STRUCTURAL ANALYSIS OF DECISION MODELS

Levent V. Orman

Johnson Graduate School of Management, Cornell University

Modern organizations rely extensively on computer-based mathematical models designed to solve various statistical, optimization, and decision making problems, or to enhance the quality of the decisions made by humans. These models do not exist in isolation, but form complex relationships with each other resulting in organizational model bases. Large model bases require extensive structuring and organization to reduce complexity. Abstraction and decomposition are the two major tools of organization, and are shown to be the basis of macro and micro structure respectively. Both abstraction and decomposition are studied within a framework of model definition based on data models and constraints.

1. Decision Models

Modern organizations rely extensively on computer-based mathematical models designed to solve various statistical, optimization, and decision making problems [3, 47]. The storage, retrieval, maintenance, and especially the structure of these models are critical to the success of these organizations. Ad hoc development of decision models, unstructured implementation using various programming languages, and textual descriptions through user’s manuals are effective only for minimally automated organizations. Large scale automation requires high level description of models to protect users from the algorithmic details. Model retrieval strategies to locate the correct models for each problem and for each maintenance task without an exhaustive search through manuals, and extensive linkages among models to eliminate duplication and to facilitate sharing [6, 24].

At another level, human organizations can also be viewed as a collection of decision models [15, 19, 48]. As in automated models, the human decision models also need to be designed, stored, retrieved, maintained, and structured carefully to ensure the success of the organization. Model description corresponds to task and job descriptions; retrieval strategies are analogous to organizational problem solving; and the model linkages are related to the organizational structure. This article will concentrate on automated models, but the parallels to human organizations will be suggested throughout.

Previous work on model management has concentrated on model description and manipulation as opposed to structure. Four general approaches to model description and manipulation are identifiable. Data model based approaches characterize models in terms of the data they process, and the data relationships they impose [6, 10]. Consequently, they provide rich semantic descriptions through classification and abstraction, but fail to incorporate the mathematical detail. Algebraic models on the other hand, are rich in mathematical detail, but fail to establish the semantic correspondence between the various mathematical constructs such as variables, indexes, and functions, and the real world entities [9, 12]. Two other approaches attempt to bridge the two basic methods to alleviate their individual shortcomings. Logic-based model management starts with a data model approach, and incorporates some of the mathematical relationships in the form of logic expressions defined on the data model [5, 8, 21, 24]. The major advantage of this approach is the formal analysis and inference techniques allowed by logic without considering the algebraic and algorithmic details of a model. Graph-based model management starts with an algebraic model, and incorporates some of the real world semantics in the form of a graph defined on the variables of the algebraic model [11, 18, 34]. The major advantage of this approach is the intuitive appeal of the graphical constructs to end users, and the intuitive manipulation allowed by graphs without considering the data values and the algorithmic details of a model. All of these approaches concentrate on the description and manipulation of models. The structure of the models such as their basic units, aggregation and partitioning to create various grouping of the units, and linkages among those groups are not well understood. Considerable work on structure has been done in other related fields such as software engineering.
implications among keywords cannot always be captured by decomposing them into primitive keywords which are often not obvious to users, and the resulting from decomposition. This design can be computationally tractable. Many issues in software engineering such as module size, module grouping and aggregation, and module linkages and calls, and many issues in organization design such as task size, task grouping and aggregation into activities, jobs, and organizational divisions, and task activity, and job linkages leading to organizational structures, are relevant to model size, model units, grouping and sharing, and model base structure. Cross fertilization is likely to provide insights into all fields involved, in addition to developing the model management field.

In this article, a data model-logic hybrid approach will be taken to model management. This approach has the dual advantage of rich semantics of data models, and the formal analysis of logic. The formal analysis will be extended to cover many of the structural issues commonly discussed in software engineering and organization design literature, but often ignored in model management. In the reverse direction, the formal analysis provided by model management will be applied to issues in software engineering and organization design which are often discussed informally. In particular, abstraction will be shown to determine the macro structure, and decomposition will be shown to define the micro structure. Abstraction will be shown to create broad categories that are useful in suppressing detail and facilitating the useability of the structure without information overload. Decomposition will be shown to facilitate sharing of models and hence lead to significant efficiencies in implementing structures. Optimum level of decomposition will be established through a trade off between the savings due to additional sharing made possible by decomposition, and the costs due to additional communication and coordination requirements resulting from decomposition. This design problem will be formulated as a variant of the traveling salesman problem, and will be shown to be computationally tractable.

A similar attempt to organize large model bases on the basis of model structure was undertaken by [29]. Our approach differs in two aspects. 1. [29] uses a classification based on keywords which are decomposed into sets of primitive keywords. Although keyword approach is practical, it is very restrictive. Semantics of keywords are often not obvious to users, and the implications among keywords cannot always be captured by decomposing them into primitive keywords (e.g., disjunctions of keywords, or a set of keywords implying another set of keywords). Our use of first order logic is more powerful, although not as intuitive as keywords. 2. [29] has a subtyping system based on subsetting of I/O domains. Subtypes are useful in inheritance and identifying substitutes, but subset based subtypes are restrictive. They do not capture a variety of relationships involving composite models which may be created by sharing inputs, joining outputs, or cascading outputs of one as inputs to the next. These relationships require more complex inferencing mechanisms.

2. Model Description

For the purposes of structural analysis, a model can be viewed as a collection of input-output data, and a collection of logic constraints that need to be satisfied by the correct inputs and outputs. Inputs and outputs of a model are data, and hence, can be expressed by the relational data model [35, 36]. The constraints describing valid input and output are database constraints, and hence can be expressed in the clausal form of first order logic [3, 37, 41, 47]. The relational model of data describing input and output consists of relations defined on named domains. A relation $R$ defined on domains $D_1, \ldots, D_n$ is a set of n-tuples (records) $d_1, \ldots, d_n$ where $d_1 \in D_1, \ldots, d_n \in D_n$ [35, 46].

**Example 2.1:** A production environment contains a product name and a demand value for each product; a resource name and the available supply quantity for each resource; and the quantity utilized of each resource by each product in the production process. The following relational database with three relations defines this production environment:

- **PRODUCT (PNAME, DEMAND)**
- **RESOURCE (RNAME, SUPPLY)**
- **PRODUCTION (PNAME, RNAME, UTILIZATION)**

The database at a given point in time may contain the following data:

<table>
<thead>
<tr>
<th>PNAME</th>
<th>DEMAND</th>
<th>RNAME</th>
<th>SUPPLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
<td>oil</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
<td>labor</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>18</td>
<td>land</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRODUCT</th>
<th>RESOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PNAME</td>
<td>RNAME</td>
</tr>
<tr>
<td>a</td>
<td>oil</td>
</tr>
<tr>
<td>b</td>
<td>labor</td>
</tr>
<tr>
<td>c</td>
<td>land</td>
</tr>
<tr>
<td></td>
<td>labor</td>
</tr>
</tbody>
</table>

| PRODUCTION |
The constraints describing valid input and output can be expressed as logic expressions in the clausal form of first order logic [37, 41]. Each expression in the clausal form is a collection of clauses where each clause is in the form of $L_1, \ldots, L_n \rightarrow R_1, \ldots, R_m$ interpreted as $L_1$ and $\ldots$ and $L_n$ implies $R_1$ or $\ldots$ or $R_m$; and each literal $L_i$ or $R_j$ is an atom. An atom is a predicate (a database relation or a built in predicate such as $<$, $\le$, $\ge$, $=$, $\ne$) with arguments that are variables, constants or functions. Variables are shown in lower case letters, functions in capital letters, and character string constants in quotes. Intuitively, all variables such as $x$ are interpreted to be universally quantified ($\forall x$), and all functions such as $F(x)$ are interpreted to be existentially quantified ($\exists F$).

The formal semantics of the clausal logic can be found in [41], and informal semantics can be shown by correspondence to the standard form of first order logic [35].

Although it is logically first order complete, the major shortcoming of the clausal logic is in expressing aggregate operators such as SUM, PRODUCT, MAXIMUM, or MINIMUM which are so critical in mathematical modeling. There is a natural extension to represent these operations also as built in predicates, since an aggregate operator is merely a relationship between a set of data items and a singleton e.g., a set of data items and their sum, a set of data items and their maximum.

Example 2.2: The maximum demand in the PRODUCT file is 20, and it is denoted by a relationship between the demand values $[10, 20, 18]$ and the value 20.

| a  | 10 | 10 20 |
| b  | 20 | 20 20 |
| c  | 18 | 18 20 |

$\text{PRODUCT}(x, D(x)) \rightarrow \text{MAXIMUM}(D(x), M)$

where $M$ is 20, and $D(x)$ takes the values $10, 20, 18$ for $x=a, b, c$ respectively. The expressions are interpreted as for each $x$ there is a $D(x)$ representing the demand for $x$, and the maximum of $D(x)$ is $M$. Similarly, the maximum utilization rate for each product relates each utilization rate to the maximum of all rates for that product:

$\text{PRODUCTION}(\text{pname}, \text{name}, U(\text{pname}, \text{name})) \rightarrow \text{MAXIMUM}(U(\text{pname}, \text{name}), M(\text{pname}))$

which states that for every $\text{pname-name}$ pair in $\text{PRODUCTION}$ there is a utilization value $U(\text{pname}, \text{name})$; for every $\text{pname}$ there is an $M(\text{pname})$ value that is the maximum of all $U(\text{pname}, \text{name})$ values with matching $\text{pname}$ (the common argument).

Clausal form of logic has been used for general purpose programming under the name “logic programming” [27], however, a model description language (MDL) based on clausal logic needs to be different from logic programming in a variety of ways:

a. Model descriptions do not have to be directly executable since they are mere descriptions of problems and their solvers, and they are used only to select, transform, combine, link, and reason about the solvers which have an independent existence. Consequently, many of the restrictions imposed on clausal logic to make it directly executable are not necessary in model management, hence, avoiding the limitations imposed on logic by those restrictions. Conversely, the procedural extensions to logic programming to repair these limitations are not necessary in model description languages, hence preserving the simple declarative semantics of logic [26, 41].

b. Model description languages have to provide abstraction capabilities to capture the complex structures exhibited by decision models. Organizational models are not solitary entities, but they relate to each other with complex exchanges, and they are often organized into large related groups forming hierarchical or network type structures. Logic programs are not suitable to describe such complex structures, since they promote uniform and minimal components for simplicity of development and programming tasks. Model description languages are not used for developing systems, and they need to emphasize semantic richness to describe complex structures.

c. Logic programs do not distinguish between inputs and outputs. In model management, this distinction is historically made, and allows a more compact and intuitive expression of model semantics. Compactness of notation is important in model management, since a simple model may require dozens of clauses for a complete description, and logic programs quickly become unwieldy and unreadable [26, 35].

A model is described in MDL as a collection of database constraints to be satisfied by all valid input and output of the model. The description of a model then has three components. Input and output are data, and they are expressed as relational database predicates with variable arguments. Constraints are logical expressions in clausal form of first order logic. The syntax of MDL is a triplet $I, O, C$ corresponding to input, output, and constraints where each one is a collection of
clauses listed in a column, and I and O are restricted to data predicates.

Example 2.3: Given a relation PRODUCT(PNAME, DEMAND) as in Example 2.1, deriving another relation HDPRODUCT(PNAME, DEMAND) containing high demand products with a demand between 100 and 1000 units can be expressed by the following model:

\[
\begin{align*}
I: & \quad \text{PRODUCT}(x,y) \\
O: & \quad \text{HDPRODUCT}(x,y) \\
C: & \quad y > 100 \\
& \quad y < 1000
\end{align*}
\]

Formally, each MDL expression is a triplet \(I, O, C\) where \(I, O,\) and \(C\) each is a clausal expression in first order logic. I and O are simple clausal expressions where each clause is an atom whose arguments are variables (such as \(\text{PRODUCT}(x,y)\)). C is an arbitrary clausal expression. The triplet is shown as a column of clauses with each clause on a separate line. Formal interpretation of the triplet \(I, O, C\) is \(I \rightarrow (O \leftrightarrow C)\); in other words, for a given input, the constraints are true for all (and only) the correct outputs.

Example 2.4: Given the same relations, the maximum demand over all products can be found by:

\[
\begin{align*}
I: & \quad \text{PRODUCT}(x,y) \\
O: & \quad \text{MDemand}(m) \\
C: & \quad y = \text{DEMAND}(x) \\
& \quad \text{MAX}(\text{DEMAND}(x), m)
\end{align*}
\]

The optimization problems can be expressed similarly using the \text{MAX} predicate.

All common decision models are expressible in MDL, since for a given input-output pair, constraints consist of arbitrary expressions in clausal logic, and clausal logic is first order complete [41]. Once the models are described using MDL, these descriptions can be stored in a database, and retrieved using logic based query languages. In particular, models that are capable of solving a specific problem (also expressed in MDL) can be located and matched to the problem at hand using logical inference. However, logical inference with a large number of models can be quite inefficient, and additional structures are necessary to facilitate efficient model management.

3. Model Abstraction

Organizational models are not always described individually in their full detail, but often organized into groups of models with high level abstract descriptions. Each group is treated as a unit at a macro level, its individual models are not distinguished from each other, and its internal structure is not visible to its external users [35]. Such organization is useful in reducing complexity and facilitating efficient problem solving. It reduces complexity since the details of individual models are hidden, and the users deal with limited abstract descriptions; and it facilitates efficient problem solving since groups serve as classifications and simplify the search procedures to locate relevant models for each problem. Such abstractions are also common in human organizations; most notably, geographic divisions, functional divisions, and product divisions of modern organizations group large numbers of decision models and decision makers into abstract structures, and serve to reduce complexity and facilitate efficient problem solving [4].

There are three types of abstraction corresponding to the three components of decision models: input, output, and constraint based abstraction. Input-based abstraction clusters all models that use a given set of input data into a single unit, and identifies the unit by that common input. For example, all models that use the product relation of Example 2.1 such as the pricing and production scheduling models can be grouped together to respond efficiently to the events modifying the \text{PRODUCT} data. A change in demand, or the introduction of a new product requires the execution of some models in this group, and all models needed are within the same group. Such input based abstractions are called "transactions" in software engineering, and they provide an efficient structure to respond to external events that trigger the execution of a large number of models. Grouping together all models that are affected by a change in input data is an efficient way of responding to external events that update data [35]. Human organizations also utilize input based abstractions. They are input-based divisions of organizations where all decision models and decision makers relevant to a particular type of input are grouped together. Most notable of these are the geographic divisions, especially of large multinationals, where the inputs and general market conditions from a particular geographic area define a division as a collection of decision models and decision makers that use those inputs. Such input-based abstractions respond to external events with considerable speed and efficiency since all decision models required to respond to the event are clustered together in one unit [20]. On the negative side, there is considerable overlap and redundancy among input-based abstract units, since many inputs trigger many of the same or similar models. A change in demand for a product, or a change in the supply of a resource in Example 2.1, will both trigger pricing and production scheduling models [47]. In a human organization, models related to the west coast operations are likely to interact with the decision models related to the midwest. Such interacting models clearly belong to and repeated in
multiple units, creating considerable redundancy. Consequently, it is very common in organizations with geographic divisions to duplicate many functions, skills, and models in multiple divisions [20, 32].

Output-based abstraction clusters all models that are needed to produce a particular output into a single unit and identifies the unit by that output. For example, all models needed to produce production schedules in a manufacturing environment can be grouped together to respond efficiently to production scheduling requests. The efficiency results from the fact that all needed models are within the same group, and no searching or linking is necessary. Such output-based abstractions are called "applications" in software engineering, and they provide an efficient structure to respond to user requests that require the execution of a large number of models. Grouping all models needed to produce an output within a single unit is an efficient way of responding to user requests for information as long as those requests do not diverge from the predefined system outputs [17]. Human organizations also utilize output-based abstractions. They are commonly called "product divisions", where all decision models and decision makers relevant to a particular product are grouped together. Most notable examples are in large diversified manufacturing companies, e.g., General Motors and its product divisions such as Chevrolet and Pontiac. Such output-based abstractions produce outputs with considerable speed and efficiency since all decision models required for an output are clustered in one unit, provided that the outputs requested are among the standard anticipated products of the organization, and not ad hoc individualized products [33]. On the negative side, there is considerable overlap and redundancy among output-based abstract units, since many outputs require many of the same or similar models. A request for a production schedule, or a pricing decision for a new product both require the demand forecasting model [47]. Similarly, in human organizations the overlap is considerable. Both Chevrolet and Pontiac divisions of General Motors use the same market analysis model to forecast the demand for their products. Consequently, it is common in organizations with product divisions to duplicate many functions, skills, and models in multiple divisions [4, 32].

Constraint-based abstraction clusters all models that enforce a particular constraint or requirement into a single unit and identifies the unit by that requirement. For example, all models that involve maximizing a linear objective function subject to linear constraints can be grouped together as a Linear Programming package for efficient implementation. Similarly, all models that require fitting a curve to discrete data points to minimize the sum of square of the error terms can be grouped together into a regression package for general and efficient implementation. All models that require the satisfaction of such mathematical criteria will invoke these abstract units. The efficiency results from the sharing of a single abstract unit by all models that require it, eliminating the redundant implementation and storage. Such constraint-based abstractions are called "generic models", and they are a major concern of the model management field [3, 10]. They provide an efficient structure to abstract out common components of decision models and make them available to all from a common and shared resource. They are most appropriate when many models are slight variations of each other with considerable overlap, or models have to be modified frequently at the margin to personalize or adjust them to the idiosyncrasies of users. Modern decision support systems are often characterized by such dynamic requirements [35, 47]. Human organizations also utilize constraint-based abstractions. They are commonly called "functional divisions", where all decision models and decision makers relevant to a particular skill or function are grouped together. The examples of functional divisions are abound in modern corporations. It would be difficult to find an organizational chart that does not refer to functional divisions such as marketing, finance, or human resources. Such function-based abstractions are easy to implement since they avoid duplication and foster sharing of skills. They also respond to environmental and technological changes efficiently since the impact is often localized due to lack of redundancy, and specialized personnel can better deal with an uncertain environment without information overload. A new forecasting technique or a sudden disruption of critical raw material supplies are dealt with much more efficiently in a functional organization, where those employees specializing in market analysis or materials purchasing incorporate the new technique or the sudden disruption into their analyses on behalf of the whole organization [2, 44]. On the negative side, sharing and nonredundancy fostered by constraint-based abstraction requires considerable communication and coordination among units, since building an application, or responding to a user request will require searching, retrieving, and linking many shared units. A request for a production schedule may involve invoking many generic models such as regression for forecasting demand, and linear programming for minimizing production cost [3, 24]. Similarly, in human organizations the communication and coordination costs are considerable. Production scheduling for
example requires extensive interaction between marketing and production divisions. Consequently, it is common in organizations with functional divisions to invest heavily into communication and control technologies, processes, and personnel [2, 27, 45].

It is possible to summarize the three types of abstraction in terms of the efficiencies they create and the costs they impose. Two general types of efficiency are short term and long term efficiency. Short term efficiency relates to the operating costs of a system when the environment is fixed or at least stable. Long term efficiency relates to the ability of the system to absorb and adapt to the changes in the environment. In software engineering, these two types of efficiency are referred to as the "operational" and "maintenance" costs. In human organizations they are called "efficiency" and "flexibility". It is also possible to identify two types of long term efficiency (flexibility): behavioral and structural. Behavioral flexibility is the ability of the system to adapt to the changing behavior of the input variables that are monitored routinely. These are parametric changes where a broad category of change is anticipated, but its exact nature and value are not known. Structural flexibility is the ability of the system to absorb unanticipated changes such as new variables, technological innovations, and environmental disruptions that alter the structure of decision models.

Output-based abstraction maximizes short term efficiency at the expense of others, since predetermined outputs can be produced efficiently by using a minimal number of units, but any change in the environment will require an overhaul of the whole system to determine the effect on each unit. Application-based systems in software engineering are notoriously difficult to maintain, and product divisions of large corporations have been noted for their inertia and resistance to structural changes, e.g., General Motors and the inability of its product divisions to respond to the changing market conditions in 1970s and 1980s [32]. Input-based abstractions maximize behavioral long term efficiency since they can accommodate changes in the values of input variables efficiently within minimal number of units, but structural changes require an overhaul of many units, and every output involves a search through many units to locate the appropriate models. Transaction based systems in software engineering are designed for efficient handling of inputs, but notoriously lacking in user friendliness and responsiveness to the changing needs of users. Similarly, geographic divisions of large corporations are locally responsive, but resistant to global structural changes [4, 19]. Finally, constraint-based abstractions maximize structural long term efficiency, since they can accommodate structural changes efficiently within a minimal number of units due to the nonredundancy characteristic, but input changes or output requests require searching and linking many models across many units. Generic models such as Linear Programming or Regression packages are notoriously difficult to use without extensive education, and require complex linkages to other packages and data files; and communication problems among functional divisions of large corporations are well documented [31, 32]. These three types of efficiency summarize the arguments for and the advantages of the three types of abstraction developed in this section.

4. Model Decomposition

Model abstraction is useful at a macro level by aggregating models into large units. However, such macro level analysis is rarely sufficient to completely organize and structure a complex system of decision models. Complex systems require complex organization involving multilevel multicomponent structures, resulting from both macro and micro level analysis. The principal tool of micro analysis is model decomposition. Decomposition enables a micro level analysis, to fine tune the broad abstract structures of macro analysis, by dividing models into smaller components, and facilitating sharing of those smaller components by multiple models. Decomposition is often touted as the basis of all structure [11]. It has been studied extensively, but no consensus exists as to why and when systems should be decomposed into smaller components. The oldest theory is the prevention of information overload. In software engineering, decomposition of systems into smaller modules is justified by the limit on complexity that can be handled by one programmer. Dividing a system into modules and assigning them to different programmers keeps the complexity per programmer below a certain threshold [17, 39]. Similarly, in organization design, information overload is the oldest theory behind all structure. Organizational activities are decomposed into smaller tasks and assigned to different employees to prevent the complexity of a job from exceeding the complexity tolerance of employees [19, 44]. This theory is simple and quite effective in explaining the cognitive limitations of humans, but it does not capture the essence of structure. It does not explain the structure often found within an individual job, nor does it explain extensive structure utilized in software systems to maximize machine efficiency even when information overload is not a concern (e.g., when large processors are assigned to
Proceedings of the 28th Annual Hawaii International Conference on System Sciences 1995

relatively small tasks). The second theory is the reduction in the complexity of a system by dividing complex units into smaller relatively independent units [25, 30, 31, 39, 40, 45]. The reduction is claimed to follow from the multiplicative effect on complexity of the variables within a unit, as opposed to the additive effect on complexity when the variables are split into multiple independent tasks. This theory explains elegantly the effect of interdependent variables on complexity in contrast to the effect of independent variables, but it fails to explain why independence of variables cannot be taken advantage within the same unit. If two variables are independent, they should be independent whether they are split into two separate units or not! Conversely, there is no obvious reason why splitting interdependent variables into separate units should reduce the total complexity, except by ignoring the interdependence of variables and producing a suboptimum solution. That trade-off between reduction of complexity and suboptimization has not been clearly established. The third theory suggests that the processors (human or machine) are more efficient in repetitive execution of the same task than in switching from task to task, leading to savings when the tasks are decomposed and processors are specialized [2, 22, 32, 48]. This theory has roots in information theory and information economics since it deals with the information content of each unit. However, the theory acknowledges that while saving switching costs, decomposition introduces communication and coordination costs in all cases except when the tasks are completely independent [28], and fails to demonstrate why the switching costs would be so significantly higher than the communication costs to provide the basis of fundamental structures.

This article adopts a variant of the third theory, and proposes "sharing" as a fundamental basis for structure. Avoiding duplication of a common task for every model that needs it can be a significant source of efficiency. However, tasks exist in a context and they are rarely identical in multiple contexts, making sharing difficult. More specifically, tasks are optimized to fit into the context of a specific model, and shared tasks cannot be optimized as effectively since they need to serve multiple objectives. This trade-off between the savings resulting from sharing, and the costs resulting from suboptimization of shared tasks is critical in determining optimum structure. The relationship to structure is through decomposition. Decomposition creates smaller components which are more readily shareable by more models. The savings produced by more sharing, and the costs resulting from the suboptimization of the shared components determine the optimum level of decomposition and the variety of structures it causes.

Intuitive Example 4.1: A commercial bank has two decision models for each mortgage loan applicant: one for a short-term loan and one for a long-term loan. Each decision model is executed for each loan applicant and each has two tests as shown below:

**Short-Term Loan Decision**
- $t_1$: homeowner test
- $t_2$: salary $> 30K$ test

**Long-Term Loan Decision**
- $t_1$: homeowner test
- $t_2$: salary $> 30K$ test
- $t_3$: life expectancy $> 20$ years test

Assume that each test is equally difficult to execute; there are 100 customers incoming daily, 60% of which are homeowners, 20% with salary $> 30K$, and 30% with life expectancy $> 20$ years. The objective is to minimize the cost of information processing which is measured in terms of the number of tests performed. If the two models are executed independently, each model would apply the most stringent test first to eliminate as many customers and as quickly as possible. The first model would execute $t_2$ first for 100 customers, and $t_1$ next for the 20 customers who have passed the first test, at a cost of 120 tests. The second model would execute $t_3$ and then $t_1$. Similarly at a cost of 130 tests. Total cost = 250 tests. Alternatively, consider the decomposition of these two models into three models, each containing only one test, with $t_1$ shared by the two decisions. To achieve the benefits of sharing, $t_1$ has to be executed first, since otherwise $t_2$ and $t_3$ will produce different sets of customers to be processed, and the benefits of sharing will be lost! With decomposition, the homeowner test will be applied only once, cost = 100; the other two tests will be applied next, each to 60 customers returned by the homeowner test, cost = 120. Total cost = 220 tests. Clearly, decomposition pays! However, by changing the homeowner percentage to 80%, the decomposed model would lead to a cost of 100 + 80 + 80 = 260 tests, clearly higher than the original models. In this case, the savings of 100 tests on home ownership comes at too great a cost of suboptimization of each model by changing the optimum sequence of administering tests.

Decomposition does not in and of itself reduce complexity, but facilitates sharing. Sharing reduces complexity since shared tasks do not have to be duplicated for every decision model that needs them. Sharing also leads to local suboptimization, since models cannot do their information processing in the optimum sequence, but have to accommodate the structure imposed by
the shared task. This imposition is realized through
the communication between the shared task and the
models that receive information from it; it is often
called "communication and coordination cost" [28].
Clearly, it is much more than the cost of
transporting messages, but it involves what the
models have to do to accommodate the messages
received.
Example 4.2. Given the database of Example 2.1,
the following models are used to find the products
that consume high and low levels of oil
respectively.

\begin{align*}
\text{PRODUCTION} (x, y, u) \\
\text{HIGH}(u) \\
c_1: y = \text{"oil"} \\
c_2: u > 10 \\
\text{PRODUCTION} (x, y, u) \\
\text{LOW}(u) \\
c_3: y = \text{"oil"} \\
c_3: u < 5
\end{align*}

Given that each constraint has the same
complexity, the total number of tests can be used as
a measure of information processing cost.
Assuming 100 records in the PRODUCTION file,
with 60% satisfying the constraint c_1, 20%
satisfying c_2, and 30% satisfying c_3. The optimum
sequence of execution is (c_2, c_1) and (c_3, c_1) in
each model, leading to a total cost of 100 + 20 +
100 + 30 = 250 tests as in Example 4.1.
Decomposing the models by factoring out the
common constraint c_1:

\begin{align*}
\text{PRODUCTION} (x, y, u) \\
\text{OIL}(u) \\
c_1: y = \text{"oil"} \\
c_2: u > 10 \\
\text{OIL}(u) \\
\text{LOW}(u) \\
c_3: u < 5
\end{align*}

where the intermediate relation OIL has 60 records,
leading to a total cost of 100 + 60 + 60 = 220 tests,
indicating savings due to decomposition. As in
Example 4.1, changing the percentage of records
satisfying c_1 to 80% raises the cost of the
decomposed model to 100 + 80 + 80 = 260
rendering decomposition undesirable.

5. Structural Design
The structural design problem involves
choosing the optimum decomposition by balancing
the benefits of sharing with the costs of model
suboptimization. To determine the cost of model
suboptimization, we will formulate a multimodel
problem, i.e., balancing the costs of deviating from
the optimum sequence in each model, against the
obvious benefits of sharing.
A decomposition is worthwhile if the
savings from sharing a decomposed model is
greater than the cost of model suboptimization
resulting from sharing. Structural design is the
problem of choosing the best decomposition given
a collection of q models partitioning a collection of
p constraints, where a constraint may appear in one
or more models. It can be formulated as a variant
of the multiple traveling salesmen problem with
\[ \sum r_i \] nodes where \( r_i \) is the number of constraints
in model \( i \), or \( q^2 \) in case of equal size models of
size \( r \). The complexity of the problem is \( r!q^r \) where
\( r \) is the number of constraints per model, and \( q \) is
the number of models. Given a set of constraints
c_1, \ldots, c_p with execution costs \( b_1, \ldots, b_p \), and
selectivities \( s_1, \ldots, s_p \); also given a set of models
m_1, \ldots, m_q where each model
m_j \subseteq \{c_1, \ldots, c_p\}, the notation is:
\[ c_u \subseteq \text{a subset of constraints of a model,} \]
\[ c_u = \{c_i : i \in u\} \subseteq m_j \text{ for some } j \in [1, \ldots, q]. \]
or a node on the network.
\[ s_u \subseteq \text{joint selectivity of the constraints in} \]
\[ c_u = \prod s_i \text{ if constraints are} \]
\[ \text{independent} \]
\[ s_{null} = 1 \]
\[ b_{uv} = b_i \text{ the cost of executing the constraint} \]
c_i where \( v = (u, i) \) for some \( i \in [1, \ldots, q]\), i.e., the cost of the arc
from node \( u \) to node \( v \) for the total
input file.
\[ x_{uv} = 1 \text{ if the constraint } c_i \text{ is executed} \]
immediately after the constraints
\[ c_u \text{ where} \]
\[ v = (u, i); \text{ i.e., moving from node } u \text{ to node } v \text{ on the network,} \]
\[ x_{uv} = 0 \text{ otherwise.} \]
The formulation as a 0-1 integer programming
problem is:

\[ \text{Minimize } \sum_{u,v} b_{uv} s_u x_{uv} \]
where \( u, v \subseteq m_j \text{ for some } j \in [1, \ldots, q] \),
and \( v = (u, i) \) for some \( i \in [1, \ldots, p] \).
subject to:

\[
\sum_{v} x_{\text{null},v} \geq 1 \\
\text{the requirement to leave the node} \\
\text{"null"}, \text{i.e., the starting node.}
\]

\[
\sum_{u} x_{u,m_j} \geq 1 \\
j = 1, \ldots, q \\
\text{the requirement to visit all models } m_j.
\]

\[
x_{v,w} \leq \sum_{u} x_{u,v} \\
\forall v,w \neq \text{null} \\
\text{the requirement for a continuous path.}
\]

The formulation is different from the multiple traveling salesmen problem since not all nodes need to be visited, but only a collection of continuous arcs need to be followed from the beginning node to the final nodes each corresponding to a model. Sharing is accommodated in the model automatically since an arc traversed multiple times is counted only once. This formulation has a network of \(q^2\) nodes since it considers all subsets of \(r\) constraints in each one of the \(q\) models. Similar to the single model case, the complexity of the problem is \(r!q\), since the variable definitions \((x_{u,v})\) restrict the feasible routes to all permutations of \(r\) constraints in each of the \(q\) models. \(r\) tends to be small, typically \(< 10\), for most decision models, since it is the number of constraints per model. \(q\) is the large critical variable, since it is the number of models in the system. The problem is computationally tractable since its complexity is linear with respect to the critical variable \(q\), and especially because it requires a one-time solution for the life of the system.

6. Conclusions

Modern organizations increasingly rely on computer-based decision models to enhance the quality of decisions. Many of the techniques used to develop, maintain, and describe decision models do not scale up efficiently, and large scale automation requires extensive restructuring of decision models for effective storage, retrieval, description, maintenance, and use. Two other fields have studied such structures: software engineering and organization design where the systems are often very large, and relationships very complex. Both literatures have devoted considerable resources to issues of structure and organization, and both literatures have independently developed a variety of techniques to study various structures. Model management is in a unique position to benefit from both literatures. Software engineering is relevant because models are implemented as software; organization design is relevant because human organizations can also be viewed as collections of decision models. Equally important is the contribution model management can make both to software engineering and organization design. Models lead themselves to formal analysis due to their narrow, well defined semantics, and their abstract and mathematical formulation. Such formal analysis is often difficult in more general and complex environments of software engineering and especially of organization design. Formal analysis of common structures such as hierarchies, abstraction, and decomposition within the narrow and well defined domain of model management may contribute to a better understanding of more complex environments of software engineering and organization design through development of formal theories and design algorithms. This article has shown that models can be defined and described formally in terms of collections of constraints to be satisfied by their input and output. The basis of structure within a model is the sequencing of these constraints for execution. The determination of the optimum sequence has been formulated as a traveling salesman problem. The basis of structure in a model library is decomposition of models, and sharing of common models made possible by such decomposition. Decomposition leads to hierarchical structures when a linear execution sequence of constraints is assumed. More complex network structures follow from parallel execution of constraints and multiple sharing. In both cases, the identification of optimum structures have been formulated as traveling salesman problems, but with increasing complexity from hierarchies to general networks.

Bibliography

6. Dolk D., Model Management and Structured Modeling: The Role of an Information