource in a DSS are as follows:

a) a database provides *facts* about the existence of objects in the problem environment and their elementary properties and relationships.

b) an analysis routine provides a particular type of inference from an given set of input facts. The input and output of these routines is in most cases quantitative.

c) a *logical domain* model describes qualitative inter-relationships in the problem domain. It provides the apparatus for defining higher level qualitative concepts from those used in the database, and provides a qualitative context for the quantitative inferences performed by analysis routines. As proposed here, the logical domain model also serves as point of contact for user interface languages and display routines. These translate the logical notation of the logical model into a form more understandable to the user. This integrating role is diagramed in Exhibit 2.

The purpose of a logical domain model is to translate from the data orientations of databases and analytic models to those of the problem domain. We now examine how this is done.

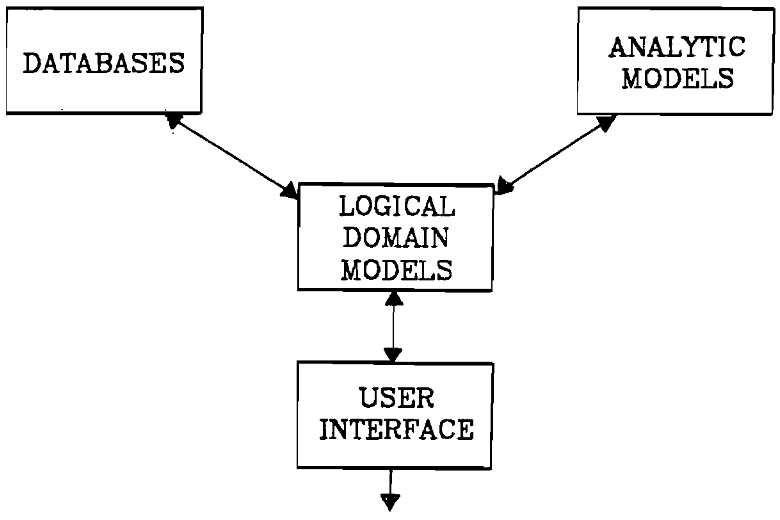

Exhibit 2.

1. Databases

The first problem is how the logical domain model is to "know" about the existence of individuals in the environment. This information is provided by the database, however not explicitly.

For instance, tuples in a database relation are often meant to correspond to distinct individuals in the environment, however this is not always the case nor is the interpretation explicit in the database structure. For example, using the earlier notation we might have the relations

@emp = {<id#,name$,salary#>}
@married = {<husband-id#,wife-id#>}
@part = {<part-id#,qty#>}

In the relation @emp, each tuple corresponds to a specific person in the environment. In the @married relation, each tuple represents a husband-wife pair, whereas in the @part relation, each tuple represents a set of parts whose cardinality is given by qty#.

Essentially, what is required is a series of logical assertions describing the "existential claims" of each database relation, i.e., what the presence of a tuple indicates about the presence of individuals in the environment. For the previous three relations, these assertions would be as follows:

$$\forall x\ (x \in \text{@person}) \rightarrow \exists y\ \text{PERSON}(y)$$
$$\forall x\ (x \in \text{@married}) \rightarrow \exists y\ \exists z\ \text{MARRIED}(y,z)$$
$$\forall x\ (x \in \text{@part}) \rightarrow \exists y\ y = \{z \mid \text{PART}(z)\}$$

Note that in their logical interpretation, database relation names take on the role of predicates on these logical individuals.

The next problem is how the character and numeric data in the database convey logical properties about these individuals.

a. Character data

Character strings seem to have two general uses in characterizing static problem domains. In one use they convey *qualitative properties* —e.g., colors, shapes of objects, organizational rank, skills of people. These translate in a straightforward way to predicate names, as described above. The other major use of character strings is a *labels*. These do not indicate properties of the object, but rather are used by members of the organizational environment to identify these objects (and one another). Examples of labels for people are first and last names, social security numbers, similarly vehicles have license and serial numbers, companies have names and SEC number etc.

Numeric data is often used in these same two ways as well. For instance, some labels consist entirely of numeric digits (e.g., a part code), though these are used essentially as character strings, without any associated arithmetic operations.

Likewise numbers are often used as classification codes, i.e., abbreviations for the names of predicates, expressing qualitative properties and relationships. For instance

MARITAL STATUS
0 = single
1 = married
2 = divorced

In this case, too, the numbers are used basically as other alphabetic symbols, without any associated arithmetic. Sometimes, these two uses are combined as "meaningful" identification codes, e.g., where the first two digits of a product code indicate its generic category, the second two a subcategory, and the remaining digits are an arbitrary assignment to distinguish the individual within this sub-class.

The other major uses of numeric data in static domains is to express *measurement*.

Measurement is a scaling of the intensity or magnitude of some feature on a number line. These scales are distinguished by the inferences that can be drawn. A *nominal* scale corresponds to a simple classification scheme--i.e., numbers are essentially abbreviations for predicate names as was just described. With an *ordinal* scale, objects are ordered along the line but intervals and absolute position are not meaningful. On an *interval* scale, inferences comparing interval lengths are allowed, but not between absolute magnitude (e.g., temparature--20°C is not twice as warm as 10°C, but a rise of 20° is twice as much as a rise of 10°). Lastly, in a *ratio* scale, inferences comparing absolute position are also permitted. Here, when speaking of measurement a ratio scale is generally implied. Logical formalization of the other scales is however straightforward.

A measurement typically involves a measurement dimension, indicating the property being scaled (e.g., height, weight) plus a unit of measure, indicating calibration of the scale (e.g., centimeters, kilograms).

One type of measurement is of course volume measurement, e.g., barrels of oil in a ship, bushels of wheat in a baxcar. Maintaining our assumption that the objects in the logic are discretely identifiable, we assume that such liquid or granular materials are contained in a discretely identifiable container.

A special case of volume measurement is cardinality, indicating the count of elementary objects in a set, e.g., the number of bolts in a bin. In this case, the unit of measure is one such elementary object.

To summarize, as a tentative taxonomy of the uses of character and numeric data to describe static domains, we have

character strings	used for	labels
character strings	used for	predicate names
numbers	used for	labels
numbers	used for	predicate names
numbers	used for	measurement

For more detailed illustration, let us consider a more elaborate form of the PERSON file, with the following rows of data:

PERSON (LAST-NAME,	ID,	SEX,	MARITAL-STATUS,	SALARY,	HEIGHT,	WEIGHT)
SMITH	12	M	0	25	1.5	80
JONES	27	F	1	50	1.55	52
ADAMS	52	F	2	20	1.7	65

Since each row of the file corresponds to an individual person, we may assign logical names to these individuals by placing arbitrary labels on each row, e.g., @a for row 1, @b for row 2, @c for row 3.

The each column of the file can be regarded as a functional mapping from each of these individuals to a character string or number, e.g,

last-name$(@a) = "SMITH"
id#(@a) = 12
sex$(@a) = "M"
marital-status#(@a) = 0

etc.

In this form, each of these functional maps serves merely in the role of a label, i.e., it associates a character string or number to the individual, but allows no further inferences.

However, as discussed above, certain of these mappings may be reinterpreted as predicates, allowing further logical inference to be described, whereas others may be interpreted as measurements, enabling us to do certain quantitative inferences.

Note that last-name$ and id# (identification number) are data specifically used for (external) identification purposes. These have little further value for inferencing.

However the next two, sex$ and marital-status#, do indicate qualitative properties that may be used in further deductions. We therefore indicate their re-interpretation as a predicate in the following way:

$$(\forall x)\text{sex\$}(x) = \text{"M"} \leftrightarrow \text{MALE}(x)$$
$$(\forall x)\text{sex\$}(x) = \text{"F"} \leftrightarrow \text{FEMALE}(x)$$
$$(\forall x)\text{marital-status\#}(x) = 0 \leftrightarrow \text{SINGLE}(x)$$
$$(\forall x)\text{marital-status\#}(x) = 1 \leftrightarrow \text{MARRIED}(x)$$
$$(\forall x)\text{marital-status\#}(x) = 2 \leftrightarrow \text{DIVORCED}(x)$$

The interdependence of the marital status predicates can be indicated as follows:

$$(\forall x)\ \sim\text{MARRIED}(x) \leftrightarrow \text{SINGLE}(x) \vee \text{DIVORCED}(x)$$

Further predicates can likewise be defined. For instance, we may want to indicate that two people are candidates for marriage:

$$(\forall x)(\forall y)\text{CAN-MARRY}(x,y)$$
$$\Longleftrightarrow \text{MALE}(x)\ \&\ \text{FEMALE}(y)\ \&\ \sim\text{MARRIED}(x)\ \&\ \sim\text{MARRIED}(y).$$

The function salary# is implicitly an annual salary in thousands of dollars. We can make this unit conversion specific by converting to to a measurement function:

$$\text{year-salary\#}(x,@\text{dollar}) := 1000 * \text{salary\#}(x)$$

We can likewise define a monthly salary as:

$$\text{month-salary\#}(x,\text{unit}) := \text{year-salary\#}(x,\text{unit})/12$$

Note that this relationship is true regardless of the currency, the unit of measure has been left as a variable.

2. *Interpretation of Analytic Models*

As observed, most analytic models do transformations on quantitative data. For simplicity, we consider only this kind. In these cases, the numbers input to and output from the model are interpreted logically as measures, i.e., as a numeric scaling of some qualitative feature. Thus the quantitative transformations are always regarded as transformation of measure functions.

A very simple example are the arithmetic functions, +, -, *, /, as used in the preceding examples.

As further examples, these can be used to define transformations of measurement units, e.g.,

$$\forall x\ \text{LENGTH\#}(x,@\text{cm}) = 2.54 * \text{LENGTH\#}(x,@\text{in})$$

Higher level measurements can also be defined, for instance the area of a rectangle

$$\forall x \; RECTANGLUAR(x) \rightarrow AREA\#(x,unit1,unit2) := LENGTH\#(x,unit1) * WIDTH(x,unit2)$$

(Note that area involves a double measurement unit.)

However, most interesting analytic models involves features not of single objects, but of collections. For instance, the function average is a property of a set of objects. Consider, e.g., the average height of all males in the population.

$$n = avg\#(\{\, ht\#(x) \mid MALE(x)\})$$

A correlation routine, corr#, takes as an argument a two place relation. Suppose we want to find the correlation between the heights of married couples. This is expressed

$$n\# = corr\#(\{< ht\#(x), ht\#(y)> \mid MARRIED(x,y)\})$$

In these cases, the result of the analysis routine was a scaler number. To illustrate a case where the result is a tuple, suppose we want a multiple regression (mr#) of weight of people as a function of height and age.

$$<a\#,b\#,c\#> = mr\#\{<wt\#(x),ht\#(x),age\#(x)> \mid PERSON(x)\}$$

To illustrate a function whose result is a set, consider the function, sort which orders a relation

$$\{<i,x,y,z>\} = sort\{<x,y,z>\}$$

D. System Operation: Asking Questions

Logical representations, such as we have shown, are generally oriented towards making assertions--i.e., expressing facts or generalizations. What is missing in this from the standpoint of DSS operation is the ability to express questions about the problem domain.

Here we consider two general types of questions--"true/false" questions and "which" questions.

A true/false question has the general form "Is it the case that P?," where P is an assertion. Possible responses are true (T), false(F) or don't know(?). Logically, the question is simply an assertion which is to be compared for compatibility with existing assertions. We therefore need to distinguish between assertions taken as axiomatic and others taken as goals to be proven. For this we use the simple device of a question mark preceding the assertion, e.g.,

?P

is read "Is it the case that P?."

'Which questions ask for a list of objects which may be either logical individuals, character strings or numbers. One way to do this is to specify a function which selects the objects in question. Recall that functions are always used in assertions as an argument to a predicate, typically as one of the arguments to the predicate "=". Outside of this context they simply result in an individual--which was the goal of the which question. For instance, the question "who is Mary's father?" is given by:

father(@mary).

This results in a logical constant name, which may be printed out directly or passed to a user interface. However, logical constants are used here as internal identifiers. Normally a user interface would want to refer to external identifiers--e.g.,

last-name$(father(@mary))

returns Mary's father's last name.

More often, however, we are interested in identifying a set of individuals, for instance the set of females:

{x | FEMALE(x)}

Again this would return their internal logical names. To obtain their last names, we specify

{last-name$(x) | FEMALE(x)}

or if we want first and last-name pairs, we write:

{<first-name$(x),last-name$(x)> | FEMALE(x)}

To identify all married couples, we specify:

{<x,y> | MARRIED(x,y)}

to request their names and ages we write

{<name$(x),name$(y),age#(x),age(y)> | MARRIED(x,y)}

As is evident, this is constructing a data relation of an intensionally identified set of individuals. Note that this method of querying is "data structure independent"--i.e., the retrieval does not depend on the structure of the database retations from which the data originally came, but rather on the association of this data to logical individuals.

Other types of analytic routines, regarded as functions, can likewise be specified directly. For instance, to obtain the correlation of height and weight of males we specify

corr#{<height#(x),weight#(x)> | MALE(x)}

to get the correlation in heights of married couples we specify:

corr#{<height#(x),height#(y) | MARRIED(x,y)}

IV. FURTHER ISSUES FOR LOGICAL MODELING

As mentioned, the logical syntax presented here is part of a larger formalism called CANDID, first developed in Lee, (1980). Listed here are some additional logical modeling issues addressed in that work.

A. Representation of Aggregate Objects

The predicate calculus, as we have described it is commonly called "first order," that is the logical individuals within it are presumed to be elementary, with no internal structure. We commented earlier that a bin or package of smaller objects could nonetheless be indicated provided the bin is considered as an individual in the calculus and its cardinality regarded just as any other numeric measurement. This is probably satisfactory for describing static views of inventories, but leads to problems when we wish to refer to properties and relationships within the aggregate--e.g., we may wish to refer to the collection of employees associated with a department, yet make assertions about separate employees within this collection as well.

The aversion for mixing assertions about individuals, sets, sets of sets, etc. in the same calculus is due originally to Russell, and has been commonly accepted within logic. More recently, Goodman (1977) has proposed an alternave, more natural framework he calls the "calculus of individuals" that treats elementary and collections of individuals all within the same first order framework. Adopting this view we may write expressions like

@a = {@b,@c,@d}

or

@a = {x | P(x)}

where the brackets are now re-interpreted from designating sets to being what Goodman calls a "summation" of these individuals. Interestingly, within this calculus the two concepts of "element of" and "subset of" are discarded in favor of a single alternative, "part of."

In the calculus presented in this paper, we have followed Goodman's suggestion as far as representing non-elementary objects (sets, relations) as other logical individuals. To keep the discussion based on familiar concepts, we did not however include his part-of and logical summation constructs. These are however useful in modeling objects constructed out of others. For instance--it is common in production to have parts combined to form sub-assemblies which in turn combine to form larger assemblies, etc. In a theory of sets these are each increasingly higher order sets whereas in the theory of individuals these are all objects of the same ontological status, which is more natural from the modeling standpoint.

Thus a bolt which is part of a carburetor is as well part of the engine and part of the car itself. In a theory of sets the car would thus be a third order set, whereas in th calculus of individuals, it is merely another first order object.

B. Representation of Time

Earlier, we limited our discussion to databases containing only current facts about the environment. An obvious extension is to consider databases containing not only current but historical data. Logical interpretation of these databases involves so-called temporal logic. One formulation, proposed by Rescher and Urquhart (1971), introduces the concept of "realization"--i.e., the time in which a particular assertion is true. For instance, if S(@vienna) is the assertion "there is snow in Vienna," then

(R t) S(@vienna)

expresses the assertion that it is realized at time t that there is snow in Vienna. When time is conceptualized as a series of discrete intervals, e.g., days, then this reduces to a predicate calculus in a straight forward way, by adding extra predicate places, e.g.,

S(@vienna, t)

or

S(@vienna, t0, t1)

where t0 and t1 are the starting and ending days of the interval. When time is conceptualized as a continuous dimension (the more common interpretation), certain additional complications arise. This problem is analogous to that of dealing with liquid objects--the predicate calculus is basically oriented towards objects that can be discretely named.

The method of dealing with it is also analogous. One may identify liquids, grains, etc. by reference to individual containers that holds them. Continuous time can also be treated by reference to individual time spans--e.g., the days, months, years named by the Gregorian calendar.

C. Change and Process

A concept related to realization is that of *change*. Logically this can be reduced to the changing in truth value of a predicate or proposition.

A useful notation is that of von Wright (1965):

(P1 T P2)

read P1 "and then" P2, where P1 and P2 are propositions. The advantage of this construct is to be able to refer to changes *generically*, without specifying when they occurred. For instance, if we assume a predicate OWN(x,y) indicating that x owns object y, we can then define a change in ownership, from x to z as:

OWN(x,y) T OWN(z,y)

A specific change of ownership is described by identifying the particular individuals involved and when it occurred, e.g., for Bill giving Tom a book in July 1, 1975

(R @july-1-75) OWN(@bill,@book) T OWN(@tom,@book).

A change is usually regarded as instantaneous or taking place during the shortest interval on the time scale. A *process*, by contrast, is a change of longer duration. Often in organizational environments we are concerned with production processes. This is described formally simply by replacing the elementary time of realization by a longer interval. An interesting aspect here is that, typically, the process does not merely transform the object, but rather several input objects are absorbed (destroyed) by the process while others are created. This necessitates the predicate of "temporal existence," say E(x).

Destruction and creation are thus expressed, respectively:

E(x) T ~E(x)
~E(x) T E(x)

D. Action and Responsibility

An action is a change "caused" by somebody. Von Wright (1967) formalizes this as a change that would not otherwise have occurred without the intercession of this agent. In combination with the previous T operator, this is denoted as

P1 T (P2 I P3)

read P1 and then P2 "instead of" P3. The identity of the intervening agent is however not indicated in this notation. Suppose we add a third place in the I connective, i.e.,

P1 T (P2 Ix P3)

We now convey the notion that partly x is *responsible* for the occurrence of state P2. This notion of responsibility is an extremely important one from the standpoint of organizational control systems.

E. Forecasts and Plans

The temporal logic discussed thus far is used for describing temporal *facts*, i.e., observation in th past or present. However, when dealing with descriptions about *future* states, these are generally not regarded as facts (we cannot observe the future), but rather as speculations, intentions, etc.

Using the constructs thus far, we can describe a *forecast* as a state or change assertion realized in the future. Correspondingly, a *plan* is an action assertion realized in the future.

A contingent plan is a plan whose actions are conditional on some forecast.

While probabilities are sometimes associated with forecasts, this is seldom done with planned actions. However, in either case we can speak of whether or not the forecast or plan was realized--i.e., when current time reaches the time of the plan or forecast. In this way, assertions about the future can be related to the deductive framework of first order logic.

F. Contractual Obligation and Logical Interpretation of Accounting Data

The concept of contractual obligation extends naturally from that of a plan. It is essentially a plan promised to some other party.

This notion of promise can be convened using a so-called "deontic logic." Again with reference to von Wright (1968), the following operators are introduced. For an action, a:

Oa means a is obligatory
Pa means a is permitted
Fa means a is forbidden.

Interestingly, these three concepts are inter-definable--for instance that *a* is obligatory is equivalent to it not permitted not to do *a*, which in turn to forbid not doing *a*.

When dealing with contracts we are basically concerned with the concept of obligation. However, in commercial transactions this is not merely a relationship created between two individuals, but as well an *object* in itself which can for instance be bought and sold (e.g., factoring of receivables, trading of bonds).

Recognizing this gives certain logical insights into the structure of accounting data, namely that many of its assets (receivables, investments in securities, licenses) and all of its liabilities (notes, bonds, grades of stock) can be elaborated as so-called *promissory objects*. Interestingly, this logical interpretation supports emerging views in theoretical economics and finance of the firm as a locus of contingent obligation.

V. CONCLUSION

The concept of a logical domain model was proposed as a qualitative inferencing framework for integrating the database and analytic model resources in a DSS. This has certain practical possibilities in enabling the DSS to incorporate certain types of "knowledge" about the problem domain.

It as well serves as a proposal for a theoretical basis of DSS research--using logical representations as an intermediate, formal specification of application related knowledge independent of computational limitations.

REFERENCES

Brachman, R.J. 1978. Theoretical Studies in Natural Language Understanding. Report No. 3888. Cambridge, Massachusetts: Bolt, Beranek and Newman.

Brachman, R.J. and B.C. Smith. eds. 1980. Special Issue on Knowledge Representation. SIGART Newsletter 70.

Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. 13: 377-387.

Codd, E.F. 1971. A Data Base Sublanguage Founded on the Relational Calculus. Proceedings of ACM SIGFIDET Workshop on Data Description, Access, and Control. San Diego, California.

Coelho, H., L.C. Cotta and L.M. Periera. 1980. How to Solve It with PROLOG. Lisbon: Laboratorio Nacional de Engenharia Civil.

Elam, J.J., J.C. Henderson and L.W. Miller. 1980. Model Management Systems: An Approach to Decision Support in Complex Organizations. In Proceedings of the First International Conference on Information Systems, Philadelphia, Pennsylvania, December 8-10, 1980, pp.98-110.

Fick, G. and R.A. Sprague, Jr. eds. Decision Support Systems: Issues and Challenges. IIASA Proceeding Series No. 11. Oxford: Pergamon

Press.

Goodman, N. 1977. The Structure of Appearance. 3rd Ed. Dordrecht, Holland: D. Reidel Publishing Co.

Lee, R.M. 1980. CANDID--A Logical Calculus for Describing Financial Contracts. Ph.D. dissertation. Philadelphia: Department of Decision Sciences, The Wharton School, The University of Pennsylvania.

Melichar, B. 1981. Nonprocedural Communication Between User and Application Software. RR-81-22. Laxenburg, Austria: International Institute for Applied Systems Analysis.

Minsky, M. 1975. A Framework for Representing Knowledge. In P.H. Winston, ed, The Psychology of Computer Vision. New York: McGraw Hill.

Nilsson, N.J. 1980. Principles of Artificial Intelligence. Palo Alto, California: Tioga Publishing.

Quillian, M.R. 1968. Semantic Memory. In Semantic Information Processing, pp.227-268. Cambridge, Massachusetts: MIT Press.

Rescher, N. and A. Urquhart. 1971. Temporal Logic. Vienna: Springer-Verlag.

Sprague, R.A., Jr. 1980. A framework for research on decision support systems. In G. Fick and R.A. Sprague, Jr. eds. Decision Support Systems: Issues and Challenges, pp.5-23. Oxford: Pergamon Press.

van Griethuysen, J.J. *et al.* eds. 1981. Concepts and Terminology for the Conceptual Schema. ISO/TC97/SC5/WG3 Preliminary report by International Organization for Standardization.

von Wright, G.H. 1965. And Next. *Acta Philosophica Fennica* Fasc. XVIII: 293-301.

von Wright, G.H. 1967. The Logic of Action--A Sketch. In N. Rescher, ed, The Logic of Decision and Action, pp.121-136. Pittsburgh: University of Pittsburgh Press.

von Wright, G.H. 1968. An Essay in Deontic Logic and the General Theory of Action. *Acta Philosophica Fennica* Fasc. XXI. Amsterdam: North Holland.

Winograd, T. 1975. Frame Representations and the Declarative-Procedural Controversy. In D.G. Bobrow and A. Collins, eds, Representation and Understanding, pp.185-220. New York: Academic Press, Inc.