최적화 모델링 언어를 위한 객체 지향 모형 관리 체계의 개발†

허순영*

Development of an Object-Oriented Model Management Framework for Computer Executable Algebraic Modeling Languages†

Soon-Young Huh*

ABSTRACT

A new model management framework is proposed to accommodate wide-spreading algebraic modeling languages (AMLs), and to facilitate a full range of model manipulation functions. To incorporate different modeling conventions of the leading AMLs (AMPL, GAMS, and SML) homogeneously, generic model concepts are introduced as a conceptual basis and are embodied by the structural and operational constructs of an Object-Oriented Database Management System (ODBMS), enabling the framework to consolidate components of DSSs(database, modelbase, and associated solvers) in a single formalism effectively. Empowered by a database query language, the new model management framework can provide uniform model management commands to models represented in different AMLs, and effectively facilitate integration of the DSS components. A prototype system of the framework has been developed on a commercial ODBMS, ObjectStore, and a C++ programming language.

Keywords:Model Management, Database Management Systems, Mathematical Programming

1. Introduction

During the last decade, an extensive body of model management studies has been developed regarding storage, retrieval, shared use, and execution of mathematical programming models. The

* 현재 한국 과학기술원 경영과학과 조교수로 재직중이다. 서울대학교 전자공학과에서 공학사(1981), 한국과학기술원 경영과학과에서 공학석사(1983), 미국 UCLA대학교 경영대학원에서 정보 시스템분야로 경영학 박사학위(1992)를 취득하였다. 주요관심분야는 객체지향데이타 베이스의 의사결정 지원, 재무 관리 및 경영 제분야에의 응용이다.

major focus of research is on the development of a single formalism to represent model and data simultaneously;two predominant approaches have evolved:artificial intelligence (AI) methods and database modeling approaches. In AI approaches, diverse model representation methods are developed on the basis of flexible symbolic knowledge representation techniques such as semantic inheritance nets [7], first order predicate calculus [6], graph [18], frame [1,4,19]. In database modeling, relational database approaches [2,5,16] are prevalent, focusing on modelbase construction using easy set-based relations. Both approaches emphasize more on the model representation and retrieval aspect than on the model execution and solution aspect, partially because the underlying formalism such as AI is relatively less efficient for the model solving tasks requiring heavy procedural computation, and partially because solver accommodation is inappropriate in the data manipulation languages such as SQL and QUEL [5,16] in database modeling. In contrast, with an emphasis on model execution, several efforts are made at a system framework [20], and object-oriented programming language (OOPL) approaches [15,17,21]. However, these underlying platforms are yet to provide satisfactory results in terms of flexibility in accommodating models, solvers, and data in a single formalism, and functionality in facilitating interactive manipulation and execution commands. Fortunately, continuous database technology advances toward next-generation database management systems, called Object-Oriented Database Management Systems (ODBMSs) [23] present us new ways to construct a robust platform. Combined with programming language paradigm and database technology, an ODBMS facilitates object types to be user-defined, persistent (as opposed to transiency in the OOPL), sharable by multiple users, and importantly, provides a nontrivial database model and manipulation query language [12].

In parallel, recently, algebraic modeling languages (AMLs) such as IFPS/OPTIMUM [22], GAMS [3], SML [10], and AMPL [8], have gained popularity from practitioners and researchers due to their efficient and stable algebraic-notation based model representation methods and user-friendly interfaces for model execution. Proliferation of the AMLs provides opportunities for decision support system (DSS), specifically its model management component, because 1) the AML models have become new organizational asset;and 2) the AMLs provide user-friendly and semantically rich modeling power in diverse modeling domains (for instance, IFPS for what-if analysis, GAMS and AMPL for optimization, and SML for general modeling), so that they become suitable as model representation sublanguages, replacing previous proprietary model representation methods.

Inspired by the emerging challenges facing model management, this paper proposes a new model management framework based on ODBMS. The new model management framework has the following objectives : 1) accommodating the diverse model semantics in the leading AMLs (GAMS, SML, and AMPL) in a general and uniform way, 2) providing a full range of functions in model

management, including creation, retrieval, update, deletion of models as well as model execution and solution capabilities, and 3) furnishing high-level manipulation commands to facilitate manipulation and integration of modelbase, solvers, and database, based on a database query language. As a conceptual basis for the framework, we establish generic model concepts based on the systems approach, to provide simplistic but flexible modeling constructs subsuming individual AML-specific concepts. An Object-oriented Structured Query Language (OSQL), is also adopted to facilitate flexible and dynamic model management.

2. Generic Model Concepts

A **generic model** is a model represented in AMLs as an abstract representation of a real world problem. The **generic model type** is an object type for the generic model. As an external interface, the generic model has a set of ports through which it interacts with its environment, i. e. , human modelers and databases, by exchanging data. A port, characterized by a **port type,** has a unique name, a set of attributes, and operations to describe the information pertaining to algebraic expressions and data values. Three types of ports exist in terms of data exchange : **input ports** (inports), **output ports** (outports), and **mid ports** (midports). The inport and outport admit and produce data, respectively, in connection with the outside environment, while the midport holds intermediate computation results, or contains constrainsts of the model, if any. The classification of ports, specifically the midport, needs discretionary judgement based on a thorough understanding of the model semantics. The external interface provides a simplified uniform view (what is required and produced through the ports) of the model, and thus relieves non-technical users from the details of the model that may look overwhelmingly complicated.

The internal view captures the detailed semantics inside the model. The ports are grouped at a higher level by some model-specific logical constructs (e. g. , set group, parameter group, etc.) called **modules.** Modules can be organized into higher-level modules. Thus, a generic model is the highest aggregation of a number of modules which, in turn, consist of a set of ports and, possibly, lower-level modules. A module, characterized by a **module type,** has a set of attributes and operations to describe its conceptual meaning, component modules, and ports. Figure 1 illustrates the relationships between the external interface and internal view. In terms of a tree structure, all the leaf nodes become the ports while the remaining internal nodes become the modules.

To the generic model, we add a set of specific **model-execution operations,** to transform the data from inports or midports into outports by invoking a solver, or to convert a generic model into a machine-readable form. When the model-execution operations are added to the generic model, the result is a **functional model** that captures the external and internal views of a decision

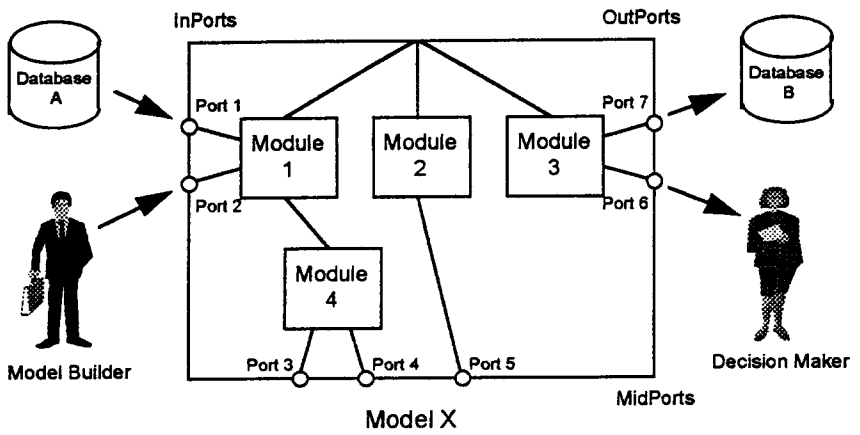

Figure 1. Conceptual Architecture of Generic Model Concepts.

model, and facilitates model-execution operations. Thus, the functional model is a specialized model which focuses on individual AML and problem-solving. Also, at the bottom of the system, there exists a solver library ; to an individual functional model, only legitimate solvers are visible to protect users from applying unauthorized, incompatible solvers to the model. Several functional models can exist at the same time as long as the types of model-execution operations differ from one another. A functional model type (or model type) is an object type of the functional model.

3. Syntax Analysis of AMPL, GAMS, and SML

AMLs' primary reliance on the algebraic notation and the indexing structures leads them to have syntactic and semantic commonalities in their model representation. From the common features, we want to extract general structures which can uniformly capture the model components independent of the idiosyncratic syntax of the individual languages, and can facilitate development of the modelbase which can store models of disparate AMLs in a single repository. To this end, in this section, we conduct a high-level syntax analysis for the three AMLs, and apply the generic model concepts in the context of the AMLs.

In the syntax analysis, square brackets, '[ ]', a vertical bar, '|', and brackets '< >' respectively denote optionality, alterneity, and repetition. A statement is different from an expression in that the statement is a syntactically complete, self-contained sentence whereas expression, as part of the statement, is an incomplete, partial assertion.

3. 1. AMPL Model

AMPL model is composed of five types of entity statements which are declared by keywords including set (set), param (parameter), var (variable), minimize/maximize (objective), and subject to (constraint). In an AMPL model, a data section is separated from the model structure, and an entity declaration statement always accompanies an assignment expression or a definition expression as part of the declaration statement. Figure 2 represents the high-level AMPL syntax of diverse types of statements.

```
set entity_name [set expression | indexing expression]; [comment]
param entity_name [{set_entity_name <, set_entity_name>}] [computation expression | qualifying expression]; [comment]
var entity_name [{set_entity_name <, set_entity_name>}] [computation expression | qualifying expression]; [comment]
subject to entity_name [{set_entity_name <, set_entity_name>}]: computation expression; [comment]
minimize|maximize entity_name : computation expression; [comment]
```

Figure 2. High-Level Syntax of AMPL Modeling Entities.

In general, each statement consists of two parts: 1) a declaration part with an entity type, entity name, and underlying indexed sets, 2) a definition part with computation or assignment expressions. From Figure 2, a general AMPL structure, uniformly capturing the five types of modeling statements, can be perceived as shown in Table 1.

fields	type	entity-name	[index]	[index/set expression]	[<expression>]	[comment]
contents	set, param, var, subject to, minimize/maximize	entity name	set entities	set expression, indexing expression,	computation expression, etc	comment

Table 1. General AMPL statement structure.

To show how a model is represented in the high-level AMPL syntax and how the generic model concepts can be practically applied, we pick a typical transportation model shown in Figure 3. The decision problem of the model is to identify optimal shipping quantity in each plant and each factory in order to minimize total transportation cost.

In Figure 3, using the comments such as SETS, PARAMETERS, VARIABLES, CONSTRAINTS, and OBJECTIVE, the individual statements are grouped together to present the abstract model structure of the transportation model at a higher level. Arranged by the comment statements, the transportation model can make other abstract model structures at modeler's will. In essence, these groups become the modules of the generic model concepts, and thus, a model, in

```
### SETS ###
set factory;                    # canning factories
set market;                     # markets

### PARAMETERS ###
param factory_sup{factory};     # supply of factory
param market_dem{market};       # demand of market
param cost{factory,market} ;    #transportation cost

### VARIABLES ###
var qty{factory, market} >= 0;  # shipment quantities

### OBJECTIVE ###
minimize total_cost:
        sum{i in factory} sum{j in market} (cost[i,j]*qty[i,j]);
### CONSTRAINTS ###
subject to supply {i in factory}:
        sum{j in market} (qty[i,j]) <= factory_sup[i];
subject to demand {j in market}:
        sum {i in factory} (qty[i,j]) >= market_dem[j];

data;
set factory     := SEATTLE SAN-DIEGO;
set market      := NEW-YORK CHICAGO TOPEKA;
param cost:     PORTLAND NEW-YORK  CHICAGO   TOPEKA  HOUSTON:=
SEATTLE            3        25        18        19        18
SAN-DIEGO         10        22        17        19        15
LOS-ANGELES        8        20        15        17        15;
param factory_sup:          SEATTLE      SAN-DIEGO LOS-ANGELES :=
                              350           600          500;
param market_dem: PORTLAND     NEW-YORK      CHICAGO       TOPEKA      HOUSTON:=
                    220           325           300           275          280;
```

Figure 3. An AMPL Transportation Model.

an abstract view, is an aggregation of several mdules. Individual modeling statements in Figure 3 identify model ports. From the external interface viewpoint, the inports of the model are the ports that admit data from users or outside the model. By virtue of explicit separation of a model structure from its data part, all ports that are instantiated by some datasets such as sets including factory and market, and parameters including freight, distance, factory−sup, and market−dem, become the inports. On the contrary, the outports of the model are the ports that will produce output values as final results of model execution; they involve the qty variable as a decision variable and the total−cost as an objective value of the model. Finally, the rest of the ports are midports; they entail all the constraints and parameters that are not instantiated within the data section (in AMPL, these parameters are termed computed parameters.) such as cost parameter, and supply and demand constraints.

3. 2. GAMS Model

GAMS model is composed of several basic entities including sets, data, variables, equations and model manipulating statements. The set, data, variable entities are declared with corresponding keywords; data assignments for the declared entities are individually followed by either separate

assignment statements or list/table data insertion commands. Declaration and definition of an equation entity are made in separate statements. GAMS model also involves a number of model manipulation statements that are not usually associated with specific entities.

In analyzing the GAMS syntax, we modify GAMS modeling conventions so that more simplistic articulation of the syntax is made possible without serious loss of GAMS modeling power : 1) a model structure is separated from data set, 2) a declaration statement includes assignment statements or definition statements if they are all associated with a single entity. Reflecting such syntactic considerations, Figure 4 delineates the high-level GAMS syntax.

entity declarations and assignment/definition statements
SCALA entity_name [comment] [assignment statement]
SET entity_name [comment] [set statement]
PARAMETER entity_name [(set_entity_name <, set_entity_name>)] [comment] [assignment statement]
TABLE entity_name (set_entity_name <, set_entity_name>) [comment] [table statement]
[range] VARIABLE entity_name [comment] [assignment statement]
EQUATION entity_name [(set_entity_name <, set_entity_name>)] [comment] .. [definition statement]

model manipulation statements
MODEL model_name detail_statement
SOLVE detail_statement
DISPLAY detail_statement
ABORT detail_statement
OPTION detail_statement
LOOP detail_statement

Figure 4. High-Level Syntax of GAMS Modeling Entities

fields	type	entity-name	[index]	[<range>]	[<expression>]	[comment]
contents	SCALA, SET, PARAMETER, TABLE, VARIABLE, EQUATION, MODEL, etc	entity name	set entities	range entity	set expression, table expression, assignment statement, definition statement, detail statement, etc	comment

Table 2. General GAMS statement structure.

Note that the high-level syntax shown in Figure 4 primarily forces such related statements (i. e. , the declaration statement and definition statement of a same entity) to be aggregated into a single statement. A general GAMS statement structure will be as shown in Table 2.

Figure 5 shows how a transportation model is actually represented in GAMS. As in the case of the AMPL model, all the statements are grouped by the entity types such as SETS, PARA-

METERS, etc. , making them modules. Under each module, individual entities are procedurally declared, assigned, or detaily defined. The port identification of the GAMS model is similar to that of the AMPL model and we omit the detailed process, but provide the summary result in the Table 6.

3. 3. SML Model

SML model is an algebraic model based on Structured Modeling (SM) framework [9] to represent analytical deterministic models by using an acyclic, attributed graph. SM provides a general modeling approach to diverse MS/OR paradigms such as mathematical programming, forecasting, regression, and simulation. Since the primary component of a structured model is a genus, the syntactic analysis of a structured model structure focuses only on the genus paragraph. Thus, the syntax of the genus (paragraph) statement is as follows :

genus−name [index−variable] [(set−genus−name 〈, set−genus−name〉)] /type/ [{index set expression}] [:range expression] [;generic rule expression] [interpretation]

The general SML statement structure can be arranged as shown in Table 3.

fields	type	entity-name	[index]	[<range>]	[<expression>]	[comment]
contents	pe, ce, a, va, f t	genus name and index variable	indexed genuses	range expression	index set expression, generic rule expression,	interpretation

Table 3. General SML statement structure.

A SML model matches well with the generic model concepts due to the modular, and hierarchical characteristics of the SM framework. Basically, the SML modules correspond to the modules of the generic model concepts while the SML genera (i. e. , plural of genus) correspond to ports. Since the calling sequence embedded among genus statements can be implemented in the ports capturing individual genera, visualization of the calling sequence among the ports, can illustrate abstract, definitional interdependence among ports (genera), supplementing the internal view of a model.

As an example, let's consider a SML transportation model shown in Figure 5.

GAMS Model	SML Model
SETS I canning factories / Seattle, SanDiego, LosAngeles / J markets /Portland,NewYork, Chicago, Topeka,Houston / PARAMETERS FACTORY-SUP(I) capacity of factory i in cases; MARKET-DEM(J) demand at market j in cases; TABLE COST(I,J) transportation cost in thousand of dollars per case; VARIABLES QTY(I,J) shipment quantities in cases TOTAL-COST total transportation costs in thousands of dollars; POSITIVE VARIABLE QTY; EQUATIONS OBJECTIVE define objective function; SUPPLY(I) observe supply limit at factory i; DEMAND(J) demand at market j; OBJECTIVE. TOTAL-COST =E= SUM(I,J)*QTY(I,J)); SUPPLY(I). SUM(J,QTY(I,J))=L= FACTORY-SUP(I); DEMAND(J). SUM(I,QTY(I,J))=G= MARKET-DEM(J) ; MODEL TRANSPORT /ALL/ ; SOLVE TRANSPORT USING LP MINIMIZING TOTAL-COST ;	&FACTORY-DATA FACTORY DATA MODULE FACTORYi /pe/ There is a list of FACTORYS. SUP(FACTORYi) /a/ {FACTORY} : Real+ For each FACTORY, there is FACTORY SUPPLY in tons. &MARKET-DATA MARKET DATA MODULE MARKETj /pe/ There is a list of MARKETS. DEM(MARKETj) /a/ {MARKET} : Real+ For each MARKET, there is MARKET DEMAND in tons. &TRANSPORTATION-DATA TRANSPORTATION DATA MODULE LINK (FACTORYi, MARKETj) /ce/ Select {FACTORY} x {MARKET} where i covers {FACTORY}, j covers {MARKET} There are some transportation LINKS from FACTORYS to MARKETS. COST (LINKij) /a/ {LINK} : Real+ Each LINK has a Transportation COST in $ /ton. QUANTITY (LINKij) /va/ (LINK) : Real+ There can be a nonnegative transportation QUANTITY in tons over each LINK. &OBJECTIVE OBJECTIVE FUNCTION MODULE TOTAL-COST (COST, QUANTITY) /f/ 1 ; @SUMi SUMj (COSTij * QUANTITYij) TOTAL COST associated with all QUANTITYS to be minimized. &CONSTRAINT CONSTRAINT MODULE T : SUPPLY (QUANTITYi, ,SUPi) /t/ {FACTORY} ; @SUMj (QUANTITYij) <= SUPi Is the total QUANTITY leaving a FACTORY less than or equal to its FACTORY-SUPPLY ? This is called the SUPPLY CONSTRAINT TEST. T:DEMAND (QUANTITY, j,DEMj) /t/ {MARKET} ; @SUMi (QUANTITYij) = DEMj Is the total QUANTITY arriving at a MARKET exactly equal to its MARKET-DEMAND? This is called the MARKET CONSTRAINT TEST.

Figure 5. Transportation models represented by GAMS and SML

In the transportation model of Figure 5, five modules are identified which are &FACTORY-DATA, &MARKET-DATA, &TRANS-DATA, &CONSTRAINT, and &OBJECTIVE. In deciding inports and outports from the individual genera, since the SML is a general-purpose AML, modelers pose an optimization problem for the model structure in advance by choosing the type of optimization (minimization or maximization), objective functions, constraints, and decision variables. Thus, in the transportation model, we choose TOTAL-COST and FLOW as objective function and decision variable respectively, and assign them as outports. In contrast, all genera which are rather close to independent parameters and element types such as primitive and compound entities (pe, ce), and attributes (a) including FACTORY, SUP, MARKET, DEM, LINK, COST become inports since they are instantiated with datasets supplied by human modeler or external data sources. The rest of genera which are similar to dependent parameters or variables

with element types such as function (f) or test (t) including COST, T : SUPPLY, and T:DEMAND become midports.

3. 4. A Core Structure Capturing a Modeling Statement of AMPL, GAMS, and SML

Fundamental reliance on the algebraic notation as underlying model representation scheme in the three modeling languages leads to three similar general statement structures as shown in Table 1, 2, and 3. From the three general statement structures, we extract a core structure that is common to any of the modeling statements of the three AMLs as follows:

type [entity−name] [⟨index⟩] [⟨expression⟩] [comment]

The syntax of the core structure includes entity type, entity name, a list of indices, a list of expressions, and a camment. The common structure provides a basis for a generic port type so that it can facilitate building specialized port types for the individual AMLs by adding language-specific attributes.

4. An Object-Oriented Database Model for the Model Management Framework

This section presents an object-oriented database model, embodying the generic model concepts into persistent object types, and facilitating diverse model management commands. The database model is composed of three generic object types, which are generic model type (GenericModelType), module type (ModuleType), and port type (PortType), and two subtypes, which are functional model types and specialized port types. Additionally, DataType is introduced for the data storage requirement in the PortType. Table 4 provides the object type definitions for the database schema. Detailed description of individual types are as follows.

4. 1. GenericModelType

In the definition of GenericModelType, we want to note that the module attribute ensures referential integrity between a generic model and its component modules via binary inverse relationship. In support of the external view of the model, the three port attributes (i. e. , inport, outport, and midport) individually hold a portion of ports that have the same interfacing roles. In fact, each port attribute forms a transitive closure of a certain type of ports ; for instance, the inport attribute refers to all inports of a model no matter to which modules the inports directly belong. In

define type GenericModelType		define type Module Type	
(structure		(structure	
string	name,	string	name,
string	comment,	string	comment,
List of ModuleType	module inverse model of ModuleType,	GenericModelType	model inverse module of GenericModelType,
list of PortType	inport, outport, midport,	list of portType	port inverse module of PortType,
list of string	solvers	ModuleType	ParentModule inverse childModule,
operation		list ofModuleType	ChildModule inverse parentModule
	GenericModelType(name),	operation	
	showIntemal(),		ModuleType(name),
	showExtemal(),		ModuleType(name, model),
customizable	display(), convert(),		ModuleType(name, port),
customizable	solve(), solve(solverName),		showModuleIntermal())
customizable	type())		

define type	PortType	define DataType	
(structure		(structure	
string	name, comment	int	lastLabel, ; nidex no. of last label
ModuleType	module inverse port of ModuleType,	array[] of string	label, ; array of abels
		int	row, col, ; row and column size
string	type, unit	arry[][] of real	value, ; arry of real
set of string	expression,	operation	
list of PortType	index,		DataType(label). ; input text labels
state	iostate,		insertLable(label), ; input text labels
DataType	data,		insertValue(value), ; input numeric values
operation			printLabel(), ; print labels
	PortType(name),		printValue(), ; printvalues
	PortType(name, comment,		PrintPair(), ; print label and value
	module, type),		matrix(range), ; output matrix
customizable	showDependency()		table(range) ; output matrix
	; index dependency of model)		; with row and column labels)

Table 4. Object Type Definitions for Generic Model Concepts.

accommodating multiple ports, like the module attribute, the individual port attributes are multi-valued object-valued attributes.

Regarding model management operations for the GenericModelType, basic DBMS operations such as insert, delete, or update are presumed to be supported by the ODBMSs. In addition, to leverage an emphasis on the multiple abstract views of a model, the GenericModelType suplies several generic operatins. The showExternal() provides the general description of a model and its external interface by describing all ports in terms of inport, outport, and midport. In contrast, the showInternal() furnishes a tree structural view of the model by enumerating all modules and their component ports. The display() reassembles the decomposed model components from the modelbase, and restores a natural form of the model as shown in Figure 3 or Figure 5, while the convert() offers a machine

readable form of the model such as MPS format, and the solve() facilitates model solving. Lastly, the type() returns the model type.

However, owing to syntactic differences in individual AMLs, specifically the last four operations need to behave differently in the context of each AML. To facilitate such adaptation of generic operations to individual AML environments, functional model types are defined to inherit all the attributes and generic operations as defined in the GenericModelType, and to additionally tailor some of the generic model-execution operations by making them specific to individual modeling environments. In the creation of the functional model type, additional attributes and operations can be further incorporated for more elaborated manipulation and solution of the model. As an example of a functional model type, AMPLModelType can be defined as follows to accommodate models represented in AMPL, and provide AMPL-specific model execution operations.

```
define type AMPLModelType : supertype GenericModelType
(structure
operation
  customizable display(), ; display the whole AMPL model
  customizable convert(), ; convert the model into a standard format
  customizable solve(), ; solve the model
  customizable type() ) ; return the model type, e. g. , "AMPL"
```

Table 5 illustrates how the three types of ports are categorized in an AMPL transportation model shown in Figure 3. Figure 5 and Table 6 show how the GenericModelType can accommodate the identical transportation models represented in GAMS and SML.

4. 2. ModuleType

The key objective of the ModuleType defined in Table 4 is to capture intermediate building blocks of the model by aggregating a set of ports and child modules, while maintaining relationship to a model to which the module belongs. To do this, in addition to name and comment attributes, the ModuleType has the model attribute for an inverse reference to the model, in fact, as a matching inverse reference attribute for the module attribute of the GenericModelType. The port attribute clusters a set of component ports, and ensures another binary referential integrity between a module and its component port. Consequently, when we update a port of a model, the port as a part of a module is also updated, and thus data integrity on the port is secured.

The parentModule and childModule attributes allow the current module to be a child module as well, possibly at the same time, a parent module of other modules, facilitating the nested module construction. The nested module construction can enhance the semantic expressiveness by making

⟨transportation, AMPLModelType, decide product shipping quantity between each factory and each market to minimize total transportation cost
inport [⟨name, (index), type, indexExpr, computeExpr, comment, unit, (module)⟩]* [⟨factory, AMPLPortType, , set, , , list of factory names, , (SETS) ⟩, ⟨market, , set, , , list of market names, , (SETS)⟩, ⟨factory−sup, (factory), param, {factory}, , supply capacity of factory, ton, (PARAMETERS)⟩, ⟨market−dem, (market), param, {market}, , demand amount of market, ton, (PARAMETERS)⟩, ⟨cost, (factory, market), param, {factory, market}, ⟩= 0, transportation cost, , (PARAMETERS)⟩], outport [⟨qty, (factory, market), var, {factory, market}, ⟩= 0, transportation quantity over links, ton, (VARIABLES)⟩, ⟨total−cost, , minimize, , sum{i in factory} sum{j in market} (cost[i,j]* qty[i,j], total transportation cost, 1000 $, (OBJECTIVE)⟩] midport [⟨cost, (factory, market), param, {factory, market}, freight * distance(factory, market) /1000, transportation cost, $ /ton, (PARAMETERS)⟩, ⟨supply, (factory), subject to, {i in factory}, sum{j in market} qty[i,j] ⟨= factory−sup[i], supply constraint test, , (CONSTRAINT)⟩, ⟨demand, (market), subject to, {j in market}, , sum{i in factory} qty[i, j] ⟩= market−dem[j], demand constraint test, , (CONSTRAINT)⟩] module [⟨name, comment, (model), (component ports)⟩] [⟨SETS, index sets, (transportation), (factory, market)⟩, ⟨PARAMETERS, parameter data, (transportation), (factory−sup, market−dem, freight, distance, cost)⟩, ⟨VARIABLES, decision variable, (transportation), (qty)⟩, ⟨OBJECTIVE, objective function, (transportation), (total−cost)⟩, ⟨CONSTRAINT, constraints, (transportation), (supply, demand)⟩] operation / showInternal(), and showExternal() are inherited from Generic Model Type customized display(), customized solve( ), / simplex customized convert(), / MPS file format

* Bold-face templates at the inport and module attributes represent simplified templates for the PortType and ModuleType. Note that all port instances are of the AMPLPortType, a subtype of the PortType. To differentiate object values from data values, all object values are enclosed by parentheses.

Table 5. The Model Structure of AMPL Transportation Model.

	GAMSModelType	SMLModelType
name	"transportation"	"transportation"
comment	"Minimizing the cost of transporting products from factories to markets"	"Minimizing the cost of transporting products from factories to markets"
module	(set), (parameter), (table), (scala), (variable), (equation), (manipulation)	(&FACTORY−DATA), (&MARKET−DATA), (&TRANS−DATA), (&RESULT)
inport	(factory), (market), (factory−sup), (market−dem), (cost)	(FACTORY), (SUP), (MARKET), (DEM), (COST)
outport	(qty), (total−cost)	(TOTAL−COST), (QUANTITY), (T:SUPPLY), (T : DEMAND)
midport	(supply−cons), (demand−cons), (model), (solve)	(LINK)

Table 6. Generic Models for GAMS and SML transportation models represented by GAMS and SML

it consistent with the natural semantics of some modeling languages such as SML. In general, the ModuleType is flexible enough to adapt itself to diverse kinds of logical constructs that are unique to individual modeling languages. In Table 5, module examples for the AMPL transportation model are shown.

4. 3. PortType

The basic structure of a port is already identified from the core structure of the modeling statements of AMLs. Additionally, the port incorporates temporary data storage to interact with the surrounding databases or other models. With such objectives, the PortType has name and comment attributes. The module attribute facilitates an inverse reference to the module to which the port is directly attached. The type attribute is used to specify the base type of a model entity which is captured as a port. The unit and expression attributes respectively define the unit of data value, and contain an algebraic expression. The index attribute can contain an index definition of the port. For example, in the AMPL model, the "cost" port contains object references to its index sets such as "factory" and "market". The iostate attribute specifies the interfacing role of the port in terms of inport, outport, or midport. Again, in Table 5, ports arranged by the external interfacing roles with the detailed descriptions are illustrated in the AMPL transportation model.

As the last attribute, the data attribute serves as a resident data repository for the port. In support of the data storage, the DataType primarily provides two independent, variable-length arrays to accommodate text labels as well as numeric values in diverse formats, ranging from a single real number to a matrix. Additionally, to leverage the interfacing role of a port, it provides various operations to facilitate data manipulation and port integration with source databases. The flexibility of the two arrays combined with user-defined, array manipulation operations allow the port to effectively accommodate nontrivial data sets such as numeric array, label array, array of label and numeric value in pairs, or numeric matrix with row and column labels. As an example, consider the inports "market" and "market-dem" of the AMPL transportation model where the markets are located in New York, Chicago, and Topeka, and their corresponding market demands are respectively 325, 300, 275 tons. To store these data sets, the "market" port employs the label array in the data attribute of the port, while the "market-dem" port employs the numeric array. In support of these input tasks, the insertLabel() and insertValue() operations are specifically provided to take an array of text labels and an array of real numbers. For output operations, the printLabel() and printValue() respectively print out text labels and numeric data values of a port in an array form. To retrieve matrix data, the matrix(range) or the table(range) can be used ;

while the matrix() retrieves only a numeric matrix, the table() operation supplements the matrix with the row and column text labels, enhancing matrix readability.

Since the PortType embodies the core parts of the three AML ports, to adequately incorporate all the language-specific data structures and operations, the PortType specializes itself to three subtypes including AMPLPortType, GAMSPortType, and SMLPortType.

5. Model Manipulation Language

As a model manipulation language (MML), a hypothetical OSQL is used, since it could easily illustrate powerful features that the relational SQL does not have : navigational capability and invocation of operations of an object type. The navigational path among individual objects is hierarchically addressed by a nested dot notation, (.), and makes a normal nested SQL to be more simplistic and intuitive. Queries presented subsequently are all simulated by the ObjectStore query language [14].

5. 1. Model Instantiation Queries

In the model management framework, the modelbase is made up of three interrelated collections of objects : MODELS, MODULES, and PORTS. The three collections are matched with three major object types : GenericModelType, ModuleType, and PortType. Though there may exist multiple functional model types, all the functional models are accommodated by the MODELS collection. To show typical model structure instantiation procedures, several commands are exemplified. The following command instantiates an AMPL model named "transportation" into MODELS.

```
INSERT AMPLModelType into MODELS
(name = "transportation",
comment = "decide shipping quantity between each factory and each market to minimize
total transportation cost")
```

The INSERT command is also used to create a module and its child port, and insert them into the MODULES and PORTS collections while interrelating them with their parent model in the MODELS collection. In the following two commands dealing with the AMPL transportation model, the first one shows how "parameter" module is created and inserted into MODULES with a proper reference to the model. The second one shows how "factory-sup" port is created and inserted into PORTS in relation to the "parameter" module. Note that inserting the module into the transportation model automatically assigns the module's port into the model's external interface.

```
INSERT ModuleType into MODULES
(name = "parameter", comment = "input parameter data", model = M)
FROM M in MODELS
WHERE M. name = "transportation" and M. type() = "AMPL"

INSERT PortType into PORTS
(name = "factory-sup", comment = "supply capacity of factory", iostate = INPORT,
module = MD)
FROM MD in MODULES
WHERE MD. name = "parameter"
```

These commands reflect the advantages of an ODBMS approach by supporting object-orientation and referential integrity. Object-orientation means that, in these commands, the parameters are specified directly by objects (e. g. , model = M, module = MD) than by indirect values ; this considerably simplifies model manipulation queries by nullifying the need for supplementary relations that is often required in a relational approach.

5. 2. Model Retrieval Queries

Once models are instantiated and accumulated in the modelbase, the modelbase can be utilized for various purposes at the individual user level. For instance, the decision maker, who prefers to handle models at the most abstract level for retrieving and running models, and understanding their outputs, can type the following command to view the external interface of a model.

```
SELECT M. showExternal()
FROM M in MODELS
WHERE M. name = "transportation" and M. type() = "AMPL"
```

For knowledgeable model builder, showInternal() is available in place of showExternal(). Depending on the model type, additional internal view of model is available ; for instance, for SMLModelType, showCallingSequence() would visualize definitional interdependencies among ports (i. e. , genera). Meanwhile, for browsing data stored in a specific port, a variety of convenient retrieval methods are available on the basis of operations encapsulated in the DataType ; we can retrieve the location and the capacity of each factory of the transportation model using the printPair() operation in the following query.

```
SELECT P. data. printPair()
FROM P in PORTS
WHERE P. name = "factory-sup" and P. model. name = "transportation"
      and P. model. type() = "AMPL"
```

Or, by specifying ranges of matrix portion in the matrix() or table() operation, various portions of a matrix can be dynamically retrieved. The following query retrieves a portion of a distance matrix that covers all column values ranging from "new york" to "topeka" of the "san diego" row of the transportation model with corresponding row and column labels.

```
SELECT P. data. matrix("san diego", "new york" : "topeka")
FROM P in PORTS
WHERE P. name = "distance" and P. model. name = "transportation"
      and P. model. type() = "AMPL"
```

For reuse or easy development of models from the existing model blocks, the model builder may be interested in examining a particular module. This view may need to include detailed port information of the module as well as the description of the module. The following query retrieves names and expressions of all ports attached to a module named "equation", of the transportation model represented in AMPLModelType.

```
SELECT M. module. port. name, M. module. port. expression
FROM M in MODELS,
WHERE M. module. name = "equation" and M. name = "transportation" and M. type() =
"AMPL"
```

5. 3. Integration among modelbase, database, and solvers

Integration of modelbase with database, in more concrete terms, refers to a direct data exchange between a functional model and a source database. Many operations of the DataType are provided for the integration. For instance, either the insertValue() or insertLabel() operation of the DataType facilitate direct data-instantiation of a model structure ; since both operations take an array of data at a time, data instantiation focusing on the data attribute of a port can be greatly simplified by matching the port and the array of data. Assume that there is a corporate database called "MarketDB" which provides product demand information of large number of individual markets. The insertion query first retrieves the market demand for the product "PROD-A" from the corporate database, and then inserts the retrieved tuples into the inport specified as "market-dem".

```
SELECT M. inport. data. insertValue(demandArray)
FROM M in MODELS
WHERE M. name = "transportation" and M. type() = "AMPL"
      and M. inport. name = "market-dem" and
      demandArray of array[] of real
      = SELECT MKT. demand
      FROM MKT in MarketDB
      WHERE MKT. product = "PROD-A"
```

On the other hand, integration of modelbase with solver operations is naturally supported in the model management framework, since the object type definition allows encapsulation of a set of solver operations inside a model type. The following example shows how to execute a solver operation available in a functional model type.

```
SELECT M. solve()
FROM M in MODELS
WHERE M. name = "transportation" and M. type() = "AMPL"
```

Depending on a system builder's choice, solve() operation can restore an executable model instance from the modelbase, transform it into a matrix form using a modeling language translator, and submit the matrix to some external optimization solvers such as MINOS and LINDO. Current ObjectStore-based implementation follows this approach. As a whole, to both the decision maker and the model builder, encapsulation of solver operations inside a functional model type can become powerful and convenient, since it can make the model solving processes less error-prone.

6. Conclusions

This paper proposes a new model management framework for models in AMLs on the basis of an ODBMS; it can accommodate both algebraic models and their problem-solving methods in a single formalism, and can facilitate a query-based user dialog interface to manipulate and integrate components of the DSS: modelbase, solvers, database. As a conceptual foundation of the framework, we establish generic model concepts based on a systems approach. The concepts aim to provide a generalized model representation scheme to capture diverse models represented in AMLs, and lead to development of a modelbase that can facilitate incorporating different AMLs ; internally, the semantic expressiveness of the concepts adaptively accommodate uniqueness of modeling conventions and model-execution methods of each AML so that users can make the best use of the advantageous features of individual AMLs. At the same time, the concepts facilitate the pro-

vision of homogeneous model management interfaces, minimizing the idiosyncratic interfaces of individual languages.

The primary insight gained in combining the generic model concepts and model management capabilities of the DSS is the applicability of ODBMS technology. Specifically, persistent object types play the key role in constructing GenericModelType, ModuleType, PortType, DataType, and functional model types; structurally, they enable users to capture the complex structures of the generic model concepts directly, and the modeling semantics of different AMLS, and to deal with all the elaborated model representation concepts and mechanisms associated with the generic model concepts consistently. Operationally, blended with a powerful programming paradigm, they also facilitate accommodation of model-execution and solution operations inside the functional model types in addition to conventional operations such as storing and retrieving models, empowering the framework to be a full-fledged model management system.

A prototype system of the model management framework is developed on a commercial ODBMS called ObjectStore. The object types and operations proposed in the framework are implemented without any conceptual distortion in C++, an object-oriented extension of the C programming languages; the hardware platform used is the Sun-4 system. Figure 6 shows an example session of the prototype system retrieving information from a SML transportation model. Technical details of the implemented system is the subject of separate documents.

```
// model manipulation query script
GenericModelType *model;
model = GenericModelType::MODELS[ (name == "transportation") && type() == SML ];
// execute the following operations
model->showExternal();
model->showModularOutline();

———————— result of the first query (...showExternal()...)————————
model: transportation
comment: Minimizing the cost of transporting products from factories to markets

[external interface]
INPORT:
    FACTORY     list of factories
    SUP         supply capacity of factory
    MARKET      list of markets
    DEM         demand amount of market
    COST        transportation cost matrix between factories and markets

OUTPORT:
    TOTAL_COST  total transportation cost
    QUANTITY    transportation quantities
    T:SUPPLY    supply constraint test
    T:DEMAND    demand constraint test

MIDPORT:
    LINK        transportation link between factories and markets

———————— result of the second query (...showModularOutline()...)————————
&FACTORY_DATA
    FACTORY
    SUP(FACTORY)
&MARKET_DATA
    MARKET
    DEM(MARKET)
&TRANS_DATA
    LINK(FACTORY,MARKET)
    QUANTITY(LINK)
    COST(LINK)
&CONSTRAINT
    T:SUPPLY(QUANTITY,SUP)
    T:DEMAND(QUANTITY,DEM)
&OBJECTIVE
    TOTAL_COST(COST,QUANTITY)
```

Figure 6. An Example Session of the Prototype System.

References

1. Binbasioglu, M., and Jarke, M., "Domain Specific DSS Tools for Knowledge-Based Model Building", *Decision Support Systems* 2 (1986) 213-223.
2. Blanning, R., "A relational Framework for Join Implementation in Model Management", *Decision Support Systems* 1 (1985) 69-82.
3. Brooke, A., Kendrick, D., and Meeraus, A., *GAMS: A User's Guide*, Scientific Press, Redwood City, CA, 1988
4. Dolk, D., and Konsynski, B., "Knowledge Representations for Model Management Systems", *IEEE Transactions on Software Engineering* 10/6 (1984) 619-628.
5. Dolk, D., "Model Management and Structured Modeling : The Role of an Information Resource Dictionary System", *Communications of ACM* 31/6 (1988) 704-718.
6. Dutta, A., and Basu, A., "An Artificial Intelligence Approach to Model Management In Decision Support Systems", *IEEE COMPUTER* 17/9 (1984) 89-97.
7. Elam, J., Henderson, J., and Miller, L., "Model Management Systems : An Approach to Decision Support in Complex Organizations", *Proceedings of the 1st Int. Conference on Information Systems* (1980) 98-110.
8. Fourer, R., Gay, D. M., and Kernighan, B. W., "A Modeling Language for Mathematical Programming", *Management Science* 36/5 (1990) 519-554.
9. Geoffrion, A., "An Introduction to Structured Modeling", *Management Science* 33/5 (1987) 547-588.
10. Geoffrion, A., "SML: A Model Definition Language for Structured Modeling", Western Management Science Institute, Working Paper 360, Anderson Graduate School of Management, UCLA, August 1990.
11. Hogan, W. W., and Weyant J. P., "Methods and Algorithms for Energy Model Composition : Optimization in a Network of Process Models", *Energy Models and Studies*, 1983, Amsterdam:North-Holand
12. Huh, S. -Y. "Modelbase Construction with Object-Oriented Constructs," *Decision Sciences* 24/2 (1993) 409-434.
13. Kendrick, D. and R. Krishnan, "A Comparison of Structured Modeling and GAMS," *Computer Science in Economics and Management* 2/1 (1989) 17-36.
14. Lamb, C., Landis, G., Orenstein, J., and Weinreb, D., "The ObjectStore Database System", *Communications of ACM* 34/10 (1991) 50-63.
15. Le Claire, B., and Sharda, R., "An Object-Oriented Architecture for Decision Support Systems", Proceedings of the 1990 *International Society for Decision Support Systems Conference* (1990) 567-586.

16. Lenard, M., "Representing Models as Data", *Journal of Management Information Systems* 2/4 (1986) 36-48.
17. Lenard, M., "An Object-Oriented Approach to Model Management", *Proceedings of the 20th Hawaii International Conference on System Science* 1 (1987) 509-515.
18. Liang, T., "A Graph-Based Approach to Model Management", *Proceedings of Seventh International Conference on Information Systems* (1986) 136-151.
19. Mannino, M. V., Greenberg, B. S., and Hong, S. N., "Model Libraries: Knowledge Representation and Reasoning", *ORSA Journal on Computing* 2/3 (1990) 287-301.
20. Muhanna, W., and Pick, R., "Composite Models in SYMMS", *Proceedings of the 21st Hawaii Intl. Conference on System Sciences* (1988) 418-427.
21. Muhanna, W. A., "An Object-Oriented Framework for Model Management and DSS Development" to appear in *Decision Support Systems*.
22. Roy, A., L. Lasdon, and J. Lordeman, " Extending Planning Languages to Include Optimization Capabilities," *Management Science* 32 (1986) 360-373.
23. Zdonik, S. and D. Maier, Readings in *Object-Oriented Database Systems*, Morgan Kaufmann, 1990.