Decision Support Systems 7 (1991) 301–314
North-Holland

PDM: A knowledge-based tool for model construction

Ramayya Krishnan

Decision Systems Research Institute, School of Urban and Public Affairs, Carnegie-Mellon University, Pittsburgh, PA 15213, USA

This paper describes PDM, a knowledge-based tool designed to help non-expert users construct Linear Programming (LP) models of Production, Distribution and Inventory (PDI) planning problems. PDM interactively aids users in defining a qualitative model of their planning problem, and employs it to generate problem-specific inferences and as input to a model building component that mechanically constructs the algebraic schema of the appropriate LP model. Interesting features of PDM include the application of domain knowledge to guide user interaction, the use of syntactic knowledge of the problem representation language to effect model revision, and in the use of a small set of primitive modeling rules in model construction.

Keywords: Artificial intelligence, Model management.

Ramayya Krishnan is Assistant Professor of Management Science and Information Systems at Carnegie Mellon University. He has a B. Tech in Mechanical Engineering from the Indian Institute of Technology, a M.S. in Operations Research, and a Ph.D. in Information Systems from the University of Texas at Austin. His research interests are in the application of symbolic and qualitative reasoning techniques. His recent work has used these techniques to develop computer-based environments that support model development activities.

1. Introduction

Models play an important role in decision support. While models drawn from several modeling traditions have been successfully integrated into computer-based decision support systems, Linear Programming (LP) models have been among the most widely used. While the quality of system support for LP modeling has improved considerably in recent years, the need to conceptualize a real-world problem in terms of abstract concepts and mathematical notation has inhibited their use by non-expert users. Several knowledge-based systems have been proposed to address these shortcomings (Binbasioglu and Jarke, 1986; Bu-Halaiga and Jain, 1988; Ma, Murphy and Stohr, 1986; Murphy and Stohr, 1986; Krishnan, 1987,1988; Muhanna and Pick, 1988).

This paper describes PDM, a knowledge-based tool that has been designed to help non-expert users construct LP models of Production, Distribution and Inventory (PDI) planning problems. PDM interactively aids users in defining a logic model of their planning problem which is used to provide qualitative [1] insights, and as input to a model building component that mechanically constructs the corresponding LP model through the application of a small set of primitive modeling rules such as material balance. The ability to construct a quantitative LP model from high level qualitative specifications is an important feature of the PDM system.

PDM has been implemented in Prolog and the chief purpose of this paper is to describe its key modules in order to document the lessons learnt in designing and implementing a knowledge-based model construction system. PDM employs alter-

[1] By qualitative, we imply a focus on representations and inferences which deal with objects, their inter-relationships, and their attributes as opposed to representations that employ numeric relationships and emphasize numeric reasoning.

nate knowledge sources (domain knowledge and model building knowledge) and a variety of knowledge representation schemes. Thus the primary focus of the paper is on the functionality to be gained from both the structure and the content of the knowledge used in PDM and the means employed to integrate the alternate knowledge sources and representation schemes.

The rest of the paper is organized as follows. Section 2 introduces the key features and components of the PDM system. Section 3 and 4 detail knowledge representation and control issues in two principal components: the front end and the model construction module. Section 5 draws conclusions and describes some avenues for future research in light of current limitations.

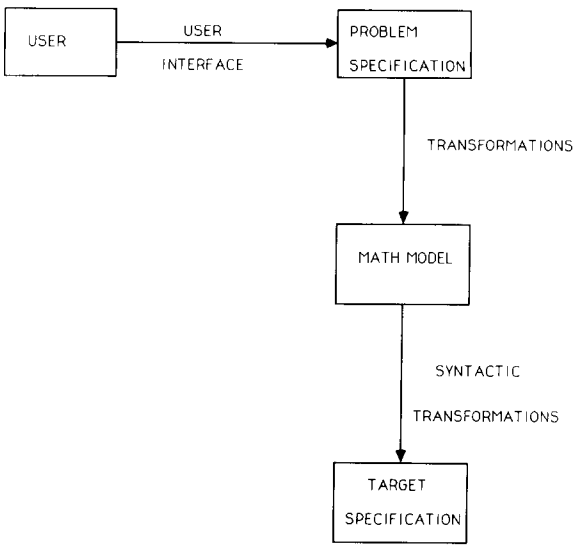

Fig. 1. The PDM system.

2. PDM: The system

PDM was designed to support non-expert users who lacked the familiarity with mathematical modeling to construct a model appropriate to their needs. The ability of non-expert users, by virtue of their familiarity with their problem, to provide qualitative problem descriptions is an important assumption underlying the approach in PDM. These qualitative descriptions when formalized within the syntax of a logical language result in a logic model of the problem. Since all the inferences flow from this representation, key features in the PDM system revolve around the processes used to obtain, represent and manipulate it. Figure 1 illustrates the important features of PDM which are summarized in the following.

(a) Qualitative descriptions of PDI planning problems are represented in a domain-specific logic-based language called PM (Krishnan, 1988). PM allows problems to be described in vocabulary familiar to the user and employs domain-specific axioms to provide problem-specific inferences. A specification in PM defines a logic model of the problem.

(b) To render the syntax of PM transparent to the user, an object-oriented dialogue system has been designed to interactively aid the user in problem description. This system employs knowledge of PM to assert sentences in response to answers obtained from the user, and domain-specific knowledge to aid problem elicitation by focusing the users attention on processes implied by the evolving description (i.e. the logic model) of the problem.

(c) The knowledge base employed by the dialogue system also explicitly encodes syntactic interdependencies between elements that make up the logic model enabling structural revision of PM problem specifications in the event of change. An important implication of this feature is the ability to structurally revise LP models, since structurally revised PM specifications yield structurally revised LP models.

(d) The algebraic schema of the LP model is constructed from the PM problem specification through the application of primitive modeling rules such as material balance. The application of these primitive modeling rules, stated in domain-independent terms, to a specific logic model in PM is facilitated by transformation rules which represent knowledge about the relationships between the generic processes that underlie the modeling rules and the specific processes that underlie PDI planning. The output of the model construction component is the LP model represented in an embedded list notation which is subsequently transformed to sentences of a mathematical modeling language called Structured Modeling (SM) (Geoffrion, 1987).

3. PDM architecture

Components in the PDM system have been grouped together and implemented as three distinct modules. (See fig. 2.) They are the front end, the model construction module and the back end. While user interaction, query answering, and model revision are handled by the front end, model building is performed by the model construction module, and syntactic transformations of the constructed LP models to Structured Modeling implemented in the back end.

The modules employ distinct knowledge sources. The front end employs domain-specific knowledge to guide user interaction and query answering, and syntactic knowledge of PM to effect model revision. On the other hand, the model construction module employs model building knowledge. The modules also use different knowledge representation schemes. While the front end employs an object-oriented scheme, the model construction component employs a forward chaining rule-based system and a set of procedures that manipulate certain object-types to perform equation building. The back end which implements a straightforward syntactic transformation into Structured Modeling is implemented as a set of Prolog procedures. The alternative knowledge representation schemes and knowledge sources employed in these modules are integrated within a blackboard type architecture, i.e., all communication between them is channeled exclusively through changes to a global database of facts.

The rest of the paper focusses on the two most important features of PDM: (a) the ability to interactively aid a user in defining and revising a high level qualitative specification of a planning problem and (b) the ability to construct the LP model from these high level specifications. Readers interested in a complete treatment of the PDM system are referred to Krishnan (1987, 1988).

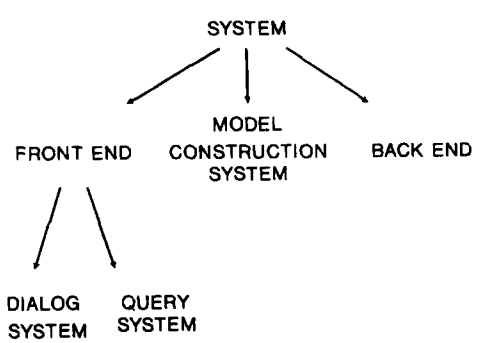

Fig. 2. System architecture.

3.1. The front end

The two important kinds of functionality offered by the front end are: (a) interactive support in the definition of a PM problem specification and (b) management and control of revisions to existing PM problem specifications. Both these features are directly influenced by the problem representation language PM. The following briefly introduces PM with a view to motivate the discussion on the knowledge base employed in the front end module.

3.2. The PM language

PM (Krishnan, 1988) is a logic-based language designed to logically model the PDI planning domain. An important feature in PM is the ability to introduce specific vocabulary as and when necessary to describe particular problems that arise in PDI planning. These user-introduced terms form the open vocabulary of PM while the rich set of generic concepts about PDI planning form part of its closed vocabulary. The following specifies a subset of the closed vocabulary of PM.

object constants: products, machines, raw-materials, regular-labor, overtime-labor, production-process, plant, distribution-center, warehouse, customer-site, purchase-yard, time, real-number, used-in, produced-by, purchased-at, stored-at, sold-at, available-at, shipped-from, unit-process-cost, unit-process-price, process-level, utilization-rate, availability, min-level, max-level

primitive predicates: basic-type, type, ftype, subtype, fsubtype, fdomain, fapply, ins-of, index, =, !=

Object constants in PM are used to name the various objects, processes and relationships that characterize the PDI domain. For example, constants such as **product** and **raw-material** name sets of objects, while others such as **purchased-at** and **stored-at** are used to name relationships. Examples of constants used to name functions are **process-level** and **unit-process-cost**. These different types of object constants in PM are distinguished and

explicitly inter-related using declarations in PM. Three important declarations are introduced using examples.

basic-type (product)
All object constants that name sets are declared using the predicate **basic-type**. The statement shown above declares that product is a type of set.

type (purchased-at, [raw-material, purchase-location, time])
Object constants that name relations are declared using the predicate **type**. The predicate is interpreted as declaring that the domain of the relation named in the first argument is defined by the cross product of the sets named in the second argument. Thus, **purchased-at** names a relation that is the purchase of **raw-materials** at **purchase-locations** across **time**.

ftype (process-cost, [commodity, location, time], [real-number])
Object constants that name functions (i.e. attributes of objects or their relationships) are declared using the predicate **ftype**. The predicate is also used to declare the domain and range of the named function. Thus, **process-cost** names a function that measures the cost of performing a given task or process using some **commodity** at a particular **location** and **time-period**.

The declarations shown above explicitly relate object constants **within** the closed and open vocabulary respectively. Relationships **between** elements of the open and closed vocabulary are declared as shown below. Object constants that name sets and relations in the open and closed vocabulary are related using the predicate **subtype**. This predicate plays a role that is similar to the generalization/specialization declarations in frames and semantic networks.

subtype (steel, product)
The example declares that the set named by **steel**, an object constant of the open vocabulary, is a subset of the set named by **product**, an object constant of the closed vocabulary. Similarly, object constants used to name functions in the open and closed vocabulary are related using the predicate **fsubtype**.

fsubtype (coal-purchase-cost, process-cost)
This predicate is interpreted as declaring that **coal-purchase-cost** is a type of **process-cost**.

Sets, relations, and functions that are named in PM are defined using the predicates **ins-of** and **fapply**.

ins-of([brown-coal], coal)
The example declares that **brown-coal** is an element of the set named by **coal**. Elements of sets and relations are declared using the predicate **ins-of**.

fapply (coal-purchase-cost, [brown-coal, dallas, 1980], 12.25)
The predicate **fapply** is interpreted as applying the function named in the first argument to a list of constants in the second argument. The output of the function application is the third argument.

A fragment of PM specification of a production problem in the steel industry is shown below.

An Example in PM

```
basic-type (steel)
basic-type (coal)
basic-type (coal-mines)
basic-type (weeks)

subtype (steel, product)
subtype (coal, raw-material)
type (coal-purchase, [coal, coal-mines, weeks])
subtype (coal-purchase, purchased-at)
ftype (coal-purchase-cost, [coal, coal-mines, weeks,
      [real-number])
fsubtype (coal-purchase-cost, process-cost)

ins-of ([brown-coal], coal)
```

The example consists of a set of declarations which specify the production of steel in a production process utilizing coal as a raw-material. An important feature of the specification is its focus on representing qualitative relationships at a high level of abstraction.

3.3 Knowledge base

As previously mentioned, helping the user define his problem in PM and managing structural revisions to a PM specification are the two important tasks performed by the front end module.

Since elements of the closed vocabulary represent generic objects, processes and attributes in PDI planning, defining a specific problem in PM requires the identification of those problem-specific concepts that are specializations of the generic domain-specific concepts. Thus aiding problem definition requires the ability to "visit" each generic concept and generate a question regarding its relevance to the users problem situation. The knowledge base (KB) in PDM enables this by structuring the domain-specific elements in PM into a graph of objects. [2] Each individual constant in PM used to name sets, relations, and functions is treated as an object. The object-based representation of the constant **purchased-at** is shown below.

purchased-at
⟨related-objects⟩:
    value: [raw-material, purchase-location, time]
⟨inv-related-to-prod⟩:
    value: [purchase-cost, purchase-level]
⟨inv-related-to-dis⟩:
    value: [purchase-cost, purchase-level, shipping-plan]
⟨process-relations⟩:
    value: [used-in, stored-at, supplied-from]
⟨to-fill-in⟩:
    rule-action: should-we-proceed
⟨if-confirmed⟩:
    rule-action: perform-tasks

Each object has five slots. The first three slots represent information about the syntactic inter-dependencies between the elements in PM. These inter-dependencies are used to effectively link the various objects in the KB to create an object graph. Each leaf node of this object graph corresponds to a constant used to name a set (i.e., declared using the predicate **basic-type**). Internal nodes of this object graph such as the example object correspond to constants used to name relations and functions (i.e., declared using the predicate **type** and **ftype** respectively). A fragment of the object graph that corresponds to the closed vocabulary is shown in fig. 3.

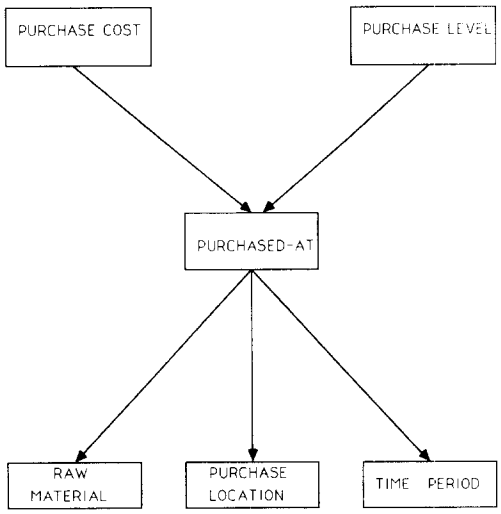

Fig. 3. An object graph.

The value fillers of the first three slots associated with an object are derived directly from the **type**, **fdomain** [3], and **joint-domain** declarations in PM. The close relationship between the object-based representation and the language is illustrated with a simple example. Consider the type declaration in PM of the purchased-at predicate shown below.

type (purchased-at, [raw-material, purchase-location, time])

The declaration relates the object constant **purchased-at** to a list of object constants that define its domain. This list of object constants is used as the filler of the **related-objects** slot of the purchased-at object.

The next two slots, **inv-related-to-prod** and **inv-related-to-dis**, represent the "inverse" of the linkages represented in the **related-objects** slot. These slots represent the set of all object constants whose type/fdomain/joint-domain declarations contain the object under consideration. Thus for example, the purchased-at object would be part of the inverse pointer slots of the raw-material, purchase-location and time objects. Specifically, the inverse pointers associated with an object, say Y, in the

[2] The term objects is used in the object-oriented programming sense. While the objects we use in PDM resemble frames in their use of slots and procedural attachments, no use is made of either inheritance or defaults.

[3] fdomain and joint domain are declarations in PM. While fdomain explicitly relates functions to its domain, joint-domain relates three relations X, Y and Z such that X names the join of the relations named by Y and Z.

object graph can be defined set theoretically as shown below.

Inv(Y) := {X: type (Y, L) & X ∈ L | fdomain (Y, X) | joint-domain (Y, U, V) & X = U or V}

This set of inverse pointers is partitioned into two non-disjoint sets as a function of the problem contexts that arise in PDI planning. Thus, the **inv-related-to-prod** slot of the purchased-at object that represents the inverse pointers in the production planning context has a filler that consists of purchase-level and purchase-cost while the equivalent **inv-related-to-dis** slot for the distribution planning context additionally consists of the shipping-plan object. The rationale is that while distribution contexts may involve both purchase and shipping, production contexts only involve purchase. This partitioning of the inverse pointers as a function of the problem context enables the selective traversal of objects in the object graph as a function of problem context enabling a focussed and structured dialogue process.

Inverse pointers enable traversal of the object graph in a "bottom-up" manner; i.e., from leaf nodes that correspond to sets to internal nodes that correspond to relations and functions. This ability is particularly important since the dialogue begins in the context of objects that correspond to sets and proceeds to contexts represented by objects that correspond to relations and functions. This is in keeping with the intuitive transition of dialogue about simple concepts to interaction about more complex concepts.

While the first three slots represent syntactic inter-dependencies, the fourth slot, **process-relations**, represents domain-specific axioms. It so happens that these axioms about the domain are of a simple structure that facilitates their representation via a slot. Consider the example shown below.

If purchased-at (X, L, T) then stored-at (X, L, T)
or supplied-from (X, L, T)
or ∃P used-in (X, P, L, T)

The axiom is non-horn and states that if a commodity X is purchased at a given location then it is either stored at that location, supplied from that location or used in a production process housed in that location. The process-relations slot represents the list of processes related to the purchase process, enabling the system to bring these related processes to the attention of the user. Finally, the last two slots represent procedural knowledge that implements the logic used in the dialogue and model revision process. Each of these slots have **rule-action** facets. In contrast to the **value** facets used in the first four slots, **rule-action** facets are active facets (much like the if-needed facets in traditional frame-based systems) that encode Prolog procedures. In the example, the slots **to-fill-in** and **if-confirmed** have rule-action facets which represent the procedures **should-we-proceed** and **perform-tasks**. The logic implemented in these facets is described later. Each object in the KB is implemented in Prolog as a set of clauses. A fragment of the purchased-at object used as an example is shown below. The general notation used is

⟨object-name⟩ (⟨slot-name⟩, ⟨facet-name⟩, ⟨value-filler⟩)

Where value-filler is either a list of objects, an object or a Prolog procedure. Thus the purchased-at object is represented as shown below.

purchased-at (related-objects, value, [raw-material, purchase-location, time])
purchased-at (to-fill-in, rule-action, should-we-proceed)

The next section describes the control logic used to manage the generation of dialogue.

3.3.1 Dialogue generation

The primary responsibility of the dialogue generation module is to interactively aid the user in defining the logic model of the problem. This is done by generating hypotheses (dialogue) in the context of situations, entities and relationships that characterize PDI planning. The traversal of the object graph to generate dialogue is performed in a bottom-up manner and characterized by two important steps:

(a) Queries are generated initially in the context of the leaf nodes (i.e. the sets) of the object graph. These queries are aimed at identifying the various types of entities in a particular planning problem and ascertaining the existence of relationships between these types of entities. They are referred to as **askable** queries since the information they ob-

tain from the user is not inferable. An example is shown below.

What are the different types of product produced in the system?
|: steel
Please supply the elements of the set "steel"
|: **[stainless-steel, tensile-steel]**

The first query requests the specification of different types of products to which the user identified steel as the only type. The next query required the user to enumerate the elements of the set steel. User responses are translated into PM sentences. The sentences asserted in response to this interaction are shown below.

subtype (steel, product)
ins-of (steel, [stainless-steel, tensile-steel])

(b) Responses to the askable queries form the kernel of the evolving PM specification of the problem. Domain-specific axioms (such as those represented in the **process-relations** slot) and other rules encoded as procedures are used to hypothesize situations implied by the evolving PM specification. Hypothesis that are confirmed by the user result in additions to the PM specification. For instance, assume that the user already indicated that coal is purchased at coal mines. The domain-specific axiom represented in the process-relations slot of the purchased-at object is used to hypothize the set of processes related to the purchase process. The user is required to confirm the existence of one or more of these related processes. The axiom and associated dialogue are shown below.

if purchased-at (X, L, T) then stored-at (X, L, T)
or supplied-from (X, L, T)
or ∃P used-in (X, P, L, T)

The axiom states that if a commodity is purchased at a location, it is either stored or supplied from that location or used in production at the same location. This axiom results in a series of queries to the user.

is coal stored at the coal mines? (Y / N)
is coal supplied-from the coal mines? (Y / N)
is coal used in production at the coal mines? (Y / N)

The illustrated use of domain axioms to guide user interaction is a novel feature of PDM and a measure of the power to be gained from a domain-specific approach. Furthermore, the requirement that the user identify at least one related process as relevant prevents several simple infeasibilities that arise in problem specifications due to errors of omission.

The "bottom up" traversal of the object graph has been implemented using an agenda scheme (Lenat, 1976).

Agenda-based Control: An agenda-based control strategy employs a queue to order the tasks at hand. In our context, the flow of dialogue is initiated and controlled by the addition of objects in the object graph to the queue which might then be sampled under a variety of queuing disciplines.

A significant advantage of the agenda scheme is the ability to tune the order in which objects are sampled thereby effecting control over the flow of dialogue. The interpreter is implemented as a simple recursive procedure in Prolog as shown below.

```
interpret ([]).
interpret ([H|T]):- process-frame (H, T, Current),
      interpret (Current).
```

The first clause represents the base case of the recursion and indicates that the interpreter halts when the agenda is empty. The second clause implements a FIFO (First In First Out) policy and considers the first object in the agenda using the procedure process-frame. This procedure activates the procedures in the **to-fill-in** and **if-confirmed** slots of the object under consideration. These procedures decide on dialogue generation and upon completion queue in objects in their appropriate inverse slots into the agenda thus ensuring the flow of dialogue. Changes to the agenda status are determined and the procedure recurses on a new binding of the agenda status.

In the context of our object graph, dialogue is initially generated in the leaf nodes. Upon completion, interior nodes that are part of inverse slots of the leaf node are added to the agenda resulting in the continued generation of dialogue among interior nodes. Dialogue generation halts when the agenda is empty.

Dialogue Generation: An important feature of the dialogue generation process is the need to generate dialogue in vocabulary familiar to the user. We have adopted a simple strategy to effect this feature.

Dialogue is generated by procedures attached to the **to-fill-in** and the **if-confirmed** slots of an

object. These procedures contain inference rules and templates of text. An example of a rule used to generate dialogue in the context of the purchased-at object is shown below.

If X is a type of raw-material and
If Y is a type of purchase-location
If Z is a type of time-period
Then ascertain the existence of a relation between
X, Y, and Z

The variables X, Y etc. are bound to user-supplied predicates that describe objects and processes specific to the problem at hand. These variables are combined with a template of text to generate dialogue using vocabulary previously supplied by the user. An example template is shown below.

is ?X purchased-at ?Y in Z?

The ?X denotes variables that are to be bound. Templates that have instantiated variables result in text. Thus ?X being bound to coal and ?Y to coal-mines and ?Z to weeks results in the dialogue shown below.

/* Comments are enclosed within these symbols */
is coal purchased-at coal-mines in weeks? (yes/no)
|: yes
Please supply a unique name to this relation
|: coal-purchase-plan
/* The PM sentence corresponding to this answer is */
/* type (coal-purchase-plan, [coal, coal-mines, weeks]) */

The first line of the dialogue queries the existence of a relation between coal and coal-mines which were user-supplied descriptions of objects and locations in his particular problem. Having ascertained the existence of the relation, the user is required to supply a name for the relationship and tuples that represent instances of the relation. As the dialogue proceeds, such interaction results in sentences in PM being asserted. These type of questions are generated for each combination of objects that satisfy the dialogue generation rules.

A useful feature of the dialogue system is the ability to save the state of the agenda and object graph midway through problem description. While saving the state of the queue suffices to save the state of the agenda, the state of the object graph is saved using a system of markings. Each object which has been investigated (i.e. dialogue generation having resulted in the addition of PM sentences) is marked. This allows the interpreter to skip over marked objects when user interaction is continued at a later time to prevent the redundant generation of dialogue. The important advantage of being able to save the state of the agenda and the object graph is the ability to work with the PDM system as and when desired.

The algorithm used to control the flow of dialogue is presented below.

ALG Process:
Do while agenda is not empty;
choose object from agenda
if object is marked /* if 1 */
then queue objects in the inverse-relation slot based on problem
context to the agenda and delete object from agenda /* end if 1 */
if object is unmarked /* if 2 */
then
  if objects in its related-objects slot are marked /* if 3 */
  then invoke perform-task and
    if new PM sentences are added /* if 4 */
    then mark object and queue in objects in inverse-relation slot
    else queue in objects in inverse-relation slot and
      delete object from agenda /* end if 4 */
  else delete frame from agenda /* end if 2, 3 */
End Do While;

Limitations: While the dialogue system amply demonstrates the functionality to be gained from a knowledge-based tool, a state of the art graphics/icon driven system would be far more user-friendly.

3.3.2. Model revision

Model specifications are constantly revised as assumptions that underlie problem specifications change. These changes in assumptions about the problem being modeled typically result in additions and/or deletions of sets, relations and functions or in additions and deletions of their respective elements. This alteration of an existing problem specification has been termed model revision.

Since elements that make up the problem specification are tightly inter-related, a change in

one part tends to affect other parts. This implies the need to control and manage the process of propagating local changes throughout a problem specification. Propagation of changes requires the explicit representation of inter-dependencies between problem elements. The object graph (i.e. the KB) in PDM supports model revision since it encodes the syntactic inter-dependencies between elements that make up the problem specification. It is used in conjunction with the agenda-based scheme described in the previous section to effect propagation of local changes. Consider a fragment of a production planning problem in PM. The sentences have been labelled for ease of reference.

sent1: subtype (steel, product)
sent2: subtype (open-hearth, production-process)
sent3: subtype (oxygen, raw-material)
sent4: subtype (oxy-used-in-open-hearth, used-in)
sent5: type (oxy-used-in-open-hearth, [oxygen, open-hearth, mill, year])

The PM specification describes the usage of oxygen in the open-hearth process used in steel-production. Now if the open-hearth process declared in sent2 were to be removed from the problem specification, the sentences labelled sent4 and sent5 should also be deleted since they are directly or indirectly related to the deleted object constant. Since each user-supplied object constant is related to a object constant of the closed vocabulary (i.e. an object in the object graph), propagation of deletions is implemented using a selective traversal of the object graph.

This traversal begins by accessing the object in the graph related to the user-supplied predicate being deleted. In the example since open-hearth, the predicate being deleted is a production-process, an object in the object graph, the production-process object is accessed and all the objects in its **inverse-relation** slot are queued into the agenda. These objects correspond to object constants of the closed vocabulary that are directly dependent on the production-process object. The interpreter examines those objects that are marked (recall that marked objects correspond to object constants of the closed vocabulary that are part of the current PM specification) and deletes all PM sentences which contain a marked object. To ensure continued propagation, objects in its (the object under consideration by the interpreter) **inverse-relations** slots are also added to the agenda. Processing halts when the agenda is empty.

The principal advantage of this approach is its focus of attention on only that part of the problem specification that needs to be changed. This is an important factor in the context of large specifications where a brute force search for sentences that need to be deleted may be infeasible.

In addition to this ability to propagate deletions, PDM also supports the addition of new sets and relations. When new sets or relations are added, there is a need to focus the users attention on the ramifications of the addition. For example, if a new type of product is added, new relationships have to be defined with existing production processes and other relevant objects involved in processes such as sales or storage. Once again the object graph and the agenda scheme are employed. When a user-supplied set or relation is added to an existing specification, the object of the closed vocabulary that it is related to (an object in the object graph) is queued into the agenda. The generation of dialogue is similar to that described previously with one major difference. Before any dialogue is generated in the context of any object in the object graph, a comparison is made between situations that are hypothesized and those that already exist in the PM specification. This prevents the redundant generation of dialogue. Once again due to the explicit representation of dependencies in the object graph, the process is focussed with only the minimal number of queries being generated.

This ability to revise model specifications is an important feature in PDM due to the flexibility offered to the user. Additionally, it also supports structural revision of the LP model since structurally revised PM specifications yield structurally revised LP model schema. This is particularly important since currently available LP modeling systems do not support this feature

4. Model construction

Linear Programming (LP) models are algebraic models. The construction of an algebraic model from a qualitative model requires the representation and application of model building knowledge.

This, in PDM, has been effected using a small set of domain-independent modeling principles

such as material balance and resource utilization. The principal insight used in model construction derives from the observation that all the measurement functions [4] used in the qualitative model to represent numeric attributes are directly transformable to variables and parameters of the algebraic model. The model building rules are strictly concerned with determining the functional form of the mathematical relationships that relate these variables and parameters.

Model building rules such as the material balance rule shown below represent generic types of mathematical relationships. These generic relationships are encoded in terms of domain-independent abstractions referred to as **canonical objects**.

If [X] is a list of inputs to a system and
If [Y] is a list of outputs to a system
Then the sum of the set of inputs $\geq$ sum of the set of outputs

Model building requires these rules to be applied to a PM specification. However, the lack of a common vocabulary presents a problem, i.e., model building rules are stated in terms of canonical objects while PM specifications are stated in terms of problem-specific vocabulary. This has been resolved by adopting a simple two step procedure. First all problem specific objects, processes and attributes that make up the PM specification are transformed into canonical objects. Rules used for this task are referred to as **transformation rules**. The canonical objects so generated are combined using the model building rules into algebraic functions and constraints that make up the schema of the LP model.

While the foregoing presented the synopsis of the logic used in model construction, the implementation in PDM takes account of certain other problems. Specifically, model building in PDM proceeds at two distinct levels: the construction level and the meta construction level. The application of transformation rules and model building rules take place at the construction level. However, these tasks at the construction level are controlled using meta-rules that encode knowledge about the order or sequencing of rule application.

[4] Examples of measurement functions are production-level, utilization-rate etc.

Meta-rules are essential to ensure correctness in model building and serve to improve the efficiency of the model building process. The following illustrates with a simple example, the knowledge representation and model building strategy employed in each of these levels.

Example

Consider a simplified steel production process which uses various types of coal to produce several types of steel. Assume that the production of each unit of steel utilizes a fixed amount of coal. Let its value be given by the parameter, coal-util-rate. Coal is supplied via purchase and let the variable coal-purchase-level represent the amount of coal purchased. Finally let the steel production level be measured by the variable steel-production level.

Given this fragment of simple problem, the capacity constraint for coal that we seek to generate is modeled using material balance resulting in the capacity constraint shown below.

sum(S) sum(P) (coal-util-rate (C, P, L, T) * steel-production-level (S, P, L, T)) $\leq$ coal-purchase-level (C, L, T)

The letters S, C, P, L and T correspond to indices for the sets steel, coal, production process, location and time-period.

The equivalent qualitative model of the problem in PM is shown below.

subtype (coal, raw-material)
subtype (steel, product)
subtype (coal-usage, used-in)
type (coal-usage, [coal, steel-production-process, mill, years])
ftype (coal-util-rate, [coal, steel-production-process, mill, years), [real-number])
fsubtype (coal-util-rate, utilization-rate)
ftype (steel-production-level, [steel, steel-production-process, mill, years), [real-number])
fsubtype (steel-production-level, activity-level)

The differences in the representation of the problem in PM and quantitatively as an algebraic model are significant. The PM representation simply declares the type of steel, coal and the production process and declares the relations and functions that characterize the problem. On the other hand, the algebraic constraint represents a mathematical relationship between numeric variables and parameters. The following briefly demon-

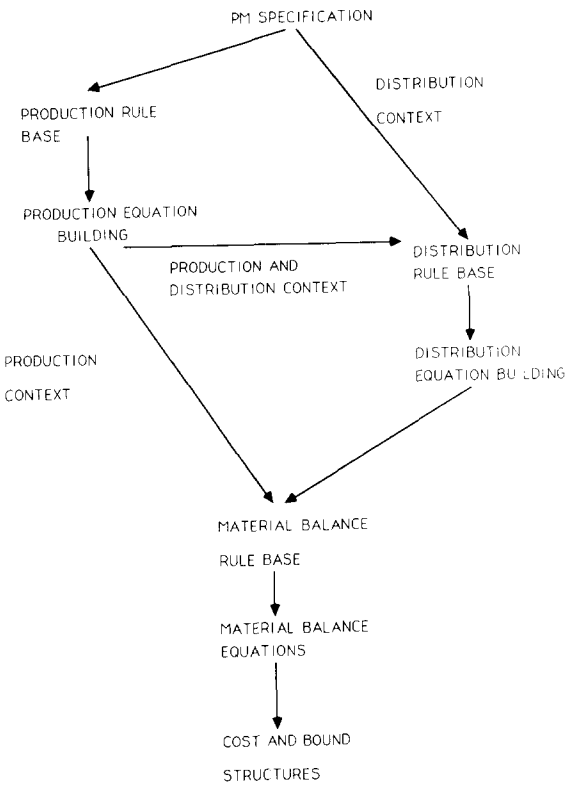

Fig. 4. Meta rules.

strates the steps used to construct the material balance constraint from the PM specification.

First since the context of the problem is production planning, a meta-rule at the meta construction level is used to suggest the application of a "transformation-in-form" rule denoted fig. 4 as the production rule base.

The "transformation in form" rule is based on a generic resource utilization process and relates the utilization-rate of a resource commodity that is input to a production process to the production level of the commodity produced as output of the same production process. The rule [5] is shown below.

If utilization-rate (X, R, RU, P, L, T) and activity-level (PL, RU, P, L, T)
and index (X, IR) and index (PL, IPL) and index (P, IP) and
index (R, Iru) and subtype (Dom, utilized-at) and
type (Dom, [R, L, T))
then gensym (RUL, utilization-level) and
fsubtype (RUL, process-level) and fdomain (RUL, Dom) and
RUL := sum(Iru) sum(IP) (X : IR * Pl : IPL)

The rule constructs the function RUL from canonical objects that are instances of the utilization-rate and activity-level objects. The application of this domain-independent rule to a PM specification requires the application of transformation rules to the PM specification. The collection of transformation rules associated with model building rules is referred to as a rule base or rule set.

The exact details of the transformation rule application are detailed and space limitations prevent a full-fledged description. The reader is referred to Krishnan (1988) and Krishnan (1987) for an indepth discussion of transformation rules. The idea behind transformation rules is simple. Essentially these rules encode knowledge about the relationships between the domain-specific processes that underlie the problem specification in PM and the generic processes that underlie the model construction rules. They yield instances of canonical objects as output. Thus, in the context of our example, the transformation rules recognize the steel-production-process as a process that utilizes coal as a resource to produce steel. After a series of transformations this leads to the generation of steel-production-level as an activity-level object and the coal-util-rate as an utilization-rate object. They are shown below.

steel-production-level
canonical-object-type:
value : activity-level
context:
value : [s, p, l, t]

coal-util-rate
canonical-object-type:
value : utilization-rate
context:
value : [c, p, l, t]

An important feature of these canonical objects are their context slots that represent index information. The assignment of indices to the elements of a PM specification is performed by index assignment procedures that are also activated by meta rules. Essentially, the strategy used in index assignment is as follows. Each set is assigned a

[5] The collection of model building rules may be found in Appendix A (available from the author).

unique symbol as index. These indices may either be supplied by users or provided by the system. Indices of named relations and functions are derived from index information associated with the sets that define their domain. An example is shown below.

if type (N-pred, [A1, ..., An]) and index (A1, IA1) and
  and ... and index (An, IAn)
then index (N-pred, [IA1, IA2, ..., IAn])

Thus type and ftype declarations in PM prove useful in index assignment. Indices set the context of an object and are used to determine if two or more objects can be combined by a given model building rule. The objects steel-production-level and coal-utilization-rate are combined using the transformation-in-form rule introduced previously to yield a new canonical object which encodes the left hand side (LHS) of the constraint we are in the process of constructing. The function built using the "transformation-in-form" rule and the canonical object that encodes it are shown below.

coal-utilization-level (c, l, t) = sum(s) sum(p)
    (coal-util-rate (c, p, l, t) *
    steel-production-level (s, p, l, t))

coal-utilization-level
    canonical-object-type:
      value: activity-level
    context:
      value: [c, l, t]
    function:
      value: [[sum, [s, p]], [coal-rate, [c, p, l, t]],
        [*], [production-level, [s, p, l, t]]]

The object is similar to the objects introduced earlier with one major exception. It has a function slot which encodes the LHS of the constraint under construction.

Upon completion of the model building rule application, control transfers once again back to the meta-level which suggests the application of the material balance rule. Once again the transformation rules associated with material balance are applied to the PM specification to yield coal-utilization-level, the canonical object described above, as an output object and the coal-purchase-level as an input object. The coal-purchase-level object is shown below.

coal-purchase-level
    canonical-object-type:
      value: output
    context:
      value: [c, l, t]

These objects are combined using the material balance rule since they represent inputs and outputs of the commodity coal to a system (the indices in the objects are used to ensure similarity in contexts) into the capacity constraint being constructed. This yields the constraint for coal shown below.

coal-utilization-level (c, m, t) $\leq$ coal-purchase-level (c, m, t)

The constraint is similar to the one derived earlier except that the LHS in the constructed constraint is itself a function that was defined earlier.

Three concluding remarks are in order. First, meta-rules were used to sequence the application of the model building rules and their associated rule sets as a function of the problem context, i.e., since the context was production planning, only the resource utilization and material balance rules and their associated transformation rules were activated. This results in significant gains in efficiency since rule sets tend to be large and transformation to yield canonical objects involves significant amount of chaining. Second, the construction process used is "bottom up". That is constraints are built from left hand sides (LHS) and right hand sides (RHS) using the material balance rule. The LHS and the RHS themselves may be functions constructed through previous rule applications. This is illustrated in our example where the LHS was a resource utilization function built from primitive canonical objects using the "transformation-in-form" rule. The generic "bottom-up" procedure used in constraint building is depicted in Fig. 5.

An important implication of this "bottom up" approach to model building is the need to sequence the application of the model building rules since the input [6] of one rule is dependent on the

[6] In the example, the material balance rule used the coal-utilization object that was the output of the "transformation-in-form" rule as an input in constraint construction.

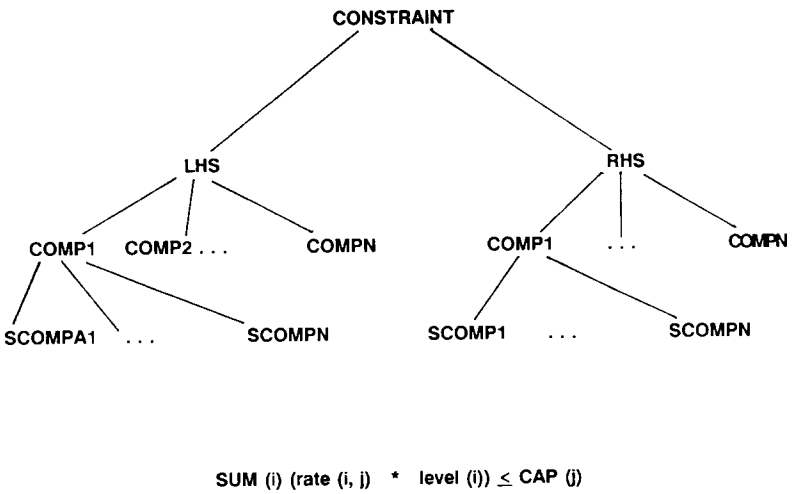

Fig. 5. Model construction strategy.

output of another to ensure correctness. Thus meta rules serve to ensure correctness of model building and to increase the efficiency by only invoking relevant rule sets. Meta rules thus serve an important role in that they explicitly represent "model construction" know how. In being the sole repository of such knowledge they support the alteration and manipulation of it in the event of change.

An important aspect of the model building strategy that we have not discussed as a result of space limitations is the role of transformation rules. Transformation rules are used to transform elements of the PM specification to canonical objects. This process is straightforward or complex depending on the availability of a rule/rules specific to the situation being modeled by the PM specification. If directly applicable rules exist, the transformation process is straightforward. However, in the absence of directly applicable rule sets, PM problem specifications are augmented with additional variables and decomposed into situations for which directly applicable rule sets are available. A common example of this case include transhipment processes which are augmented with additional variables into transportation processes and material balance processes for which directly applicable rules exist. This process of diagnosing situations that need augmentation and decomposing them into situations that are transformable into canonical objects is an important non-trivial feature of transformation rules. The interested reader is referred to Krishnan (1987, 1988) for a detailed description of the model construction process.

5. Conclusions

The main contribution of this paper has been the description of two key modules in the PDM system with an emphasis of the structure and content of the knowledge employed to effect user-interaction, model revision and automatic model construction. The novel features in PDM that were detailed included a description of the close interaction between the object-oriented system employed in the front end and the logic-based language PM used in problem representation, the use of domain-specific axioms to guide user interaction, and the application of domain-independent model building rules to simulate a "first principles" approach to automated model construction.

There are two principal limitations in PDM. First, the user interaction system is textual and cumbersome and a graphics based tool would greatly help the type of non-expert users that PDM hopes to support. Another limitation is the restriction to linear models and the PDI planning domain. An useful extension would be a domain-independent logic modeling language that would be used to specify mathematical and qualitative models within an uniform framework. Research is underway on all these issues.

References

[1] Binbasioglu, M. and Jarke, M. (1986), Domain specific tools for knowledge based model building, Decision Support Systems, 2, 1, pp. 213–223.

[2] Bu-Halaiga, M., and Jain, H. (1988), An Interactive Plan Based Procedure for Model Integration in DSS, Proceedings of the Twenty First Hawaii Conference on the System Sciences, IEEE Press.

[3] Geoffrion, A.M. (1987), Introduction to Structured Modeling, Management Science, 33, 5, pp. 547–588.

[4] Krishnan, R. (1987), Knowledge Based Aids for Model Construction, Unpublished PhD Thesis, University of Texas, Austin, TX 78712.

[5] Krishnan, R. (1988), Automated Model Construction: A Logic Based Approach, Annals of Operations Research, Special Issue on Linkages between AI and OR, 21, pp. 195–226.

[6] Lenat, D. (1976), AM: An AI Approach to Discovery in Mathematics as Heuristic Search, Rept-STAN-CS-80-814, Stanford University, CA.

[7] Ma, P., Murphy, F., Stohr, E. (1986), The Science and Art of Formulating Linear Programs, to appear in IMA Journal of Mathematics In Management.

[8] Murphy, F., Stohr, E. (1986), An Intelligent System for Formulating Linear Programs, Decision Support Systems, 2, 1.

[9] W.A. Muhanna and Pick, R. (1988), Composite Models in SYMMS, Proceedings of the Twenty First Hawaii Conference on the System Sciences, IEEE Press.