

DECISION SUPPORT SYSTEMS: AN OBJECT-  
ORIENTED CONCEPTUAL ARCHITECTURE

By

BRIAN PHILLIP LE CLAIRE

Bachelor of Arts  
Ripon College  
Ripon, Wisconsin  
1982

Master of Business Administration  
University of Wisconsin-Oshkosh  
Oshkosh, Wisconsin  
1984

Submitted to the Faculty of the  
Graduate College of the  
Oklahoma State University  
in partial fulfillment of  
the requirements for  
the Degree of  
DOCTOR OF PHILOSOPHY  
December, 1989

Thesis  
1989D  
L462d  
cop-2

DECISION SUPPORT SYSTEMS: AN OBJECT-  
ORIENTED CONCEPTUAL ARCHITECTURE

Thesis Approved:

*Ramesh Sharda*

Thesis Adviser

*Marilyn G. Kletke*

*Wayne A. Sautter*

*D. E. H. H.*

*Noeman R. Durham*

Dean of the Graduate College

## ACKNOWLEDGMENTS

I would like to express my deepest gratitude to the members of my committee: Professor Ramesh Sharda; Professor Marilyn G. Kletke; Professor Wayne A. Meinhart; and Professor George E. Hedrick. I would like to commend them on their endless support during the embryonic stages of my dissertation and notably at its completion. I would also like to thank Professor Eui-Ho Suh for his help in making this dissertation a reality.

I feel especially indebted to Ramesh Sharda who served as a stellar example of a dedicated researcher and educator. His guidance and encouragement at times often served as the only spark of inspiration leading me down what seemed a long and arduous path. I can only hope to acquire his uncanny wisdom and intellectual finesse in my academic pursuits.

I would like to thank and apologize to my wife, Beth, for frequently and selfishly relying on her unfailing patience. I heavily depended on, and received, her support in the final stages of this dissertation and will never be able to fully thank her. Although my newborn son Nathan is too young to remember the time I stole away from him during

my dissertation, I hope that someday he will know how much joy he provided me during its completion.

I would like to thank my parents for their unceasing support through the years. They have made incredible sacrifices in providing me with an education and for this I am truly indebted to them. I aspire to be as successful a parent as they have been for me.

I am also grateful for the support of Vinit Verma and Dave Davis. Both Dave and Vinit proved to be very supportive friends during my tenure at Oklahoma State University. I am deeply saddened in knowing that we must part company. Nevertheless, I look forward to the times that we will once again wile away the hours discussing such topics as computers, programming efficiencies, racquetball, or whatever comes to mind.

Finally, I would like to thank Professor William A. Alexander Jr., my Ripon College adviser. He knew exactly how to deal with me when as a second semester Freshman I walked into his office and told him I wanted a doctorate. He taught me to question, to learn, and to understand. I shall, perhaps, never forgive him for the suddenness with which he left this world. I find solace in knowing that he was successful, however, in directing me toward my goal. It is in his memory that I dedicate this dissertation.

## TABLE OF CONTENTS

Chapter	Page
I.	INTRODUCTION..... 1
	Introduction..... 1
	Background of the Problem..... 2
	Statement of the Problem Situation..... 3
	Purpose of the Study..... 4
	Substantive Assumptions of the Study.... 5
	Rationale and Theoretical Framework..... 7
	Statement of Hypotheses..... 9
	Scope and Delimitations of the Study.... 10
	Outline of the Dissertation..... 10
II.	LITERATURE REVIEW..... 12
	Introduction..... 12
	Object-Oriented System Concepts..... 12
	Object-Oriented System
	Architecture..... 15
	Abstraction Concepts..... 16
	Historical Perspective
	of Abstraction..... 17
	Encapsulation..... 19
	Objects..... 21
	Object Roles..... 22
	Object Relationships..... 23
	Object Subsystem..... 24
	Message Subsystem..... 25
	Methods Subsystem..... 27
	Methods Handler..... 27
	Methods..... 28
	Instance Stores..... 28
	Method Types..... 29
	Inheritance Concepts..... 30
	Decision Support System Concepts..... 34
	The Evolutionary Nature of
	Decision Support Systems..... 41
	Decision Support System Definition. 43
	Decision Support System
	Characteristics..... 44
	Decision Support System Categories. 45

Chapter	Page
Decision Support System	
Architecture.....	47
Advantages of a Decision Support	
System Approach.....	52
Shortcomings of Decision Support	
System Designs.....	53
Data Management System Concepts.....	54
Traditional Data Models.....	55
Distinguishing	
Characteristics.....	55
Relational Data Model.....	56
Limitations of the	
Traditional Data Models.....	59
Semantic Data Models.....	61
Distinguishing	
Characteristics.....	62
Semantic Data Model	
Components.....	64
Entity-Relationship Model.....	69
Model Management System Concepts.....	71
Traditional Approaches to Model	
Management.....	74
Model Management System	
Objectives.....	74
Distinguishing	
Characteristics.....	76
Structured Modeling.....	77
Elemental Structure.....	77
Generic Structure.....	79
Modular Structure.....	80
Structured Model.....	82
 III.	
OBJECT-ORIENTED RELATIONAL DATA MODEL	
MANAGEMENT SYSTEM.....	85
Introduction.....	85
Object-Oriented Relational Data Model	
Fundamentals.....	86
Object-Oriented Relational Data	
Model Schema Development.....	87
Object-Oriented Relational Data	
Model Schema Abstraction.....	91
Attribute Syntax.....	92
 IV.	
OBJECT-ORIENTED MODEL MANAGEMENT SYSTEM.....	95
Introduction.....	95
Object-Oriented Structured Modeling	
Fundamentals.....	95
Model Schema Development.....	97

Chapter	Page
Model Schema Abstraction.....	108
Attribute Syntax.....	113
Entity Object Syntax.....	115
Relationship Object Syntax....	117
Model Object Syntax.....	119
Model Abstraction Benefits....	121
Model Acyclicity Verification.....	123
V. MESSAGE PROTOCOLS.....	128
Introduction.....	128
Organization of Classes.....	128
Object Class Versus Object	
Instance Access Mechanisms.....	133
General Characteristics of Instance	
Objects.....	135
Object Class Identifiers.....	135
Attribute Information.....	136
Attribute Definitions.....	136
Attribute Access	
Mechanisms.....	139
Overriding Derived	
Attributes.....	139
Object Instance Identifier List....	140
Related Issues.....	141
Productions.....	141
Contexts.....	143
Object Dependencies.....	144
Class Message Protocols.....	145
Metamodel Class.....	147
Class Message Protocols.....	147
Instance Message Protocols....	148
Entity Class.....	162
Class Message Protocols.....	163
Instance Message Protocols....	163
Relationship Class.....	164
Class Message Protocols.....	166
Instance Message Protocols....	166
Model Class.....	171
Class Message Protocols.....	173
Instance Message Protocols....	174
Relation Class.....	182
Class Message Protocols.....	184
Instance Message Protocols....	185
VI. PROTOTYPE DESCRIPTION.....	193
Introduction.....	193
Implementation Environment.....	193
Software.....	194



Chapter	Page
Hardware.....	195
User Interface.....	195
Message Level.....	196
Window Level.....	199
Window Components.....	203
Window Label.....	203
Edit Pane.....	204
Cell Pane.....	206
DSS Browser Level.....	220
Window Level Data and Model Distinctions.....	223
VII.    FUTURE RESEARCH DIRECTIONS.....	229
VIII.   SUMMARY AND CONCLUSIONS.....	232
LITERATURE CITED.....	234

## LIST OF FIGURES

Figure		Page
1.	An Object Subsystem.....	16
2.	Object Subsystems.....	24
3.	Message Subsystem.....	27
4.	Methods Subsystem.....	29
5.	An Inheritance Hierarchy.....	31
6.	A Management Information Systems Framework.....	36
7.	A Decision Support System Framework.....	37
8.	A Connotational View.....	41
9.	The Learning Cycle Model.....	43
10.	Decision Support System Orientation.....	46
11.	Decision Support System Components.....	48
12.	Decision Support System Architecture.....	49
13.	Decision Support System Architecture Revisited.....	50
14.	Decision Support System Levels of Technology....	51
15.	Decision Support System Roles.....	52
16.	Source Relation Example.....	58
17.	Link Relation Example.....	58
18.	Attribute Example.....	65
19.	Aggregation Example.....	66
20.	Grouping Example.....	67

Figure	Page
21. Overlapping Generalization Example.....	68
22. Covering Generalization Example.....	69
23. Derived Schema Components Example.....	70
24. Transportation Model Genus Graph.....	80
25. Source Point and Link Generic Paragraphs.....	81
26. Source Data Module Paragraph.....	81
27. Transportation Model Modular Structure.....	82
28. Transportation Model Modular Outline.....	83
29. Transportation Model Schema.....	83
30. Source Relation Class.....	88
31. Link Relation Class.....	89
32. Data Model Schema Abstraction Syntax Notation...	91
33. Attribute Syntax.....	92
34. Relational Data Model Abstraction General Syntax.....	93
35. Data Model Schema Abstraction Example.....	94
36. Entity-Relationship Diagram.....	99
37. Simplified Class-Instance Diagram.....	100
38. Class-Instance Diagram with Identifier Aggregates.....	102
39. Model Class-Instance Diagram with Instance Attributes.....	104
40. Complete Class-Instance Diagram.....	106
41. Simplified Class-Instance-Model Diagram.....	107
42. Class-Instance-Model Diagram with Instance Attributes.....	109
43. Complete Class-Instance-Model Diagram.....	110

Figure	Page
44. Model Schema Development Steps.....	111
45. General Linear Programming Class-Instance-Model Diagram.....	112
46. Attribute Syntax.....	113
47. Attribute Syntax Examples.....	114
48. Entity Syntax.....	115
49. Entity Syntax Example.....	117
50. Relationship Syntax.....	117
51. Relationship Syntax Example.....	119
52. Model Syntax.....	120
53. Transportation Model Schema Abstraction.....	122
54. General Linear Programming Model Schema Abstraction.....	124
55. Calling Sequence Determination Rules.....	126
56. Algorithm for Verifying Model Acyclicity.....	127
57. Class Object Hierarchy.....	130
58. Creating an Entity Class.....	197
59. Message Level Flowchart.....	199
60. Window Level Flowchart.....	201
61. Source Point Class Window.....	202
62. Transportation Model Class Window.....	202
63. Window Label Menu.....	205
64. Edit Pane Menu.....	206
65. Cell Menu.....	208
66. Column Heading Menu.....	209
67. Row Heading Menus.....	211

Figure	Page
68.	Solution Process Flowchart..... 213
69.	Solution Process Object Interactions..... 214
70.	New Instance Window Cell Pane Menu..... 215
71.	Class and Instances Window Cell Pane Menus..... 217
72.	Window Level Object Interactions..... 219
73.	DSS Browser Window..... 221
74.	List Pane Menu..... 221
75.	DSS Browser Level Flowchart..... 222
76.	Suppliers Relation Window..... 224
77.	New Tuple Cell Pane Menu..... 226
78.	Relation Cell Pane Menu..... 226
79.	Relation Row Heading Menu..... 226
80.	Suppliers Class Window..... 227

## CHAPTER I

### INTRODUCTION

#### Introduction

Mason and Mitroff (1973) informally define information as knowledge for the purpose of taking effective action. Sprague (1987) maintains that the purpose of an organizational information system is to improve the performance of its information workers through the application of information technology. Both the formal and the informal exchange of information, aided by information technology, forms the basis of all organizational activity (Barret and Konsynski 1982, Rathwell and Burns 1985).

Decision Support Systems (DSSs) provide one form of information technology capable of storing, retrieving, presenting, and manipulating data and models in an online, real-time manner. DSSs rely on the intellect of the information worker at all stages of the problem solving process and are different from traditional computer-based approaches to problem solving. Traditional approaches primarily deal with repetitive and routine problem situations which have little need for novelty in the decision making process. Accordingly, the information

worker in the role of the decision maker commonly uses a DSS to solve less well structured, underspecified problems which tend to be novel with no apparent, clear way of solving them.

### Background of the Problem

The development of a DSS is iterative, adaptive, and evolutionary because of its argued need for flexibility. Researchers tend to agree that the most important components of a DSS are (Bonczek, Holsapple, and Whinston 1980a, Sprague 1980): (1) models; (2) data; and (3) the user. The hardware and software employed should facilitate the integration of data and models. Considerable research is directed toward resolving this issue, however, because of a lack of sufficiently general conceptual and theoretical foundations this goal has not been realized (Sprague 1980, Dolk and Konsynski 1984, Ahn and Grudnitski 1985, Klien, Konsynski, and Beck 1985, Blanning 1986, Dolk 1986, Konsynski and Sprague 1986, Lenard 1986).

Researchers often attack this deficiency in the coordination and integration of disparate DSS components from one perspective or another. Holsapple and Whinston (1987) assert that an object-oriented (O-O) notion of the environment within which the DSS functions allows for the coordinated interplay among diverse and related concepts

and is potentially an important one concerning the flexibility, power, and convenience of DSSs.

#### Statement of the Problem Situation

Historically, researchers have viewed DSSs as either data-oriented or model-oriented (Alter 1977, Bonczek, Holsapple, and Whinston 1979, Dolk 1986, Elam and Konsynski 1987). According to Dolk (1986) the information systems community has traditionally emphasized the data-oriented nature of information systems whereas the modeling community, characterized by the fields of operations research and management science, has focused on the algorithms and procedural requirements for solving models. Thus, the tendency of these disciplines is to concentrate on one component of the DSS with the consequence that the DSS user often encounters problems in integrating data and models. For example, DSS users must recollect and reorganize data for each run of a model (Bonczek, Holsapple, and Whinston 1980a).

Researchers have moved away from this fragmented view of decision support. Current research strives to abstract the whole process of data and model management. Such abstraction mechanisms hope to achieve the goal of integrating data and model management systems such that the user is unaware of whether he or she is directing a data retrieval operation or modeling process. Suh and Hinomoto



(1989) suggest a relational approach which integrates the three DSS components under a relational framework. They propose the concept of a relational dialogue base using ideas analogous to those found in relational database and relational model base approaches forwarded by Codd (1970) and Blanning (1985), respectively.

Researchers, however, encounter the problem of orchestrating such ideas into a well designed whole in order to realize an efficient, workable system. Thus, the current study undertakes the problem of developing a conceptualized DSS architecture which incorporates such ideas as data abstraction, model abstraction, and information hiding. This is accomplished by applying O-O notions in the development of such an architecture.

#### Purpose of the Study

The primary purpose of this study is the merging and integration of previously separate tools into a unified whole which represents a conceptual architecture for a DSS. Chung (1984) argues that the design architecture for any given system should consist of different levels of abstraction which may be conceived of as a continuum from conceptual constructs, to operational constructs, and then to implementation constructs. The proposed architecture provides support for all three of these levels of

abstraction. We achieve this support through the specific application of O-O information system ideas.

As a consequence, we direct this study at solving the problem of integrating data and models across various areas (e.g., functional areas) such that the solution techniques provide a mechanism for the coordination of data and model components. Through the application of techniques from different fields, including artificial intelligence, we make the modeling process flexible using an O-O approach to data management and to model development and design. O-O applications follow a modular design where modelers use such design techniques to organize a system into a set of increasingly complex modules (Fuerst and Martin 1984).

Thus, this study undertakes the following objectives:

- (1) to develop an O-O relational data model;
- (2) to develop an O-O structured model;
- (3) to develop message protocols which allow the DSS user, O-O relational data model, and O-O structured model to interact with one another;  
and
- (4) to develop a prototype O-O DSS in a personal computing environment which employs the ideas developed in (1), (2), and (3).

#### Substantive Assumptions of the Study

Gorry and Scott Morton (1971) argue that the missing ingredient in problem formulation is the ability of the modeler to elicit from the decision maker his or her view of the organization and its environment, and the ability to

formalize models of this view. As a result, the process of model definition must be dominated by the decision maker where relevant models are most often the un verbalized models used by the decision makers of the organization (Gorry Scott Morton 1971). This is generally supported by the accepted precept of system design which states that systems have a higher probability of succeeding if users are involved in their development (Fuerst and Martin 1984).

DSS models usually are not laborious to build and users mostly prefer to construct models according to their own way of thinking (Wagner 1981). DSS users create, modify, and discard DSS models according to their weekly needs and whims (Huber 1983). This is reinforced by two principles (Mason and Mitroff 1973): (1) decision makers need information that is geared to their psychology not to that of the system designers; and (2) decision makers need a method of generating evidence that is geared to their problems and to those of the system designers. Gorry and Scott Morton (1971) contend that an understanding of managerial activity is a prerequisite for effective systems design and implementation. Thus, we assume that decision makers should play an integral role in the model creation process.

Huber (1983) notes two conclusions from his review of cognitive style research in management information system (MIS) and DSS design. First, he argues that at present the

available literature on cognitive style is an unsatisfactory basis for deriving operational design guidelines. Second, further cognitive style research is unlikely to provide a satisfactory body of knowledge from which to derive such guidelines. As Bahl and Hunt (1984) discuss, each theory of decision making tends to emphasize different aspects or different perspectives of the general process of making and of implementing decisions. They argue that no single theory of decision making adequately deals with the entirety of the phenomenon. Alavi and Henderson (1981) support this perspective. Thus, we assume that we can ignore theories of cognitive style and decision making processes in developing our architecture.

#### Rationale and Theoretical Framework

Ackoff (1967) argues that no information system should be carried out unless the users for whom it is intended are trained to evaluate and hence control it rather than be controlled by it. A solution forwarded by Ackoff (1967) is to have the user participate in the design of the system thereby assuring the user's ability to evaluate its performance by comparing its output with what was predicted. As a consequence, the user of a DSS should play a much more active and controlling role in the design and development of the system. Andriole (1982) believes the design of DSSs should be completely user and task driven.

The provision of end user control and a simple means for model building requires the flexibility of adding, deleting, or changing DSS functions at the discretion of the user. DSSs should lend themselves to rapid modification to meet the needs of a particular decision maker in each new situation (Rathwell and Burns 1985). This implies the use of a DSS shell which allows users to modify existing features or develop new ones. This suggests an evolutionary approach to system development where the decision maker is the iterative designer of the system since no one can anticipate all conceivable design possibilities or potentially relevant data and modeling needs before design starts. Keen (1980) believes that the evolutionary nature of a DSS is of central conceptual and practical importance. This flexibility allows the DSS to support multiple styles of decision makers solving several different types of tasks (Ahn and Grudnitski 1985).

Finally, the development of a conceptual architecture is helpful in several ways: (1) it organizes a complex subject; (2) it further identifies the relationships between the parts; and (3) it suggests areas for further research. As Blanning (1986) states, an important component of any effective approach to decision support is a theoretical component.

## Statement of Hypotheses

The solution to the problem discussed above, namely the development of a conceptualized DSS architecture which incorporates such ideas as data abstraction, model abstraction, and information hiding, requires the need for a flexible design that can easily adapt to current needs. There should be a dependence upon a generalized system approach where specific systems are built from general systems. This has the advantage of relative ease of understanding since the specific systems are based on the same principles encountered in the general systems (Bonczek, Holsapple, and Whinston 1980b).

Abstraction allows the construction of such complex systems. Abstraction provides a meaningful way of managing complexity and guarantees continuity. The conceptual development of an application involving complexity is perhaps most appropriately handled using a powerful abstraction mechanism, such as provided by an O-O approach.

O-O information systems emphasize objects as the unit of access and manipulation. O-O information systems provide mechanisms to define, create, and relate objects and object interactions. Such systems use abstraction and information hiding, the hiding of design decisions about those abstractions, in order to reduce complexity. O-O information systems deal with a complex idea or real world

system through the construction of a set of independent abstractions.

Thus, given the need for data abstraction, model abstraction, and information hiding, an O-O approach to a conceptualized DSS architecture is a natural choice.

### Scope and Delimitations of the Study

We are concerned with the development of a conceptual architecture for the support of DSS design and do not implement an actual system. We do, however, develop a prototype which employs several fundamental O-O concepts. Furthermore, our prototype provides support for data and model representations but does not support the ideas of model selection or model sequencing (see Bu-Hulaiga and Jain 1988).

The design of the given conceptual architecture suggests a system which does not restrict itself to consideration of problems in a given application area. The proposed architecture is very general in nature allowing for specific incorporation of certain concepts not directly discussed.

### Outline of the Dissertation

Chapter II presents an in-depth review of related literature. Chapter III discusses an O-O relational data model management system which employs fundamental

relational data modeling concepts, data model schema development, and data model schema abstraction. An O-O model management system is discussed in Chapter IV. This chapter presents structured modeling ideas from an O-O perspective and introduces model schema development and model schema abstraction concepts. Chapter V defines several class objects and their associated message protocols necessary for the operationalization of the ideas forwarded in Chapters III and IV. Chapter VI reports on a prototype developed in a personal computing environment employing the message protocols defined in Chapter V. Chapter VII suggests several possible future research directions. Finally, Chapter VIII summarizes and concludes the present study.



## CHAPTER II

### LITERATURE REVIEW

#### Introduction

This chapter presents a review of the literature related to concepts encountered in the study of object-oriented (O-O) information systems, decision support systems (DSSs), data management systems, and model management systems. First, we discuss relevant O-O system ideas and present a formalized O-O architecture. Next, we review various issues involved in the design of DSSs. In the following section we examine assorted data management system notions, specifically Codd's (1970) relational data model and Chen's (1976) entity-relationship model. Finally, we present several model management system issues with an emphasis on Geoffrion's (1987) structured modeling approach to model management.

#### Object-Oriented System Concepts

O-O information systems are the result of a synthesis of many diverse ideas within the computer science field (Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund 1984). Specifically, O-O programming is responsible for the

development of many O-O ideas. O-O programming differs from a procedural style of programming in that the role of data is more central. That is, the shape of the data determines the way the operation behaves (Jenkins, Glasgow, and McCrosky 1986). An operation in procedural programming receives data and is considered the dual of O-O programming (Korth 1986). Data-driven programming moves a problem solution away from the machine domain and places it closer to the problem domain.

The programming language Simula, developed in the middle 1960's, introduced the class concept which is central to O-O notions and was the immediate predecessor to O-O programming (Rentsch 1982, Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund 1984, Stefik and Bobrow 1986). An outcropping of the ideas carried out in Simula resulted in a programming system known as Smalltalk.

The Smalltalk programming system emerged in the early 1970's delineating several O-O concepts. For instance, Smalltalk introduced the term "object-oriented" and perhaps serves as the best current example of an O-O programming language (Rentsch 1982, Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund 1984). Smalltalk was but one part of a broader effort to explore the ways in which people manipulate information and communicate with machines (Shoch 1979). Designers of Smalltalk were influenced from its

inception by Alan Kay's vision of the future, the Dynabook (Cox 1986).

Bergin and Greenfield (1988) argue that any programming language which provides for the notion of an abstract data type (a set of well defined actions on a collection of data structures) and supports the ability to enclose and separate such a type from other types, provides the basis for consideration as an O-O programming language. As a result of this argument O-O programming may be considered either revolutionary or evolutionary (Cox 1984). According to Jacky and Kalet (1987) new dialects of certain languages (e.g., LISP, C, and Pascal) provide support for O-O ideas. Furthermore, they state that several O-O techniques exist for the languages CLU, Ada, and even Fortran.

Several researchers have applied O-O notions to other areas within the information systems literature. The use of such ideas in office information systems development and management appeared as early as 1984 (Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund 1984, Lyngbaek and McLeod 1984). Borgida (1985) proposes the use of O-O concepts in the development of information systems at the conceptual level. Borgida, Greenspan, and Mylopoulos (1985) introduce the use of O-O ideas as a basis for knowledge representation. Recently, much attention has been directed at O-O ideas within the area of artificial intelligence

because of their similarity to existing techniques for knowledge representation, such as frames, and for their use in knowledge acquisition (Casais 1988, Wegner 1988). For similar reasons, O-O ideas have generated considerable interest within the database community during the last few years (Bancilhon 1986, Kim and Lochovsky 1989).

Certain information system areas have successfully applied O-O concepts but only on a limited scale because of the relative unfamiliarity of these concepts. Much of the research within the O-O area is undertaken using an implicit O-O system model as a consequence of this unfamiliarity. In the next section we present an explicit O-O system architecture.

#### Object-Oriented System Architecture

LeClaire and Suh (1988) present a common framework for O-O systems in an attempt to provide a unified paradigm to aid in understanding relevant concepts and bestow researchers with an explicitly formalized O-O system architecture. According to them an O-O system consists of two components: (1) objects; and (2) a message bus.

Rentsch (1982) stresses that objects "are the sole inhabitants of an otherwise empty universe" (p. 53). Hence, objects exist alongside other objects and are the only observable entities within the object universe. Objects are the basic unit of construction used in building

O-O systems. Figure 1 presents a diagrammatic view of such a system. The message bus is a conceptual representation which serves the purpose of providing a logical interface between objects. Presented below is a discussion of the message bus, objects, and their associated ideas.

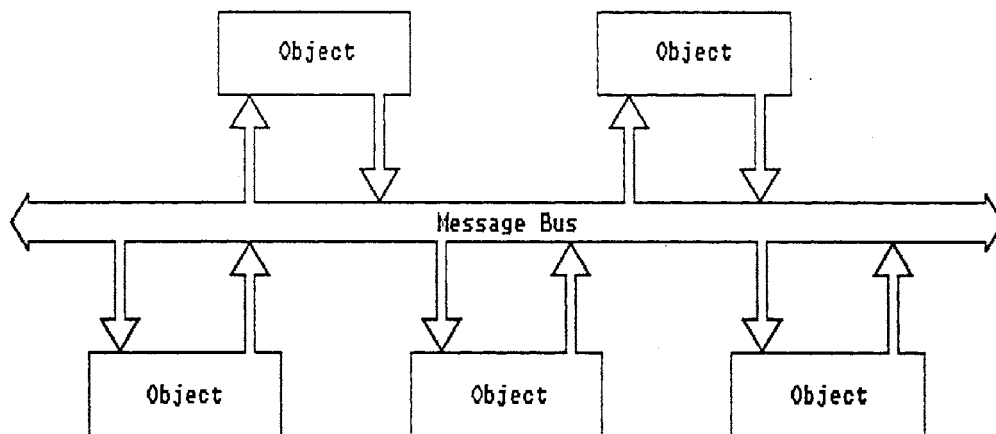


Figure 1. An Object Subsystem

### Abstraction Concepts

Large O-O systems use layers of abstraction in their design and, as a result, a review of abstraction concepts is pertinent as objects represent abstractions of the entities in these systems (Borgida 1985, Booch 1986). Abstraction provides the means to manage complexity and

involves the specification of a system that emphasizes certain system details while suppressing others. This specification is nothing more than a mathematical description of the underlying thought (Liskov and Zilles 1975).

### Historical Perspective of Abstraction

Shoch (1979) notes that Plato's theory of Forms is an example of the early use of abstraction. A study of relevant historical concepts in computer science is justified since, as shown above, most O-O system ideas arose because of work in that discipline. Not surprisingly, modern programming's primary way of controlling complexity is through abstraction.

Abbott (1987) presents an evolutionary perspective of abstraction wherein he argues that abstraction has progressed through several stages in moving toward O-O applicability. According to him this movement is important in that "the history of software development has been the continuing abstraction of programs away from the computer and toward the problem" (p. 664). These evolutionary stages are:

- (1) procedural abstraction;
- (2) syntax abstraction;
- (3) data abstraction; and
- (4) process abstraction.

Procedural abstraction, the first form of software abstraction, is the isolation of certain, possibly parameterized sequences of code. This allows for functional representation where there is no loss of meaning in terms of what the code purports to do. Procedural abstraction was perhaps the first step toward a unified structured programming paradigm. It allows for the expression of simple mathematical functions using a single programming statement. For instance, the statement sqr(x) calculates the square root of the given value.

Instead of translating the problem into machine terms, the use of syntax abstraction allows programmers to remain closer to the problem domain. This is particularly relevant when evaluating arithmetic expressions. No longer is it necessary to assemble a sequence of instructions to add two numbers; simply expressing them in an arithmetic expression such as x+y is sufficient.

Data abstraction emphasizes data rather than control and packages each data structure and its associated operations in a single module. A unit external to the one which manages an abstract data type owns the data type.

The ability to express concurrent processes is known as process abstraction. Procedural, syntax, and data abstractions provide for a fundamental idea known as modularity. Modularity is the design of reusable and modifiable pieces of subroutines with the intent to keep

together related things, such as data structures and procedures (Ahlsen, Bjornerstedt, Britts, Hulten, Soderlund 1984, Stefik and Bobrow 1986).

### Encapsulation

O-O systems extend certain abstraction ideas, modularity in particular, to an idea known as encapsulation. When coupled with process abstraction, the viewing of objects as independent entities, modularity provides the basis for encapsulation. Encapsulation achieves both abstraction and information hiding which are fundamental to O-O systems (Booch 1986, Cox 1986, Bancilhon 1988). The intent of hiding design decisions about abstractions during the decomposition of a system is known as information hiding (Parnas 1972). Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund (1984) argue that information hiding is synonymous with encapsulation.

The uniqueness of an object is determined by its external relations and is independent of its internal representation. This focus on an external view of objects achieves encapsulation. Viewing the actions of objects rather than their intrinsic behavior provides a natural metaphor for that behavior (MacLennan 1982, Rentsch 1982).

The separation of specification from implementation ensures that objects contain the operations necessary to deal with themselves and thus these operations are only



accessible through the given object. As a result, the object which owns the abstract data type also manages it (Buzzard and Mudge 1985). This encapsulation tends to enhance the understandability and maintainability of objects because of the localization of operations (Booch 1986). An object successfully separates external specification from internal implementation by protecting properties used only for purposes internal to the object from outside access (Ahlsen, Bjornerstedt, Britts, Hulten, and Soderlund 1984, Blaha, Premerlani, and Rumbaugh 1988).

This forms the basis for the establishment of protection domains where the effect of an operation within a closed system, for example an object, remains confined to that closed system. According to Buzzard and Mudge (1985), protection domains provide for a secure and error tolerant execution environment. The dependencies between objects are thus decoupled thereby restricting intentional and unintentional modifications from proliferating throughout the system. The localization of design decisions to the object level reduces the scope that a change in an object will have upon the system through the encapsulation of operations done by objects at a primitive level (Booch 1986). In other words, objects should define "the object, the whole object, and nothing but the object" (Booch 1986, p. 216).

## Objects

Booch (1986) postulates that an object has six fundamental characteristics. An object is an entity which:

- (1) exists through time;
- (2) is characterized in behavioral terms, that is, by the actions that it displays and those it requires of other objects;
- (3) is an instance of some, possibly anonymous, class;
- (4) is denoted by a name or identifier;
- (5) has restricted visibility of and by other objects; and
- (6) may be viewed either by its specification or by its implementation.

Thus, an object is a thing that exists, has identity, and is not inert matter. An object "is an active, alive, intelligent entity" (Rentsch 1982, p. 53). Objects correspond to real world entities and as such exist in time, are changeable, have state, and may be created, destroyed, and shared (MacLennan 1982). Two objects that have different substance maintain their identity even though they may have the same form. In other words, two objects occupy separate regions of space even though they may be uniform in every other possible way. In this sense an object may exist without having a unique identifier and thus be distinct from other objects. The issue of object identity continues to be debated (Bancilhon 1986, Bancilhon 1988). It is necessary, however, to be able to distinguish

between various objects and a unique identifier serves this purpose.

Objects represent the primitive elements within an O-O system and are natural metaphors for model building in that each is a capsule of state and behavior (Cox 1984, Cox 1986, Stefik and Bobrow 1986). Such systems, as depicted in Figure 1, use objects to model "some entity, or activity, or, more generally, some concept in the world being modeled" (Borgida, Greenspan, and Mylopoulos 1985, p. 85) and emphasize objects as the unit of access and manipulation. Many of the ideas underlying these characteristics are discussed at length below.

### Object Roles

Objects fulfill certain roles during their existence as do the real world entities which they model. Objects may assume any one of these roles at any given time. Certain objects, however, may only play one role during their existence. Three object roles are (Booch 1986): (1) actor; (2) agent; and (3) server.

An actor is any object which does not serve other objects but which requests their service in fulfilling a given task. An object is acting as an agent when it serves another object. Much like an actor, an agent may request an action be undertaken by another object. An object is functioning as a server when acting for another object.

Unlike an actor, a server is unable to request the service of another object. Thus, an actor strictly directs other objects, a server suffers at the hands of other objects, and an agent may act in either capacity at any point in time.

### Object Relationships

Blaha, Premerlani, and Rumbaugh (1988) define a relationship as a logical binding between objects. They identify three different relationships which may exist between objects. These are: (1) generalization relationship; (2) aggregation relationship; and (3) association relationship.

Generalization, defined by Smith and Smith (1977b), regards a set of similar objects as a generic object. Blaha, Premerlani, and Rumbaugh (1988) agree and extend this idea to O-O modeling by defining a generalization relationship as an is-a relationship which partitions a collection of objects into mutually exclusive subclasses.

Aggregation, as described by Smith and Smith (1977a, 1977b), is an abstraction which allows a relationship between objects to be thought of as a higher-level, named object. Blaha, Premerlani, and Rumbaugh (1988) define this type of relationship as an aggregation relationship such that an object is treated as an assembly component or part-of relationship. Thus, object aggregation is the process

of combining low-level objects into composite objects expressed at a higher-level.

An association relationship is analogous to the notion of an instance of a relationship set as used in Chen's (1976) entity-relationship model. Thus, an association relationship relates two or more independent objects.

### Object Subsystem

Two subsystems comprise an object (see Figure 2): (1) a message subsystem; and (2) a methods subsystem. Each of these subsystems is discussed at length below.

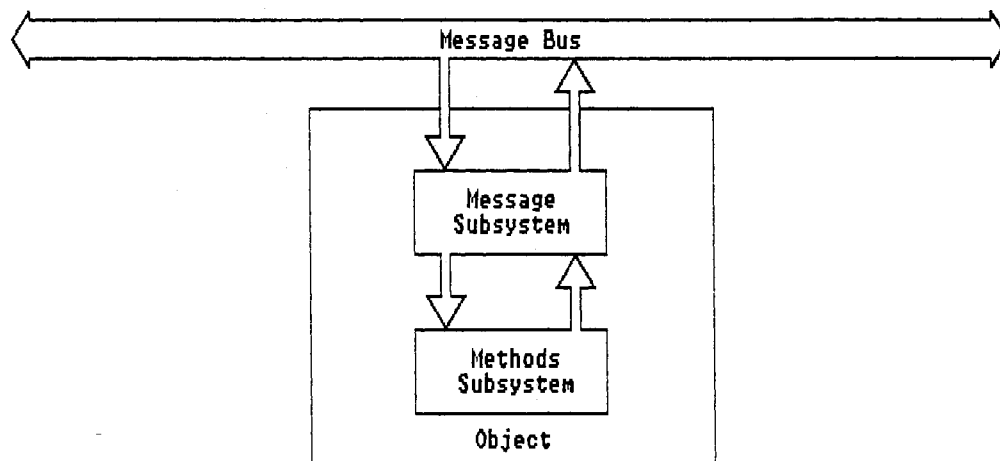


Figure 2. Object Subsystems

### Message Subsystem

The environment manipulates objects by selecting them and communicating to them which requests to fulfill. An object may at times be entirely self-sufficient and thus able to accomplish the requested task. The capability to initiate the fulfillment of a request or propagate this request in the event an object is incapable of carrying it out on its own requires that objects have a means of communicating with other objects. Objects achieve this integration through a message sending/receiving capability. The message subsystem of an object provides this ability.

A message is the specification of a request to be fulfilled by an object. Messages serve to initiate processing and request information. Objects pass messages to other objects across the message bus using a preestablished message protocol. A message protocol is a collection of messages to which an object will respond. The specification of an object name, a method name, and possible parameters is an example of the structure of a message. Message passing between objects is the dual of a functional call of a method name (Jenkins, Glasgow, and McCrosky 1986, Stefik and Bobrow 1986).

The use of protocols allows for a uniform interface between objects and leads to polymorphism. The communication between objects through a well defined interface forms the basis for information hiding as

discussed above. Polymorphism is characteristic of O-O systems because of the interchangeability of objects. This interchangeability is a result of the use of message protocols in that different objects are invoked in the same manner. As a result, objects in one application may effectively be used in another. Booch (1986) states that "reusable software components tend to be objects or classes of objects" (p. 220) and as such an application may be carried out through functional composition rather than decomposition. This results in a large reduction in the complexity of systems. A consequence of this reduction is systems which are easier to build, test, and maintain (Bhaskar 1983).

The message subsystem has two components (see Figure 3): (1) a message receiver; and (2) a message sender. The message receiver responds to messages communicated across the message bus which are directed at the given object. The message receiver retrieves a message from the message bus and passes it along to the methods subsystem. A message queue may be used to buffer messages. The methods subsystem determines whether it is capable of fulfilling the communicated request and, consequently, other message sending may be necessary. Any object can fulfill a request by any message through the direction of message flows to other objects. A central thread of control cannot be identified in an O-O system because of message forwarding,

object independence, and object autonomy. The message sender allows the object to propagate a message to other objects, request specific tasks to be carried out by other objects in fulfilling its task, and to send confirmation about task completion.

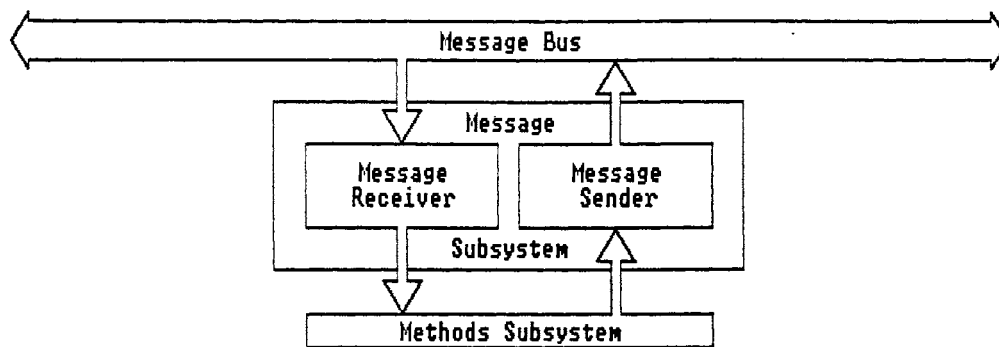


Figure 3. Message Subsystem

### Methods Subsystem

The methods subsystem is comprised of three components (see Figure 4): (1) a methods handler; (2) one or more methods; and (3) zero or more instance stores. Each of these components is elaborated upon below.

Methods Handler. Whereas the message subsystem serves as the interface unit between two objects, the methods handler serves as an interface between the message



subsystem and specific methods. The methods handler is responsible for receiving a message from the message subsystem and determining whether a method exists which will be able to fulfill the given request. An object uses the methods handler in order to complete a request in the event no such method exists within the given object or should a method be unable to fulfill the request without relying on another object. This process is known as delegation (Wegner 1988). Thus, the methods handler, in combination with the message subsystem, allows a "call by desire" implementation (Rentsch 1982).

Methods. The behaviors manifested by objects are known as methods. A method is simply the function which carries out the response to a message. Methods allow for the hiding of information by concealing the way in which an object satisfies a request. Furthermore, methods achieve data abstraction by implementing data manipulation and handling outside of the visibility of the object universe. As a result, methods have natural side effects and tend to modify the state of an object. Using methods the environment determines what is done rather than how it is done (Cox 1984).

Instance Stores. Objects, like the real world entities they represent, have the ability to save state by using methods and instance stores. Objects manipulate the instance stores which they own. They may be dynamically

created or changed during the life of an object. Instance stores form the local database upon which methods act and exist only within the body of an object (MacLennan 1982, Methfessel 1987, Casais 1988).

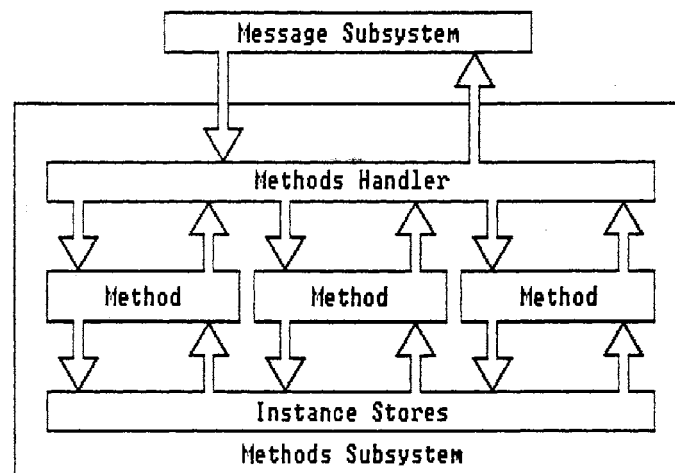


Figure 4. Methods Subsystem

Method Types. Booch (1986) identified three types of operations which may be carried out by a method: (1) constructor; (2) selector; and (3) iterator. A constructor is an operation which alters the current state of the object. In this sense, a constructor operates upon instance stores. An operation that evaluates the current state of an object is a selector. A selector causes an

object to act or display behavior. All parts of an object are visited using an operation known as an iterator.

### Inheritance Concepts

Fundamentally, objects which are similar in nature may be grouped together and in doing so form a class. A class is nothing more than a collection of homogeneous objects expressed at an appropriate level of abstraction. Classes exist in the same sense as objects and are objects at a metalevel, called class objects.

As stated above, an object is an instance of a class. The word instance, when used as a noun, refers to objects which are not classes and are called instance objects. It is possible to have any number of instances of otherwise identical objects. Each object, whether a class object or instance object, may have several instance stores which are private to that object. There are two types of instance stores: (1) existence stores; and (2) class stores. Existence stores allow objects to save state. Existence stores are available within both class objects and instance objects. Class stores provide class objects the ability to store values describing all instances of the class.

The phrase instance of generally refers to the relationship between an object, either a class object or an instance object, and its class. Figure 5 shows an inheritance hierarchy where a specific source point in a

transportation model is an example of an instance object. A source point instance object in this example has two instance stores: (1) sourceName; and (2) supply. A specific source point in the transportation model uses the sourceName existence store for identification purposes. The supply existence store furnishes the quantity available at the specific source point.

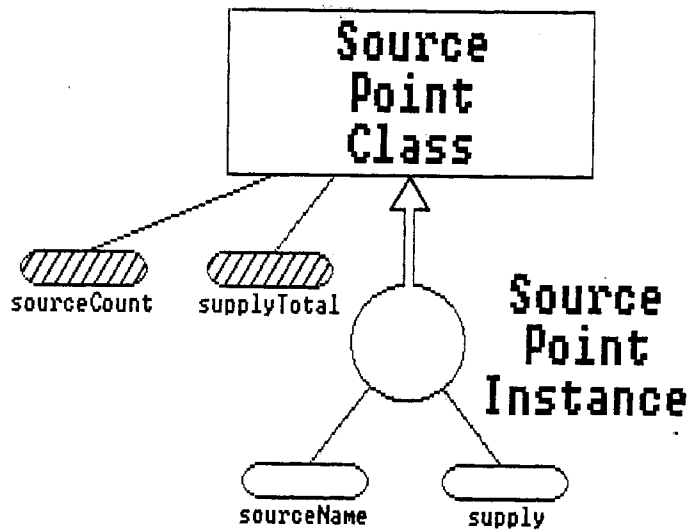


Figure 5. An Inheritance Hierarchy

The collection of individual source points comprises the source point class object. As is evident in Figure 5, the source point class object has two instance stores: (1) sourceCount; and (2) supplyTotal. The instance store

sourceCount is an example of an existence store. This store describes the class object by giving the number of corresponding instance objects. On the other hand, supplyTotal is a class store which describes the total supply of all source point instance objects. Thus, this store describes a characteristic common to all instances of the class.

A given source point instance object is an instance of the source point class object. The source point class object is the superclass of the source point instance objects. A superclass is a class that is above a given object in the inheritance hierarchy. An instance object can never be a superclass object since no objects in the inheritance hierarchy may exist directly below it. The source point class object is an instance of the entity class object. Here the source point class object is a subclass of the entity class object. A subclass is a class that is directly below a given class in the inheritance hierarchy.

Perhaps one of the greatest benefits of an O-O system is the ability of an object to garner the characteristics of the class to which it belongs. Newer classes are built upon older, less specialized classes using inheritance. Inheritance distinguishes O-O systems from other systems (Cox 1984, Wegner 1988). Inheritance implies that an object shares the characteristics common to its class and

permits an incremental sharing of object attributes such as behavior, knowledge, or implementation. This sharing of attributes appears to be a useful device to abbreviate object descriptions and allows a generic object to own a common attribute rather than replicating it many times at lower levels (Borgida 1985, Bic and Gilbert 1986).

According to Stefik and Bobrow (1986), addition allows for the introduction of new instance stores and methods which do not appear in a newly instantiated object's superclasses. Substitution, often called overriding in the O-O literature, is the respecification of an instance store or method which already appears in the inheritance hierarchy. In this sense, stepwise refinement by specialization is possible wherein an attempt is undertaken to define the most general classes first followed by incrementally specializing subclasses. This results in incremental system development and easier replacement of system components and is possible because of the object relationships discussed above.

As previously implied, object instantiation makes inheritance possible. Instantiation is the process of creating a new object and provides for inheritance by attaching the generic attributes of its superclass to that object. Generally, the superclass object provides the ability to instantiate new objects and is implemented as a method. A factory object is such a superclass object in

that it produces new objects (Cox 1984, Cox 1986). The ability of an object to receive its attributes from more than one superclass is known as multiple inheritance.

### Decision Support System Concepts

Gorry and Scott Morton (1971) introduced the term "decision support system" to the management information systems (MIS) community. In their discussion of a MIS framework they note the significance of three managerial levels introduced by Anthony (1965). These levels are: (1) the strategic planning level; (2) the management control level; and (3) the operational control level.

Strategic planning level managers determine the organization's objectives, changes in these objectives, the resources used to attain them, and the policies that are to govern resource acquisition, use, and disposition. The effective and efficient obtainment and use of resources in achieving organizational goals is the concern of management control level managers. Operational control level managers ensure that specific tasks are carried out effectively and efficiently. Gorry and Scott Morton (1971) argue that managerial level determines information use.

Gorry and Scott Morton (1971) also focus attention on problem solving by restating a three phase problem solving process originated by Simon (1960). These phases are: (1)

the intelligence phase; (2) the design phase; and (3) the choice phase.

The decision maker searches the environment for conditions calling for decision in the intelligence phase. The decision maker invents, develops, and analyzes possible courses of action during the design phase. During the choice phase the decision maker selects a particular course of action from the ones identified in the previous phase. Bonczek, Holsapple, and Whinston (1979) argue that the problem solving process requires decision makers to have power. Power allows the decision maker to exercise some authority or directive force. Through power the decision maker can successfully complete the choice phase.

Also, problem structure influences the information system (IS) user. Gorry and Scott Morton (1971) extend Simon's (1960) programmed and nonprogrammed problem types in delineating three new problem types. These problem types are: (1) structured; (2) semi-structured; and (3) unstructured.

Structured problems exist when all three phases of the problem solving process are highly structured. There is no need for novelty in the decision making process as these problems tend to be repetitive and routine. As a result, procedures exist so that each time the problem arises there is no need to deal with it uniquely. Semi-structured problems involve a greater degree of unstructuredness in



that only one or two of the problem solving phases is highly structured. Unstructured problems are, on the other hand, encountered when all three phases of the problem solving process are highly unstructured. These problems are rather novel with no apparent, clear way of solving them.

Structured decision systems (SDSs) and DSSs should handle structured and unstructured problems, respectively, according to Gorry and Scott Morton (1971). This relationship is shown in Figure 6. Figure 6 also incorporates the three levels of managerial activity identified by Anthony (1965). It is clear that all three levels of managerial activity require decision support in the form of a DSS.

	Operational Control	Management Control	Strategic Planning
Structured	Accounts Receivable Order Entry Inventory Control	Budget Analysis - Engineering Costs Short-Term Forecasting	Tanker Fleet Mix Warehouse and Factory Location
Semi-Structured	Production Scheduling Cash Management	Variance Analysis - Overall Budget Budget Preparation	Mergers and Acquisitions New Product Planning
Unstructured	PERT/COST Systems	Sales and Production	Research and Development Planning

Figure 6. A Management Information Systems Framework

Sprague (1980) discusses the idea of task interdependency, originally described by Thompson (1967), in view of DSSs. Hackathorn and Keen (1981) argue that the Gorry and Scott Morton (1971) MIS framework should include this third dimension (see Figure 7). The three types of task interdependency are: (1) independent; (2) sequential interdependent; and (3) pooled interdependent.

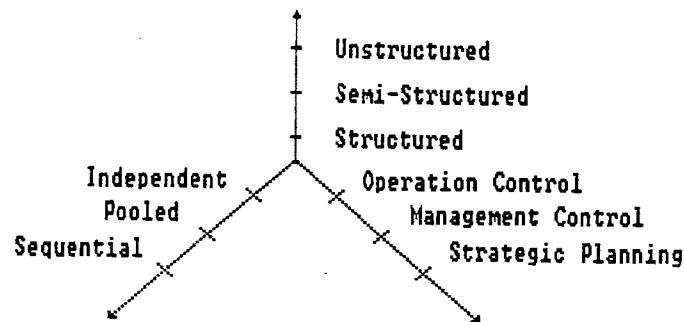


Figure 7. A Decision Support System Framework

The decision maker has, in an independent decision situation, full responsibility and authority to make a complete, implementable decision. In a sequential interdependent decision situation, however, the decision maker makes part of a decision and then passes the decision on to another decision maker. Finally, in a pooled interdependent decision situation, the decision must result

from negotiation and interaction among several decision makers.

Hackathorn and Keen (1981) describe three levels of decision support related to task interdependency. These levels of decision support are: (1) personal support; (2) organizational support; and (3) group support.

Personal decision support focuses on a specific user or class of users confronted with a distinct task or decision. As a result, independent decision situations are the target of personal decision support. An organizational task or activity involving a sequence of operations and actors is the aim of organizational decision support. Thus, sequential interdependent decision situations are the subject of organizational decision support. Finally, group decision support focuses on a group of people. Each person in the group engages in separate but highly interrelated tasks. As a consequence, pooled interdependent decision situations are the focus of group decision support.

Problem solving often involves both data handling and mathematical modeling capabilities (Wang and Courtney 1984). Each approach in isolation has evolved in the DSS literature. Sprague (1987) elaborates on this evolution by suggesting that data processing has followed four distinct stages. These data processing evolutionary stages are:

- (1) data in programs;
- (2) file management;

- (3) database approach; and
- (4) query languages.

In the first stage, the inclusion of data in programs, it became possible to create simple reporting mechanisms such as transaction summaries. The next stage, file management, permitted batch reporting facilities. The database approach stage provided decision makers with a more flexible reporting facility through the logical integration of separate files. Finally, the introduction of query languages gave decision makers the opportunity to do ad hoc reporting.

Sprague (1987) also identifies five stages, similar to the data processing stages, associated with the modeling evolution. These modeling evolutionary stages are:

- (1) symbolic models;
- (2) computational engines;
- (3) computer models;
- (4) modeling systems; and
- (5) interactive models.

Symbolic modeling involved the use of linear and nonlinear equations in an attempt to model the environment. In the next stage users employed computers as computational engines helpful in solving symbolic models. Computers became the model rather than simply solving it during the next stage of evolution through such methods as simulation. Modelers next developed modeling systems such as

statistical or mathematical programming systems in an effort to handle classes of models. Finally, shareable computer time made interactive modeling a possibility. Unfortunately, interactive modeling has led to stand-alone programs with different data requirements, different data formats, and little linkage between models.

It is difficult to discern what, if any, significant contributions DSS ideas make to the field of MIS. Keen (1980) argues that DSSs point toward a synthesis of the MIS and management science (MS) fields. As is seen in Figure 8, Sprague (1980) distinguishes between DSSs and MIS where DSSs have a decision focus while MIS have an information focus. Huber (1981) states that MIS answer "What is" questions while DSSs answer "What if" questions. It was primarily out of the weaknesses of MIS that DSSs developed. Vierck (1981) identifies several weaknesses of MIS:

- (1) they addresses repetitive problems;
- (2) they addresses primarily internal data;
- (3) they are not well oriented to answer the top executive's questions; and
- (4) they lack depth, flexibility, and the power to analyze unstructured problems.

As Parker and Al-Utaibi (1986) note, DSSs involve decisions where there is sufficient structure for computer and analytic aids to be of value, however, the decision maker's judgement is essential. Thus, this involves the creation of a support tool which does not attempt to

automate the decision process, predefine objectives, or impose solutions and which is under the control of the decision maker.

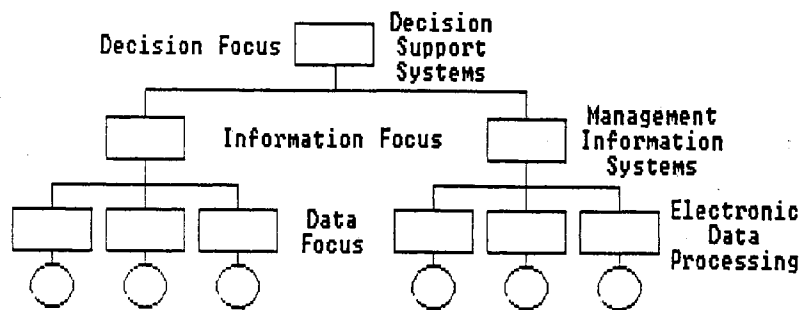


Figure 8. A Connotational View

### The Evolutionary Nature of Decision Support Systems

The evolutionary nature of a DSS is of central conceptual and practical importance. Bonczek, Holsapple, and Whinston (1980b) stress the need for a flexible system that can adapt to change concerning current needs. They argue that a general system should be tailored to specific needs thereby achieving ease of understanding.

A traditional approach often does not rely on user input and instead depends on an analyst's expertise to

ensure appropriate problem conceptualization, model definition, and solution generation (Alavi and Henderson 1981). An evolutionary approach maximizes user input by beginning with simplistic models and iteratively updating these models based on actual use. This direct feedback reduces the system's shift from its predefined objectives (Ahn and Grudnitski 1985). Alavi and Henderson (1981) found that an evolutionary implementation strategy is more effective than a traditional one in their study of approaches to DSS design and implementation.

They also argue that a DSS user must participate in four types of activities in order for effective DSS implementation. These user activities are:

- (1) involvement in new, concrete experiences;
- (2) observation and reflection on those experiences;
- (3) creation of ideas that integrate these observations into theories; and
- (4) usage of these theories to make decisions and solve problems.

They call the repetitive way of moving from one activity to the next the Learning Cycle (LC) model. Figure 9 depicts the LC model. Alavi and Henderson (1981) argue that such a process-oriented evolutionary implementation strategy is more effective when implementing an analytical model. Thus, any DSS employing analytical models should ensure the user's ability to follow such a process. Furthermore, as Wang and Courtney (1984) point out, the

changing nature of the decision environment causes DSSs to have a very short life cycle compared to conventional computer-based ISSs. Thus, a DSS must be easily adapted to environmental change which implies a high frequency of adaptive redesigns.

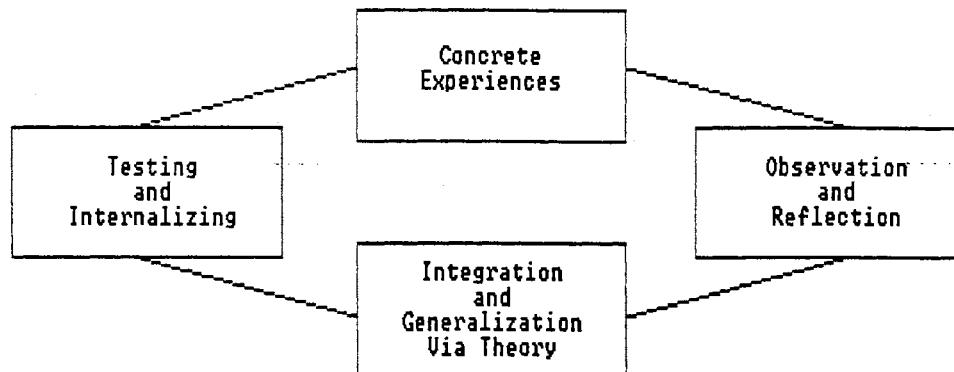


Figure 9. The Learning Cycle Model

DSSs represent an important extension of many ideas found in the study of ISSs. Several key ideas relevant to the current study are examined below.

#### Decision Support System Definition

Many researchers have forwarded competing DSS definitions in the literature. Two of the more descriptive definitions given are:



- (1) Sprague (1980) characterizes DSSs as interactive computer-based systems which help decision makers use data and models to solve unstructured problems; and
- (2) Watson and Hill (1983) define a DSS as an interactive system that provides the user with easy access to decision models and data in order to support semi-structured and unstructured decision making tasks initiated and controlled by the user.

We define a DSS as a user initiated and controlled interactive computer-based system that employs data and models to solve semi-structured and unstructured problems. Thus, the examination of DSSs should address the three topics of data management, computation management, and user interface as suggested by Bonczek, Holsapple, and Whinston (1980a).

#### Decision Support System Characteristics

DSSs have several distinguishing characteristics according to Sprague (1980). First, semi-structured and unstructured problems, addressed more often by managerial control and strategic planning level managers, are the focus of DSSs. Operational control level managers also face such problems but less often.

Additionally, DSSs combine the use of models or analytical techniques with traditional data access and retrieval functions. This characteristic is perhaps one of the least understood because no strong theoretical underpinnings exist which describe this interaction.

Sprague (1980) also characterizes DSSs as easy to use by noncomputer people in an interactive mode. Thus, there is a need to incorporate a data and model transparent user interface. There is a move to abstract the whole process of data and modeling so that the user is unconcerned whether a data or modeling operation is being specified. Furthermore, the user interface must allow the user to describe the system in terms familiar to the modeled operation (Fuerst and Martin 1984).

Finally, DSSs emphasize flexibility and adaptability. This is because of the need to accommodate changes in the decision making environment and the decision making approach of the user.

#### Decision Support System Categories

Historically, DSS design followed one or the other of two orientations: (1) data-oriented design; or (2) model-oriented design. Each design approach emphasizes operations related to its orientation. Alter (1977) was perhaps the first researcher to distinguish between these approaches. Figure 10 shows these orientations on opposite ends of a continuum as identified by Alter (1977).

A data-oriented approach to DSS design supports the user-model interface by treating the representation of a model and its solution as part of a database (Dolk 1986). DSS design approaches which augment existing data models in

order to include modeling capabilities foster this orientation. A data-orientation, however, artificially restricts the domain of model management.

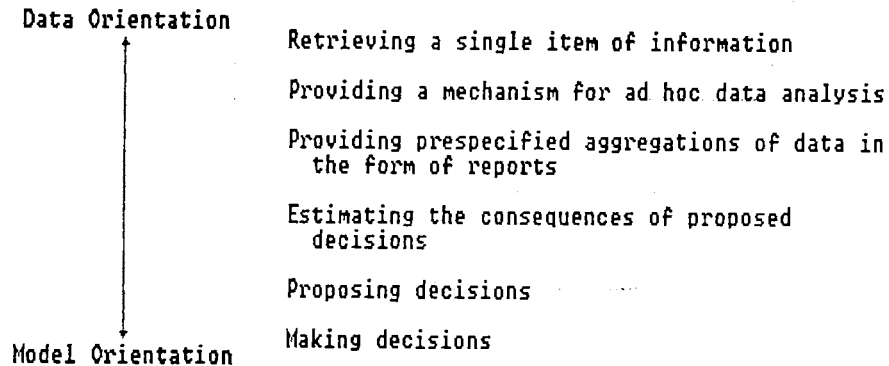


Figure 10. Decision Support System Orientation

A model-oriented DSS design approach focuses on modeling situations. Specifically, model-oriented DSS accomplish model formulation, involving the generation of potential data analyzing algorithms, by modifying and combining various known program modules (Bonczek, Holsapple, and Whinston 1979).

According to Elam and Konsynski (1987), model management is a specific body of research within the DSS field. The identification of those tasks required to build and to use models in an interactive problem solving

environment is a concern of this research. The provision of software support for doing these tasks is also an interest of model management researchers. Unfortunately, the model as a stand-alone system, the most recent evolutionary stage in model development, tends to hide the true relationships between models and data (Dolk 1986).

#### Decision Support System Architecture

Researchers tend to agree that the most important components of a DSS are (Bonczek, Holsapple, and Whinston 1980a, Sprague 1980): (1) models; (2) data; and (3) the user. Bonczek, Holsapple, and Whinston (1980a) discuss the flow of commands and information in a DSS. According to them, commands flow from the user to models, from the user to data, and from models to data. Furthermore, they stated that information, in the form of responses, flows from data to the user, from data to models, and from models to data. Bonczek, Holsapple, and Whinston (1980a) argue that a language for directing computations ensures the flow of information from models to the user. In addition, a language for directing data retrieval makes possible the flow of information from data to the user and from data to models. Figure 11 conveys these relationships.

Bonczek, Holsapple, and Whinston (1980b), in later research, claim that the principle components of a generic DSS are: (1) language system; (2) knowledge system; and

(3) problem processing system. Figure 12 depicts the interrelationships of these components. Although these components seem to differ somewhat from those identified by Sprague (1980) (see Figure 13), they are analogous to one another. The dialogue generation and management system, called language system by Bonczek, Holsapple, and Whinston (1980b), is responsible for coordinating the user's interactions with the other two systems. An action language allows the user to communicate computational and retrieval commands to the other two systems. A presentation language lets the other two systems respond to the user.

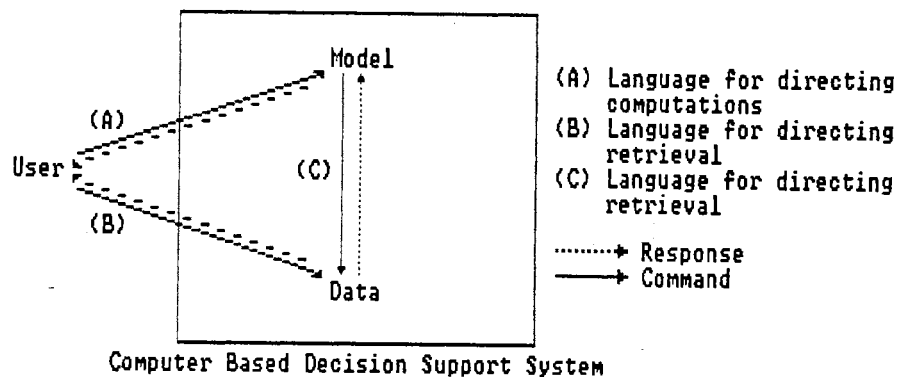


Figure 11. Decision Support System Components

The model base management system, similar to the problem processing system, allows the user to create, maintain, and manipulate a wide variety of models. The model base management system provides specific support for the use of models across all managerial levels and offers the user various model building blocks from which new models may be constructed. Sprague (1980) argues that models "be imbedded in an information system with the database as the integration and communication mechanism between them" (p. 17).

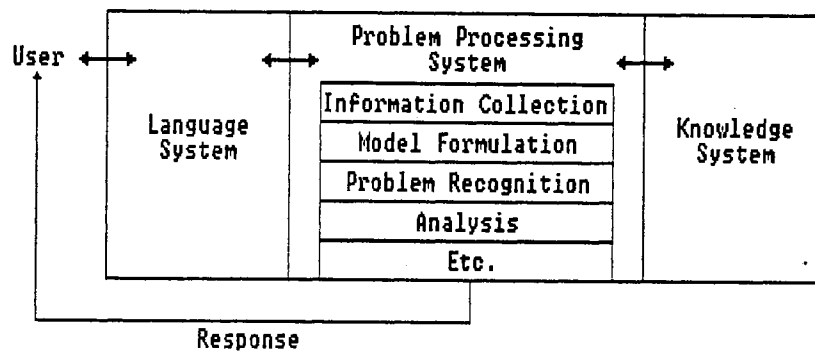


Figure 12. Decision Support System Architecture

The database management system lets the DSS user create, update, and perform inquiry and retrieval operations on the DSS database. This system is akin to the knowledge system proposed by Bonczek, Holsapple, and

Whinston (1980b). Successful DSSs require a database which is logically separate from other operational databases (Sprague 1980).

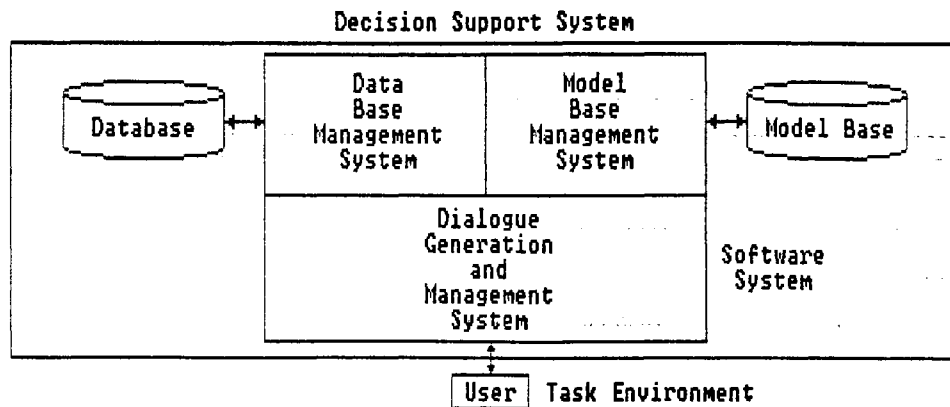


Figure 13. Decision Support System Architecture Revisited

Besides the components of a DSS, Sprague (1980) delineates three levels of DSS technology. The three levels of DSS technology are: (1) specific DSS; (2) DSS generator; and (3) DSS tools.

The system which genuinely supports the decision making process, an actual information systems application, is a specific DSS. On the other hand, a DSS generator is a package of hardware and software which provides the capacity to promptly and readily build a specific DSS. Finally, DSS tools are hardware or software elements which

ease the development of a specific DSS or a DSS generator. Figure 14 shows how these levels of technology are related.

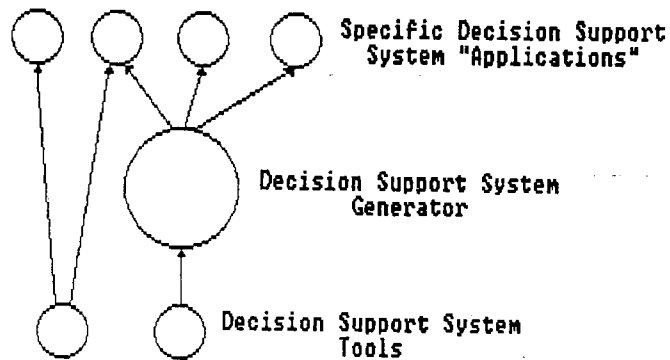


Figure 14. Decision Support System Levels of Technology

Sprague (1980) also specifies five evolving roles in DSS design and use as they relate to DSS technology (see Figure 15). These roles are:

- (1) manager/user;
- (2) intermediary;
- (3) DSS builder;
- (4) technical supporter; and
- (5) toolsmith.

The manager/user is the person faced with the problem for whom decision support is necessary. An intermediary is anyone who helps the manager/user. The DSS builder uses



the resources of a DSS generator to construct a specific DSS with which the manager/user or intermediary interacts directly. A technical supporter accumulates additional resources as needed for a DSS generator. The toolsmith develops new technology or improves the efficiency of existing technology for either specific DSS or DSS generators. Sprague (1980) emphasizes that a single individual may act in any given role at any given time.

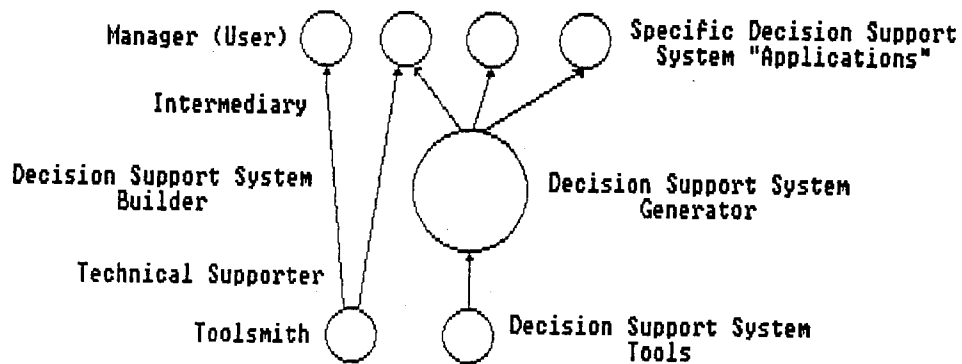


Figure 15. Decision Support System Roles

### Advantages of a Decision Support System Approach

Users reap several benefits when using a DSS. These advantages include:

- (1) decreased cost and time;
- (2) increased structuredness;
- (3) improved collaboration; and
- (4) changed focus of discussions.

Vazsonyi (1978) argues that the use of a DSS leads to decreased cost and time required to perform the various phases of decision making. In addition, a DSS increases the applicability and efficiency of structuring managerial situations. DSSs also improve the collaboration between the manager/user, operations research/MS, and the IS analyst. Finally, decision-analytic DSSs further improve discussion by letting decision makers focus on a quantitative model instead of each other (Adelman 1984).

#### Shortcomings of Decision Support System Designs

Existing DSSs have several drawbacks related to their design. These design shortcomings include:

- (1) modeling incompatibilities;
- (2) model updating;
- (3) data restructuring;
- (4) poor documentation; and
- (5) intermediary dependency.

Bonccek, Holsapple, and Whinston (1980a) argue that models are not easily combined. Generally, users do not develop models using modules that otherwise might be

combined to form other modules as the need arises. Furthermore, there is difficulty in updating models and modifying their uses. In addition, users must continually recollect and restructure data for each run of a model. This interrupts the communication between different models. Poor documentation characterizes DSSs according to Bonczek, Holsapple, and Whinston (1981).

Perhaps the greatest shortcoming of DSSs is the user's reliance on an intermediary. Andriole (1982) argues that this reliance leads to increasing man-computer alienation by the user. The intermediary becomes a surrogate problem solver and in doing so consciously or unconsciously manipulates the problem solving process. This results in a system which is not user understandable. The user adopts a machine rather than environmental orientation (Klein 1986). As a consequence, the user does not understand the modeling process because of lack of involvement.

#### Data Management System Concepts

The need to organize data in a well defined, rigorous manner has led to the development of many data models. A data model is a collection of mathematically well defined ideas that helps to consider and express the static and dynamic properties of data intensive applications. Brodie (1984) argues that data models and modeling concepts are central to information systems.

Thus, any data model applied within an information system must construct a representation which captures both static and dynamic processes. This implies that an information system must be capable of satisfying information requirements not only at design time, but also as these requirements change through time. The degree of success for various data models differs in this regard.

### Traditional Data Models

Three prevalent data models together form a class of models called the traditional data models. These models are: (1) the hierarchical data model; (2) the network data model; and (3) the relational data model. Historically, many practical applications have successfully used the traditional data models (Abiteboul and Hull 1987). Several distinguishing characteristics describe these record-based data models.

### Distinguishing Characteristics

According to Kent (1979), a record-based data model assumes that records provide an excellent tool for processing information that fits a certain pattern. A record is a fixed linear sequence of field values which conform to a static description. Generally speaking, field names have no semantic meaning and simply serve as placeholders for the data stored within the field.

Records, because of their predetermined length and static nature, tend to be machine-oriented constructs and provide a limited, yet desirable degree of flexibility. Of the three traditional data models, only the relational data model is discussed below because of its relevance to the present study.

### Relational Data Model

Codd (1970) proposed what is called the relational data model. The mathematical concept of relations serves as the basis for the relational data model. A relation is a set of tuples where this set varies over time. A tuple is simply the concatenation of a set of attributes. Each tuple in a given relation has the same set of attributes. The particular sequence of attributes within a tuple and tuples within a relation is irrelevant. The number of attributes defined for a relation is the degree of the relation. The value set from which attribute values are drawn is known as the domain of the attribute. Two or more attributes may have a common underlying domain. Another interpretation of a relation is that it is a subset of the Cartesian product of the domains across the various attributes.

Relations have two more properties beyond the ordering of attributes and tuples discussed above. The first is that, assuming all domains are atomic in that they are not

themselves relations, all entries in the relation are atomic values. Second, there is no duplication of tuples in a given relation.

Date (1986) uses several informal terms when referring to various formal relational data model definitions. The term table refers to a specific relation. A tuple is a row or record within a table. A column or field of a table is an attribute of the relation. A domain is the pool of legal values from which column values are drawn. Finally, one row is distinguished from another in a table using a unique identifier, called a primary key.

Figure 16 shows a relation called source. The source table (relation) has two unique rows (tuples). Furthermore, this table has three columns (attributes) and hence is of degree three. The first column name, Source Name, is distinct from the other two column names and serves as the unique identifier (primary key) for this table. Figure 17 shows a second relation called link. This table also has three columns, however, the first two columns form the unique identifier for this table. The concatenation of the unique identifiers of the tables participating in a relationship symbolically represents the relationship between two or more tables. The column values for the Source Name column and Destination Name column come from the same pool of legal values (domain). Date (1986) calls the list of attribute names for a relation the

heading of the relation. The body of a relation is the collection of tuples which comprise the relation.

<u>Source Name</u>	<u>Interpretation</u>	<u>Supply</u>
DAL	Dallas	20,000
CHI	Chicago	42,000

Figure 16. Source Relation Example

<u>Source Name</u>	<u>Destination Name</u>	<u>Link Cost</u>
DAL	PITTS	23.50
DAL	ATL	17.75
DAL	CLEV	32.45
CHI	PITTS	17.60
CHI	CLEV	25.75

Figure 17. Link Relation Example

A relational database is a time-varying collection of data which may be accessed and updated as if organized as a collection of time-varying tabular relations of assorted degrees defined on a given set of simple domains (Codd 1979). As a result, the relational data model consists of: (1) a collection of time-varying tabular relations with the properties discussed above; (2) insert, update, and delete rules formally known as entity and referential integrity rules; and (3) a relational algebra used both as a data definition and data manipulation language.

The relational data model emphasizes several advantages in its design (Clemons 1985). The relational data model is very easy to use because of its mathematical rigor in the definition of data representations, operators, and simplicity of data structures. Furthermore, there tends to be an absence of performance detail and implementation clutter. Binary and higher-order relationships between entities are captured with equal facility. One-to-one, one-to-many, and many-to-many relationships may be directly represented. The user perceives the data in a relational data model as tables and nothing but tables because of the foregoing advantages according to Date (1986). In addition, the relational operators available to users allow them to generate new tables from old tables.

### Limitations of the Traditional Data

#### Models

The primary purpose of a data model is to serve as a mechanism for representing data and relationships. Each of the traditional models fails to accomplish this objective in one significant way or another.

The hierarchical data model allows only one relationship, either directly or indirectly, to exist between two entities over time. Furthermore, no attributes for relationships may be represented as there is no need to



create names for relationships and, thus, there exist no entities to which to attach those attributes.

The network data model represents relationships as named sets where these names allow for the existence of several direct and indirect relationships between two entities. As with the hierarchical data model, however, there is no practicable support for attributes of relationships.

The relational data model represents entities and relationships using relations. This allows for the specification of attributes for relationships, however, there is limited support for semantics.

Kent (1979), in a discussion of the weaknesses of record-based models, identifies several pitfalls of such approaches. Regardless of how well record-based data models provide natural constructs for representing information which fits a specific pattern, certain information does not easily fit into a record structure. A result of this limitation is that record structures assume a horizontal and vertical homogeneity in data. Each record assumes horizontal homogeneity of a given type in that each contains the same fields; vertically in that a given field contains the same "kind" of information in each record. The solutions developed for the homogeneity problem tend to introduce problems in that data integrity is threatened, where such integrity is crucial, and the final data

structure employed bears little resemblance to the semantic structure of the underlying relationships. Furthermore, these solutions usually result in the creation of a predefined structure for dealing with entities which is very stable and thus violates the need for an evolving data model. Finally, a precise data model should distinguish carefully between the structure of entities being modeled and the various structures of names which might be associated with them.

Generally, there is an inability of the three models to capture the true meaning of the data organized within the model. Semantic modeling provides richer data structuring capabilities for database applications. This leads to the next evolution in terms of the direct representation of entities and relationships between entities as captured by the information system; a class of data models known as semantic data models.

### Semantic Data Models

The traditional data models may be classified as syntactic data models in that the structures employed fail to model the semantics of the information accurately and unambiguously as evidenced in the modeling environment. Hainaut and Lecharlier (1974) argue that such database systems have only a limited power of representation compared with the semantic structure of the information

describing a real system. A class of data models known as semantic data models evolved in order to capture more of the meaning of the data within the model itself. Semantic data models describe data in a very abstract and understandable manner. In other words, moving from traditional data modeling concepts to semantic data modeling achieves an evolutionary step away from the machine domain toward the problem domain. The definition of the structure of the data and the operational environment in which it exists is a concern of semantic data models (Hawryszkiewicz 1983). Several features which distinguish semantic data models from record-based data models are discussed below.

#### Distinguishing Characteristics

The need for conceptual schema design tools led to the introduction of early semantic data models. A conceptual schema could be designed with a semantic data model and then transformed into one of the traditional models for implementation. Semantic data models initially emphasized the need to model data relationships that arose in typical database applications because of this. Traditional data models, however, still lacked the power of representation afforded by a semantic approach to data modeling and, as a progression, semantic data modeling approaches to database systems were undertaken. Several distinguishing features

of semantic data models are (Hull and King 1987): (1) an increased separation of conceptual and physical components; (2) a decreased semantic overloading of relationship types; and (3) an availability of convenient abstraction mechanisms.

The access paths available to end users tend to mimic the logical structure of the database schema directly in record-based data models (Clemons 1985). In contrast, semantic data models allow users to focus their attention directly on abstract objects and, in turn, on the conceptual relationships modeled in a semantic schema. This results in an increased separation of conceptual and physical components.

Record-based data models provide only two or three constructs for representing data interrelationships whereas semantic data models provide several constructs. Thus, record-based data models tend to be semantically overloaded in that several types of relationships and entities must be represented by the same constructs. For example, entities and relationships in the relational data model must be represented using relations in both cases. This restriction is not apparent in a semantic data modeling environment.

Semantic data models provide a variety of convenient mechanisms for viewing and accessing the schema at different levels of abstraction. Semantic data models

provide a much richer framework for defining derived schema components and applying such constructs as aggregation, grouping, and generalization. Record-based data models tend to simulate objects and attributes by interrelating records of different types with such semantically meaningless mechanisms as logical and physical pointers.

### Semantic Data Model Components

Semantic data model components include (Date 1983, Hull and King 1987):

- (1) objects;
- (2) attributes;
- (3) type constructors;
- (4) generalization constructors; and
- (5) derived schema components;

An object is the actual entity of interest within the modeling environment. The idea of what comprises an object is usually confusing and "so we blithely define an object or entity as anything (concept, event, object, etc.) worth recording in the database that meets the information and processing requirements" (Brodie 1984, p. 23). The definition of what constitutes an object in the semantic data modeling literature is virtually identical to the one used in the O-O literature.

An object may have zero or more attributes. An attribute in a semantic data model is analogous to an

instance store in an O-O system. There are usually two dimensions of attributes identified: (1) degree of value; and (2) degree of owner. There are two differing degrees of value: (1) single valued; or (2) multivalued. A single valued attribute is an attribute owned by an object which has a single, identifiable value whether null or nonnull. On the other hand, a multivalued attribute is an attribute which may contain more than a single value whether null or nonnull. In Figure 18 the attribute hasAddress and isResidenceOf are examples of single valued and multivalued attributes, respectively.

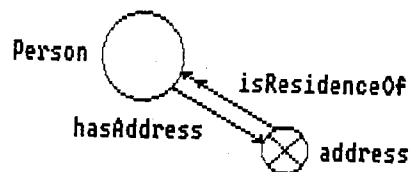


Figure 18. Attribute Example

The degree of owner refers to the object which owns the attribute and has two forms: (1) entity; or (2) type. An object which owns an attribute exclusively describing some characteristic of that object is an entity attribute. This is comparable to an existence store. By comparison, the attribute of an object defined over the class of that

object is known as a type attribute. This form of attribute is equivalent to a class store.

Semantic data models usually employ two type constructors: (1) aggregation; and (2) grouping, otherwise called association. Aggregation, formally presented by Smith and Smith (1977a, 1977b), allows a relationship between objects to be thought of as a higher-level, named object. Thus, aggregation is the process of combining low-level objects into composite objects expressed at a higher-level. The aggregate linkName, represented by a circle with an "x" through it in Figure 19, is an aggregation of sourceName and destinationName.

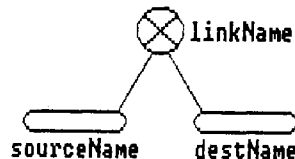


Figure 19. Aggregation Example

Grouping constructs a set of objects of the same type and corresponds to a single valued attribute of an object. The single valued attribute body, depicted in Figure 20 as a circle with an "\*" through it, is a grouping of tuples.

Generalization, an idea also introduced by Smith and Smith (1977b), is an abstraction construct in which a set of similar objects is regarded as a generic object and forms an is-a relationship between two objects. Thus, generalization expresses the relationship between a class and instances of that class. Two types of generalization are: (1) overlapping generalization; and (2) covering generalization. Overlapping generalization results in the partitioning of a generic class into various subclasses which have the potential to overlap. For example, in Figure 21 the superclass Vehicle has several subclasses: Motorized Vehicle; Land Vehicle; and Air Vehicle. Here an automobile belongs to the Motorized Vehicle and the Land Vehicle subclasses. Thus, the subclasses defining the superclass are not necessarily mutually exclusive.

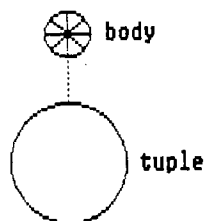


Figure 20. Grouping Example



Covering generalization results in the partitioning of classes into mutually exclusive and collectively exhaustive categories whereby the subclasses cover the superclass. Figure 22 presents an example of this form of generalization. In this instance there exists a superclass called Convoy. Several subclasses also exist such as Pacific Convoy and Atlantic Convoy. A ship, however, cannot physically belong to both convoys at once and, thus, the combination of both convoys covers the superclass.

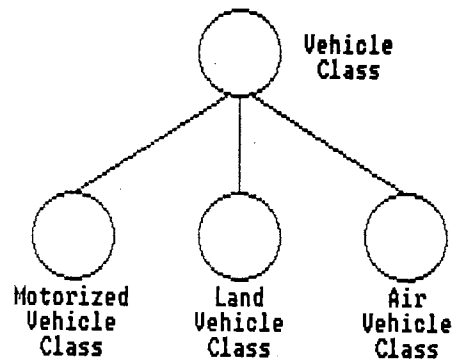


Figure 21. Overlapping Generalization Example

Finally, various semantic data models discuss the idea of derived schema components. A derived schema component requires the specification of the intension rather than the extension of the particular component. Historically, database management systems required users to specify the

extension of the database. Users simply specify the intension of the database and the extension of the database follows through this specification using the idea of derived schema components. As is seen in Figure 23 there are two types of derived schema components: (1) derived schema subtypes; and (2) derived schema attributes. Here the class object Pet Lover is a derived schema subclass defined as a pet owner who owns at least three pets. Furthermore, the derived schema attribute number is the cardinality of the set own for a specific pet lover.

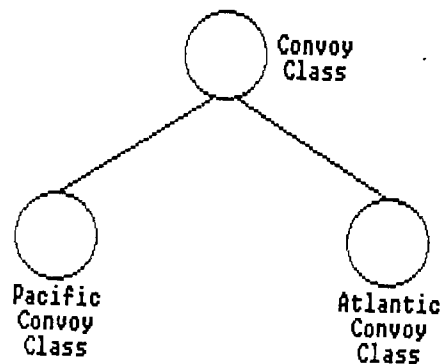


Figure 22. Covering Generalization Example

### Entity-Relationship Model

The entity-relationship model (E-R), proposed by Chen (1976), is one of the first truly semantic data models to

appear and is oriented toward user needs and expectations rather than machine efficiency (Bic and Gilbert 1986). The E-R model incorporates some of the important semantic information about the application environment and may be viewed as a generalization of the three traditional data models.

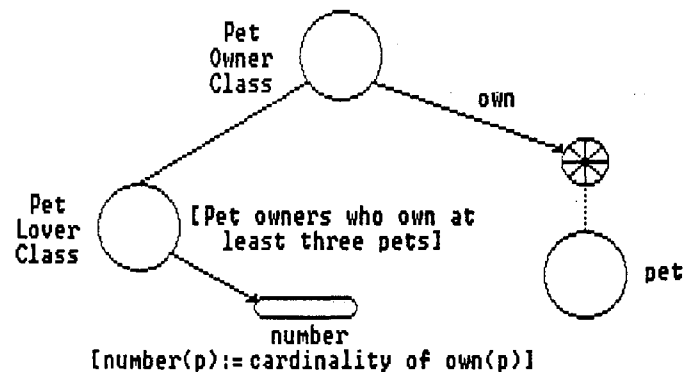


Figure 23. Derived Schema Components Example

The basic components of the E-R model are entity sets and relationship sets where each entity set and relationship set represents some generic classification of entities and relationships, respectively. The natural view that the world consists of entities and relationships serves as the basis for these ideas. Both entity sets and relationship sets may have properties, called attributes,

associated with them. There is a predicate associated with each entity set to test whether a particular entity belongs to it. A relationship set is a mathematical relation among some entities where each is taken from an entity set and each tuple of entities is a relationship.

Some other data model implements the actual database after the designer uses the E-R model as a database design tool. A pictorial design tool called E-R diagramming simplifies this design process.

E-R modeling in adopting a top-down approach, together with its various extensions and derivations, is a significant improvement over the traditional data models. Unfortunately, it is not always easy to categorize objects as either entities or relationships and, as a result, some information may not easily be captured as either an entity or a relationship.

#### Model Management System Concepts

Klein, Konsynski, and Beck (1985) define a model as any abstraction of reality applied to problem solving. Klein (1986) suggests that researchers develop procedures to make the management of models possible in order for future DSSs to fulfill the flexibility characteristic. According to Blanning (1983), a principle area of DSS research is the development of various frameworks for model

management systems. Typically these frameworks are similar to those developed for database management systems.

Keen (1980) notes that the assumption made by managers that most models are unrealistic, abstract, and intimidating is probably correct. Keen (1980) contends, however, that model management research has the potential to make models practical, concrete, and useful to managers. Managers have come to use models as instruments to transform data into information for aiding decision making. Thus, Dolk and Konsynski (1984) believe that models are another valuable resource, not unlike data, which must be managed.

The regard for models as an important organizational resource requiring effective management serves as the basis for much research into model management systems (Blanning 1983, Chung 1984). Model management provides a logical view of information that separates the users of the information from the physical aspects of information storage and processing (Blanning 1986).

Dolk (1986) identifies two levels of modeling activity in organizations: (1) informal; and (2) formal. Informal modeling occurs on an unplanned basis and usually is the result of individual resourcefulness. Formal modeling, however, is a direct result of organizational policy which defines and supports organizational planning, control, and operation. The organizational dimension of model

management is a consequence of the transition from informal to formal modeling. This transition solidifies the need to control the modeling resource.

Fuerst and Martin (1984) observe that an accepted precept of systems design is that user involvement in systems development leads to a higher probability of system success. Furthermore, user involvement ensures the opening and continuation of a communications channel which should lead to shared understanding. As a consequence, the process of defining the problem must be dominated by the managers involved. This allows the manager to address the correct problem and hence select the best model formulation. Vazsonyi (1978) points out that DSSs leave the problem structuring process to the manager. Thus, models tend to be individual and result from a modeling process as opposed to the application of a model.

Unfortunately, most modeling languages are written in computer languages which only computer programmers can understand. As a result, developers of model management systems should design systems in a top-down manner allowing for differing degrees of user expertise (Wang and Courtney 1984). Vazsonyi (1978, 1982) argues for the abstraction of the modeling process such that modelers may develop concrete objects to serve as model representations.

## Traditional Approaches to Model Management

Bu-Hulaiga and Jain (1988) identify several prevalent approaches to model management:

- (1) model abstraction;
- (2) structured modeling;
- (3) logic based approaches;
- (4) semantic networks;
- (5) graph based approach;
- (6) relational database approach; and
- (7) expert system and subroutine approaches.

Model representation, model selection, and model sequencing are the concern of model management systems. Most of these model management approaches provide model representation facilities but do not provide for model selection or sequencing. Various procedures for model selection and model sequencing, however, are present in the literature (Klein 1986, Bu-Hulaiga and Jain 1988). Several distinguishing characteristics of model management systems in general are described below.

### Model Management System Objectives

From the foregoing discussion, it is apparent that model management systems have three key objectives. These objectives are: (1) presentation of a semantically based modeling language; (2) incorporation of a flexible and

dynamic modeling component; and (3) centralization of model management functions.

Developing model representations using a semantically based modeling language allows the DSS user to model the environment in familiar terms. Thus, the modeling process is not restricted to programmers or technicians who are the only ones capable of understanding the modeling language employed. This ability to express models semantically leads to increased productivity and improved communications between model users (Lenard 1987).

The incorporation of a flexible and dynamic modeling component allows for the creation of modeling classes. The model management system permits users to create instances of these model classes dynamically for personal use. Flexible interfaces between models, data, and users allow the DSS to deal with much of the detail work done by the system. DSSs historically have provided a shareable data organization that is both static and intolerant from a model standpoint (Klein, Konsynski, and Beck 1985).

Finally, the need to centralize model management functions and insure the integrity, consistency, currency, and security of model bases in a multiuser environment is crucial (Applegate, Chen, Konsynski, and Nunamaker 1986). This need arises out of the realization that models are resources and, like other resources, require organizational centralization and control.



### Distinguishing Characteristics

Dolk and Konsynski (1984) argue that modelers should view model management systems as the counterpart of a database management system. As a consequence, the characteristics of a model management system are:

- (1) to manage a large number of model representations;
- (2) to establish independence between the model representation and problem solver invoked to solve the model;
- (3) to separate data representations from model representations; and
- (4) to provide flexible, easy access to model representations by non-modelers.

Dolk and Konsynski (1984) note that model management systems must be general enough to handle many different classes of models thereby requiring the system to handle a large number of model representations. The separation of model representation from problem solver allows for the development of a representation which does not a priori bias the representation scheme with a specific solution technique. The separation of data representations from model representations permits the database management system to fuel the model representation. This distinction, however, lets database users change data structures without requiring corresponding changes to model representations and vice versa. Finally, since most DSS users prefer to "do their own thing" according to their own way of

thinking, model management systems must be flexible enough to allow non-modelers easy access to model representations presented in natural terms (Wagner 1981).

### Structured Modeling

Structured modeling, proposed by Geoffrion (1987), aims to provide a formal mathematical framework and computer-based environment for conceiving, representing, and manipulating a wide variety of models. Structured modeling uses a hierarchically organized, partitioned, and attributed acyclic graph to represent a model or a model class. Structured modeling follows several guidelines in model development. These guidelines are:

- (1) incorporate important data development processes directly into the model;
- (2) document definitional interdependencies;
- (3) use stepwise refinement;
- (4) compose models from validated submodels; and
- (5) exploit parallel structure.

Structured modeling focuses on three basic structural levels. These levels are: (1) elemental structure; (2) generic structure; and (3) modular structure. Each of these structural levels is discussed at length below.

#### Elemental Structure

Discrete elements comprise a structured model. Elemental structure intends to capture all the definitional

detail of a specific model instance. Elements may call one another. Each call represents a definitional reference. In other words, a call shows the participation of one element's definition in the definition of another element. For all intents and purposes, a call shows a functional dependency between elements.

Geoffrion (1987) identifies five distinct elements in structured modeling. These elements are:

- (1) primitive entity;
- (2) compound entity;
- (3) attribute;
- (4) function; and
- (5) test.

A primitive entity element has no associated value and generally represents things or concepts postulated as primitives of the model. A specific source point or destination point in a transportation model is an example of a primitive entity.

Compound entity elements also have no associated value but represent things or concepts that are defined in terms of other things or concepts. In other words, compound entity elements represent a relationship between primitive entity elements. A link in a transportation model is an example of a compound entity. A source point primitive element and a destination point primitive element define a link compound element.

Geoffrion (1987) draws a distinction between two types of attribute elements. The first type, called fixed attribute element, has a constant value and generally represents properties of things or concepts. A variable attribute element, however, also generally represents properties of things or concepts but determines its value through the solution of the model. A source point supply amount is an example of a fixed attribute element. The flow across a link is an example of a variable attribute element.

Function elements have a value that is dependent according to a definite rule on the values of called elements, and generally represents calculable properties and more complex aspects of models. The total cost of a transportation model is an example of a function element.

Finally, identical in nature to function elements, test elements are boolean valued. The test to determine if demand requirements are met at a destination point in a transportation model is an example of a test element.

### Generic Structure

Generic structure focuses on capturing the natural familial groupings of elements. Geoffrion (1987) argues that, mathematically, this is accomplished by partitioning all elements of a given type into genera. Each element is a cell of the partition. The modeler uses the idea of

generic similarity, meaning that every element in a genus calls elements in the same foreign genera, to organize genera. Figure 24 shows a genus graph for a transportation model.

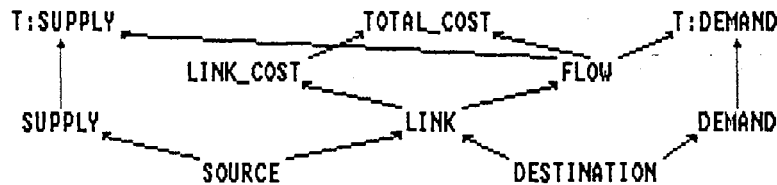


Figure 24. Transportation Model Genus Graph

The modeler communicates generic structure through a specific syntax developed by Geoffrion (1987). Each generic structure is encapsulated in a generic paragraph. Figure 25 shows two generic paragraphs, one for the source point primitive element generic structure and the other for the link compound element generic structure.

### Modular Structure

A modular structure attempts to organize generic structure hierarchically to the extent that this seems appropriate and useful. The basic notion is to group genera into conceptual units called modules. Geoffrion

(1987) argues that modelers should group modules into higher order modules according to some commonality or semantic relatedness. The modeler communicates modular structure through module paragraphs. Figure 26 is an example of a source data module paragraph from the transportation model example. Figure 27 is a modular structure for the transportation model.

```

SOURCEi /pe/ There is a list of SOURCES.

LINK(SOURCEi,DESTINATIONj) /ce/ Select {SOURCE} * {DESTINATION}
where i covers {SOURCE}, j covers {DESTINATION} There are some
transportation LINKS from SOURCES to DESTINATIONS. There must be at least
one LINK incident to each SOURCE, and at least one LINK incident to each
DESTINATION.

```

Figure 25. Source Point and Link Generic Paragraphs

```

&SDATA SOURCE DATA

SOURCEi /pe/ There is a list of SOURCES.

SUPPLY(SOURCEi) /a/ {SOURCE}: R+ Every SOURCE has a SUPPLY CAPACITY
measured in tons.

```

Figure 26. Source Data Module Paragraph

Structured modeling does not permit all forms of modular structure, however. Listed modular structures must satisfy a monotone ordering, that is, an indented list

representation with no forward references. Forward references exist when genera higher in the list call those lower in the list. Figure 28 is an example of a modular outline for the transportation model which satisfies the monotone ordering qualification.

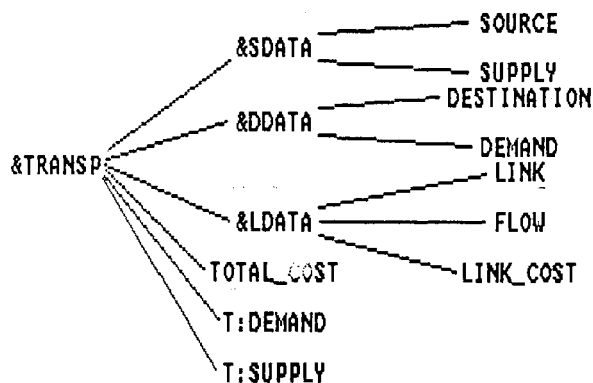


Figure 27. Transportation Model Modular Structure

### Structured Model

Thus, a structured model is an elemental structure together with a generic structure satisfying similarity and having a monotone modular structure. Figure 29 presents a model schema for the transportation model example.

```

&TRANSP
  &SDATA
    SOURCE
    SUPPLY
  &DDATA
    DESTINATION
    DEMAND
  &LDATA
    LINK
    FLOW
    LINK_COST
TOTAL_COST
T:DEMAND
T:SUPPLY

```

Figure 28. Transportation Model Modular Outline

**&TRANSP** TRANSPORTATION MODEL

**&SDATA** SOURCE DATA

**SOURCE<sub>i</sub>** /pe/ There is a list of SOURCES.

**SUPPLY(SOURCE<sub>i</sub>)** /a/ {SOURCE}: R+ Every SOURCE has a SUPPLY CAPACITY measured in tons.

**&DDATA** DESTINATION DATA

**DESTINATION<sub>j</sub>** /pe/ There is a list of DESTINATIONS.

**DEMAND(DESTINATION<sub>j</sub>)** /a/ {DESTINATION}: R+ Every DESTINATION has a nonnegative DEMAND measured in tons.

**&LDATA** LINK DATA

**LINK(SOURCE<sub>i</sub>,DESTINATION<sub>j</sub>)** /ce/ Select {SOURCE} \* {DESTINATION} where *i* covers {SOURCE}, *j* covers {DESTINATION} There are some transportation LINKS from SOURCES to DESTINATIONS. There must be at least one LINK incident to each SOURCE, and at least one LINK incident to each DESTINATION.

**FLOW(LINK<sub>ij</sub>)** /va/ {LINK}: R+ There can be a nonnegative transportation FLOW (in tons) over each LINK.

**LINK\_COST(LINK<sub>ij</sub>)** /a/ {LINK}: R Every LINK has a TRANSPORTATION COST RATE for use in \$/ton.

**TOTAL\_COST(COST, FLOW)** /f/;  $\sum_i \sum_j (\text{LINK\_COST}_{ij} * \text{FLOW}_{ij})$  There is a TOTAL COST associated with all FLOWS.

**T:SUPPLY(FLOW<sub>i.</sub>, SUPPLY<sub>i</sub>)** /t/ {SOURCE};  $\sum_j (\text{FLOW}_{ij}) (= \text{SUPPLY}_i$  Is the total FLOW leaving a SOURCE less than or equal to its SUPPLY CAPACITY? This is called the SUPPLY TEST.

**T:DEMAND(FLOW<sub>.j</sub>, DEMAND<sub>j</sub>)** /t/ {DESTINATION};  $\sum_i (\text{FLOW}_{ij}) = \text{DEMAND}_j$  Is the total FLOW arriving at a DESTINATION exactly equal to its DEMAND? This is called the DEMAND TEST.

Figure 29. Transportation Model Schema



Lenard (1987) proposes the use of structured modeling as a basis for a model management system. Lenard (1987) borrows ideas from O-O programming to help delineate the objects constituting a structured model. Lenard (1987) currently is constructing a prototype system with a restricted set of function rules and a limited range of operations to show the feasibility of using structured model management to manage at least linear programming models.

This approach applies structured modeling ideas directly, however, O-O programming is seen as a vehicle leading to its successful implementation. O-O notions are not directly applied to the model representations themselves.

## CHAPTER III

### OBJECT-ORIENTED RELATIONAL DATA MODEL MANAGEMENT SYSTEM

#### Introduction

This chapter presents the ideas of object-oriented (O-O) relational data modeling. O-O relational data modeling applies pertinent relational data modeling concepts (Codd 1970) using an O-O approach (LeClaire and Suh 1988). This allows users to treat relations as objects and exploit relational operators using messaging.

The O-O relational data model differs from O-O database models which support forms of local behavior in a manner similar to O-O programming languages (Hull and King 1987). The goal of O-O data modeling is to provide constructs for capturing more of the semantics of an application environment than is possible with a traditional data model (King 1984). The O-O relational data model presented simply provides O-O decision support system (DSS) users with access to a relational database management scheme.

## Object-Oriented Relational Data Modeling Fundamentals

The mathematical concept of relations serves as the basis for the relational data model developed by Codd (1970). A set of time varying tuples defines a relation. A tuple is simply the concatenation of a set of attributes. The same set of attributes comprises each tuple in a given relation. The particular sequence of attributes within a tuple and tuples within a relation is irrelevant. The domain of an attribute is the value set from which attributes draw their values. Two or more attributes may have a common domain.

Relations have two further properties. First, all entries in the relation represent atomic values. Second, there is no duplication of tuples. A subset of attributes in a relation serves to distinguish one tuple from another. Relational theory calls this subset of attributes the primary key of the relation. A relational database is a time-varying collection of data which the user accesses and updates as if organized as a collection of time-varying tabular relations of assorted degrees defined on a given set of simple domains (Codd 1979). Objectives of the relational data model include (Clemons 1985):

- (1) ease of use;
- (2) mathematical rigor in the definition of data representation and operators;

- (3) simplicity of data structures;
- (4) generality; and
- (5) absence of performance detail and implementation clutter.

We present an O-O view of the relational data model. This representation allows the users of an O-O DSS to interact with a relational data modeling component using an O-O approach. As a consequence, we combine the benefits of an O-O approach with those of the relational data model. An O-O approach to relational data modeling takes advantage of the aforementioned objectives by providing the user with an O-O representation of relations. This representation is achieved through simple data model abstraction which progresses from data model schema development to data model schema abstraction. These ideas are discussed below.

#### Object-Oriented Relational Data Model

##### Schema Development

Figure 30 presents an O-O relational data model (R-D-M) diagram. The octagon symbol represents the Source Relation class. This class object has the ability to create instance objects which represent tuples of the relation. Such an instance object is called a tuple instance object. Figure 31 depicts the Link Relation class. The collection of all specific relation classes represents the relational database (a relational database is a collection of time-varying relations). The collection

of instance objects within one of these classes represents all the tuples of the relation.

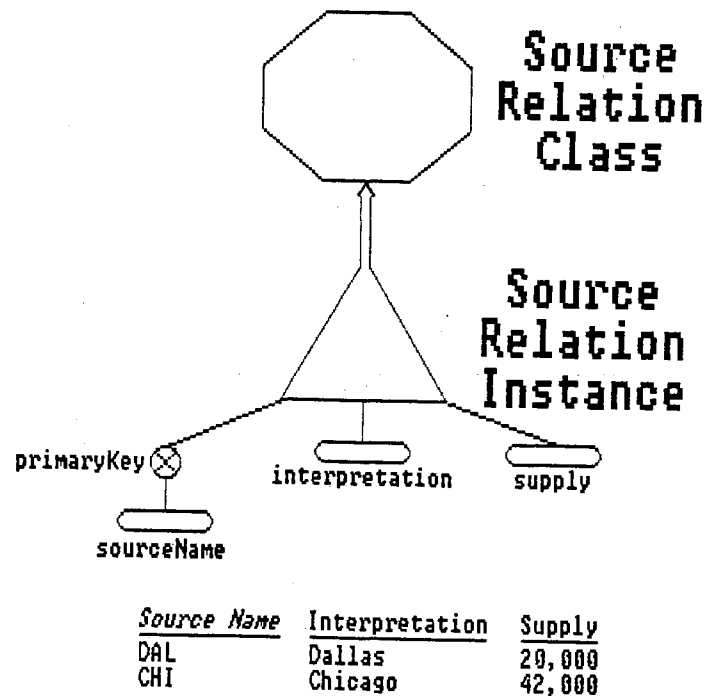


Figure 30. Source Relation Class

The data modeler expresses information concerning the contents of a specific relation class object (including its instance objects) using instance stores called attributes. An attribute obtains its value from the user through observation, measurement, or calculation. In Figures 30 and 31 an empty oval depicts a fixed attribute. A fixed attribute is user-determined and, thus, the user supplies

the attribute value. This type of attribute is identical to that proposed by Chen (1976).

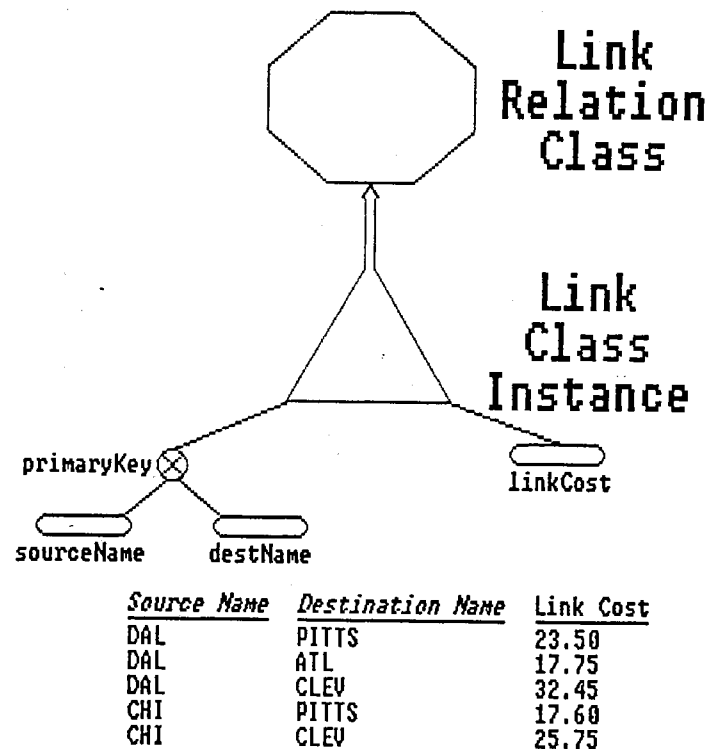


Figure 31. Link Relation Class

Certain restrictions on specific relation class object attributes are imposed, however. Specifically, no class attributes may be defined for any specific relation class object (e.g., Source Relation class or Link Relation class). Furthermore, all tuple instance object attributes for a specific relation class object must be fixed.

Specific relation class objects enforce uniqueness among their instance objects. That is, duplicate tuple instance objects are not allowed. Each tuple instance object has an aggregate attribute which serves as a unique identifier for the tuple instance object. This aggregation, called primaryKey, consists of zero or more attributes defined for the tuple instance object. The circle symbol in Figures 30 and 31 with an "x" through it refers to this aggregation. Aggregation, presented by Smith and Smith (1977a, 1977b), allows a relationship between objects to be thought of as a higher level object.

The value of the primaryKey aggregate is the aggregation of the values for the attributes defining the primaryKey aggregate. As a consequence, all tuple instances are distinguishable from one another based on this aggregate. The R-D-M diagram in Figure 30 shows that the sourceName identifier differentiates one tuple instance object from another. The unique instance identifier for the specific relation class appearing in Figure 31 is the aggregation of two attributes: (1) sourceName; and (2) destName. The primaryKey aggregate may be empty, in which case the specific relation class object takes the aggregate of all the instance object attributes to determine uniqueness.

Zero or more attributes may exist outside of the primaryKey aggregate. Two such attributes are defined for

the Source Relation class: (1) interpretation; and (2) supply. Only one non-key attribute is defined for the Link Relation class, linkCost. The aggregate of all instance attribute names defined for a specific relation class object is the heading of the relation. The body of a relation is the grouping of all tuple instance objects.

This approach to data management assumes that the user proceeds through the various stages of information requirements analysis, relational design, and normalization. The O-O relational data model simply provides access to a relational database management scheme.

#### Object-Oriented Data Model Schema

##### Abstraction

The second step in defining a specific relation class object is data model schema abstraction. Data model schema abstraction permits the user to specify data model particulars. First, however, a brief discussion of syntax notation is required. Figure 32 shows the syntax notation used in data model schema abstraction.

is fixed	Required Notation
<i>ATTRIBUTE-NAME</i>	User Specified Required Parameter
<i>[range]</i>	User Specified Optional Parameter

Figure 32. Data Model Schema Abstraction  
Syntax Notation



### Attribute Syntax

Relational data model schema abstraction begins at the attribute level. Figure 33 shows the abstraction syntax for the only allowable kind of attribute, fixed, appearing in a R-D-M diagram. The data modeler specifies the name of the attribute, such as ATTRIBUTE-NAME, and declares its kind as fixed.

*ATTRIBUTE-NAME is fixed of-type TYPE [range];*

Figure 33. Attribute Syntax

Attribute values are drawn from a given value set as specified by TYPE. Each attribute may have an optional range statement, [range], which follows the attribute type specification. Thus, a range statement may restrict the allowable set of attribute values for a given attribute.

Note that a semicolon (;) terminates a fixed attribute statement. All data model schema abstraction statements end in this manner. Figure 34 gives the general syntax for a R-D-M diagram abstraction. From this figure it is clear that the user of an O-O relational data model specifies

three items. These items are: (1) RELATION-NAME; (2) primaryKey aggregate list; and (3) non-key attribute list.

```

RELATION-NAME is-a relation:-
  instance attributes:-
    primaryKey is aggregate-of:-
      KEY ATTRIBUTE LIST;
    end primaryKey;
    NON-KEY ATTRIBUTE LIST;
  end instance;
end RELATION-NAME;

```

Figure 34. Relational Data Model Abstraction  
General Syntax

Figure 35 illustrates the data model schema abstraction for the Source Relation class and Link Relation class objects. The source data model schema abstraction is an example of a single attribute, sourceName, serving as a the primaryKey aggregate. The link data model schema abstraction uses two attributes, sourceName and destName, to define the primaryKey aggregate.

The source data model schema abstraction in Figure 35 has a non-key attribute list containing two attributes. These attributes are: (1) interpretation; and (2) supply. The interpretation attribute has no range qualifier whereas the supply attribute must be nonnegative. The link data model schema abstraction has a single non-key attribute, linkCost. This attribute cannot be negative as is evident

by the `[link_cost >= 0]` range qualifier. Note that in both data model schema abstractions the specific relation class object is an instance of the Relation class (e.g., source is-a relation).

```

source is-a relation:-
  instance attributes:-
    primaryKey is aggregate-of:-
      sourceName is fixed of-type string;
    end primaryKey;
    interpretation is fixed of-type string;
    supply is fixed of-type integer [supply >= 0];
  end instance;
end source;

link is-a relation:-
  instance attributes:-
    primaryKey is aggregate-of:-
      sourceName is fixed of-type string;
      destName is fixed of-type string;
    end primaryKey;
    linkCost is fixed of-type float [linkCost >= 0];
  end instance;
end link;

```

Figure 35. Data Model Schema Abstraction Example

What will become evident in subsequent chapters is that the process of schema development followed by schema abstraction is specifically chosen for the purpose of integrating data and model perspectives. That is, the user will be able to regard a specific relation class object as a data object or may treat that object in a manner identical to a model object.

## CHAPTER IV

### OBJECT-ORIENTED MODEL MANAGEMENT SYSTEM

#### Introduction

This chapter introduces the ideas of object-oriented (O-O) structured modeling. O-O structured modeling applies relevant structured modeling concepts (Geoffrion 1987) and model abstraction ideas for the purpose of model representation in a decision support system (DSS) environment using an O-O approach (LeClaire and Suh 1988). Additionally, we use pertinent semantic data modeling notions (LeClaire and Chahande 1988) to enhance the transition from a structured modeling approach for model representation to an O-O structured modeling one.

#### Object-Oriented Structured Modeling

##### Fundamentals

O-O structured modeling provides a formal framework and computer-based environment for conceiving, representing, and manipulating an assortment of models. As a result, the objectives of O-O structured modeling are identical to those of structured modeling (Geoffrion 1987).

Dolk (1988) identifies several characteristics which should be present in any modeling system. Modeling systems should support a conceptual framework which defines a general model structure. Such systems should enforce independence of model representation from both model solution operators and underlying data associated with specific model instances.

Furthermore, modeling systems should be able to capture a wide range of operations research/management science mathematical models and other conceptual models encountered in the database design and software engineering fields. This implies the need to support a general modeling life cycle. Finally, modeling systems must have full use of data management facilities as contained in database management systems.

O-O structured modeling has each of these characteristics. O-O structured modeling ensures this by following a three phase process in model development and representation. These phases are: (1) model schema development; (2) model schema abstraction; and (3) model acyclicity verification.

Each of these development phases is discussed in detail below. We use a simple model, the Hitchcock-Koopmans transportation model, as an explicative example of this process. The reader is cautioned, however, that O-O structured modeling can capture a wide range of models.

## Model Schema Development

The entity-relationship (E-R) model, introduced by Chen (1976), serves as the basis for model schema development. Chen (1976) proposes that database design users employ the E-R model for database design and description. The E-R model adopts a more natural view that the real world consists of entities and relationships. The application of an E-R approach to model development is not new (see Blanning 1986). Geoffrion (1987) notes that structured modeling subsumes E-R modeling.

Chen (1976) argues that an entity is any "thing" in the modeling environment which may be distinctly identified. For instance, a specific person, automobile, or dog is an entity. E-R modelers classify similar entities into entity sets. People, cars, and pets may serve as entity sets for the foregoing entities. There is a predicate associated with each entity set to test whether an entity belongs to that set. In the transportation model example there are two entity sets: (1) source points, perhaps from which finished goods originate; and (2) destination points, to which the finished goods arrive. Specific source points and specific destination points (e.g., Dallas and Denver) are examples of entities in the transportation model example.

A relationship set, according to Chen (1976), is a mathematical relation among  $n$  entities taken from an entity

set. Each tuple of entities in a relationship set represents a specific relationship. Thus, a relationship is an association among specific entities. For example, the project-worker relationship set relates the two entity sets employee and project. In the transportation model example a single relationship set exists, called link. Each link relates one source point with a single destination point.

A relationship set may have one of three possible mappings between entities in the participating entity sets. These mappings are: (1) 1:1; (2) 1:N; or (3) N:M. The first mapping (1:1) relates an entity in the first entity set with at most one entity in the second entity set. The relationship set marriage is an example of such a mapping. The second mapping (1:N) relates an entity in the first set with any number of entities in the second set but not vice versa. A relationship set such as department-employee has such a mapping if an employee may not belong to more than a single department. Finally, the third mapping (N:M) relates any number of entities in the first set with any number of entities in the second set. In the transportation model example the relationship set link has an N:M mapping of source point entities to destination point entities.

O-O structured modeling fosters a view that models consist of entities and relationships. This allows

modelers to design and describe models beginning with an E-R approach. As a consequence, O-O structured modeling adopts a top-down approach to model development using semantic information to organize the model representation. Model schema development begins with the identification of key entity sets and relationship sets encountered in the modeling environment.

There are two entity sets in the transportation model example (source points and destination points) and one relationship set (link) of mapping N:M. Figure 36 applies an E-R diagrammatic technique (see Chen 1976) to illustrate the transportation model entity sets and relationship set. A box denotes an entity set and a diamond symbolizes a relationship set. Notice that the diagram includes the mapping of the relationship.

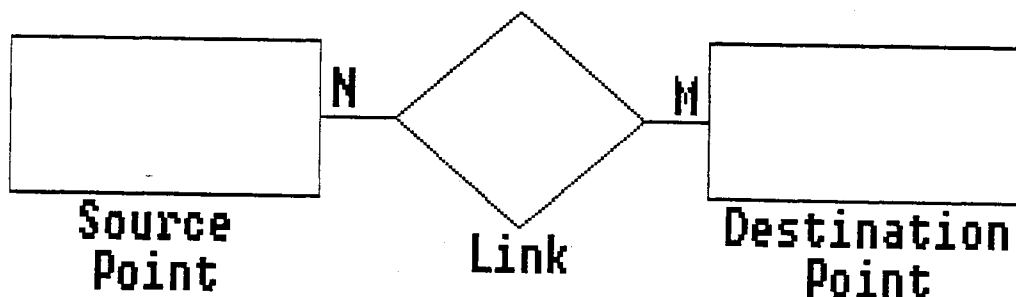


Figure 36. Entity-Relationship Diagram



The next step in model schema development is to draw a distinction between an entity/relationship set and members of that set. The O-O ideas of class object and instance object are important in showing this contrast. Figure 37 shows that what once were considered entity/relationship sets are now considered entity/relationship class objects. In this figure circles represent instance objects of each class object. The relationship of an instance object to its respective class object, known as an instance-of relationship, is shown using an arrow. Figure 37 is a simple Class-Instance (C-I) diagram. Two specific entity class objects, the Source Point class and the Destination Point class, appear as boxes in this figure. A single specific relationship class object also appears in Figure 37, called the Link class, and is shown using a diamond.

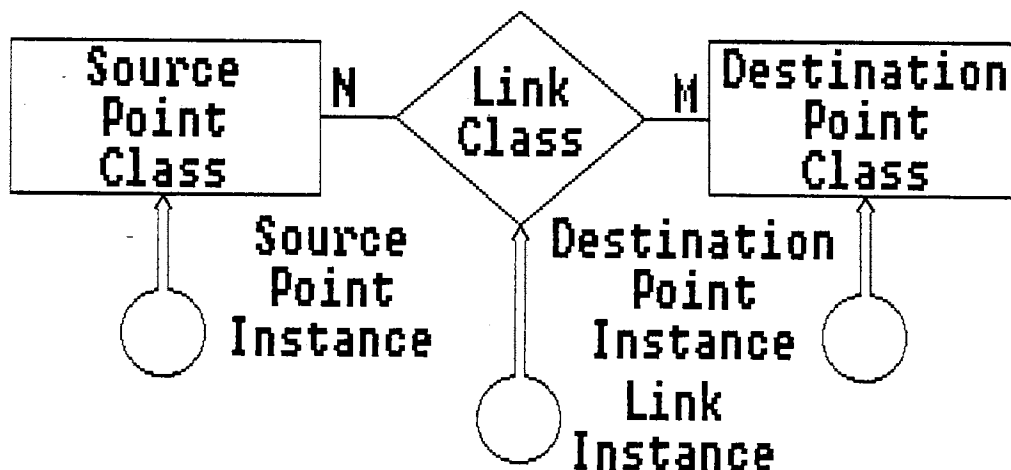


Figure 37. Simplified Class-Instance Diagram

The modeler expresses information about an entity or a relationship as an entity instance store or a relationship instance store called an attribute. An attribute obtains its value from the user through observation, measurement, or calculation. According to Chen (1976), attributes are drawn from a value set where different attribute values may come from the same value set.

Each entity instance object must have a unique identifier. This identifier is constructed using aggregation. Aggregation, formally presented by Smith and Smith (1977a, 1977b), allows a relationship between objects to be thought of as a higher-level, named object. An instance identifier is the aggregate of one or more instance level attributes defined for the specific entity class object. Thus, all instance objects of a specific entity class object are distinguishable from one another using this identifier.

Figure 38 shows a C-I diagram of the transportation model example which contains the relevant identifier aggregations. Ovals represent attributes in a C-I diagram and the circle symbol with an "x" through it refers to aggregation. The identifier aggregation for an instance of the Source Point class is the single instance attribute sourceName. The identifier aggregation of the Destination Point class is the single attribute destName.

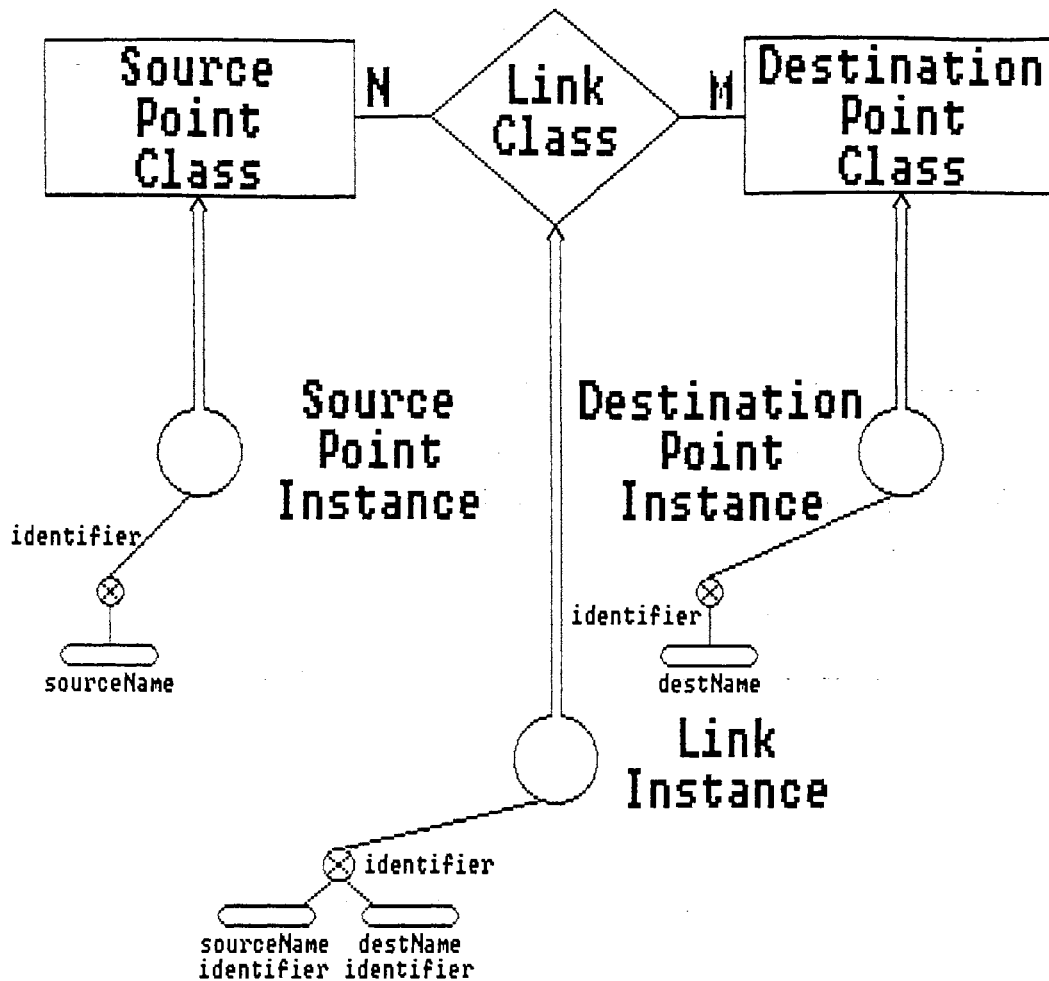


Figure 38. Class-Instance Diagram with Identifier Aggregates

The identifier aggregation for an instance object of the Link class requires some explanation. The relationship between a given Source Point class instance object and a given Destination Point class instance object may be regarded as an aggregate of the corresponding Source Point class and Destination Point class instance object identifiers. Since these identifiers are sourceName and

destName, respectively, they form the instance identifier used by the Link class.

Therefore, the aggregation of the entity instance object identifiers participating in a relationship represents the instance identifier for the specific relationship class object. The mapping of the relationship restricts the possible set of aggregate values permissible across all relationship instance objects.

Model schema development continues with the identification of the remaining specific entity instance and specific relationship instance object attributes. Figure 39 illustrates those entity/relationship instance object attributes determined to be relevant at this level of model development. Notice that a distinction is drawn between three different kinds of attributes.

An empty oval depicts a fixed attribute. A fixed attribute is user-determined and, thus, the user supplies the attribute value to the model. This type of attribute is identical to that proposed by Chen (1976) and a fixed attribute element as defined by Geoffrion (1987). The demand attribute of a Destination Point class instance object is an example of a fixed attribute.

An oval filled with diagonal lines represents a derived attribute. A derived attribute determines its value based on the value of other attributes in the model. The idea of derived schema attributes in a semantic data

modeling environment (LeClaire and Chahande 1988) serves as the basis for including derived attributes. O-O structured modeling can represent function and test elements from structured modeling (Geoffrion 1987) using derived attributes. The supplyTest attribute of a Source Point instance object is a derived attribute.

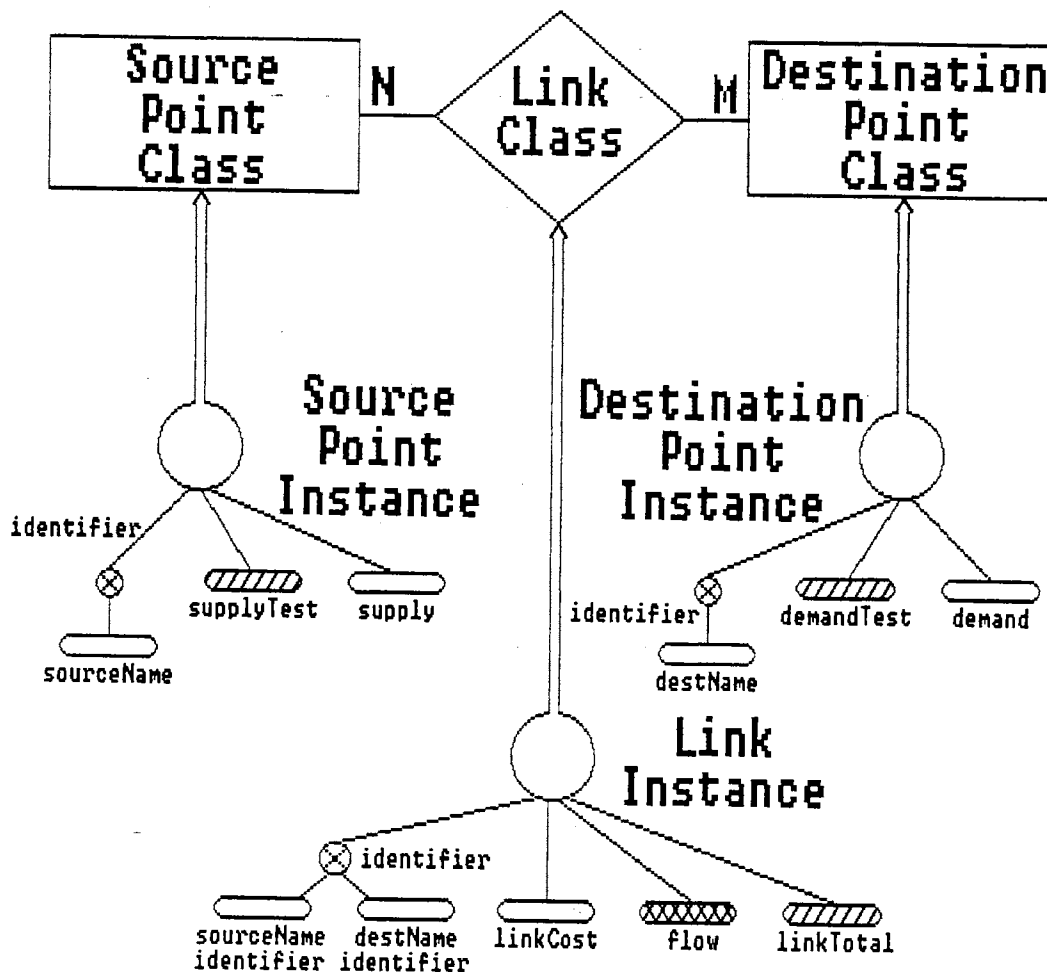


Figure 39. Model Class-Instance Diagram with Instance Attributes

Finally, a cross-hatched oval denotes a solver-derived attribute. A solver-derived attribute receives its value from the problem solver invoked to solve the model and, thus, is analogous to a variable attribute element from structured modeling (Geoffrion 1987). The flow attribute of a Link instance object is an example of a solver-derived attribute.

A C-I diagram is complete once the modeler adds specific entity class and specific relationship class object attributes. A specific class object attribute records information regarding all instances of the specified class using class stores. A specific class object attribute may also record information specific to the object using existence stores. Figure 40 presents several entity/relationship class object attributes. The derived attribute demandTotal, an attribute of the Destination Point class object, records the total demand for all Destination Point instance objects.

The addition of a specific model class object and its corresponding instance object representation to a C-I diagram transforms it into a C-I-Model (C-I-M) diagram. Figure 41 shows a simplified C-I-M diagram of the transportation model example. An octagon and a triangle represent specific model class objects and specific model instance objects, respectively. As with specific entity class and specific relationship class objects and instances

of those specific class objects, a specific model instance object uses an instance-of arrow to indicate its relationship to its class object. Furthermore, a C-I-M diagram also requires model instance objects to have a unique identifier aggregation. The aggregation of a single instance attribute, modelName, serves as the instance identifier in Figure 41.

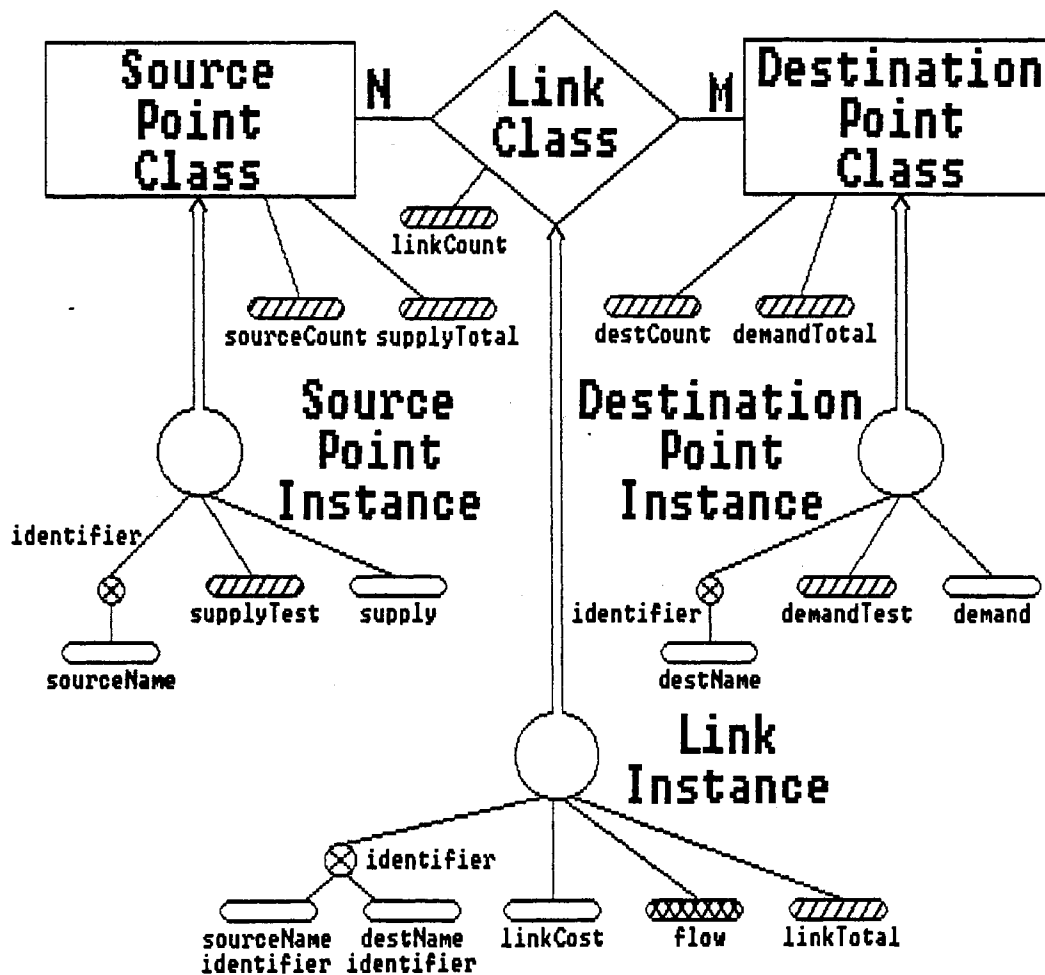


Figure 40. Complete Class-Instance Diagram

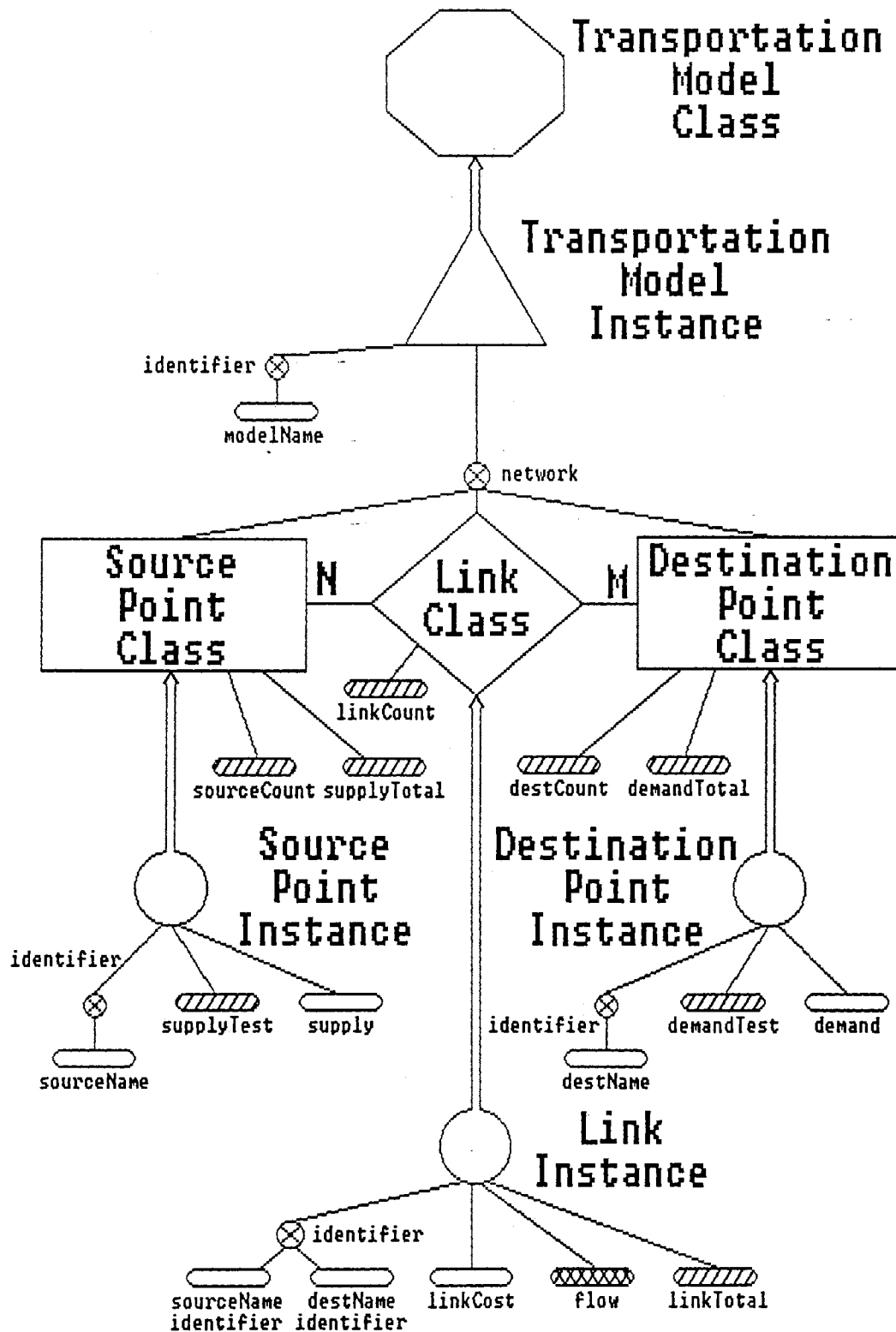


Figure 41. Simplified Class-Instance-Model Diagram



A single, named attribute of each specific model instance object serves as the aggregation of all entity/relationship class objects. Figure 41 shows this as the network attribute of a Transportation Model instance object. Figure 42 includes the addition of all specific model instance object attributes. Figure 42 adds a single derived attribute, totalCost.

The final step in model schema development is to include specific model class object attributes. This completes model schema development. Figure 43 shows the final transportation model schema. Note that this figure depicts only one model class object attribute, modelCount.

Model schema development is not a linear process and, as a result, several iterations may be necessary. The stepwise nature of this process, however, helps to ensure successful schema development. Figure 44 describes each step necessary in model schema development. Figure 45 shows a complete model schema for a general linear programming model. The next phase of model development is model schema abstraction. This is discussed in the following section.

#### Model Schema Abstraction

Dolk and Konsynski (1984) introduced model abstraction as one approach to model management. Dolk and Konsynski (1984) argue that model abstraction regards models as data

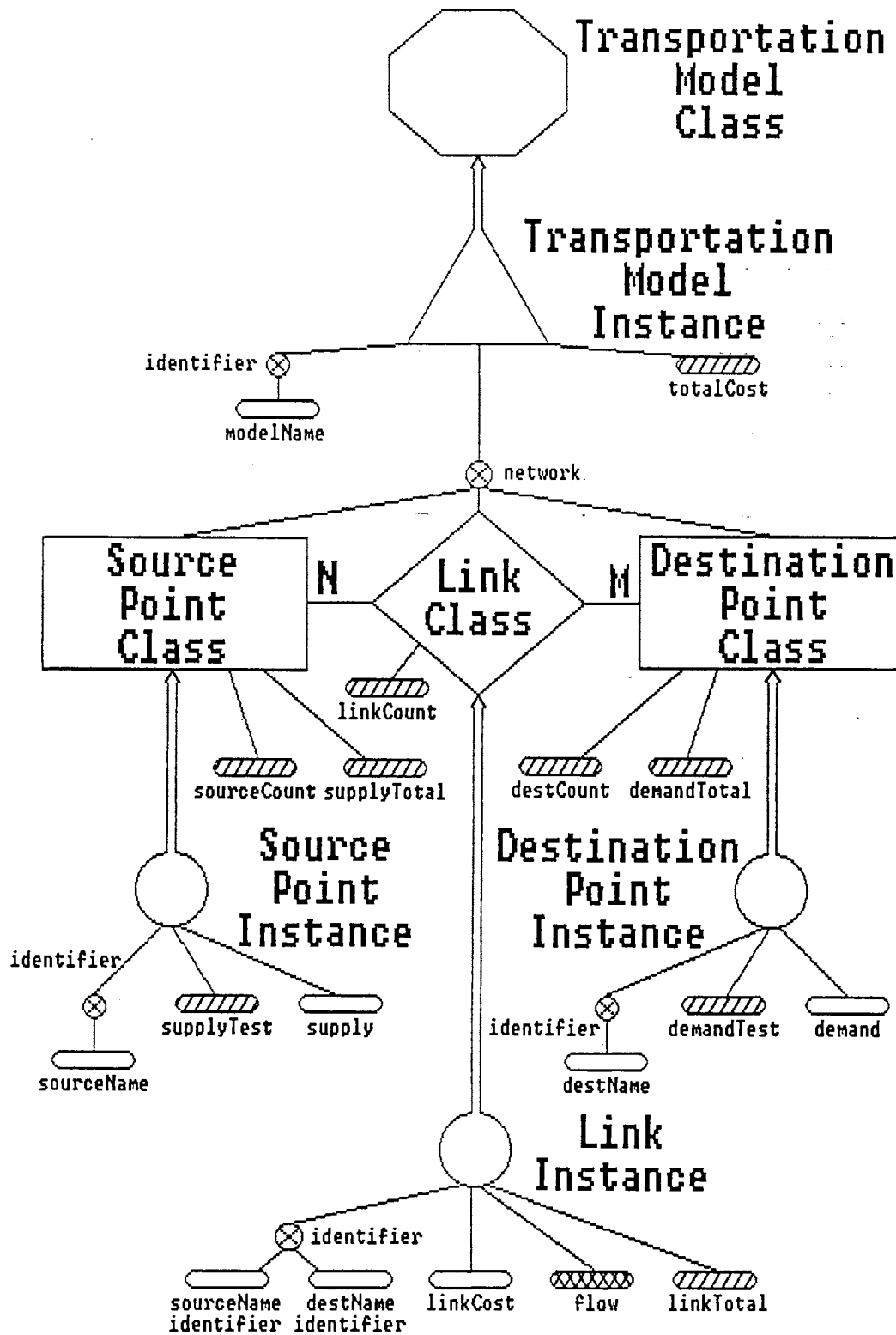


Figure 42. Class-Instance-Model Diagram with Instance Attributes

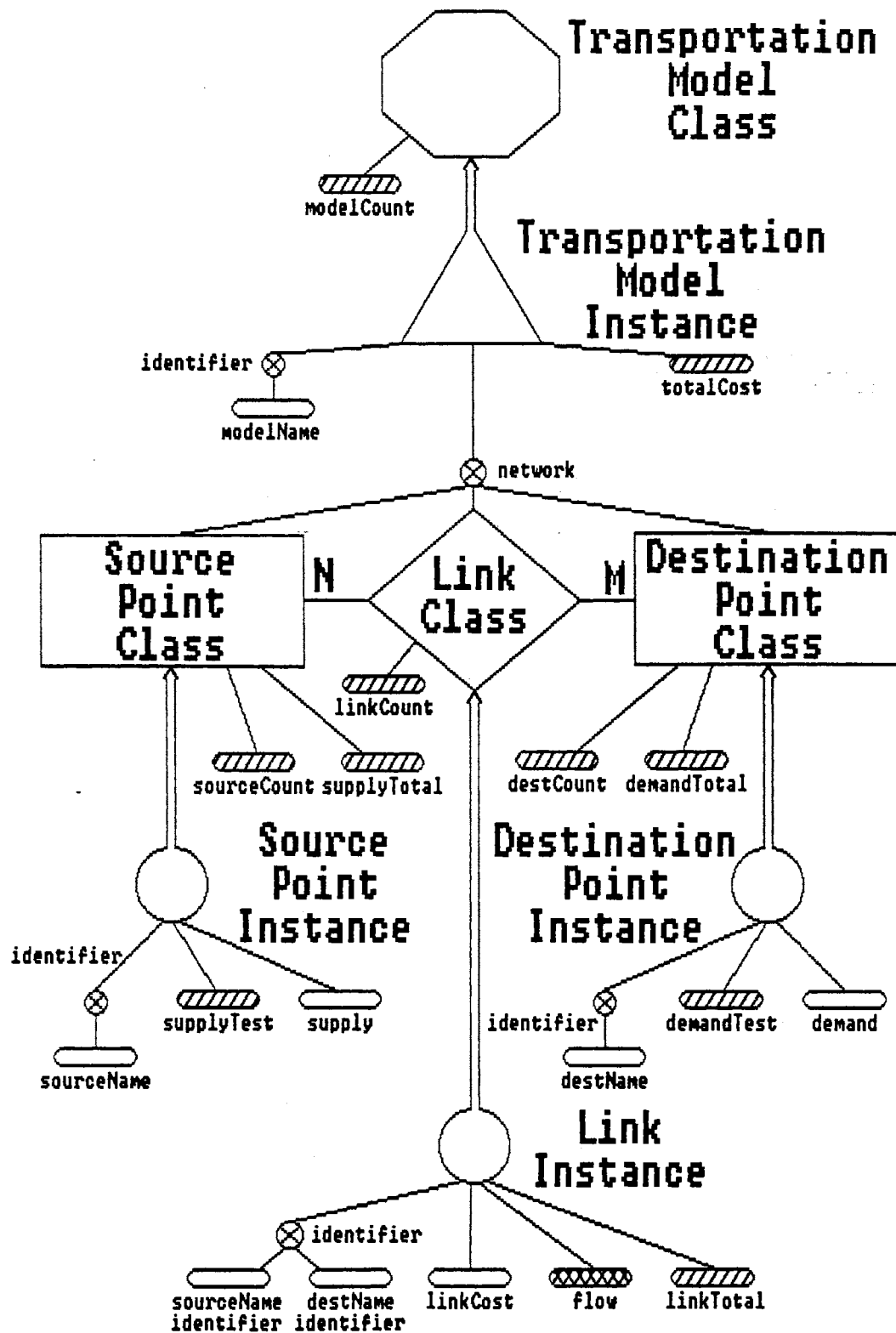


Figure 43. Complete Class-Instance-Model Diagram

- 1.) Identify key entities and relationships between those entities which participate in the model
  - a.) Determine the mapping of each relationship (e.g., 1:1, 1:N, or N:M)
  - b.) Diagram each entity and the relationships between the various entities using an entity-relationship (E-R) diagram
- 2.) Draw a distinction between entity/relationship classes and instances of those classes
  - a.) Diagram this distinction using a modified E-R diagram known as a Class-Instance-(C-I) diagram
  - b.) For each entity instance determine an aggregation of attributes which are to serve as the unique identifier for each instance and add it to the diagram
  - c.) For each relationship add an aggregation, to serve as the instance identifier, consisting of the entity instance identifiers participating in the relationship instance; the value of each relationship identifier is subject to the mapping restrictions
  - d.) For each entity/relationship instance determine the remaining attributes and add them to the diagram
  - e.) For each entity/relationship class determine the appropriate attributes at this level and add them to the diagram
- 3.) Move from the C-I level to the model level
  - a.) Diagram this using a modified C-I diagram known as a C-I-Model (C-I-M) diagram
  - b.) Determine an aggregation of attributes which are to serve as the unique identifier for each instance and add it to the diagram
  - c.) Regard the aggregate of all entities and relationships between those entities in the C-I diagram as a named attribute of the model instance
  - d.) Determine all other instance attributes and add them to the diagram
  - e.) Determine the model class attributes and add them to the diagram

Figure 44. Model Schema Development Steps

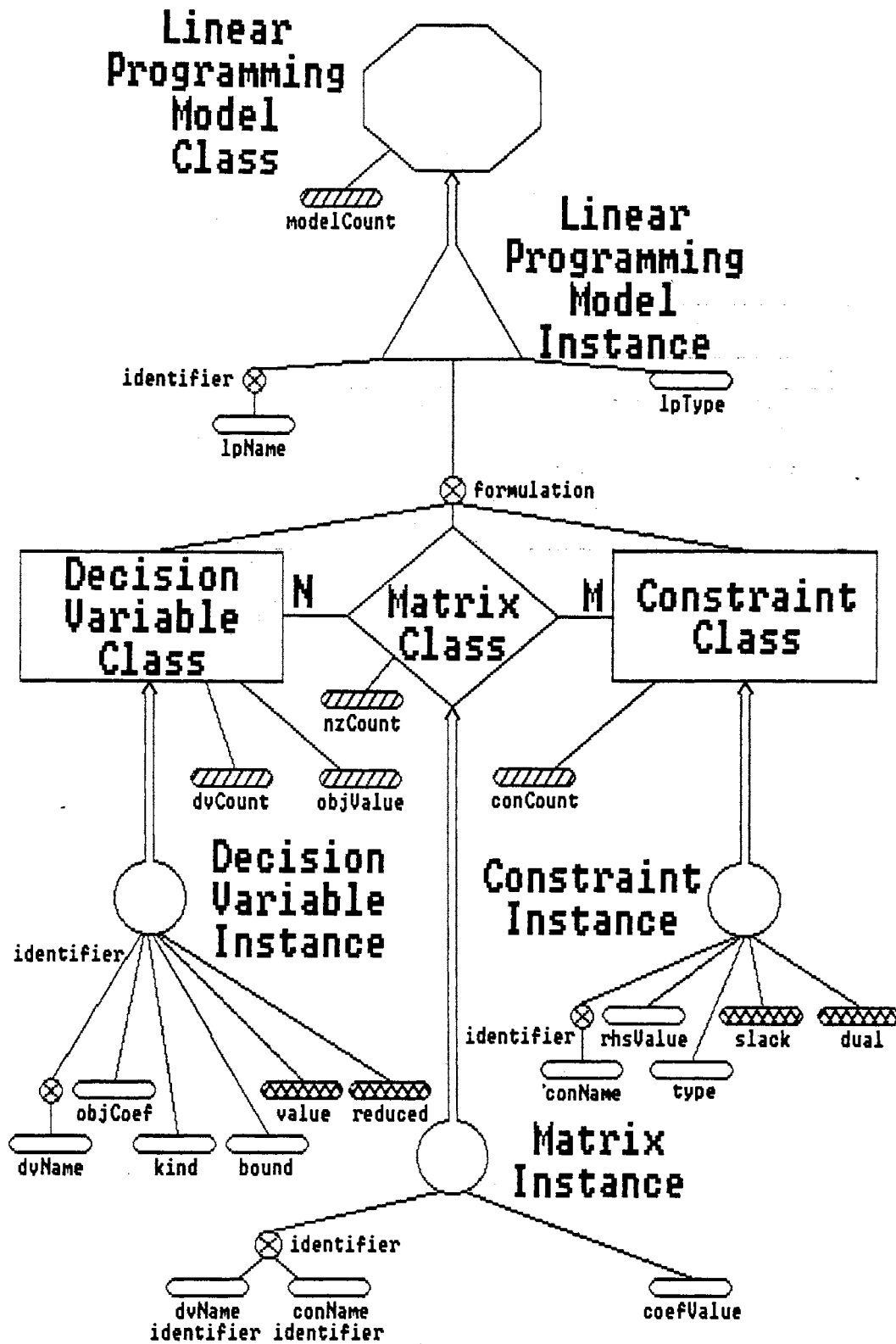


Figure 45. General Linear Programming Class-Instance-Model Diagram

and model management as a corollary of data management, both conceptually and in implementation terms.

First, a brief discussion of syntax notation is required. Figure 32 shows the syntax notation used in data model schema abstraction. This same syntax notation is relevant for model schema abstraction.

### Attribute Syntax

Model schema abstraction begins at the attribute level. Figure 46 shows the model abstraction syntax for the various kinds of attributes appearing in a C-I-M diagram. These attribute kinds are: (1) fixed; (2) derived; and (3) solver-derived.

The modeler specifies the name of an attribute, such as ATTRIBUTE-NAME in Figure 46, and declares its kind. The modeler may declare one of three different kinds for each attribute. This declaration must be one of the following: (1) is fixed of-type; (2) is derived of-type; or (3) is solver-derived of-type.

```

ATTRIBUTE-NAME is fixed of-type TYPE [range];

ATTRIBUTE-NAME is derived of-type TYPE [range]:-
  ATTRIBUTE-NAME := DERIVATION;
end ATTRIBUTE-NAME;

ATTRIBUTE-NAME is solver-derived of-type TYPE [range];

```

Figure 46. Attribute Syntax

Attribute values, regardless of kind, are drawn from a given value set as specified by TYPE. Each attribute may have an optional range statement, [range], which follows the attribute type specification. Thus, the range statement may restrict the set of allowable attribute values for a given attribute.

Note that a semicolon (;) terminates each statement for fixed attributes and solver-derived attributes. All model abstraction statements end in this manner. The derived attribute statement is a compound statement and, thus, ends with a continuation symbol (: -). This marks the beginning of an attribute derivation. An attribute derivation describes how a particular attribute determines its value. An end statement terminates a derived attribute statement. Figure 47 gives examples of the various kinds of attribute syntax that are encountered in the abstraction of the transportation model example.

```
supply is fixed of-type integer [supply >= 0];  
linkTotal is derived of-type float [linkTotal >= 0]:-  
  linkTotal := linkCost * flow;  
end linkTotal;  
flow is solver-derived of-type integer [flow >= 0];
```

Figure 47. Attribute Syntax Examples

## Entity Object Syntax

The modeler uses a specific syntax to develop the abstract representation for each entity. Figure 48 depicts the formal syntax for an entity in a C-I-M diagram.

```

ENTITY-NAME is-a entity:-
  class attributes:-
    ATTRIBUTE LIST;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      IDENTIFIER ATTRIBUTE LIST;
    end identifier;
    NON-IDENTIFIER ATTRIBUTE LIST;
  end instance;
end ENTITY-NAME;

```

Figure 48. Entity Syntax

Each specific entity abstraction begins with an entity class name. This name serves to identify the specific entity class as it is a subclass of the more general Entity class. This relationship is indicated by the ENTITY-NAME is-a entity statement. Thus, both the Source Point class and Destination Point class are subclasses of this more general class.

In addition, note that a named entity abstraction is a compound statement (there is a continuation symbol). No formal definition is given for a named entity class object except to say that it is an object which has class level



attributes and instance level attributes. This implies that an entity class object has the ability to instantiate new instances having the instance level attributes.

A class level attribute definition section appears following the class name declaration for the specific entity class object. The class attributes statement within which a list of attributes appears announces a class level attribute definition section. The modeler terminates the class level attribute section with an end class statement.

Instance object attribute definitions begin with the instance attributes statement. The instance object identifier aggregation definition immediately follows the instance attributes statement. Furthermore, any attribute defined within this aggregation statement must be a fixed attribute. The modeler may restrict the value of any of these attributes using a range statement, however. A list of additional instance level attributes follows this statement. An end instance statement concludes the instance level attribute section.

An end statement completes a specific entity class object definition. Figure 49 gives an example entity syntax for the Source Point class in the transportation model example.

```

sourcePoint is-a entity:-
  class attributes:-
    sourceCount is derived of-type integer [sourceCount >= 0]:-
      sourceCount := cardinality of instances;
    end sourceCount;
    supplyTotal is derived of-type integer [supplyTotal >= 0]:-
      supplyTotal := SUM (supply[*]);
    end supplyTotal;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      sourceName is fixed of-type string;
    end identifier;
    supplyTest is derived of-type boolean:-
      supplyTest := (SUM (flow[sourceName,*]) <= supply);
    end supplyTest;
    supply is fixed of-type integer [supply >= 0];
  end instance;
end sourcePoint;

```

Figure 49. Entity Syntax Example

### Relationship Object Syntax

Each relationship in a model also has a specific syntax used to develop its abstract representation. Figure 50 depicts the formal syntax for a relationship in a C-I-M diagram.

```

RELATIONSHIP-NAME is-a relationship [NAME LIST]:-
  class attributes:-
    ATTRIBUTE LIST;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      IDENTIFIER ATTRIBUTE LIST[exists];
    end identifier;
    NON-IDENTIFIER ATTRIBUTE LIST;
  end instance;
end RELATIONSHIP-NAME;

```

Figure 50. Relationship Syntax

Notice that there is substantial similarity to an entity abstraction. Each relationship class is named and is a subclass of the Relationship class (as is evident by the RELATIONSHIP-NAME is-a relationship statement), has class level attributes, and instance level attributes. There are two important differences, however. First, the specific relationship class object declaration statement includes [NAME LIST]. Second, each attribute appearing in the identifier aggregate has the [exists] qualifier appended to its definition.

The name list for a specific relationship class object consists of two items for each specific entity class object participating in the definition of the relationship. First, the class name of the specific entity class object is given. Second, the mapping of the specific entity class object follows its name. Each of these items is separated with a colon (:). The mapping specification is one of two options: (1) one; or (2) many.

Each entry of a specific entity class object appearing in the name list is separated using a comma (,). For example, in Figure 51 the Link class object has as its name list: [sourcePoint:many,destPoint:many]. This indicates that two specific entity class objects, sourcePoint and destPoint, participate in the link relationship with a mapping of N:M (many to many).

```

link is-a relationship [sourcePoint:many,destPoint:many]:-
  class attributes:-
    linkCount is derived of-type integer [linkCount >= 0]:-
      linkCount := cardinality of instances;
    end linkCount;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      sourceName is fixed of-type string [exists];
      destName is fixed of-type string [exists];
    end identifier;
    linkCost is fixed of-type float [linkCost >= 0];
    flow is solver-derived of-type integer [flow >= 0];
    linkTotal is derived of-type float [linkTotal >= 0]:-
      linkTotal := linkCost * flow;
    end linkTotal;
  end instance;
end link;

```

Figure 51. Relationship Syntax Example

The modeler explicitly defines the relationship instance object identifier as an aggregation using the is aggregate-of statement. An instance attribute definition list follows this statement. This list replicates the identifier attribute definition lists of the specific entity objects participating in the relationship. The [exists] statement qualifies each identifier attribute listed thereby ensuring that no reference to a nonexistent specific entity instance object occurs. An end statement terminates the identifier aggregation. All the ideas discussed above are shown in Figure 51.

### Model Object Syntax

The modeler uses a specific syntax to develop an abstract representation for each model. Figure 52 depicts the formal syntax for a model in a C-I-M diagram.

```

MODEL-NAME is-a model:-
  class attributes:-
    ATTRIBUTE LIST;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      IDENTIFIER ATTRIBUTE LIST;
    end identifier;
    NON-IDENTIFIER ATTRIBUTE LIST;
  AGGREGATE-NAME is aggregate-of:-
    ENTITY-NAME is-a entity:-
      class attributes:-
        ATTRIBUTE LIST;
      end class;
      instance attributes:-
        identifier is aggregate-of:-
          IDENTIFIER ATTRIBUTE LIST;
        end identifier;
        NON-IDENTIFIER ATTRIBUTE LIST;
      end instance;
    end ENTITY-NAME;
  RELATIONSHIP-NAME is-a relationship [NAME LIST]:-
    class attributes:-
      ATTRIBUTE LIST;
    end class;
    instance attributes:-
      identifier is aggregate-of:-
        IDENTIFIER ATTRIBUTE LIST[exists!];
      end identifier;
      NON-IDENTIFIER ATTRIBUTE LIST;
    end instance;
  end RELATIONSHIP-NAME;
end AGGREGATE-NAME;
end instance;
end MODEL-NAME;

```

Figure 52. Model Syntax

As with the entity and relationship syntaxes, each specific model class object declares a class name. This is shown in Figure 52 by the MODEL-NAME is-a model statement. Note that any named model class object is an instance of the generic Model class object. Every specific model class

object has the ability to instantiate new instances which have the defined instance level attributes.

Each specific model class object has a class level attribute section and an instance level attribute section. The specific model class definition is not unlike a specific entity definition with one exception. An is aggregate-of statement defines a named aggregation of all the specific entity and specific relationship class objects diagrammed in the C-I diagram. The modeler specifies an entity/relationship definition list in this aggregation. This list contains nothing more than the entity abstractions and relationship abstractions of the objects found in the C-I diagram. An end statement terminates the specific entity and specific relationship aggregation.

An end statement also terminates a specific model abstraction. Figure 53 shows the complete model schema abstraction for the transportation model example. Figure 54 presents the complete model schema abstraction for the general linear programming model diagrammed in Figure 45.

#### Model Abstraction Benefits

Some of the benefits of model abstraction identified by Dolk and Konsynski (1984) are: (1) it enforces the separation of model description and model solution; (2) it enforces model and data independence; and (3) it provides

```

transportationModel is-a model:-
  class attributes:-
    modelCount is derived of-type integer [modelCount >= 0]:-
      modelCount := cardinality of instances;
    end modelCount;
  end class;
instance attributes:-
  identifier is aggregate-of:-
    modelName is fixed of-type string;
  end identifier;
  totalCost is derived of-type float [totalCost >= 0]:-
    totalCost := SUM (linkTotal[*]);
  end totalCost;
  network is aggregate-of:-
    sourcePoint is-a entity:-
      class attributes:-
        sourceCount is derived of-type integer [sourceCount >= 0]:-
          sourceCount := cardinality of instances;
        end sourceCount;
        supplyTotal is derived of-type integer [supplyTotal >= 0]:-
          supplyTotal := SUM (supply[*]);
        end supplyTotal;
      end class;
      instance attributes:-
        identifier is aggregate-of:-
          sourceName is fixed of-type string;
        end identifier;
        supplyTest is derived of-type boolean:-
          supplyTest := (SUM (flow[sourceName,*]) <= supply);
        end supplyTest;
        supply is fixed of-type integer [supply >= 0];
      end instance;
    end sourcePoint;
    destPoint is-a entity:-
      class attributes:-
        destCount is derived of-type integer [destCount >= 0]:-
          destCount := cardinality of instances;
        end destCount;
        demandTotal is derived of-type integer [demandTotal >= 0]:-
          demandTotal := SUM (demand[*]);
        end demandTotal;
      end class;
      instance attributes:-
        identifier is aggregate-of:-
          destName is fixed of-type string;
        end identifier;
        demandTest is derived of-type boolean:-
          demandTest := (SUM (flow[*],destName) == demand);
        end demandTest;
        demand is fixed of-type integer [demand >= 0];
      end instance;
    end destPoint;
  end network;
end transportationModel;

```

Figure 53. Transportation Model Schema Abstraction

```

link is-a relationship [sourcePoint:many,destPoint:many]:-
  class attributes:-
    linkCount is derived of-type integer [linkCount >= 0]:-
      linkCount := cardinality of instances;
    end linkCount;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      sourceName is fixed of-type string [exists];
      destName is fixed of-type string [exists];
    end identifier;
    linkCost is fixed of-type float [linkCost >= 0];
    flow is solver-derived of-type integer [flow >= 0];
    linkTotal is derived of-type float [linkTotal >= 0]:-
      linkTotal := linkCost * flow;
    end linkTotal;
  end instance;
end link;
end network;
end instance;
end transportationModel;

```

Figure 53 (Continued). Transportation Model Schema Abstraction

for the development of a model management system as an analog of a database management system.

The combination of model schema development and model schema abstraction addresses the required characteristics of a model management system. Thus, O-O structured modeling serves as a useful medium for model management. One last issue addressed by structured modeling is model cyclicity. O-O structured modeling examines this in the next section.

#### Model Acyclicity Verification

Structured modeling uses acyclic directed graphs to display the model schema thereby assuring the acyclicity of



```

generalLPModel is-a model:-
  class attributes:-
    modelCount is derived of-type integer [modelCount >= 0]:-
      modelCount := cardinality of instances;
    end modelCount;
  end class;
  instance attributes:-
    identifier is aggregate-of:-
      lpName is fixed of-type string;
    end identifier;
    lpType is fixed of-type string [(lpType == 'max') or (lpType == 'min')];
    formulation is aggregate-of:-
      decisionVariable is-a entity:-
        class attributes:-
          dvCount is derived of-type integer [dvCount >= 0]:-
            dvCount := cardinality of instances;
          end dvCount;
          objValue is derived of-type float:-
            objValue := SUM (objCoef[*] * value[*]);
          end objValue;
        end class;
        instance attributes:-
          identifier is aggregate-of:-
            dvName is fixed of-type string;
          end identifier;
          objCoef is fixed of-type float;
          kind is fixed of-type string [(kind == 'continuous') or
            (kind == 'integer') or (kind == 'binary')];
          bound is fixed of-type float;
          value is solver-derived of-type float;
          reduced is solver-derived of-type float;
        end instance;
      end decisionVariable;
    end formulation;
  end instance;
  constraint is-a entity:-
    class attributes:-
      conCount is derived of-type integer [conCount >= 0]:-
        conCount := cardinality of instances;
      end conCount;
    end class;
    instance attributes:-
      identifier is aggregate-of:-
        conName is fixed of-type string;
      end identifier;
      rhsValue is fixed of-type float;
      type is fixed of-type string [(type == '>=') or (type == '<=') or
        (type == '=')];
      slack is solver-derived of-type float;
      dual is solver-derived of-type float;
    end instance;
  end constraint;

```

Figure 54. General Linear Programming Model Schema Abstraction

```

matrix is-a relationship [decisionVariable:many,constraint:many];-
class attributes:-
  nzCount is derived of-type integer [nzCount >= 0];-
  nzCount := cardinality of instances where (coefValue ~= 0);
end nzCount;
end class;
instance attributes:-
  identifier is aggregate-of:-
    dvName is fixed of-type string [exists];
    conName is fixed of-type string [exists];
  end identifier;
  coefValue is fixed of-type float;
end instance;
end matrix;
end formulation;
end instance;
end generalLPModel;

```

Figure 54 (Continued). General Linear Programming Model Schema Abstraction

modeling calls. O-O structured modeling does not enjoy this benefit as there may be inherent cyclicity as one derived attribute may call another which ultimately calls the first attribute.

Acyclicity verification involves the development of an attribute list. This list is represented as a set which includes all attributes in the model abstraction. Each element of the set (each attribute) has an associated calling sequence. This calling sequence also may be represented as a set. A calling sequence is nothing more than the set of attributes on which the given attribute is functionally dependent. Figure 55 depicts various rules the modeler may use to determine the calling sequence for a given attribute.

**Fixed Attributes:****Class Level**

- 1.) The calling sequence is represented by an empty set

**Instance Level**

- 1.) The calling sequence is represented by an empty set for model instance and entity instance identifiers
- 2.) The calling sequence is represented by a set consisting of each entity instance identifier attribute participating in the definition of the relationship
- 3.) The calling sequence is represented by a set consisting of the specific instance identifier attribute

**Derived Attributes:****Class Level**

- 1.) The calling sequence is represented by the set of attributes referenced in the attribute derivation

**Instance Level**

- 1.) The calling sequence is represented by the union of the specific instance identifier attribute and the set of attributes referenced in the attribute derivation

**Solver-Derived Attributes:****Class Level**

- 1.) The calling sequence is represented by an empty set.

**Instance Level**

- 1.) The calling sequence is represented by a set consisting of the specific instance identifier attribute

Figure 55. Calling Sequence Determination Rules

Figure 56 presents an algorithm which the modeler may use to determine whether circular references occur within the model. This algorithm is named Warshall after its inventor (Aho, Hopcroft, and Ullman 1983). It begins with

an adjacency matrix, called adjacency. This is an  $n \times n$  matrix where each attribute appears along both axes of the matrix. If attribute  $i$  calls attribute  $j$ , a one is placed in the  $i,j$  entry in adjacency; otherwise enter a zero. Once the adjacency matrix is constructed, Warshall's algorithm may be used to determine if cyclicity occurs. This is indicated by the appearance of a one in the diagonal of the transitive closure matrix called closure. A one in the diagonal of closure implies that at some point an attribute calls itself.

```

for i := 1 to n do
  for j := 1 to n do
    closure[i,j] := adjacency[i,j];
for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      if closure[i,j] = 0 then
        closure[i,j] := (closure[i,k] and closure[k,j])

```

Figure 56. Algorithm for Verifying Model Acyclicity

The concepts of model schema development and model schema abstraction represent the first architectural step in proper system design (Chung 1984). The next two steps involve operationalization and implementation. Chapters V and VI attack these issues.

## CHAPTER V

### MESSAGE PROTOCOLS

#### Introduction

This chapter proposes a minimal set of class objects required to support an object-oriented (O-O) approach to decision support systems (DSSs) as discussed in the preceding chapters. In addition, we present an associated collection of message protocols defined for each class object which allow for the creation and manipulation of instance objects capable of representing the general entity, relationship, model, and relation class concepts introduced earlier. These protocols allow the user and other objects in the system to access the class attributes and instance attributes of these classes. Chapter VI describes a prototype implemented in a personal computing environment which uses these protocols.

#### Organization of Classes

We define five class objects which are necessary to implement the concepts of O-O data model schema abstraction and O-O model schema abstraction. These class objects are:

- (1) Metamodel class;

- (2) Entity class;
- (3) Relationship class;
- (4) Model class; and
- (5) Relation class.

Together these classes are placed into the single inheritance hierarchy shown in Figure 57. This figure also includes an additional abstract class object, Object, which is the superclass of all classes and defines the protocol common to all objects in the object universe. This hierarchical organization implies that the class methods and instance methods of both the Object class and the Metamodel class are inherited by the Entity, Relationship, Model, and Relation classes. The Metamodel class is also an abstract class in the sense that it defines instance level methods which are inherited by its subclasses but itself does not have any instance objects.

Each of these subclass objects has the ability to create instance objects specific to the subclass. Furthermore, although these instance objects are instantiated by different classes they have several characteristics in common. These general traits are inherited from the Metamodel class object.

Inheritance from the Metamodel class provides instances of these subclass objects with the capability to model nonspecific class level and instance level object interactions. Consequently, a single instance object of

any of the four subclass objects (Entity, Relationship, Model, and Relation) has the ability to represent both class level and instance level attributes as well as has the ability to perform both class level and instance level operations on the class modeled by the instance object. A reference to a class or an instance of a class modeled within such a singular object uses the prefix object.

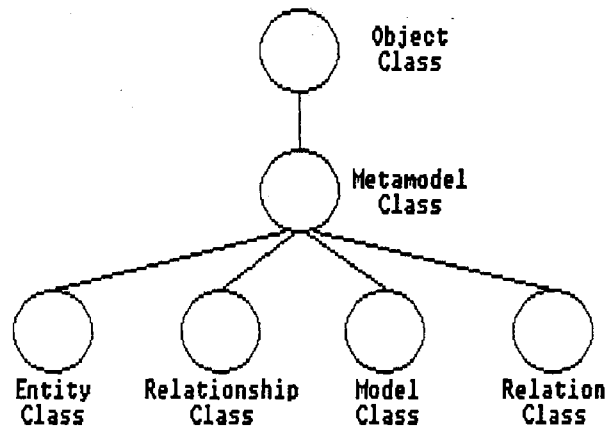


Figure 57. Class Object Hierarchy

For example, the Entity class object is the specific object on which the definition of the Source Point class in Figure 49 depends (note the sourcePoint is-a entity statement). The Entity class object creates an entity instance object in order to model the Source Point class. This instance object is referred to as the source point

instance object (since it is an instance object which models the Source Point class). The term object class refers to the specific class modeled by an instance object. Thus, the object class modeled by the source point instance object is the Source Point class. Adding a new source point to the Source Point class is an example of an object class level operation. The source point instance object provides this capability.

The source point instance object is also responsible for managing the interactions between the instances of the Source Point class (e.g., a specific source point) and other objects in the object universe. Consequently, an instance object of one of the four subclass objects performs a dual function. First, such an instance object represents the class level attributes of the object class which it models. Second, such an instance object is responsible for the creation and maintenance of the instance objects of this same class. An instance created by such an instance object is called an object instance (as the object instance is an instance of the object class).

Access to the value of the instance attribute supply (see Figure 49) must be provided for any source point created by the Source Point class. This is an example of an object instance level operation. Thus, operations on specific source points affect object instances whereas operations on the Source Point class affect the object



class. The Entity class object creates an instance object which has the capability to handle both of these kinds of operations.

We present another example of these ideas using the Model class object. The Model class object is the specific object on which the definition of the Transportation Model class in Figure 53 depends (note the transportationModel is-a model statement). The Model class object creates an instance object (called a transportation model instance object) which then serves to represent the Transportation Model class defined in this figure. Formulating a new transportation model would involve the creation of a new object instance. This is an object class level operation. On the other hand, solving a transportation model formulation is an example of an object instance level operation. Again, the source point instance object and the transportation model instance object must perform object class level and object instance level operations.

In summary, the four subclass objects discussed above create instance objects to model the classes defined in either a data model schema abstraction or a model schema abstraction. A class appearing in an abstraction and modeled by such an instance object is called the object class. Instances of a class appearing in an abstraction are instances of the object class and are called object instances. Thus, an instance object models an object class

and, consequently, is responsible for creating, manipulating, and removing object instances.

## Object Class Versus Object Instance

### Access Mechanisms

The user and other objects in the system gain access to the values of the general properties of the modeled classes (object class attributes) by passing messages to the specific instance object (e.g., the transportation model instance object). We develop message protocols which allow access to specific properties (object instance attributes) indirectly through the instance object since it is responsible for maintaining the instances of the object class. In this manner the instance object becomes solely responsible for managing the class attributes and instance attributes of the class that it models. Thus, as stated above, the source point instance object manages both the Source Point class attributes and its instance attributes.

We propose two mechanisms, one direct and the other indirect, which the user may use to refer to a specific object instance appearing in an instance object (that is, refer to an instance of the modeled class). The first mechanism allows the user to specify an object instance index in accessing a specific object instance. An object instance index refers to the chronological order in which the instance object creates the particular object instance.

For example, the user would refer to the first transportation model formulation (object instance) of the transportation model instance object as object instance one, the second by two, and so on.

The fifth object instance would become the fourth object instance should the user remove object instance four (dispose of a specific transportation model formulation), the sixth object instance would then become the fifth object instance, and so forth. This means of accessing object instances has the obvious shortcoming that an object instance cannot be uniquely identified by its object instance index as the object instance set is not guaranteed to remain static through time (various transportation models will be formulated, retained, and disposed of as needed).

Alternatively, the user may specify an object instance identifier list which the instance object uses to uniquely identify each object instance. All object class definitions outlined in the previous chapters require that object instances have unique object instance identifiers. Consequently, we provide messages which allow the user to access an object instance either through its object instance identifier or its associated object instance index.

## General Characteristics of Instance Objects

Instance objects created by the Entity, Relationship, Model, and Relation class objects have certain general characteristics which we discuss in this section.

Individual differences between these objects are discussed in the appropriate message protocol sections below.

### Object Class Identifiers

An object class identifier must be specified at the time that one of the four subclass objects creates an instance object. An object class identifier statically names the instance object. For example, the class name link serves as the object class identifier for the Link class definition appearing in Figure 51.

Entity and relationship instance objects, however, are not required to have unique object class identifiers. This is a necessary condition as model object instances duplicate entity and relationship instance object definitions at the time that they are created and thus would require the specification of unique object class identifiers. Dropping the uniqueness restriction for these instance objects allows the model instance object to create model object instances without having to alter the entity and relationship object class identifiers appearing in the

object entity definition and object relationship definition lists (an explanation of these lists appears below).

Model and relation instance objects, on the other hand, are required to have unique object class identifiers. Creation of an instance object, either for a new model object class or new relation object class, fails if the object class identifier is not unique to the other instance objects for the given subclass object. For example, an attempt to create a new model instance object with an object class identifier of transportationModel would fail if a preexisting model instance object has the same object class identifier.

#### Attribute Information

The message protocols developed below provide the user with the power to define, access, and in certain cases override attributes and their associated values. Each of these issues is further discussed in the sections which follow.

#### Attribute Definitions

Object class level and object instance level attributes for a new instance object are defined and communicated to the subclass responsible for creating the new instance object. The subclass object uses this information to organize the object class attributes and

object instance attributes for the new instance object. Through time the new instance object uses the relevant object instance level information to create new object instances for the object class which has the specified object instance level attribute characteristics.

For example, the source point instance object has two object class attributes (see Figure 49): (1) sourceCount; and (2) supplyTotal. In addition, the source point instance object creates three object instance attributes for each new object instance (e.g., instance of the Source Point class; refer to Figure 49): (1) sourceName; (2) supplyTest; and (3) supply. The information concerning these attributes, obtained from the previously developed abstraction, is passed to the Entity class object at the time it creates the source point instance object.

All class attribute definitions are collected together into an object class attribute definition list. The object class attribute definition list establishes the object class attributes represented within the instance object. Likewise, the object instance attribute definition list specifies the object instance attributes represented within the instance object. Both of these definition lists are derived from the corresponding schema abstraction. For each attribute in the schema abstraction there is an equivalent attribute definition appearing in the list.

This definition contains the following attribute information:

- (1) name;
- (2) kind;
- (3) type;
- (4) range; and
- (5) derivation.

An attribute name must be unique to its given list. In other words, no two attribute names may be the same in either the object class attribute list or the object instance attribute list. The object class attribute list may, however, contain an attribute name found in the object instance attribute list and vice versa. The attribute kind specifies whether the attribute is fixed, derived, or solver-derived. The attribute type details the data type of the attribute. For example, an attribute may be a string, float, integer, boolean, or any other valid type as determined by the Metamodel class. Attribute ranges are optional and when specified restrict the allowable set of values that an attribute may possess. Finally, an attribute derivation must be provided if an attribute is solver-derived. The attribute derivation is used to compute the associated value of the attribute at the time that it is accessed.

### Attribute Access Mechanisms

Attributes, whether at the object class level or object instance level, are accessed through either of two mechanisms in a manner similar to that provided for object instance level access. Attributes may be accessed through the specification of the attribute name or by giving the index of the attribute as it appears in the corresponding object attribute definition list used to define the instance object.

For example, the object class attribute supplyTotal may be accessed by giving its name, supplyTotal, or by specifying the index two since it is the second object class attribute appearing in the object class attribute definition list. Attribute indexes are unique, unlike object instance indexes, since attributes may not be removed once the instance object is created.

### Overriding Derived Attributes

Our message protocols allow the user to override any attribute derivation. There are several reasons for incorporating this feature into the system. An attribute derivation may, for example, compute a value which violates either the type or the range specified for the attribute. Likewise, other derived attributes may depend on the violated derivation and thus may also be affected.



Consequently, the user may specify an override value for any derived attribute and may also disable or enable overrides for an entire object class within the instance object. An instance object returns an override value without computing the value of a derivation when overrides are enabled and an override is defined for the associated derived attribute. As a result, a given instance object only computes a derivation when overrides are disabled or no override is specified for the specific derived attribute.

#### Object Instance Identifier List

The Metamodel class object enforces object instance uniqueness. That is, no two object instances within an instance object may have the same values across all object instance attributes. The object instance identifier list, if defined for an instance object, specifies the object instance attributes with which the instance object determines object instance uniqueness. For example, the source point instance object uses the single object instance attribute sourceName as its object instance identifier list. Uniqueness is determined across all object instance attributes if no object instance identifier list is defined for the instance object.

Entity and model instance objects are required to have non-varying object instance identifier lists. Relationship

instance objects, on the other hand, derive their non-varying object instance identifier list by aggregating the object instance identifier lists of the entity instance objects which participate in defining the relationship instance object. Contrarily, relation instance objects may have time-varying object instance identifier lists.

### Related Issues

Three additional concepts require discussion. The ideas of object class level and object instance level productions, context objects, and object dependencies are presented below. All of these issues are relevant to the discussion of message protocols for the proposed system.

### Productions

In his development of graph-based modeling systems, Jones (1988) defined the set of allowable editing operations on graphs, drawn from the field of graph-grammars, as productions. As used presently, productions permit the user to create tailored operations which build on the set of message protocols provided for each of the five class objects. Productions provide the user with the capacity to construct a sequence of operations for model instances which may be invoked through a single message. Furthermore, productions allow the user to incorporate

model specific actions not otherwise available through model abstraction.

A user may define two types of productions: (1) object class level productions; and (2) object instance level productions. Object class level productions are defined for an object class within an instance object and may affect all object instances of the object class. For example, the transportation model instance object may have an object class production named newSource which when executed creates, for a specified model object instance, a new source point object instance in the object class (Source Point class) and subsequently create a new link object instance in the link relationship instance object for every existing destination point in the destination point instance object. Thus, this production allows the user to create a new source point instance and link that new point to all existing destination points in a specific transportation model formulation. This sequence of events requires multiple message passing which, using the newSource production, may be achieved through a single message.

The user may define productions specific to a given object instance thereby restricting the scope of the production. For example, suppose that a user would like to perform several database operations such as selection and projection on several relations and use the results as

inputs to a specific object instance of a transportation model. By defining an object instance production, perhaps named build, the user may literally build inputs to the model permitting changes in the database to be incorporated into the model. Likewise, object instance productions accomplish a sequence of operations through the passage of multiple messages. This process is instigated when the user passes a single message to the model instance object indicating the desire to execute an object instance production.

### Contexts

The concept of contexts is taken directly from Smalltalk. A context is an object which contains a sequence of Smalltalk messages invoked when the context object is passed a message to evaluate itself. Frequently, the value of a context is used to perform conditional branching or testing. Another proposed use of contexts is in the creation of productions. Thus, a production is simply a valid context defined for an object instance or object class. Several messages below utilize contexts, referred to as blocks, which are evaluated within the corresponding method and are used to perform conditional testing or conditional message passing.

## Object Dependencies

Object dependencies naturally arise from the proposed O-O approach to DSSs. Two specific dependencies are: (1) entity-relationship dependencies; and (2) model-entity/relationship dependencies.

Relationships are naturally dependent on entities. Object instances of relationships may not be created without the existence of the entity object instance participating in the new relationship object instance. Consequently, relationship instance objects must verify the existence of these entity object instances at the time they are created. Furthermore, relationships are also dependent on entity classes especially when entity object instances are removed from an entity object class which participates in the relationship instance.

Models are inherently dependent on the entities and relationships which participate in the model. Changes in the underlying entity and relationship object classes may also affect a model object instance. Specifically, the model object instance may require the generation of a new solution should any one of the underlying objects change state.

These conditions require that objects have the capability to establish object dependencies. Message protocols are provided below to permit the user and objects themselves to create these dependencies. In addition,

these objects can respond to messages communicating changes in superior objects.

### Class Message Protocols

The message syntax used below borrows heavily from Smalltalk. A message in Smalltalk consists of (Smalltalk/V 1987):

- (1) identifying the object to which the message is sent (the receiver of the message);
- (2) identifying the additional objects that are included in the message (the message arguments);
- (3) specifying the desired operation to be performed (the message selector); and
- (4) accepting the single object that is returned as the message answer.

We use two types of message patterns in describing our protocols: (1) unary; and (2) keyword. Unary message patterns have no arguments. For example, instanceCount is a unary message where instanceCount is the selector for the message. The methods handler uses the selector to determine which method to invoke or whether to pass the message up the inheritance hierarchy. Keyword message patterns, on the other hand, are messages with one or more arguments. The keyword selector new: has a single argument. This argument follows the colon (:) which appears at the end of the selector. The message new:aValueList passes the selector new: to the receiver using the argument object aValueList. The methods handler

passes the message arguments to the selected method which then uses these objects in fulfilling the specified request.

The keyword selector for multiple arguments is distributed through the message pattern in parts. A part of the keyword selector appears before each argument. For instance, the keyword selector forEntity:do: should have two accompanying arguments. Note that this selector has two colons where an argument follows each colon in the message pattern. The message sender of the sending object passes the message forEntity:anEntity do:aBlock to the receiver whose methods handler uses the selector forEntity:do: to identify the appropriate method and passes the two arguments anEntity and aBlock to the selected method.

In explaining our proposed message protocols we state the name of the applicable class object, its class description, which object it inherits from, and which objects inherit from it. We also detail each class level and instance level message defined for the class. It is important to note that inheritance of methods applies and, as a result, superclass methods are available to a subclass object unless specifically overridden within the subclass. Finally, certain objects return a single object as an answer to the message sent to the receiver. Where appropriate this object is discussed.

## Metamodel Class

The Metamodel class object is an abstract class and is perhaps the most important object of the five class objects discussed. It provides a common protocol for defining and accessing object class attributes and object instance attributes necessary for the implementation of the Entity, Relationship, Model, and Relation instance objects so critical to the proposed O-O DSS. No messages are ever passed directly to this object. Rather, all messages received by this object are passed along the inheritance hierarchy.

Inherits From:        Object

Inherited By:        Model Entity Relationship Relation

### Class Message Protocols

classHavingIdentifier:aClassIdentifier

Answer the instance object for the receiver which has aClassIdentifier as its object class identifier.

newClass:aClassIdentifier

classAttributes:classAttributeList

instanceAttributes:instanceAttributeList

instanceIdentifier:identifierAttributeList

Create a new instance object having the object class identifier aClassIdentifier, object class attributes defined in classAttributeList, object instance attributes defined in instanceAttributeList, and where object instance identifiers are represented by the concatenation of the object instance attributes which appear in identifierAttributeList. Answer the new instance object initialized.



## Instance Message Protocols

### classAttributeCount

Answer the number of object class attributes defined for the receiver.

### classAttributeForIndex:anIndex

Answer the attribute name appearing at position anIndex in the object class attribute definition list for the receiver.

### classAttributeHasOverride:anAttribute

Answer true if the object class attribute named anAttribute in the receiver has a defined override value and overrides are enabled, else answer false. This message is only valid for derived attributes.

### classAttributes

Answer the object class attribute names for the receiver.

### classAttributesForIndexes:anIndexList

Answer the receiver attribute names for attributes appearing at the positions specified in anIndexList within the object class attribute definition list.

### classDerivations

Answer the object class attribute derivations for the receiver's derived attributes.

### classIndexHasOverride:anIndex

Answer true if the attribute at position anIndex in the object class attribute definition list for the receiver has a defined override value and overrides are enabled, else answer false. This message is only valid for derived attributes.

### classKinds

Answer the object class attribute kinds for the receiver.

`classRanges`

Answer the object class attribute ranges for the receiver.

`classTypes`

Answer the object class attribute types for the receiver.

`dependentEntityHavingClass:aClassIdentifier`

Answer the dependent entity instance object having aClassIdentifier as its object class identifier. Answer nil if no such object is found.

`dependentModelHavingClass:aClassIdentifier`

Answer the dependent model instance object having aClassIdentifier as its object class identifier. Answer nil if no such object is found.

`dependentRelationshipHavingClass:aClassIdentifier`

Answer the dependent relationship instance object having aClassIdentifier as its object class identifier. Answer nil if no such object is found.

`do:aBlock`

For each set of object instance values occurring for the receiver, evaluate the context aBlock using that set as the argument to the context.

`forAttributes:anAttributeList do:aBlock`

For each set of object instance values defined by the object instance attribute names appearing in anAttributeList occurring for the receiver, evaluate the context aBlock using that set as the argument to the context.

`forDependentEntitiesDo:aBlock`

For each dependent entity instance object occurring in the receiver, evaluate the context aBlock using that object as the argument to the context.

forDependentModelsDo:aBlock

For each dependent model instance object occurring in the receiver, evaluate the context aBlock using that object as the argument to the context.

forDependentRelationshipsDo:aBlock

For each dependent relationship instance object occurring in the receiver, evaluate the context aBlock using that object as the argument to the context.

forIdentifierDo:aBlock

For each object instance identifier for the object instances of the receiver, evaluate the context aBlock using that object instance identifier as the argument to the context. This is a valid message only if an object instance identifier list is defined for the receiver.

forIndexes:anIndexList do:aBlock

For each set of object instance values defined for the object instance attributes appearing at the positions specified in anIndexList within the object instance attribute definition list occurring for the receiver, evaluate the context aBlock using that set as the argument to the context.

hasDependencyWith:anObject

Answer true if anObject is a direct or indirect dependent of the receiver, else answer false. Indirect dependency occurs when a dependent of the receiver has anObject as a dependent, and so on.

identifierForClass

Answer the object class identifier for the receiver.

indexesOfClassAttributes:anAttributeList

Answer the index positions in the object class attribute definition list of the attribute names appearing in anAttributeList for the receiver.

indexesOfInstanceAttributes:anAttributeList

Answer the index positions in the object instance attribute definition list of the attribute names appearing in anAttributeList for the receiver.

indexOfClassAttribute:anAttribute

Answer the index position in the object class attribute definition list of the attribute name anAttribute in the receiver.

indexOfInstanceAttribute:anAttribute

Answer the index position in the object instance attribute definition list of the attribute name anAttribute in the receiver.

initialize:aClassIdentifier  
 classAttributes:classAttributeList  
 instanceAttributes:instanceAttributeList  
 instanceIdentifier:identifierAttributeList

Initialize the object class identifier using aClassIdentifier, initialize the object class attribute definition list using classAttributeList, initialize the object instance attribute definition list using instanceAttributeList, and construct the object instance identifier list using identifierAttributeList. Answer the receiver initialized. The corresponding method for this message may only be invoked once, at the time that the new instance object is created.

instanceAttributeCount

Answer the number of object instance attributes defined for the receiver.

instanceAttributeForIndex:anIndex

Answer the attribute name appearing at position anIndex in the object instance attribute definition list for the receiver.

instanceAttributeHasOverride:anAttribute

Answer true if the object instance attribute named anAttribute in the receiver has a defined override value and overrides are enabled, else answer false. This message is only valid for derived attributes.

instanceAttributes

Answer the object instance attribute names for the receiver.

instanceAttributesForIndexes:anIndexList

Answer the receiver attribute names for attributes appearing at the positions specified in anIndexList within the object instance attribute definition list.

instanceCount

Answer the number of object instances in the receiver.

instanceDerivations

Answer the object instance attribute derivations for the receiver's derived attributes.

instanceHasIdentifier:anIdentifier

Answer true if an object instance occurring in the receiver has an object instance identifier value of anIdentifier, else answer false. This is a valid message only if an object instance identifier list is defined for the receiver.

instanceHasValues:aValueList forAttributes:anAttributeList

Answer true if an object instance occurring in the receiver has the values of aValueList for the attributes names in anAttributeList, else answer false.

instanceHasValues:aValueList forIndexes:anIndexList

Answer true if an object instance occurring in the receiver has the values of aValueList for the attributes appearing at the positions specified in anIndexList within the object instance attribute definition list, else answer false.

instanceHavingIdentifier:anIdentifier

Answer the object instance index for the receiver of the object instance having an object instance identifier value of anIdentifier. Answer zero if no such object instance is found. This is a valid message only if an object instance identifier list is defined for the receiver.

instanceHavingValues:aValueList  
forAttributes:anAttributeList

Answer the object instance index for the receiver of the object instance having the values of aValueList

for the attribute names in anAttributeList. Answer zero if no such object instance is found.

instanceHavingValues:aValueList forIndexes:anIndexList

Answer the object instance index for the receiver of the object instance having the values of aValueList for the attributes appearing at the positions specified in anIndexList within the object instance attribute definition list. Answer zero if no such object instance is found.

instanceIdentifier

Answer the object instance attributes used to define the object instance identifier list for the receiver. Answer nil if no object instance identifier list is defined for the receiver.

instanceIdentifierFor:anInstance

Answer the object instance identifier for the object instance index anInstance in the receiver. This message is valid only if an object instance identifier list is defined for the receiver.

instanceIndexHasOverride:anIndex for:anInstance

Answer true if the attribute at position anIndex in the object instance attribute definition list for object instance index anInstance in the receiver has a defined override value and overrides are enabled, else answer false. This message is only valid for derived attributes.

instanceKinds

Answer the object instance attribute kinds for the receiver.

instanceRanges

Answer the object instance attribute ranges for the receiver.

instanceTypes

Answer the object instance attribute types for the receiver.

makeClassDerivation:aDerivation forAttribute:anAttribute

Answer a new derivation using the context aDerivation for the object class attribute named anAttribute in the receiver. Any existing override for the attribute is removed. This message is valid only for derived attributes.

makeClassDerivation:aDerivation forIndex:anIndex

Answer a new derivation using the context aDerivation for the attribute name appearing at position anIndex in the object class attribute definition list of the receiver. Any existing override for the attribute is removed. This message is valid only for derived attributes.

makeClassRange:aRange forAttribute:anAttribute

Answer a new range using the context aRange for the object class attribute named anAttribute in the receiver. This message is invalid if the current value of the attribute violates the new range.

makeClassRange:aRange forIndex:anIndex

Answer a new range using the context aRange for the attribute name appearing at position anIndex in the object class attribute definition list of the receiver. This message is invalid if the current value of the attribute violates the new range.

makeInstanceDerivation:aDerivation forAttribute:anAttribute

Answer a new derivation using the context aDerivation for the object instance attribute named anAttribute in the receiver. Any existing override for the attribute is removed for all object instances of the receiver. This message is valid only for derived attributes.

makeInstanceDerivation:aDerivation forIndex:anIndex

Answer a new derivation using the context aDerivation for the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver. Any existing override for the attribute is removed for all object instances of the receiver. This message is valid only for derived attributes.

`makeInstanceIdentifier:identifierAttributeList`

Answer a new object instance identifier list comprised of the object instance attributes appearing in identifierAttributeList. Attribute names may be in any order, occur only once in the list, and the corresponding attribute kinds must be fixed. The new object instance identifier list is accepted only if uniqueness of object instances holds true.

`makeInstanceRange:aRange forAttribute:anAttribute`

Answer a new range using the context aRange for the object instance attribute named anAttribute in the receiver. This message is invalid if the current value of the attribute for any object instance violates the new range.

`makeInstanceRange:aRange forIndex:anIndex`

Answer a new range using the context aRange for the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver. This message is invalid if the current value of the attribute for any object instance violates the new range.

`makeObjectADependent:anObject`

Answer the object anObject after making it a dependent of the receiver.

`new:aValueList`

Create a new object instance of the receiver after initializing its fixed and solver-derived attribute values with the values of aValueList. A new object instance is created if all values are acceptable for type and range. A new object instance must be unique. Uniqueness is determined by the object instance identifier list if one is defined for the receiver, otherwise it is determined by the combination of all the values in the object instance attribute definition list. Answer true if the new object instance is created, else answer false.

`notifyDependentsOfChange:aChange`

For dependents of the receiver which respond to anObject:changedWith:, notify the dependent of a change in the receiver by sending the dependent the



message anObject:receiver changedWith:aChange. Answer the receiver.

overrideClassAttribute:anAttribute usingValue:aValue

Override the value of the object class attribute named anAttribute with the value aValue in the receiver. This message is valid only for derived attributes and when overrides are enabled for the receiver. The data type of aValue must agree with the type specified for the attribute. Answer the override value.

overrideClassIndex:anIndex usingValue:aValue

Override the value of the attribute name appearing at position anIndex in the object class attribute definition list of the receiver with the value aValue. This message is valid only for derived attributes and when overrides are enabled for the receiver. The data type of aValue must agree with the type specified for the attribute. Answer the override value.

overrideDisable

Disable override operations for the receiver. No override values are removed from the receiver. Answer the receiver.

overrideEnable

Enable override operations for the receiver. Previously defined overrides are reinstated. Answer the receiver.

overrideInstanceAttribute:anAttribute usingValue:aValue  
for:anInstance

Override the value of the object instance attribute named anAttribute with the value aValue for the object instance index anInstance in the receiver. This message is valid only for derived attributes and when overrides are enabled for the receiver. The data type of aValue must agree with the type specified for the attribute. Answer the override value.

overrideInstanceIndex:anIndex usingValue:aValue  
for:anInstance

Override the value of the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver with the value aValue for the object instance index anInstance. This

message is valid only for derived attributes and when overrides are enabled for the receiver. The data type of aValue must agree with the type specified for the attribute. Answer the override value.

#### remove:anInstance

Remove the object instance whose object instance index is anInstance from the receiver. All relevant overrides are also removed. Answer the object instance identifier of the removed object instance. Answer nil if no object instance identifier list is defined for the receiver.

#### removeAllClassOverrides

Remove all object class overrides from the receiver. This message is valid only if overrides are enabled. Answer the receiver.

#### removeAllInstanceOverrides

Remove all object instance overrides from the receiver. This message is valid only if overrides are enabled. Answer the receiver.

#### removeAllInstances

Remove all object instances of the receiver. All object instance overrides are also removed. Answer the object instance identifiers of the removed object instances. Answer nil if no object instance identifier list is defined for the receiver.

#### removeInstances:anInstanceList

Remove any object instance from the receiver whose object instance index appears in the object instance index list anInstanceList. All relevant overrides are also removed. Answer the object instance identifiers of the removed object instances. Answer nil if no object instance identifier list is defined for the receiver.

#### removeObjectAsDependent:anObject

Remove the object anObject as a dependent of the receiver. Answer the object.

removeOverrideOfClassAttribute:anAttribute

Remove the override of the object class attribute named anAttribute. This message is valid only for derived attributes, when an override is defined for the attribute, and when overrides are enabled for the receiver. Answer the override value.

removeOverrideOfClassIndex:anIndex

Remove the override of the attribute appearing at position anIndex in the object class attribute definition list of the receiver. This message is valid only for derived attributes, when an override is defined for the attribute, and when overrides are enabled for the receiver. Answer the override value.

removeOverrideOfInstanceAttribute:anAttribute  
for:anInstance

Remove the override of the object instance attribute named anAttribute for the object instance index anInstance in the receiver. This message is valid only for derived attributes, when an override is defined for the attribute, and when overrides are enabled for the receiver. Answer the override value.

removeOverrideOfInstanceIndex:anIndex for:anInstance

Remove the override of the attribute appearing at position anIndex in the object instance attribute definition list of the receiver for the object instance index anInstance. This message is valid only for derived attributes, when an override is defined for the attribute, and when overrides are enabled for the receiver. Answer the override value.

updateClassAttribute:anAttribute usingValue:aValue

Update the value of the object class attribute named anAttribute with the value aValue in the receiver. This message is valid only for fixed or solver-derived attributes. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateClassAttributes:anAttributeList  
usingValues:aValueList

Update the values of the object class attributes named in anAttributeList with the values in aValueList in the receiver. This message is valid only for fixed or

solver-derived attributes. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateClassIndex:anIndex usingValue:aValue

Update the value of the attribute name appearing at position anIndex in the object class attribute definition list of the receiver with the value aValue. This message is valid only for fixed and solver-derived attributes. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateClassIndexes:anIndexList usingValues:aValueList

Update the values of the attribute names appearing at the positions specified in anIndexList in the object class attribute definition list of the receiver with the values in aValueList. This message is valid only for fixed or solver-derived attributes. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateInstanceAttribute:anAttribute usingValue:aValue  
for:anInstance

Update the value of the object instance attribute named anAttribute with the value aValue for the object instance index anInstance in the receiver. This message is valid only for fixed or solver-derived attributes. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateInstanceAttributes:anAttributeList  
usingValues:aValueList for:anInstance

Update the values of the object instance attributes named in anAttributeList with the values in aValueList for the object instance index anInstance in the receiver. This message is valid only for fixed or solver-derived attributes. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding

types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateInstanceIndex:anIndex usingValue:aValue  
for:anInstance

Update the value of the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver with the value aValue for the object instance index anInstance. This message is valid only for fixed and solver-derived attributes. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateInstanceIndexes:anIndexList usingValues:aValueList  
for:anInstance

Update the values of the attribute names appearing at the positions specified in anIndexList in the object instance attribute definition list of the receiver with the values in aValueList for the object instance index anInstance. This message is valid only for fixed or solver-derived attributes. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateInstanceUsingValues:aValueList for:anInstance

Update the values of the fixed and solver-derived attributes in the order specified in the object instance attribute definition list of the receiver with the values in aValueList for the object instance index anInstance. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding types and ranges of the fixed and solver-derived attributes of the receiver. Answer the new values.

valueOfClassAttribute:anAttribute

Answer the value of the object class attribute named anAttribute in the receiver.

valueOfClassIndex:anIndex

Answer the value of the attribute name appearing at position anIndex in the object class attribute definition list of the receiver.

valueOfInstanceAttribute:anAttribute for:anInstance

Answer the value of the object instance attribute named anAttribute for the object instance index anInstance in the receiver.

valueOfInstanceIndex:anIndex for:anInstance

Answer the value of the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver for the object instance index anInstance.

valuesFor:anInstance

Answer the values of all the object instance attributes for the object instance index anInstance in the receiver.

valuesOfClassAttributes:anAttributeList

Answer the values of the object class attributes named in anAttributeList in the receiver.

valuesOfClassIndexes:anIndexList

Answer the values of the attribute names appearing at the positions specified in anIndexList in the object class attribute definition list of the receiver.

valuesOfInstanceAttributes:anAttributeList for:anInstance

Answer the values of the object instance attributes named in anAttributeList for the object instance index anInstance in the receiver.

valuesOfInstanceIndexes:anIndexList for:anInstance

Answer the values of the attribute names appearing at the positions specified in anIndexList in the object instance attribute definition list of the receiver for the object instance index anInstance.

## Entity Class

The Entity class object has the capability of producing entity instance objects possessing characteristics specific to the entity classes encountered in model schema development and model schema abstraction. For example, each model object instance of the transportation model instance object requires two entity object classes: (1) the Source Point class; and (2) the Destination Point class. Each of these classes is represented as an instance of the Entity class.

This class provides the protocols necessary to create and access entity instance objects. Furthermore, certain methods are overridden at this level in order to account for the special needs of its objects. Specifically, an instance object must, when an object instance is removed from the object class, notify all dependent relationship instance objects that the removed object instance is no longer a member of the entity object class. This allows the relationship object to remove its object instances which are dependent on the removed entity object instance. For example, all links in a transportation model formulation which are defined in terms of a removed source point must also be removed from the model formulation.

Inherits From:        Metamodel Object

Inherited By:        (None)

Class Message Protocols

classHavingIdentifier:aClassIdentifier

Entity instance objects are not required to have unique object class identifiers and as such the receiver answers nil.

newEntity:entityIdentifier  
 classAttributes:classAttributeList  
 instanceAttributes:instanceAttributeList  
 instanceIdentifier:identifierAttributeList

Create a new entity instance object having the object class identifier entityIdentifier, object class attributes defined in classAttributeList, object instance attributes defined in instanceAttributeList, and where object instance identifiers are represented by the concatenation of the object instance attributes which appear in identifierAttributeList. The object instance identifier list must be nonempty. Answer the new instance object initialized.

newClass:aClassIdentifier  
 classAttributes:classAttributeList  
 instanceAttributes:instanceAttributeList  
 instanceIdentifier:identifierAttributeList

This message is disallowed because a method specific to the Entity class object exists for creating new entity instance objects.

Instance Message Protocols

makeInstanceIdentifier:identifierAttributeList

Entity instance objects are required to have unchanging object instance identifier lists and as such the sender is not allowed to change the receiver's object instance identifier list.

notifyDependentsOfChange:aChange

For dependents of the receiver which respond to entity:changedWith:, notify the dependent of a change in the receiver by sending the dependent the message entity:receiver changedWith:aChange. Answer the receiver.



### remove:anInstance

Remove the object instance whose object instance index is anInstance from the receiver. All relevant overrides are also removed. For all dependent relationship instance objects remove object instances in which the removed entity object instance identifier appears. Answer the object instance identifier of the removed object instance.

### removeAllInstances

Remove all object instances of the receiver. All object instance overrides are also removed. For all dependent relationship instance objects remove object instances in which the removed entity object instance identifiers appear. Answer the object instance identifiers of the removed object instances.

### removeInstances:anInstanceList

Remove any object instance from the receiver whose object instance index appears in the object instance index list anInstanceList. All relevant overrides are also removed. For all dependent relationship instance objects remove object instances in which the removed entity object instance identifiers appear. Answer the object instance identifiers of the removed object instances.

## Relationship Class

The Relationship class object has the ability to produce relationship instance objects possessing characteristics specific to the relationships encountered in model schema development and model schema abstraction. For example, each model object instance of the transportation model instance object requires a single relationship object class, the Link class. The Link class is represented as an instance of the Relationship class (it creates a link instance object). The Relationship class

provides the protocols necessary to create and access relationship instance objects. Furthermore, certain methods are overridden at this level in order to account for the special needs of its objects.

The Relationship class object requires an object entity mapping definition list in addition to an object class identifier, object class attribute definition list, and object instance attribute definition list in order to create a new relationship instance object. An object entity mapping definition list contains a two element entry for each entity which participates in the relationship. This entry consists of: (1) an entity instance object; and (2) the mapping of the entity into the relationship. The entity instance object must be a valid object instantiated by the Entity class object and the mapping must be either: (1) one; or (2) many. No fewer than two entities may be specified in an object entity mapping definition list and each entity instance object class identifier must be unique to the list. Finally, the new relationship instance object makes itself a dependent of all the entity instance objects participating in its creation.

Inherits From:        Metamodel Object

Inherited By:        (None)

Class Message Protocols

classHavingIdentifier:aClassIdentifier

Relationship instance objects are not required to have unique object class identifiers and as such the receiver answers nil.

newClass:aClassIdentifier  
 classAttributes:classAttributeList  
 instanceAttributes:instanceAttributeList  
 instanceIdentifier:identifierAttributeList

This message is disallowed because a method specific to the Relationship class object exists for creating new relationship instance objects.

newRelationship:relationshipIdentifier  
 classAttributes:classAttributeList  
 instanceAttributes:instanceAttributeList  
 entities:entityList

Create a new relationship instance object having the object class identifier relationshipIdentifier, object class attributes defined in classAttributeList, and object instance attributes defined in instanceAttributeList. Each entity instance object participating in the relationship and its corresponding mapping are defined as a matched entry in entityList. Relationship object instance identifiers are represented by the concatenation of the object instance identifier lists of the entity instance objects participating in the relationship. At least two entities must participate in the definition of a relationship instance object and must have unique entity instance object class identifiers. The new relationship instance object makes itself a dependent of all the entity instance objects participating in its creation. Answer the new instance object initialized.

Instance Message Protocols

entityAttributesForClass:aClassIdentifier

Answer the object instance identifier list for the entity instance object participating in the receiver and having aClassIdentifier as its object class identifier.

entityAttributesForIndex:anIndex

Answer the object instance identifier list for the entity instance object participating in the receiver and appearing at position anIndex in the object entity mapping definition list.

entityClasses

Answer the object class identifiers of the entity instance objects participating in the receiver.

entityClassForIndex:anIndex

Answer the object class identifier for the entity instance object participating in the receiver and appearing at position anIndex in the object entity mapping definition list.

entityCount

Answer the number of entity instance objects participating in the receiver.

entityIndexForClass:aClassIdentifier

Answer the index position of the entity instance object having the object class identifier aClassIdentifier in the object entity mapping definition list for the receiver.

entityInstanceIdentifierFor:anInstance  
forEntityClass:aClassIdentifier

Answer the object instance identifier for the entity instance object participating in the receiver and having aClassIdentifier as its object class identifier at instance index position anInstance.

entityInstanceIdentifierFor:anInstance  
forEntityIndex:anIndex

Answer the object instance identifier for the entity instance object participating in the receiver and appearing at position anIndex in the object entity mapping definition list at instance index position anInstance.

entityMappings

Answer the mappings of the entity instance objects participating in the receiver.

forEntityClass:aClassIdentifier do:aBlock

For each object instance identifier for the entity instance object participating in the receiver and having aClassIdentifier as its object class identifier, evaluate the context aBlock using that identifier as the argument to the context.

forEntityIndex:anIndex do:aBlock

For each object instance identifier for the entity instance object participating in the receiver and appearing at position anIndex in the object entity mapping definition list, evaluate the context aBlock using that identifier as the argument to the context.

initialize:relationshipName

classAttributes:classAttributeList  
instanceAttributes:instanceAttributeList  
instanceIdentifier:identifierAttributeList

Initialize the object class identifier using relationshipName, initialize the object class attribute definition list using classAttributeList, and initialize the object instance attribute definition list using instanceAttributeList. Construct the receiver object instance identifier list from entity information provided in identifierAttributeList. Answer the receiver initialized. The corresponding method for this message may only be invoked once, at the time that the new object instance is created.

instanceHasIdentifier:anIdentifier  
forEntityClass:aClassIdentifier

Answer true if an object instance occurring in the receiver for the entity instance object having aClassIdentifier as its object class identifier has an object instance identifier value of anIdentifier, else answer false.

instanceHasIdentifier:anIdentifier forEntityIndex:anIndex

Answer true if an object instance occurring in the receiver for the entity instance object appearing at position anIndex in the object entity mapping definition list has an object instance identifier value of anIdentifier, else answer false.

instanceHavingIdentifier:anIdentifier  
forEntityClass:aClassIdentifier

Answer the object instance index of the receiver for the object instance having an object instance identifier value of anIdentifier for the entity instance object having aClassIdentifier as its object class identifier. Answer zero if no such object instance is found.

instanceHavingIdentifier:anIdentifier  
forEntityIndex:anIndex

Answer the object instance index of the receiver for the object instance having an object instance identifier value of anIdentifier for the entity instance object appearing at position anIndex in the object entity mapping definition list. Answer zero if no such object instance is found.

instancesHavingIdentifier:anIdentifier  
forEntityClass:aClassIdentifier

Answer the object instance indexes of the receiver for the object instances having an object instance identifier value of anIdentifier for the entity instance object having aClassIdentifier as its object class identifier. Answer nil if no such object instances are found.

instancesHavingIdentifier:anIdentifier  
forEntityIndex:anIndex

Answer the object instance indexes of the receiver for the object instances having an object instance identifier value of anIdentifier for the entity instance object appearing at position anIndex in the object entity mapping definition list. Answer nil if no such object instances are found.

makeInstanceIdentifier:identifierAttributeList

Relationship instance objects are required to have unchanging object instance identifier lists and as such the sender is not allowed to change the receiver's object instance identifier list.

new:aValueList

Create a new object instance of the receiver after initializing its fixed and solver-derived attribute values with the values of aValueList. A new object

instance is created if all values are acceptable for type and range. Each entity object instance identifier is verified to exist in the participating entity instance objects and mapping restrictions are enforced before the new values are accepted. A new object instance must be unique. Answer true if the new object instance is created, else answer false.

notifyDependentsOfChange:aChange

For dependents of the receiver which respond to relationship:changedWith:, notify the dependent of a change in the receiver by sending the dependent the message relationship:receiver changedWith:aChange. Answer the receiver.

removeInstancesHavingIdentifier:anIdentifier  
forEntityClass:aClassIdentifier

Remove from the receiver all object instances having an object instance identifier value of anIdentifier for the entity instance object having aClassIdentifier as its object class identifier. Answer the object instance identifiers of the removed object instances. Answer nil if no such object instances are found.

removeInstancesHavingIdentifier:anIdentifier  
forEntityIndex:anIndex

Remove from the receiver all object instances having an object instance identifier value of anIdentifier for the entity instance object appearing at position anIndex in the object entity mapping definition list. Answer the object instance identifiers of the removed object instances. Answer nil if no such object instances are found.

removeInstancesHavingIdentifiers:anIdentifierList  
forEntityClass:aClassIdentifier

Remove from the receiver all object instances having an object instance identifier value appearing in anIdentifierList for the entity instance object having aClassIdentifier as its object class identifier. Answer the object instance identifiers of the removed object instances. Answer nil if no such object instances are found.

removeInstancesHavingIdentifiers:anIdentifierList  
forEntityIndex:anIndex

Remove from the receiver all object instances having an object instance identifier value appearing in anIdentifierList for the entity instance object appearing at position anIndex in the object entity mapping definition list. Answer the object instance identifiers of the removed object instances. Answer nil if no such object instances are found.

## Model Class

The Model class object provides a set of message protocols for creating, manipulating, and accessing models. Model instance objects describe the inherent structure of the model in the form of entities and relationships. The creation of model object instances permits the user or system to formulate specific instances of models which vary according to model inputs.

Model instance objects require two pieces of information in addition to object class identifier, object class attribute definition list, object instance attribute definition list, and object instance identifier list. These are: (1) an object entity definition list; and (2) an object relationship definition list.

An object entity definition list consists of an object class identifier, object class attribute definition list, object instance attribute definition list, and an object instance identifier list. Entities appearing in the entity definition list must have unique object class identifiers.



An object relationship definition list consists of an object class identifier, object class attribute definition list, object instance attribute definition list, and a mapping list where each entity object class identifier participating in the relationship and its corresponding mapping are defined as a matched entry in this list. Relationships appearing in the object relationship definition list must have unique object class identifiers.

A model instance object creates new entity instance and relationship instance objects according to the object entity definition and object relationship definition lists each time it creates a new model object instance. When this happens the model instance object also makes itself a dependent of each of these new entity instance and relationship instance objects. Several messages are provided which allow access to these superior instance objects.

The support of object level production capabilities within the Model class provides the user the potential to tailor his or her model representation to incorporate model specific behavior. Furthermore, object instance level productions allow the user to include instance specific behavior.

Models are solved by invoking a solver object. This object has the ability to interpret the model structure and retrieve its desired input from the model object instance

through message passing. Furthermore, the solver object can return the specific model outputs to the appropriate solver-derived attributes of the model object instance.

A model instance object uses a user specified solver object when a model object instance (model formulation) requires a new solution. The user may specify a default solver object which becomes the object class solver and which is used when no solver object is defined for a specific object instance. The user may also provide an object instance solver unique to a given model object instance. This allows the user to solve one model formulation with a given solver object, to solve another formulation with a different solver object, and to solve a model object instance having no object instance solver using the default solver object (the object class solver). Several messages are provided which allow access to these solver objects.

Inherits From:        Metamodel Object

Inherited By:        (None)

#### Class Message Protocols

```
newClass:aClassIdentifier
  classAttributes:classAttributeList
  instanceAttributes:instanceAttributeList
  instanceIdentifier:identifierAttributeList
```

This message is disallowed because a method specific to the Model class object exists for creating new model instance objects.

```
newModel:modelIdentifier classAttributes:classAttributeList
instanceAttributes:instanceAttributeList
instanceIdentifier:identifierAttributeList
entities:entityList relationships:relationshipList
```

Create a new model instance object having the object class identifier modelIdentifier, object class attributes defined in classAttributeList, object instance attributes defined in instanceAttributeList, and where object instance identifiers are represented by the concatenation of the object instance attributes which appear in identifierAttributeList. The object instance identifier list must be non-empty. A complete object entity definition for each entity defined in the model appears in entityList. An object entity definition list consists of an object class identifier, object class attribute definition list, object instance attribute definition list, and an object instance identifier list. Entities appearing in the object entity definition list must have unique object class identifiers. A complete relationship definition for each relationship defined in the model appears in relationshipList. An object relationship definition list consists of an object class identifier, object class attribute definition list, object instance attribute definition list, and a mapping list where each entity object class identifier participating in the relationship and its corresponding mapping are defined as a matched entry in this list. Relationships appearing in the object relationship definition list must have unique object class identifiers. Answer the new instance object initialized.

### Instance Message Protocols

```
classHasProductionNamed:name
```

Answer true if an object class production named name is defined for the receiver.

```
classHasProductions
```

Answer true if any object class productions are defined for the receiver.

```
classProductionHavingName:name
```

Answer the object class production named name defined for the receiver.

`classProductionNames`

Answer the object class production names of the object class productions defined for the receiver.

`classProductionNamesDo:aBlock`

For each object class production name of an object class production defined for the receiver, evaluate the context aBlock using that name as the argument to the context.

`entitiesFor:anInstance do:aBlock`

For each entity instance object in the receiver participating in the object instance having object instance index anInstance, evaluate the context aBlock using that instance object as the argument to the context.

`entity:anEntity changedWith:aChange`

An entity instance object has changed. Determine which object instance is affected and change state to show that this object instance requires solving. Answer the receiver.

`entityClasses`

Answer the object class identifiers for the entity definitions appearing in the object entity definition list in the receiver.

`entityClassForIndex:anIndex`

Answer the entity object class identifier for the entity participating in the receiver and appearing at position anIndex in the object entity definition list.

`entityCount`

Answer the number of entity definitions appearing in the object entity definition list.

`entityHavingClass:aClassIdentifier for:anInstance`

Answer the entity instance object having aClassIdentifier as its object class identifier at object instance index position anInstance in the receiver.

entityHavingIndex:anIndex for:anInstance

Answer the entity instance object appearing at position anIndex in the object entity definition list at object instance index position anInstance in the receiver.

entityIndexForClass:aClassIdentifier

Answer the index position of the entity having the entity object class identifier aClassIdentifier in the object entity definition list of the receiver.

executeClassProductionNamed:name usingValue:aValue

Execute the object class production named name in the receiver passing it the object having value aValue. Answer the result of executing the production.

executeInstanceProductionNamed:name usingValue:aValue  
for:anInstance

Execute the object instance production named name at object instance index position anInstance in the receiver passing it the object having value aValue. Answer the result of executing the production.

initialize:modelName classAttributes:classAttributeList  
instanceAttributes:instanceAttributeList  
instanceIdentifier:identifierAttributeList

Initialize the object class identifier using aClassIdentifier, initialize the object class attribute definition list using classAttributeList, and initialize the object instance attribute definition list using instanceAttributeList. Construct the receiver object entity definition list, object relationship definition list, and object instance identifier list from information provided in identifierAttributeList. The object class solver is initialized to nil. Answer the receiver initialized. The corresponding method for this message may only be invoked once, at the time that the new instance object is created.

instanceHasProductionNamed:name for:anInstance

Answer true if an object instance production named name at object instance index position anInstance is defined for the receiver.

instanceHasProductionsFor:anInstance

Answer true if any object instance productions at object instance index position anInstance are defined for the receiver.

instanceHasSolverFor:anInstance

Answer true if an object instance solver at object instance index position anInstance is defined for the receiver.

instanceHavingEntity:anEntity forClass:aClassIdentifier

Answer the object instance index of the receiver for the object instance having the entity instance object anEntity which has aClassIdentifier as its object class identifier. Answer zero if no such object instance is found.

instanceHavingEntity:anEntity forIndex:anIndex

Answer the object instance index of the receiver for the object instance having the entity instance object anEntity appearing at position anIndex in the object entity definition list. Answer zero if no such object instance is found.

instanceHavingRelationship:aRelationship  
forClass:aClassIdentifier

Answer the object instance index of the receiver for the object instance having the relationship instance object aRelationship which has aClassIdentifier as its object class identifier. Answer zero if no such object instance is found.

instanceHavingRelationship:aRelationship forIndex:anIndex

Answer the object instance index of the receiver for the object instance having the relationship instance object aRelationship appearing at position anIndex in the object relationship definition list. Answer zero if no such object instance is found.

instanceProductionHavingName:name for:anInstance

Answer the object instance production named name at object instance index position anInstance defined for the receiver.

instanceProductionNamesFor:anInstance

Answer the object instance production names of the object instance productions at object instance index position anInstance defined for the receiver.

instanceProductionNamesDo:aBlock for:anInstance

For each object instance production name of an object instance production at object instance index position anInstance defined for the receiver, evaluate the context aBlock using that name as the argument to the context.

instancesHavingProductions

Answer the object instance indexes in the receiver for object instances which have object instance production defined. Answer nil if no such object instances are found.

instancesHavingSolvers

Answer the object instance indexes in the receiver for object instances which have object instance solvers defined. Answer nil if no such object instances are found.

makeClassProduction:aProduction named:name

Answer a new production using the context aProduction for the object class production named name in the receiver. Any existing object class production using this name is removed.

makeClassSolver:aSolver

Make the solver object aSolver the default object instance solver. Answer the new object class solver for the receiver.

makeInstanceIdentifier:identifierAttributeList

Model instance objects are required to have unchanging object instance identifier lists and as such the sender is not allowed to change the receiver's object instance identifier list.

makeInstanceProduction:aProduction named:name  
for:anInstance

Answer a new production using the context aProduction for the object instance production named name at the object instance index position anInstance in the receiver. Any existing object instance production using this name in that object instance is removed.

makeInstanceSolver:aSolver for:anInstance

Make the solver object aSolver the default object instance solver at the object instance index position anInstance. Any existing object instance solver already specified is removed. Answer the new object instance solver for the receiver.

model:aModel changedWith:aChange

A model instance object has changed. Determine which object instance is affected and change state to show that this object instance requires solving. Answer the receiver.

new:aValueList

Create a new object instance of the receiver after initializing its fixed and solver-derived attribute values with the values of aValueList. A new object instance is created if all values are acceptable for type and range. Each model instance object creates new entity instance objects and relationship instance objects for the new object instance in accordance with the object entity definition and object relationship definition lists. A new object instance must be unique. Answer true if the new object instance is created, else answer false.

notifyDependentsOfChange:aChange

For dependents of the receiver which respond to model:changedWith:, notify the dependent of a change in the receiver by sending the dependent the message model:receiver changedWith:aChange. Answer the receiver.

relationship:aRelationship changedWith:aChange

A relationship instance object has changed. Determine which object instance is affected and change state to show that this object instance requires solving. Answer the receiver.



## relationshipClasses

Answer the object class identifiers for the relationship definitions appearing in the object relationship definition list in the receiver.

## relationshipClassForIndex:anIndex

Answer the object class identifier for the relationship participating in the receiver and appearing at position anIndex in the object relationship definition list.

## relationshipCount

Answer the number of relationship definitions appearing in the object relationship definition list.

## relationshipHavingClass:aClassIdentifier for:anInstance

Answer the relationship instance object having aClassIdentifier as its object class identifier at object instance index position anInstance in the receiver.

## relationshipHavingIndex:anIndex for:anInstance

Answer the relationship instance object appearing at position anIndex in the object relationship definition list at object instance index position anInstance in the receiver.

## relationshipIndexForClass:aClassIdentifier

Answer the index position of the relationship having the relationship object class identifier aClassIdentifier in the object relationship definition list of the receiver.

## relationshipsFor:anInstance do:aBlock

For each relationship instance object in the receiver participating in the object instance having object instance index anInstance, evaluate the context aBlock using that instance object as the argument to the context.

## remove:anInstance

Remove the object instance whose object instance index is anInstance from the receiver. All relevant overrides and object instance productions are also

removed. Answer the object instance identifier of the removed object instance.

`removeAllClassProductions`

Remove all object class productions defined for the receiver. Answer the receiver.

`removeAllInstanceProductions`

Remove all object instance productions defined for the receiver. Answer the receiver.

`removeAllInstanceProductionsFor:anInstance`

Remove all object instance productions defined for the receiver at object instance position anInstance. Answer the receiver.

`removeAllInstances`

Remove all object instances of the receiver. All object instance overrides and object instance productions are also removed. Answer the object instance identifiers of the removed object instances.

`removeAllInstanceSolvers`

Remove all object instance solvers defined for the receiver. Answer the receiver.

`removeClassProductionNamed:name`

Remove the object class production having name name from the receiver. This message is valid only if such a production exists. Answer the removed production.

`removeInstanceProductionNamed:name for:anInstance`

Remove the object instance production having name name from the receiver at object instance index position anInstance. This message is valid only if such a production exists. Answer the removed production.

`removeInstances:anInstanceList`

Remove any object instance from the receiver whose object instance index appears in the object instance index list anInstanceList. All relevant overrides and object instance productions are also removed. Answer the object instance identifiers of the removed object instances.

`removeInstanceSolverFor:anInstance`

Remove the object instance solver from the receiver at object instance index position anInstance. Answer the solver.

`resolve:anInstance`

Answer true if the object instance at object instance index position anInstance for the receiver requires solving, else answer false.

`solve:anInstance`

Solve the object instance at object instance index position anInstance in the receiver. Answer the receiver.

`solveAll`

Solve all object instances in the receiver. Answer the receiver.

`solverFor:anInstance`

Answer the object instance solver object for the object instance at object instance index position anInstance in the receiver. Answer the object class solver if no object instance solver is defined for the object instance.

## Relation Class

The Relation class object provides the user with the capability to access and manipulate relations using two distinct approaches. First, the user may approach this class from the perspective of a relational database user. The user may access the information stored in the object using familiar terms such as insert, delete, update, tuple, column, attribute, degree, key, and so on. Thus, from this perspective the Relation class provides the typical database functions of insert, delete, and update.

The user may, quite to the contrary, access this information in terms identical to those used for accessing entity, relationship, and model instance level and class level attributes. From this perspective the relation is a class which defines no class attributes and where all instance attributes are fixed. Thus, the Relation class object manages instance objects using the same general concepts applied to entity, relationship, and model instance objects.

More specifically, the Relation class object permits its object instance identifier list to vary through time, has no defined class attributes, and implements additional instance level messages to perform relational algebra.

The user specifies three items when creating a new instance of this class: (1) a relation name; (2) an attribute definition list; and (3) a key attribute list. The relation name and key attribute items are nothing other than an object class identifier and object instance identifier list as discussed above. The attribute definition list requires some explanation.

An attribute definition list contains a list of the attributes defined for the new relation instance object. An attribute definition contains the following attribute information: (1) name; (2) type; and (3) range. These are identical in meaning to those previously presented. As stated above, all attributes for a relation instance object

are fixed and, consequently, it is not necessary to specify this information.

Attributes of a relation instance object, from the perspective of a relational database user, may be accessed by specifying the attribute name or by specifying the column within the relation that the attribute appears. Stating the column of an attribute is equivalent to specifying an attribute index. Moreover, a tuple may be accessed using either a tuple index or by designating a primary key value associated with the desired tuple. This is comparable to giving an object instance index or declaring an object instance identifier for the object instance.

Inherits From:        Metamodel Object

Inherited By:        (None)

#### Class Message Protocols

```
newClass:aClassIdentifier
  classAttributes:classAttributeList
  instanceAttributes:instanceAttributeList
  instanceIdentifier:identifierAttributeList
```

This message is disallowed because a method specific to the Relation class object exists for creating new relation instance objects.

```
newRelation:relationName attributes:attributeList
```

Create a new relation instance object having the object class identifier relationName, no object class attributes, object instance attributes defined in attributeList, and defining an empty object instance identifier list. Answer the new instance object initialized.

newRelation:relationName attributes:attributeList  
 primaryKey:keyAttributeList

Create a new relation instance object having the object class identifier relationName, no object class attributes, object instance attributes defined in attributeList, and where object instance identifiers are represented by the concatenation of the object instance attributes which appear in keyAttributeList. Answer the new instance object initialized.

### Instance Message Protocols

attribute:anAttribute for:aTuple

Answer the value of the object instance attribute named anAttribute for the object instance index aTuple in the receiver.

attributeForColumn:aColumn

Answer the attribute name appearing at position aColumn in the object instance attribute definition list for the receiver.

attributes:anAttributeList for:aTuple

Answer the values of the object instance attributes named in anAttributeList for the object instance index aTuple in the receiver.

attributesForColumns:aColumnList

Answer the receiver attribute names for attributes appearing at the positions specified in aColumnList within the object instance attribute definition list.

column:aColumn for:aTuple

Answer the value of the attribute name appearing at position aColumn in the object instance attribute definition list of the receiver for the object instance index aTuple.

columnForAttribute:anAttribute

Answer the index position in the object instance attribute definition list of the attribute name anAttribute in the receiver.

columns:aColumnList for:aTuple

Answer the values of the attribute names appearing at the positions specified in aColumnList in the object instance attribute definition list of the receiver for the object instance index aTuple.

columnsForAttributes:anAttributeList

Answer the index positions in the object instance attribute definition list of the attribute names appearing in anAttributeList for the receiver.

degree

Answer the number of object instance attributes defined for the receiver.

delete:aTuple

Remove the object instance whose object instance index is aTuple from the receiver. Answer the object instance identifier of the removed object instance. Answer nil if no object instance identifier list is defined for the receiver.

difference:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all object instances in the receiver and not in aRelation. The object aRelation must be a valid relation instance object which is union compatible with the receiver.

divideby:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The object instance attribute definition list for the new instance object consists of the object instance attributes not in aRelation but which appear in the receiver. If for all object instances in aRelation, there exist object instances in the receiver where the set of object instances from aRelation are present with fixed values for the object instance attributes not in aRelation, these fixed values become object instances in the new relation instance object. The object instance attribute types for aRelation must match an equal number of object instance attribute types appearing lastly in the object instance attribute definition list for the receiver. Furthermore, aRelation must be

a valid relation instance object and there must be at least one object instance in aRelation.

equiJoin:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all possible concatenated pairs of object instances, one from the receiver and the other from aRelation, such that each pair of the object instances has equal values for the object instance attributes which have the same object instance attribute definitions for both relation instance objects. Duplicate object instance attributes are eliminated from the new relation instance object. The object aRelation must be a valid relation instance object.

forColumns:aColumnList do:aBlock

For each set of object instance values defined for the object instance attributes appearing at the positions specified in aColumnList within the object instance attribute definition list occurring for the receiver, evaluate the context aBlock using that set as the argument to the context.

heading

Answer the object instance attribute names for the receiver.

includes:aValueList

Answer true if an object instance occurring in the receiver has the values of aValueList for all object instance attributes, else answer false.

includes:aValueList forAttributes:anAttributeList

Answer true if an object instance occurring in the receiver has the values of aValueList for the attributes names in anAttributeList, else answer false.

includes:aValueList forColumns:aColumnList

Answer true if an object instance occurring in the receiver has the values of aValueList for the attributes appearing at the positions specified in aColumnList within the object instance attribute definition list, else answer false.



includesKey:aKey

Answer true if an object instance occurring in the receiver has an object instance identifier value of aKey, else answer false. This is a valid message only if an object instance identifier list is defined for the receiver.

indexOf:aValueList

Answer the object instance index for the receiver of the object instance having the values of aValueList for all object instance attributes. Answer zero if no such object instance is found.

indexOf:aValueList forAttributes:anAttributeList

Answer the object instance index for the receiver of the object instance having the values of aValueList for the attribute names in anAttributeList. Answer zero if no such object instance is found.

indexOf:aValueList forColumns:aColumnList

Answer the object instance index for the receiver of the object instance having the values of aValueList for the attributes appearing at the positions specified in aColumnList within the object instance attribute definition list. Answer zero if no such object instance is found.

indexOfKey:aKey

Answer the object instance index for the receiver of the object instance having an object instance identifier value of aKey. Answer zero if no such object instance is found. This is a valid message only if an object instance identifier list is defined for the receiver.

insert:aValueList

Create a new object instance of the receiver after initializing its attribute values with the values of aValueList. A new object instance is created if all values are acceptable for type and range. A new object instance must be unique. Uniqueness is determined by the object instance identifier list if one is defined for the receiver, otherwise it is determined by the combination of all the values in the object instance attribute definition list. Answer

true if the new object instance is created, else answer false.

intersection:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all object instances in the receiver which also appear in aRelation. The object aRelation must be a valid relation instance object which is union compatible with the receiver.

join:aRelation where:aBlock relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all possible concatenated pairs of object instances, one from the receiver and the other from aRelation, such that the context aBlock evaluates to true for each pairing of the object instances. The object aRelation must be a valid relation instance object.

key:aTuple

Answer the object instance identifier for the object instance index aTuple in the receiver. This message is valid only if an object instance identifier list is defined for the receiver.

makeRange:aRange forAttribute:anAttribute

Answer a new range using the context aRange for the object instance attribute named anAttribute in the receiver. This message is invalid if the current value of the attribute for any object instance violates the new range.

makeRange:aRange forIndex:anIndex

Answer a new range using the context aRange for the attribute name appearing at position anIndex in the object instance attribute definition list of the receiver. This message is invalid if the current value of the attribute for any object instance violates the new range.

name

Answer the object class identifier for the receiver.

## primaryKey

Answer the object instance attributes used to define the object instance identifier list for the receiver. Answer nil if no object instance identifier list is defined for the receiver.

## primaryKey:keyAttributeList

Answer a new object instance identifier list comprised of the object instance attributes appearing in keyAttributeList. Attribute names may be in any order and occur only once in the list. The new object instance identifier list is accepted only if uniqueness of object instances holds true.

## product:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all possible concatenated pairs of object instances in the receiver and in aRelation. The object aRelation must be a valid relation instance object.

## projectAttributes:anAttributeList relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of the object instance values for the object instance attributes specified in anAttributeList across all object instances. Uniqueness of object instances in the new relation instance object is enforced.

## projectColumns:aColumnList relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of the object instance values for the attributes appearing at the positions specified in aColumnList within the object instance attribute definition across all object instances. Uniqueness of object instances in the new relation instance object is enforced.

## ranges

Answer the object instance attribute ranges for the receiver.

select:aBlock relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of object instances in the receiver such that the context aBlock evaluates to true for the object instance.

tuple:aTuple

Answer the values of all the object instance attributes for the object instance index aTuple in the receiver.

tupleCount

Answer the number of object instances in the receiver.

type

Answer the object instance attribute types for the receiver.

union:aRelation relationName:name

Answer a new relation instance object having name as an object class identifier. The new relation instance object consists of all object instances which appear in either the receiver or aRelation. The object aRelation must be a valid relation instance object which is union compatible with the receiver.

update:aValueList for:aTuple

Update the values of the attributes in the order specified in the object instance attribute definition list of the receiver with the values in aValueList for the object instance index aTuple. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateAttribute:anAttribute value:aValue for:aTuple

Update the value of the object instance attribute named anAttribute with the value aValue for the object instance index aTuple in the receiver. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed.

Uniqueness of object instances is enforced. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateAttributes:anAttributeList values:aValueList  
for:aTuple

Update the values of the object instance attributes named in anAttributeList with the values in aValueList for the object instance index aTuple in the receiver. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

updateColumn:aColumn value:aValue for:aTuple

Update the value of the attribute name appearing at position aColumn in the object instance attribute definition list of the receiver with the value aValue for the object instance index aTuple. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data type and value of aValue must agree with the type and range specified for the attribute. Answer the new value.

updateColumns:aColumnList values:aValueList for:aTuple

Update the values of the attribute names appearing at the positions specified in aColumnList in the object instance attribute definition list of the receiver with the values in aValueList for the object instance index aTuple. If an object instance identifier list is defined for the receiver, an update on an identifier attribute is disallowed. Uniqueness of object instances is enforced. The data types and values of aValueList must agree with the corresponding types and ranges of the attributes specified in anAttributeList. Answer the new values.

## CHAPTER VI

### PROTOTYPE DESCRIPTION

#### Introduction

This chapter describes a prototype we developed using the object-oriented (O-O) principles and message protocols presented in previous chapters. We discuss the implementation environment detailing the software and hardware employed. This is followed by a description of the three levels of user support provided by the prototype. Finally, we compare two differing approaches the user may adopt in managing data within the prototype.

#### Implementation Environment

We chose to develop our prototype in a personal computing environment. Advances in personal computer hardware and software make their use as tools for the support of personal decision making a realistic possibility. Consequently, the successful implementation of our proposed concepts using such a popular tool as a personal computer may enhance the viability of O-O decision support systems (DSSs).

## Software

Several O-O programming languages exist for personal computers (e.g., Actor, Objective-C, C++, and Smalltalk among others). We selected the Smalltalk/V Object-Oriented Programming System (copyright Digital Inc., 1986) to develop our prototype.

Smalltalk/V is both a system for creating Smalltalk programs and an environment for using a personal computer. It describes itself as a mode-less environment which uses windows, pop-up menus, and an optional mouse in order to simplify computer use. Smalltalk/V also provides its own components, including the Smalltalk/V source code, as building blocks for the user to create his or her own applications.

Smalltalk/V offers such desirable features as late binding, operator overloading, garbage collection, inheritance, and class add-on support. The Float class in Smalltalk/V serves as an example of the benefits afforded by class add-on support. The Float class requires a floating point coprocessor in order to perform floating point operations. The personal computer that we used to develop the prototype does not have a floating point coprocessor yet we required floating point support. A Float class add-on package was purchased thereby providing the needed supporting routines in software.

The five class objects discussed in Chapter V were added to Smalltalk/V as were the various class level and instance level methods also needed to implement the proposed message protocols. We added additional methods where necessary; specifically to implement the window level user interface discussed below.

#### Hardware

We used an IBM AT compatible personal computer in developing our prototype. This machine had one megabyte of main memory, a twenty megabyte hard disk drive, and a clock speed of eight megahertz. There was no floating point coprocessor support as stated above. A mouse was used to enhance the capabilities of Smalltalk/V although this was not a requirement.

#### User Interface

Our prototype provides three levels of support for user interaction. Each level permits varying degrees of assistance in creating, manipulating, and removing instances of the four subclass objects (Entity, Relationship, Model, and Relation). Each of these levels is discussed at length below.



## Message Level

Smalltalk/V requires that all instance objects exist independently of the class objects responsible for creating them. The space occupied by an object which is created but not referenced from anywhere in the system is collected by what is called a garbage collector and is returned to the system.

Therefore, the owner of an instance object, typically the user, becomes responsible for maintaining the existence of this object. This is frequently accomplished in Smalltalk/V by saving the instance object in a global variable. All global variables contain a single object which other objects in the system may pass messages to simply by using the global variable name as the receiver of the message.

Figure 58 presents a message which creates a new instance of the Entity class object. The global variable AnEntity (global variable names in Smalltalk/V begin with an uppercase letter) saves the new entity instance object from being collected by the garbage collector. In this example, the Entity class creates a new instance object (to model the Source Point class) having an object class identifier of sourcePoint and two object class attributes: (1) sourceCount; and (2) supplyTotal. Both of these attributes are derived, integers, and are restricted to having nonnegative values.

```

AnEntity := Entity newEntity:'sourcePoint'
classAttributes:#(
  ('sourceCount' 'Derived' 'Integer' '[sourceCount >= 0]'
   '[sourceCount := class instanceCount]')
  ('supplyTotal' 'Derived' 'Integer' '[supplyTotal >= 0]'
   '[supplyTotal := 0.
   1 to:(class instanceCount) do:[anInstance |
   supplyTotal := supplyTotal + (class valueOfInstanceAttribute:'supply'
   for:anInstance)]])')
instanceAttributes:#(
  ('sourceName' 'Fixed' 'String' '')
  ('supply' 'Fixed' 'Integer' '[supply >= 0]'))
instanceIdentifier:#('sourceName')

```

Figure 58. Creating an Entity Class

Their derivation statements differ, however. The sourceCount attribute uses the derivation: [sourceCount := class instanceCount]. The object reference class may be used in any derivation statement and assumes the value of the instance object in which the derivation statement is defined. The object reference instance is meaningful in object instance attribute derivation statements and takes on the value of the object instance index associated with the specific object instance.

The sourceCount derivation states that the value returned when the message instanceCount is sent to the new entity instance object (represented in the derivation by the class reference) becomes the value of the sourceCount object class attribute. Chapter V defines the return value of this message as the number of object instances in the receiver. In other words, sourceCount is the number of source points defined in the Source Point class. The

supplyTotal derived object class attribute iterates over each instance totaling the values of the object instance supply attributes.

There are two object instance attributes defined for this new entity instance object: (1) sourceName; and (2) supply. Both of these attributes are fixed, one a string and the other an integer, and the supply attribute must be nonnegative. The single object instance attribute sourceName is used as the object instance identifier list for the instance object.

The user may, following instantiation of the new object, create new object instances, remove them, access them, and update them by simply passing messages using the AnEntity global variable. Furthermore, any other object in the system has access to this object and thus may access the object class through message passing. For example, the user or any other object in the system might pass the message instanceAttributes (e.g., AnEntity instanceAttributes) to obtain a list of object instance attributes defined for the new instance object. In this case the instance object AnEntity would answer the two attribute names sourceName and supply. This process of user interaction is depicted by the flowchart shown in Figure 59.

A major disadvantage of this level of object access is that the user must deal with class objects and instance

objects at a very low level. It does, however, provide the user with complete access to all available methods in order to manipulate these objects. This is not the case with the next two levels of user interaction.

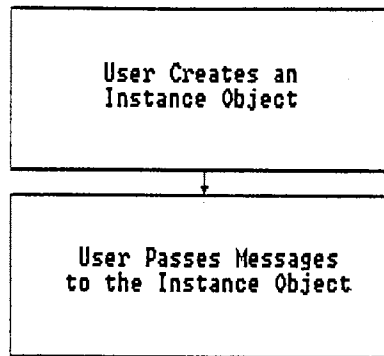


Figure 59. Message Level Flowchart

#### Window Level

Windows provide the major interface between the user and Smalltalk/V. Windows allow for the programming of menus which permit the user to select from a variety of operations or tasks to perform. This reduces the level of message sending that the user becomes involved in as the selection of a menu option may in turn lead to a sequence of message passing unbeknownst to the user.

Our prototype uses a single window implementation which presents information in the window using an electronic spreadsheet approach. We developed this window implementation specifically for the purposes of this study. In order to access a window, the user must first create the instance object (as in Figure 58) used to model the specific object class and then may open a window in order to create, remove, and update object instances. The user may open three different windows corresponding to a single instance object using the following messages: (1) openClass; (2) openInstances; and (3) openNew.

The openClass message opens a class window which shows the object class attributes for the associated instance object. The openInstances message opens an instances window which shows the object instance attributes for all the object instances defined for the receiver. The openNew message opens a new instance window which permits the user to organize a new object instance and subsequently attempt to create that new object instance in the object class. Figure 60 shows a flowchart detailing this level of user interaction. In Smalltalk/V the term scheduling refers to the process used by a special object, called the Scheduler, to determine a precedence ordering of windows defined within the system. Scheduling a window causes the system to display the window and show it as the topmost window in the display screen.

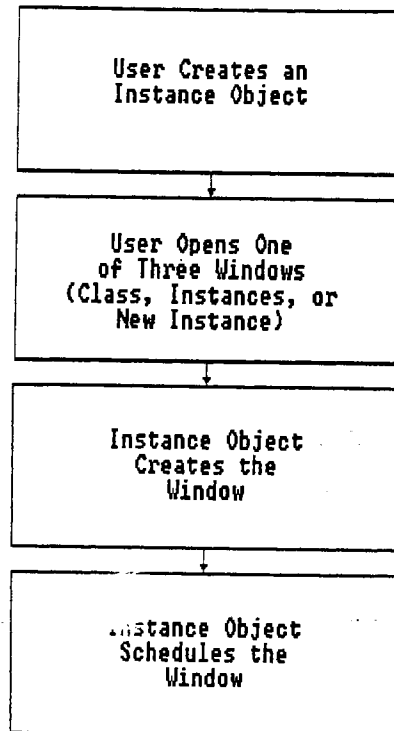


Figure 60. Window Level Flowchart

Figure 61 presents all three windows which may be opened for a single instance object. Only one window may be active at a time in Smalltalk/V. The sourcePoint Instances window is the active window and appears to the middle right of this figure. This window was opened using the message AnEntity openInstances. The other windows were accessed and corresponding new object instances created through window level mechanisms. Several window oriented concepts are discussed at length below using the windows shown in Figures 61 and 62.

(Untitled) - sourcePoint Class			sourceCount Derivation Statement	
			[sourceCount := class instanceCount]	
Menu	sourceCount	supplyTotal		
Values	4	58000		

(Untitled) - sourcePoint Instances		
Menu	sourceName	supply
1	Dallas	18000
2	Chicago	23000
3	Denver	7500
4	Boston	17500

(Untitled) - sourcePoint New Instance		
17500		
Menu	sourceName	supply
New	Boston	17500

Figure 61. Source Point Class Window

(Untitled) - transportationModel Instances		
Menu	modelName	totalCost
1	Widgets	1613658
2	Gadgets	0

supply Range Statement			
[supply >= 8]			

transportationModel - Widgets - sourcePoint Instances			
Menu	sourceName	supplyTest	supply
1	Dallas	false	18000
2	Chicago	true	23000
3	Denver	true	7500
4	Boston	false	17500

transportationModel - Widgets - link Instances					
1069800					
Menu	sourcePoint	destPoint	linkCost	flow	linkTotal
1	Boston	Stillwater	23.45	18000	234500
2	Dallas	Stillwater	53.49	28000	1069800
3	Boston	Tulsa	13.45	23000	309350

Figure 62. Transportation Model Class Window

## Window Components

Each of the three windows discussed above has three components. These components are: (1) window label; (2) edit pane; and (3) cell pane. Each of these components has a corresponding menu. This menu is accessed by placing the cursor (represented by an arrow in Smalltalk/V but which is absent for all figures depicting Smalltalk/V display screens) anywhere within the component area of the window and pressing the right mouse button. Where appropriate these menus are discussed at length below.

### Window Label

A window label appears at the top of a window and serves two purposes. First, a window label identifies the window to the user. For example, the window shown in the upper left-hand corner of Figure 61 reflects the values of the object class attributes defined for the Source Point class.

A window label also shows an optional title. Three windows in Figure 61 show the title (Untitled). The use of a title allows one object to indicate possession of the instance object displayed within the window. Figure 62, for example, shows a window with the label transportationModel - Widgets - link Instances. This window shows the object instances of the relationship object class link which is defined for the model object



instance Widgets, a member of the model object class transportationModel.

Second, the window label indicates which window is the active window. An inactive window displays its label in white whereas the active window displays its label in black. Thus, the sourcePoint Instances window is the active window in Figure 61.

Figure 63 shows the menu associated with a window label. The cycle option causes the active window to become inactive and another window, determined by the system scheduler, becomes active. This allows the user to move through windows which may be completely overlapped by other windows. The frame option lets the user change the location and size of a window. The move option, on the other hand, lets the user change the location of a window but not its size.

#### Edit Pane

The edit pane appears directly below the label of a window. The contents of a selected cell are displayed within the edit pane. The sourcePoint New Instance window, shown in the lower left-hand corner of Figure 61, shows the value 17500 in the edit pane associated with this window. This represents the value of the object instance attribute supply for the proposed new object instance. The corresponding value cell is displayed in black to indicate

that it has been selected for editing purposes. The edit pane permits the user to edit text using several system features such as copy, cut, paste, save, search, and replace.



Figure 63. Window Label Menu

These features are made available to the user through the edit pane menu. Figure 64 shows two edit pane menus. The first menu, on the left of Figure 64, has the option next menu which causes the second menu, on the right, to be displayed. Selecting save notifies the cell pane that the user wishes to save the edited text. It becomes the responsibility of the cell pane to act appropriately in saving this text. The cell pane saves the edited text as an object attribute value within the instance object (or a pending object instance attribute value when organizing a new object instance).

Changes made to either a derived attribute or an object instance attribute which participates in the object instance identifier list are disallowed by the

corresponding cell panes. This is consistent with the Metamodel class definition of these attributes (e.g., a derived attribute may only change when the attributes on which it depends change whereas updates of attributes comprising an object instance identifier are not valid). The values of these attributes may, however, be shown within the edit pane but may not be changed.

restore	print
copy	search
cut	search back
paste	replace all
show it	again
do it	
save	
next menu	

Figure 64. Edit Pane Menu

### Cell Pane

The remaining portion of a window displays the cell pane. The cell pane consists of a collection of same-sized cells much like those encountered in an electronic spreadsheet. The user interacts with a cell pane by selecting various cells. The user selects a cell by placing the cursor anywhere within the cell and pressing

the left mouse button. A cell pane has four kinds of cells:

- (1) menu cell;
- (2) column heading cells;
- (3) row heading cells; and
- (4) value cells.

The menu cell, labeled Menu in a cell pane, permits the user to perform various tasks directly related to the characteristics of the cell pane. Figure 65 shows the menu which pops-up when the user selects the menu cell. The goto option allows the user to specify the row and column coordinates of a cell which is then displayed in the upper left-hand corner of the cell pane. The home option displays row one and column one in the upper left-hand corner of the cell pane. The corner option displays the last row and last column defined in the upper left-hand corner of the cell pane. The up, down, left, and right options allow the user to move through the cell pane either one row or column, page, or to the end of the cell pane in the specified direction. The width option allows the user to change the width of the cells in the cell pane and the reverse option exchanges the foreground and background colors of the cell pane.

Column heading cells serve two purposes. First, they display the attribute names associated with the values appearing in the columns of the cell pane. For example, in

Figure 61 the supply object instance attribute values for the first four object instances of the Source Point class are 10,000, 23,000, 7,500, and 17,500 respectively.

Second, selecting a column heading cell causes a menu to pop-up permitting the user to view information concerning the selected attribute.

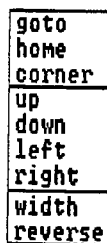


Figure 65. Cell Menu

Figure 66 shows one such menu. The show attribute option presents the user with a Smalltalk/V menu message. A menu message is simply a one line menu which the user may select by placing the cursor in the menu area and pressing the left mouse button or may cancel by pressing the left mouse button outside of the menu area. This menu message displays the full attribute name, an optional {Identifier} flag indicating that the attribute participates in the object instance identifier list, the attribute kind, and the attribute type. The structure of the menu message is

very similar to the attribute definition appearing in a schema abstraction.

show attribute
show range
show derivation

Figure 66. Column Heading Menu

Selecting the show range option causes a Smalltalk/V window to open which contains the range statement associated with the selected attribute. The window labeled supply Range Statement in Figure 62 is such a window. Similarly, selecting show derivation displays the attribute's derivation statement within a window, as is the case for the window labeled sourceCount Derivation Statement in Figure 61. Note that the show derivation option will only appear for derived attributes.

Selecting a column heading cell for each of the three windows described (class, instances, and new instance) invokes the same actions. Selecting a row heading cell varies significantly according to which window the user is viewing. Furthermore, the class and new instance windows only show a single row in the cell pane. For the class window the values of this row are the various values of the

object class attributes and selecting the corresponding row heading cell, labeled Values, has no effect. The values shown in the first row of the new instance window are those which the user edits prior to creating the new object instance. In addition, only fixed and solver-derived attributes appear in the new instance window since these are the only attributes used to define a new object instance (see the new:aValueList message in Chapter V). As with the class window, selecting the corresponding row heading cell, labeled New for this window, has no effect.

Selecting a row heading cell for an instances window causes one of the two menus in Figure 67 to appear. The two item menu on the left of this figure appears for instance objects created by the Entity, Relationship, and Relation class objects. The remove option permits the user to remove the selected object instance from the object class. If the user decides to remove an object instance a menu message appears requiring him or her to confirm the removal of the object instance. The user confirms the removal by selecting the menu message or may avoid removing the object instance by pressing the left mouse button outside of the menu message. This has the same effect as selecting the cancel menu option.

The right-hand menu in Figure 67 appears in a model instances window when the user selects a row heading cell. This menu has four options in addition to remove and

cancel. First, the open entity and open relationship options have similar effects. If the user selects the open entity (open relationship) option a menu appears listing object class identifiers for the entities (relationships) defined in the object entity (relationship) definition list of the model instance object. Selecting one of these object class identifiers causes the instances window of the entity (relationship) instance object created for the selected model object instance to open. This allows the user access to a model object instance's associated entity instance and relationship instance objects.

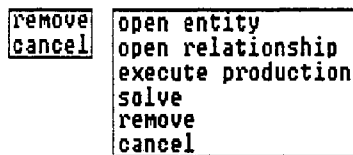


Figure 67. Row Heading Menus

The execute production option appears only if productions are defined for the selected model object instance. Selecting this option causes another menu to appear listing the names of the various productions defined for the this model object instance. The user is able to execute a production by selecting one of the names



appearing in this list. This causes a prompter to appear allowing the user to specify an optional input value for the production. A prompter is a labeled, one line window which will not relinquish control to the system without the user specifically accepting its input or canceling the prompter.

Finally, the solve option appears only if the selected model object instance requires a new solution. A new solution for this model object instance is necessary when any of the underlying entity instance or relationship instance objects changes in some manner (see the discussion of the Model class in Chapter V). Figure 68 shows a flowchart representing the process undertaken by a model instance object when one of its object instances requires a new solution. Figure 69 shows the various interactions which occur between the objects involved in generating a new solution for a model object instance. The tail of an arrow in this figure indicates which of the two objects is in the role of an actor (see the discussion concerning object roles in Chapter II). The head of an arrow indicates that an object is in the role of a server when the object interaction occurs. For example, the model instance object (in the actor role) requests that the solver object (in the server role) generate a new solution for a specific model object instance (this is shown by the Requests solution of object instance arrow in Figure 69).

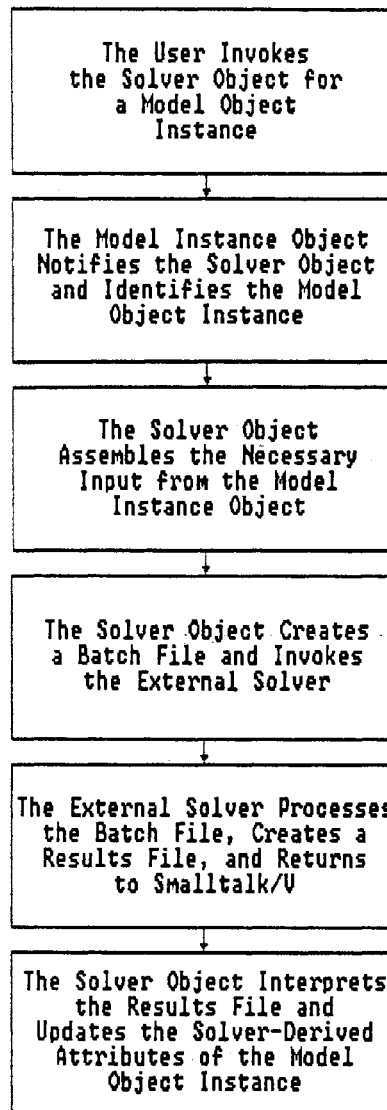


Figure 68. Solution Process Flowchart

The value cells of the cell pane show the attribute values corresponding to the object class, object instances, and new instance for the class, instances, and new instance windows respectively. The user may select any of these cells which, as discussed above, causes the corresponding

value to be displayed in the edit pane. Figure 62 shows the value 1069800 in the edit pane of the transportation - Widgets - link Instances inactive window. This is the value of the object instance attribute linkTotal for the second object instance. Selecting a value cell causes the selected cell to be displayed in inverse (foreground and background colors reversed).

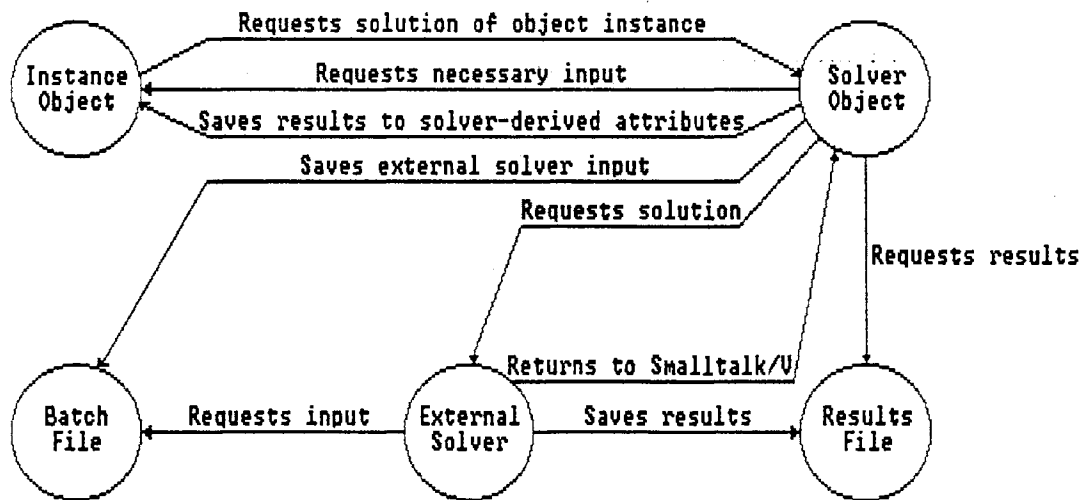
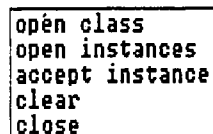


Figure 69. Solution Process Object Interactions

The user may select the save option in the edit pane menu (see Figure 64) thereby invoking an update operation on the selected attribute. Selecting any other cell in the cell pane causes a forced save operation. Saving the value of the edit pane is disallowed for derived attributes and

object instance attributes which participate in the object instance identifier list. Saving a value in the edit pane of a new instance window has no immediate effect on the object class. The object class is only affected when the user specifically accepts the collection of values in this window by selecting the accept instance option appearing in its cell pane menu.

The particular cell pane menu displayed when the user presses the right mouse button within a cell pane varies according to the type of window viewed (e.g., class, instances, or new instance window). For the new instance window the corresponding cell pane menu appears in Figure 70. The open class and open instances options permit the user to open these windows for the given instance object. Each of these options is absent from the menu if the corresponding window is already open.



```
open class
open instances
accept instance
clear
close
```

Figure 70. New Instance Window Cell Pane Menu

The accept instance option takes the values appearing in the New row for the various fixed and solver-derived

attributes of the object class and attempts to create a new object instance using these values. A menu message appears indicating whether the instance object was able to create the new object instance. Failure to create the new object instance may be due to a type violation, range violation, or duplicate object instance.

The clear option simply erases the edit pane and any values saved in the value cells appearing in the New row. The close option closes the window. Any values appearing in the New row or edit pane are discarded.

Figure 71 shows the cell pane menus for the class and instances windows. The left two menus appear when overrides are enabled for the given instance object. The disable overrides option permits the user to disable overrides for the specific instance object. This has the same effect as sending the instance object the overrideDisable message (refer to the explanation of the Metamodel class in Chapter V). The remove class overrides option is present if at least one override has been defined for a class attribute. Likewise, the remove instance overrides option is present if at least one override has been defined for an object instance attribute of any object instance. Selecting the remove class overrides (remove instance overrides) option causes the message removeAllClassOverrides (removeAllInstanceOverrides) to be sent to the instance object.

disable overrides remove class overrides remove instance overrides remove override execute production open entity open class open instances new instance close	disable overrides remove class overrides remove instance overrides make override execute production open entity open class open instances new instance close	enable overrides execute production open entity open class open instances new instance close
---	---	--

Figure 71. Class and Instances Window Cell Pane Menus

The remove override option appearing in the left-hand menu in Figure 71 is present when the selected value cell is a derived attribute and an override value for the attribute is in effect. This option permits the user to remove the previously set override value. The menu in the middle of Figure 71 shows the option make override. This option, like the remove override option, is present when the selected value cell is a derived attribute and, contrarily, when no override value for the attribute is in effect. This option allows the user to define an override for a derived attribute. These options are present in the cell pane menus for both the class and instances windows.

The execute production option, however, is present only for the class window. Like its object instance counterpart, this option appears only if productions are defined for the object class. Selecting this option causes another menu to appear listing the names of the various productions defined for the object class. The user is able

to execute a production by selecting one of the names appearing in this list. This causes a prompter to appear allowing the user to specify an optional input value for the production.

The open entity option is present only for the instances window of a relationship instance object. If the user selects the open entity option a menu appears listing object class identifiers for the entities defined in the object entity mapping definition list of the relationship instance object. Selecting one of these object class identifiers causes the instances window of the corresponding entity instance object to open. This allows the user to access a relationship instance object's associated entity instance objects.

The open class, open instances, and new instance options will appear only if the corresponding windows (e.g., class, instances, and new instance) are not already open. Selecting one of these options will open the associated window for the given instance object. The close option closes the window.

The right-hand menu of Figure 71 shows the menu options available when overrides are disabled. The only new option is the enable overrides option. This option, present only if overrides have been disabled, allows the user to enable overrides. This has the effect of sending the overrideEnable message to the specific instance object.

Figure 72 shows the various interactions which occur between the user, window label, edit pane, and cell pane. The tail of an arrow in this figure, as in Figure 69, indicates which of the two objects is in the role of an actor whereas the head of an arrow indicates that an object is in the role of a server. For example, the cell pane (in the actor role) requests that the instance object (in the server role) return the value of an attribute displayed by the cell pane (shown by the Requests text for cell values arrow in Figure 72). The final user interaction level builds on the first two levels and is discussed below.

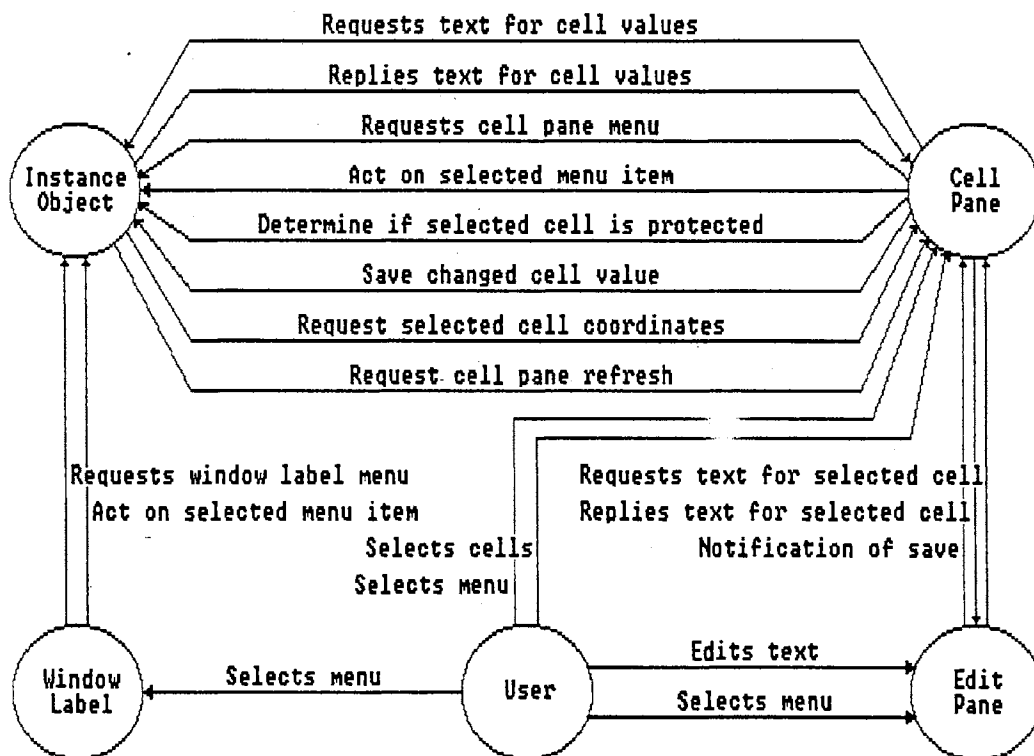


Figure 72. Window Level Object Interactions



## DSS Browser Level

The final user interaction level is the DSS Browser level. The DSS Browser presents the user with a Smalltalk/V window which contains a single pane, called a list pane. A list pane presents the user with a list of items from which he or she may select a single item. The selected item is shown in inverse. The user may scroll through the list and act on a selected item through the list pane menu. A list pane differs from a menu in that a menu immediately acts on the selected item whereas a list pane waits for the user to specify some action, through the list pane menu, to be taken for the selected item.

A DSS Browser window has two window components: (1) window label; and (2) list pane. Each of these components has an associated menu which is activated as discussed above. The menu for the window label is identical to the menu shown in Figure 63.

Figure 73 shows a DSS Browser window which contains a series of object class identifiers. In this figure the DSS Browser window is the active window and the transportationModel list item is selected. Each object class identifier shown in the list pane of a DSS Browser window comes from either a model instance or a relation instance object. The user may select any object class identifier appearing in the list pane. The list pane menu, which appears in Figure 74, permits the user to perform one

of two actions. First, the open option allows the user to open the instances window of the instance object having the selected object class identifier. Thus, the user may open a model or a relation in an equivalent manner. The close option closes the DSS Browser window.

The screenshot shows the DSS Browser window with a list of model classes on the left and two data tables on the right.

**DSS Browser**

- assignmentModel
- canneries
- generalLPModel
- link
- networkModel
- transportationModel
- warehouses

**(Untitled) - transportationModel Instances**

Item	modelName	totalCost
1	Widgets	2683450
2	Gadgets	0

**transportationModel - Widgets - link Instances**

Item	sourcePoint	destPoint	linkCost	flow	linkTotal
1	Boston	Stillwater	23.45	10000	234500
2	Dallas	Stillwater	53.49	40000	2139600
3	Boston	Tulsa	13.45	23000	309350

Figure 73. DSS Browser Window

open  
close

Figure 74. List Pane Menu

The DSS Browser window is a special window created by the DSS class object. The message open is sent to the DSS class object which collects all model instance and relation instance objects, determines their object class identifiers, and constructs the list used by the list pane. Thus, a single message is required by the user to gain access to all the models and data defined in the system. Figure 75 presents a flowchart which describes the process undertaken by the user for this level of interaction.

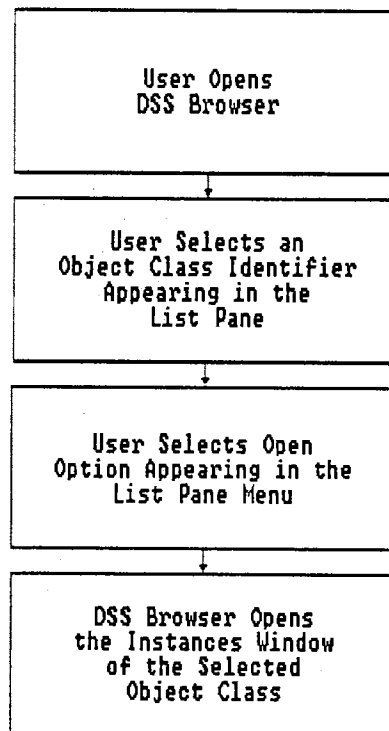


Figure 75. DSS Browser Level Flowchart

## Window Level Data and Model Distinctions

The previous sections draw no distinction between the way in which the user accesses relation instance objects and other subclass (Entity, Relationship, and Model) instance objects. Recall from Chapter V that the user may pass general messages to access object class and object instance attributes regardless of the superclass object. The user may, however, treat relation instance objects in a more specific manner. That is, the user may act as though he or she is dealing specifically with a relation rather than an instance of the Metamodel class. The Relation class accomplishes this by providing a message level view which corresponds to a relational database approach.

Concurrently, the Relation class also provides a similar window level view. From this perspective the user may open two different windows corresponding to a single relation using the following messages: (1) open; and (2) openNew.

The open message opens a relation window which shows the relation attributes for all the tuples defined for the receiver. The openNew message opens a new tuple window which permits the user to organize a new tuple and subsequently attempt to insert the new tuple into the relation.

Figure 76 shows both these windows for the suppliers relation. The active window in this figure is labeled suppliers Relation. Note that the window labeled suppliers New Tuple is a new tuple window and that the row heading cell for this window is labeled Insert. Thus, these windows employ terms consistent with relational database concepts. The window label menu for both these windows is identical to that shown in Figure 63.

status Range Statement				
[status >= 0]				
suppliers Relation				
Blake				
Item	sNumber	sName	status	city
1	S1	Smith	20	London
2	S2	Jones	10	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens
suppliers New Tuple				
Boston				
Item	sNumber	sName	status	city
Insert	S6	Nathan	20	Boston

Figure 76. Suppliers Relation Window

Selecting the row heading cell in a new tuple window has no effect. Selecting a column heading cell in this window and in a relation window displays the menu in Figure

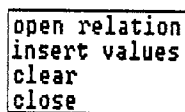
66. The show derivation option is always absent as the definition for a relation instance object given in Chapters III and V states that all its associated object instance attributes must be fixed. The show attribute option presents the user with a menu message showing the attribute name, an optional {Key} flag indicating that the attribute participates in the primary key, and the attribute type. The structure of the menu message is very similar to the attribute definition appearing in a data schema abstraction.

The cell pane menu for a new tuple window is shown in Figure 77. The open relation option is present if the corresponding relation window is not already open. Selecting this option causes the relation window to open. The insert values option performs the same function as the accept values option shown in Figure 70. This is also the case for the clear and close options.

The cell pane menu for a relation window appears in Figure 78. The new tuple option is present if the corresponding new tuple window is not already open. Selecting this option permits the user to add new tuples to the relation. The close option closes the relation window.

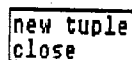
Selecting a row heading cell in a relation window pops-up the menu appearing in Figure 79. Notice that the delete option is analogous to the remove option in the menu shown in Figure 67. Thus, the user may, if desired,

interact with the data component of the O-O DSS using either the object class approach defined by the Metamodel class object or may use the relational database approach defined by the Relation class. Support for either of these two approaches is provided at both the message level and window level of user interaction.

A rectangular menu box with a black border containing four text items stacked vertically: "open relation", "insert values", "clear", and "close".

```
open relation
insert values
clear
close
```

Figure 77. New Tuple Cell Pane Menu

A rectangular menu box with a black border containing two text items stacked vertically: "new tuple" and "close".

```
new tuple
close
```

Figure 78. Relation Cell Pane Menu

A rectangular menu box with a black border containing two text items stacked vertically: "delete" and "cancel".

```
delete
cancel
```

Figure 79. Relation Row Heading Menu

status Range Statement				
[status >= 0]				

(Untitled) - suppliers Instances				
Item	sNumber	sName	status	city
1	S1	Smith	20	London
2	S2	Jones	10	Paris
3	S3	Blake	30	Paris
4	S4	Clark	20	London
5	S5	Adams	30	Athens

suppliers New Tuple				
Boston				
Item	sNumber	sName	status	city
Insert	S6	Nathan	20	Boston

Figure 80. Suppliers Class Window

Figure 80 shows the same instance object displayed in Figure 76, the suppliers instance object, using the Metamodel approach. Note that the active window in Figure 76 is labeled suppliers Relation and that all menuing within this window uses relational database definitions. The active window in Figure 80, on the other hand, is labeled suppliers Instances and all user interactions with this window use terms defined by an O-O approach to DSS. Consequently, the user may interact with the same object using two different approaches.



In conclusion, the three levels of user interaction (message level, window level, and DSS Browser level) permit varying degrees of access to the objects which exist in an O-O DSS. Furthermore, our prototype shows that an O-O DSS is a realistic possibility.

## CHAPTER VII

### FUTURE RESEARCH DIRECTIONS

The present study shows that an object-oriented (O-O) decision support system (DSS) is a viable possibility. Regardless, there are several directions which future research endeavors might pursue.

First, the data component of the current O-O DSS relies heavily on relational data modeling concepts. While a relational data modeling approach has significant benefits, it also has obvious limitations as discussed in Chapter II. An O-O data modeling environment would more naturally permit an O-O DSS user to incorporate semantic information present in the task environment. This would also extend the capabilities of the data component by including object level behaviors. The inclusion of an O-O data modeling component rather than an O-O relational data modeling component would perhaps improve the current architectural design.

The current study ignores the issues of model selection and model sequencing. Additional research might suggest possible ways of addressing the problems of model selection and model sequencing or possibly integrate an O-O

DSS model representation scheme into a system currently automating these processes.

Also, an automatic problem solver selection mechanism should be addressed. Currently the user must specify which object is to serve as the problem solver object. In an automated process the O-O DSS would scan the model representation and automatically select the problem solver object best able to handle the requirements of the given model.

The proposed O-O DSS suggests a better design than existing systems because of its ability to integrate data and models as well as presenting DSS components in a more natural light as objects. One of the strongest arguments associated with an O-O approach is that it more naturally models the environment than traditional approaches. Empirical support must be provided for this argument. Thus, the direction of an empirical investigation into the effectiveness of an O-O approach to DSS design as compared to existing systems is another area of possible future research.

The current implementation provides the underpinnings for a DSS driver. A DSS driver exists in a two layer DSS. A specific DSS relies on the DSS driver to support the data and the modeling functions of the DSS through a standard set of predefined operations. The message protocols defined for the current implementation provide this

standard set of operations necessary to implement a specific DSS using the O-O DSS as the DSS driver. Thus, the fundamental set of objects described in Chapter V along with their related set of message protocols could be used to quickly develop specific DSSs. This permits the DSS builder to focus on the user interface rather than the basal functions of the DSS. Other research efforts might address the viability of the current design architecture in providing this level of support.

Finally, a considerable amount of ongoing research addresses the topic of object sharing (Kim and Lochovsky 1989). Object sharing issues are addressed in several O-O research areas such as data modeling and office information systems. This raises the broader question of whether an O-O DSS design may be extended to include group decision making support in the form of an O-O group DSS (GDSS). GDSSs are the focus of extensive research (for example, DeSanctis and Gallupe 1987, Burns, Rathwell, and Thomas 1987, Gray 1987, Kraemer and King 1988). Extending O-O DSS concepts to GDSS architectural design should also be addressed by future research.

An O-O approach to DSS design is obviously replete with future research directions. This suggests that O-O DSS design issues provide a strong basis for developing a future research agenda.

## CHAPTER VIII

### SUMMARY AND CONCLUSIONS

Chung (1984) states that the design for any system should consist of different levels of abstraction which may be conceived as a continuum from conceptual constructs, to operational constructs, and then to implementational constructs. We introduced an object-oriented (O-O) decision support system (DSS) architecture which permits the O-O DSS user to progress through these three levels of abstraction in designing data and models for the DSS. We made this possible in part through the introduction of an O-O relational data model capable of handling the data component of a DSS. In addition, we delineated an O-O structured model representation scheme to manage the model component of the DSS.

Aided by a proposed diagrammatic technique the user creates either a data model schema or model schema. The user then abstracts from the schema using either data model schema or model schema abstraction. This process of data model schema and model schema development followed by data model schema and model schema abstraction permits the user

to develop conceptual constructs representative of the task environment.

We also proposed a set of class objects arranged in an inheritance hierarchy and having corresponding message protocols. These protocols provide the O-O DSS user, O-O relational data model, and O-O structured model with the power to interact with one another. Essentially this endows the user and the O-O DSS with the ability to operationalize these representations.

Finally, we constructed a prototype O-O DSS in a personal computing environment. This prototype was capable of implementing our proposed class objects and message protocols. The prototype attests to the feasibility of an O-O DSS and shows that the third level of abstraction, implementation, proposed by Chung (1984) is a realistic possibility. Thus, the prototype shows that DSS users can effectively implement their data model schema and model schema abstractions. At this level we see that data and models may be treated in a likewise manner. Furthermore, data may be differentially viewed in a traditional sense or in a manner not unlike models.

## LITERATURE CITED

- Abbott, R. J. Knowledge Abstraction. Communications of the ACM, Volume 30, Number 8 (August 1987), 664-671.
- Abiteboul, S., and Hull, R. IFO: A Formal Semantic Database Model. ACM Transactions on Database Systems, Volume 12, Number 4 (December 1987), 525-565.
- Ackoff, R. L. Management Misinformation Systems. Management Science, Volume 14, Number 4 (December 1967), B 147-B 156.
- Adelman, L. Real-Time Computer Support for Decision Analysis in a Group Setting: Another Class of Decision Support Systems. Interfaces, Volume 14, Number 2 (March-April 1984), 75-83.
- ✓ Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C., and Soderlund, L. An Architecture for Object Management in OIS. ACM Transactions on Office Information Systems, Volume 2, Number 3 (July 1984), 173-196.
- Ahn, T., and Grudnitski, G. Conceptual Perspectives on Key Factors in DSS Development: A Systems Approach. Journal of Management Information Systems, Volume 2, Number 1 (Summer 1985), 18-32.
- Aho, A., Hopcroft, J., and Ullman, J. Data Structures and Algorithms. Addison-Wesley Publishing Company, Reading, MA, 1983.
- Alavi, M., and Henderson, J. C. An Evolutionary Strategy for Implementing a Decision Support System. Management Science, Volume 27, Number 11 (November 1981), 1309-1323.
- Alter, S. A Taxonomy of Decision Support Systems. Sloan Management Review, Volume 19, Number 1 (Fall 1977), 39-56.
- Andriole, S. J. The Design of Microcomputer-Based Personal Decision-Aiding Systems. IEEE Transactions on Systems, Man, and Cybernetics, Volume SMC-12, Number 4 (July/August 1982), 463-469.

- Anthony, R. N. Planning and Control Systems: A Framework for Analysis. Harvard University Graduate School of Business Administration, Boston, MA, 1965.
- Applegate, L. M., Chen, T. T., Konsynski, B. R., and Nunamaker, Jr., J. F. Knowledge Management in Organizational Planning. Journal of Management Information Systems, Volume 3, Number 4 (Spring 1987), 20-38.
- Bahl, H. C., and Hunt, R. G. Decision-Making Theory and DSS Design. Data Base, Volume 15, Number 4 (Summer 1984), 10-14.
- Bancilhon, F. A Logic-Programming/Object-Oriented Cocktail. SIGMOD Record (ACM), Volume 15, Number 3 (September 1986), 11-21.
- Bancilhon, F. Object-Oriented Database Systems. In Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (Austin, TX, March 21-23). ACM, New York, 1988, pp. 152-162.
- Barrett, S., and Konsynski, B. Inter-Organization Information Sharing Systems. MIS Quarterly, (Special Issue 1982), 93-105.
- Bergin, J., and Greenfield, S. What Does Modula-2 Need to Fully Support Object Oriented Programming? SIGPLAN Notices (ACM), Volume 23, Number 3 (March 1988), 73-82.
- Bhaskar, K. S. How Object-Oriented is Your System? SIGPLAN Notices (ACM), Volume 18, Number 10 (October 1983), 8-11.
- Bic, L., and Gilbert, J. P. Learning From AI: New Trends in Database Technology. Computer, Volume 19, Number 3 (March 1986), 44-54.
- Blaha, M. R., Premerlani, W. J., and Rumbaugh, J. E. Relational Database Design Using an Object-Oriented Methodology. Communications of the ACM, Volume 31, Number 4 (April 1988), 414-427.
- Blanning, R. W. What is Happening in DSS? Interfaces, Volume 13, Number 5 (October 1983), 71-80.
- Blanning, R. W. A Relational Framework for Information Management. In Decision Support Systems: A Decade in Perspective. McLean, E. R., and Sol, H. G. (Eds).



Elsevier Science Publishers B. V., Amsterdam, 1986, 25-40.

- Blanning, R. W. An Entity-Relationship Approach to Model Management. Decision Support Systems, Volume 2, Number 1 (March 1986), 65-72.
- Bonczek, R., Holsapple, C. W., and Whinston, A. B. Computer-Based Support of Organizational Decision Making. Decision Sciences, Volume 10, Number 2 (April 1979), 268-291.
- Bonczek, R. H., Holsapple, C. W., and Whinston, A. B. The Evolving Roles of Models in Decision Support Systems. Decision Sciences, Volume 11, Number 2 (April 1980), 337-356.
- Bonczek, R. H., Holsapple, C. W., and Whinston, A. B. Future Directions for Developing Decision Support Systems. Decision Sciences, Volume 11, Number 4 (November 1980), 616-631.
- Bonczek, R. H., Holsapple, C. W., and Whinston, A. B. A Generalized Decision Support System Using Predicate Calculus and Network Data Base Management. Operations Research, Volume 29, Number 2 (March-April 1981), 263-281.
- Booch, G. Object-Oriented Development. IEEE Transactions on Software Engineering, Volume SE-12, Number 2 (February 1986), 211-221.
- Borgida, A. Features of Languages for the Development of Information Systems at the Conceptual Level. IEEE Software, Volume 2, Number 1 (January 1985), 63-72.
- Borgida, A., Greenspan, S., and Mylopoulos, J. Knowledge Representation as the Basis for Requirements Specifications. Computer, Volume 18, Number 4 (April 1985), 82-91.
- Brodie, M. L. On the Development of Data Models. In On Conceptual Modelling. Brodie, M. L., Mylopoulos, J., and Schmidt, J. W. (Eds). Springer-Verlag, NY, 1984, 19-48.
- Bu-Hulaiga, M. I., and Jain, H. K. An Interactive Plan-Based Procedure for Model Integration in DSS. In Proceedings of the Twenty-First Hawaii International Conference on System Sciences, 1988.

- Burns, A., Rathwell, M. A., and Thomas, R. C. A Distributed Decision-Making System. Decision Support Systems, Volume 3, Number 2 (June 1987), 121-131.
- Buzzard, G. D., and Mudge, T. N. Object-Based Computing and the Ada Programming Language. Computer, Volume 18, Number 3 (March 1985), 11-19.
- Casais, E. An Object Oriented System Implementing KNOs. In Proceedings Conference on Office Information Systems (Palo Alto, CA, March 23-25). ACM, New York, 1988, pp. 284-290.
- Chen, P. P. The Entity-Relationship Model: Toward a Unified View of Data. ACM Transactions on Database Systems, Volume 1, Number 1 (March 1976), 9-36.
- Chung, C.-H. A Network of Management Support Systems. OMEGA International Journal of Management Science, Volume 13, Number 4 (1984), 263-276.
- Clemons, E. K. Data Models and the ANSI/SPARC Architecture. In Principles of Database Design Volume I: Logical Organizations. Yao, S. B. (Ed). Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985, 66-114.
- Codd, E. F. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Volume 13, Number 6 (June 1970), 377-387.
- Codd, E. F. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems, Volume 4, Number 4 (December 1979), 397-434.
- Cox, B. J. Message/Object Programming: An Evolutionary Change in Programming Technology. IEEE Software, Volume 1, Number 1 (January 1984), 50-61.
- Cox, B. J. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley Publishing Company, Reading, MA, 1986.
- Date, C. J. An Introduction to Database Systems: Volume II. Addison-Wesley Publishing Company, Reading, MA, 1983.
- Date, C. J. An Introduction to Database Systems: Volume I. (4th Ed). Addison-Wesley Publishing Company, Reading, MA, 1986.

- DeSanctis, G., and Gallupe, R. B. A Foundation for the Study of Group Decision Support Systems. Management Science, Volume 33, Number 5 (May 1987), 589-609.
- Dolk, D. R. Data as Models: An Approach to Implementing Model Management. Decision Support Systems, Volume 2, Number 1 (March 1986), 73-80.
- Dolk, D. R. Model Management and Structured Modeling: The Role of an Information Resource Dictionary System. Communications of the ACM, Volume 31, Number 6 (June 1988), 704-718.
- Dolk, D. R., and Konsynski, B. R. Knowledge Representation for Model Management Systems. IEEE Transactions on Software Engineering, Volume SE-10, Number 6 (November 1984), 619-628.
- Elam, J. J., and Konsynski, B. Using Artificial Intelligence Techniques to Enhance the Capabilities of Model Management Systems. Decision Sciences, Volume 18, Number 3 (Summer 1987), 487-502.
- Fuerst, W. L., and Martin, M. P. Effective Design and Use of Computer Decision Models. MIS Quarterly, Volume 8, Number 1 (March 1984), 17-26.
- Geoffrion, A. M. An Introduction to Structured Modeling. Management Science, Volume 33, Number 5 (May 1987), 547-588.
- Gray, P. Group Decision Support Systems. Decision Support Systems, Volume 3, Number 3 (September 1987), 233-242.
- Gorry, G. A., and Scott Morton, M. S. A Framework for Management Information Systems. Sloan Management Review, Volume 12, Number 1 (Fall 1971), 55-70.
- Hackathorn, R. D., and Keen, P. G. W. Organizational Strategies for Personal Computing in Decision Support Systems. MIS Quarterly, Volume 5, Number 3 (September 1981), 21-27.
- Hainaut, J.-L., and Lecharlier, B. An Extensible Semantic Model of Data Base and its Data Language. In Proceeding of the IFIP Congress. North-Holland Publishing Company, Amsterdam, 1974, 1026-1030.
- Hawryszkiewicz, I. T. A Semantic Design Method. IEEE Transactions on Software Engineering, Volume SE-9, Number 4 (July 1983), 373-384.

- DeSanctis, G., and Gallupe, R. B. A Foundation for the Study of Group Decision Support Systems. Management Science, Volume 33, Number 5 (May 1987), 589-609.
- Dolk, D. R. Data as Models: An Approach to Implementing Model Management. Decision Support Systems, Volume 2, Number 1 (March 1986), 73-80.
- Dolk, D. R. Model Management and Structured Modeling: The Role of an Information Resource Dictionary System. Communications of the ACM, Volume 31, Number 6 (June 1988), 704-718.
- Dolk, D. R., and Konsynski, B. R. Knowledge Representation for Model Management Systems. IEEE Transactions on Software Engineering, Volume SE-10, Number 6 (November 1984), 619-628.
- Elam, J. J., and Konsynski, B. Using Artificial Intelligence Techniques to Enhance the Capabilities of Model Management Systems. Decision Sciences, Volume 18, Number 3 (Summer 1987), 487-502.
- Fuerst, W. L., and Martin, M. P. Effective Design and Use of Computer Decision Models. MIS Quarterly, Volume 8, Number 1 (March 1984), 17-26.
- Geoffrion, A. M. An Introduction to Structured Modeling. Management Science, Volume 33, Number 5 (May 1987), 547-588.
- Gray, P. Group Decision Support Systems. Decision Support Systems, Volume 3, Number 3 (September 1987), 233-242.
- Gorry, G. A., and Scott Morton, M. S. A Framework for Management Information Systems. Sloan Management Review, Volume 12, Number 1 (Fall 1971), 55-70.
- Hackathorn, R. D., and Keen, P. G. W. Organizational Strategies for Personal Computing in Decision Support Systems. MIS Quarterly, Volume 5, Number 3 (September 1981), 21-27.
- Hainaut, J.-L., and Lecharlier, B. An Extensible Semantic Model of Data Base and its Data Language. In Proceeding of the IFIP Congress. North-Holland Publishing Company, Amsterdam, 1974, 1026-1030.
- Hawryszkiewicz, I. T. A Semantic Design Method. IEEE Transactions on Software Engineering, Volume SE-9, Number 4 (July 1983), 373-384.

- Klein, G. Developing Model Strings for Model Managers. Journal of Management Information Systems, Volume 3, Number 2 (Fall 1986), 94-110.
- Klein, G., Konsynski, B., and Beck, P. O. A Linear Representation for Model Management in a DSS. Journal of Management Information Systems, Volume 2, Number 2 (Fall 1985), 42-54.
- Konsynski, B., and Sprague, Jr., R. H. Future Research Directions in Model Management. Decision Support Systems, Volume 2, Number 1 (March 1986), 103-109.
- Korth, H. F. Extending the Scope of Relational Languages. IEEE Software, Volume 3, Number 1 (January 1986), 19-28.
- Kraemer, K. L., and King, J. L. Computer-Based Systems for Cooperative Work and Group Decision Making. Computing Surveys, Volume 20, Number 2 (June 1988), 115-146.
- LeClaire, B., and Chahande, A. A Framework for the Study of Semantic Data Models. Working Paper Series #88-15, Oklahoma State University (November 1988).
- LeClaire, B., and Suh, E.-H. Object-Oriented Concepts in Information Systems. Working Paper Series #88-8, Oklahoma State University (August 1988).
- Lenard, M. L. Representing Models as Data. Journal of Management Information Systems, Volume 2, Number 4, (Spring 1986), 36-48.
- Lenard, M. L. Fundamentals of Structured Modeling. In NATO Advanced Study Institute Mathematical Models for Decision Support (Val d'Isere, Haute Savoie, France, July 26-August 6, 1987), Wed 3-Wed 23.
- Liskov, B. H., and Zilles, S. N. Specification Techniques for Data Abstractions. IEEE Transactions on Software Engineering, Volume SE-1, Number 1 (March 1975), 7-19.
- Lyngbaek, P., and McLeod, D. Object Management in Distributed Information Systems. ACM Transactions on Office Information Systems, Volume 2, Number 2 (April 1984), 96-122.
- MacLennan, B. J. Values and Objects in Programming Languages. SIGPLAN Notices (ACM), Volume 17, Number 12 (December 1982), 70-79.

- Mason, R. O., and Mitroff, I. I. A Program for Research on Management Information Systems. Management Science, Volume 19, Number 5 (January 1973), 475-487.
- Methfessel, R. Implementing an Access and Object Oriented Paradigm in a Language that Supports Neither. SIGPLAN Notices (ACM), Volume 22, Number 4 (April 1987), 83-93.
- Parker, B. J., and Al-Utaibi, G. A. Decision Support Systems: The Reality That Seems Hard to Accept? OMEGA International Journal of Management Science, Volume 14, Number 2 (1986), 135-143.
- Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, Volume 15, Number 12 (December 1972), 1053-1058.
- Rathwell, M. A., and Burns, A. Information Systems Support for Group Planning and Decision-Making Activities. MIS Quarterly, Volume 9, Number 3 (September 1985), 255-271.
- Rentsch, T. Object Oriented Programming. SIGPLAN Notices (ACM), Volume 17, Number 9 (September 1982), 51-57.
- Shoch, J. F. An Overview of the Programming Language Smalltalk-72. SIGPLAN Notices (ACM), Volume 14, Number 9 (September 1979), 64-73.
- Simon, H. A. The New Science of Management Decision. Harper and Row, New York, NY, 1960.
- Smalltalk/V Tutorial and Programming Handbook. Digital Inc., Los Angeles, CA, 1987.
- Smith, J. M., and Smith, D. C. P. Database Abstractions: Aggregation. Communications of the ACM, Volume 20, Number 6 (June 1977), 405-413.
- Smith, J. M., and Smith D. C. P. Database Abstractions: Aggregation and Generalization. ACM Transactions on Database Systems, Volume 2, Number 2 (June 1977), 105-133.
- Sprague, Jr., R. H. A Framework for the Development of Decision Support Systems. MIS Quarterly, Volume 4, Number 4 (December 1980), 1-26.
- Sprague, Jr., R. H. DSS in Context. Decision Support Systems, Volume 3, Number 3 (September 1987), 197-202.

- Stefik, M., and Bobrow, D. G. Object-Oriented Programming: Themes and Variations. The AI Magazine, Volume 6, Number 4 (Winter 1986), 40-62.
- Suh, E.-H., and Hinomoto, P. Use of a Dialogbase for Integrated Relational Decision Support Systems. Decision Support Systems (forthcoming 1989).
- Thompson, J. D. Organizations in Action. Mc Graw-Hill, New York, NY, 1967.
- Vazsonyi, A. Decision Support Systems: The New Technology of Decision Making? Interfaces, Volume 9, Number 1 (November 1978), 72-77.
- Vazsonyi, A. Decision Support Systems, Computer Literacy, and Electronic Models. Interfaces, Volume 12, Number 1 (February 1982), 74-78.
- Vierck, R. K. Decision Support Systems: An MIS Manager's Perspective. MIS Quarterly, Volume 5, Number 4 (December 1981), 35-48.
- Wagner, G. R. Decision Support Systems: The Real Substance. Interfaces, Volume 11, Number 2 (April 1981), 77-86.
- Wang, M. S.-Y., and Courtney, Jr., J. F. A Conceptual Architecture for Generalized Decision Support System Software. IEEE Transactions on Systems, Man, and Cybernetics, Volume SMC-14, Number 5 (September/October 1984), 701-711.
- Watson, H. J., and Hill, M. M. Decision Support Systems or What Didn't Happen with MIS. Interfaces, Volume 13, Number 5 (October 1983), 81-88.
- Wegner, P., (Editor). Workshop on Object-Oriented Programming: ECOOP 1987, Paris, June 18, 1987. SIGPLAN Notices (ACM), Volume 23, Number 1 (January 1988), 16-37.

VITA

Brian P. LeClaire

Candidate for the Degree of  
Doctor of Philosophy

Thesis: DECISION SUPPORT SYSTEMS: AN OBJECT-ORIENTED  
CONCEPTUAL ARCHITECTURE

Major Field: Business Administration

Biographical:

Personal Data: Born in Buffalo, New York, September  
22, 1960, the son of Leo W. and Barbara H.  
LeClaire.

Education: Graduated from Medfield Senior High  
School, Medfield, Massachusetts, in June 1978;  
received Bachelor of Arts in Psychology from  
Ripon College, Ripon, Wisconsin, in May 1982;  
received Master of Business Administration from  
the University of Wisconsin-Oshkosh, Oshkosh,  
Wisconsin, in August 1984; completed requirements  
for Doctor of Philosophy degree at Oklahoma State  
University in December 1989.

Professional Experience: Graduate Assistant, Graduate  
College, University of Wisconsin-Oshkosh,  
September 1982 to August 1984; Graduate Teaching  
Assistant, Department of Management, Oklahoma  
State University, August 1984 to August 1989;  
Assistant Professor of Management Information  
Systems, School of Business Administration,  
University of Wisconsin-Milwaukee, Milwaukee,  
Wisconsin, August 1989 to Present.