1987

A GENERIC SHELL APPROACH FOR KMOWLEDGE ELICITATION AMD REPRESENTATION IM IDSS

Rafael Lazimy
University of Wisconsin, Madison

A GENERIC SHELL APPROACH FOR KNOWLEDGE ELICITATION AND REPRESENTATION IN IDSS

Rafael Lazimy
Graduate School of Business
University of Wisconsin, Madison

ABSTRACT

This study focuses on issues of knowledge representation and elicitation in Intelligent DSS (IDSS) environments. The types, characteristics, levels of logical view, and the levels of specificity and abstraction of "passive" and "active" knowledge in IDSS are discussed. A language for knowledge description, whose syntactical objects are entities, relationships, transformations, and constraints, and which allows four levels of specificity and abstraction is proposed. Then, a graphical, semantic model for the *conceptual-schema* representation of passive and active knowledge, called the *extended ERA Model*, is presented. Finally, it is argued that a multi-paradigm programming environment is required for the *information-schema* representation of the different types of knowledge in IDSS, and to support reasoning, inference, and inheritance. A LOOPS implementation of the knowledge representation and elicitation model is described in detail.

1. INTRODUCTION

The last few years have witnessed an upsurge of interest and research in Decision Support Systems (DSS). A trend toward the design of Intelligent DSS (IDSS) has recently emerged. It is rightfully argued that a DSS should possess artificially intelligent capabilities (reasoning, inference, control) that would effectively help decision-makers in all phases of the decision making process.

This study focuses on the issues of knowledge representation and the process of knowledge elicitation in an IDSS environment. In Section 2 we discuss the types, characteristics, levels of logical view, and the levels of specificity and abstraction of knowledge in the IDSS's Problem-Domain Knowledge-Base (PDKB). The concept of generic shells is introduced, and the organization of the PDKB is presented. Generic shells provide useful conceptual templates for eliciting and representing both data ("passive") and procedural ("active") knowledge in the problem domain. In Section 3, we present the basic concepts of our model for knowledge representation. The basic, syntactical objects in the knowledge representation language are entities, relationships, transformations, and constraints. Four levels of specificity and abstraction are then identified: instances, classes, subschemas, and schemas. Based on these concepts, a graphical, semantic model for the *conceptual schema* representation of passive and active knowledge, called the *extended ERA model*, is described in Section 4. A product-mix domain is used to illustrate the model.

In Section 5, we suggest that a multi-paradigm programming environment is required in order to represent and manipulate the different types of knowledge in an IDSS environment and to support reasoning, inference, and inheritance requirements. Consequently, we chose Xerox's LOOPS (Lisp-Object-Oriented-Programming-System) (Bobrow and Stefik 1981) as the implementation environment for the IDSS's Problem-Domain Knowledge-Base (PDKB) and its knowledge elicitation system. The *object-oriented* paradigm in LOOPS is used to create objects (frames) in the problem domain and organize them in an inheritance network. The *data-oriented* paradigm is used to create active values that specify procedures to be invoked when the value of a variable (frame slot) marked as "active" is accessed. The *procedure-oriented* paradigm is used to build Interlisp procedures that compute transformations and constraints in the problem domain. The *rule-oriented* paradigm is used to trigger the execution of procedures for computing transformations and constraints. Rules are also used in the inference mechanism that interprets the user's input about instance-level objects in his application, instantiates classes in the generic schema, and builds a representation of the user's instance-level problem. The implementation using LOOPS is described in detail, using the product-mix domain for illustration purposes. Finally the Knowledge Elicitation and Representation System (KERS) is described, along with the process of knowledge elicitation and representation and its LOOPS's implementation. KERS is designed as an expert system in order to facilitate, simplify, and expedite the process of knowledge elicitation and representation.

2. THE PROBLEM-DOMAIN KNOWLEDGE-BASE

In this section, we discuss the types, characteristics, levels of logical view, and levels of specificity and abstraction of knowledge in an IDSS environment. Also, we introduce the concept of generic shells and present the organization of the Problem-Domain Knowledge-Base (PDKB).

2.1 Types, Characteristics, and Levels of Logical View of Knowledge

The knowledge system in an IDSS environment contains both transactional information and (what we may term) "conceptual information." *Transactional information* is passive, static, descriptive information about simple objects (entities, relationships) in the problem domain. *Conceptual information*, on the other hand, is structural information and abstract conceptualization about the problem domain, and about models of analysis and decision support. The objects of concern are complex entities, sophisticated relationships, abstract structures, procedures, and so on. Since the main function of conceptual knowledge in a DSS context is to support various analysis, modeling, and decision-making activities, the objects of concern are usually in the form of models: models of the problem domain and models for analysis and decision support. Models of the *problem domain* provide a conceptual representation of certain facets/subsystems in the enterprise's universe of discourse (e.g., conceptual models of a production system, a financial system, etc.). Conceptual models for *analysis and decision support* are abstract conceptualizations that provide selective, focused perception of reality, in forms that allow symbolic representation and manipulation (e.g., assessment, diagnosis, and strategic planning models; mathematical programming, statistical models; etc.). Consequently, conceptual information in a DSS context is decision-oriented and modeling-oriented, and the objects of interest are *active* entities.

Traditional data structure models in the database field are designed for the representation of transactional information only. To build a knowledge representation model sufficiently rich for representing both transactional ("passive") data and conceptual ("active") information requires the integration and combination of certain AI knowledge representation models with certain data structure models.

There are several distinct levels of logical view of knowledge, and a model for knowledge representation should be concerned with each of them:

1. The *conceptual schema* is the definition of knowledge about the problem domain as it exists in our minds. It represents our conceptual view of both transactional and conceptual knowledge.

2. The *information schema* specifies the data structures for the organization of transactional information and conceptual knowledge.

3. The *internal schema* defines the physical storage strategies for storing instances of the information schema.

In addition to representing different levels of logical view of knowledge, it is also highly desirable to represent knowledge at *different levels of specificity and abstraction* in an IDSS environment. This will simplify and facilitate the conceptualization and modeling activities, as well as the process of knowledge elicitation, inference, and representation, as will be seen later. Consequently, a model for knowledge representation in IDSS should allow the organization (and manipulation) of objects in a class-subclass hierarchy, where the most generic objects in a class lattice are highest in the hierarchy, and the most specific objects described by a class are lowest.

2.2 Generic Shells and the Problem-Domain Knowledge-Base

The Knowledge System (KS) in our IDSS design is composed of two parts: the PDKB and the Model-Domain Knowledge-Base (MDKB). In this study we focus on the PDKB, since our main concern is problem-domain knowledge elicitation and representation.

Based on the previous observations concerning the types, characteristics, levels of logical view, and levels of specificity and abstraction of knowledge in IDSS, we propose that the PDKB have the following structure (see Figure 1):

1. *Generic Shell Knowledge-Base* (GSKB), composed of:
   1.1 Generic Conceptual Schema
   1.2 Generic Information Schema
   1.3 Rules and Procedures

2. *Instance-Level Knowledge-Base* (ILKB), composed of:
   2.1 Instance-Level Conceptual Schema
   2.2 Instance-Level Information Schema

3. *Database*

A *generic shell* consists of a *conceptual schema* and an *information schema* representation of a *generic problem* in a particular domain. The assumption underlying the use of generic shells is that decision problems can usually be classified into classes of problems, since most of the instance-level problems in a particular generic class have some common structure, characteristics, and (possibly) common analysis and solution procedures. Examples of generic problems in a production/manufacturing domain include product-mix, scheduling, and distribution; examples in a marketing domain include product-planning, pricing, and media-planning. The concept of generic shell provides a useful conceptual template for eliciting and representing knowledge about instance-level problems/applications. Furthermore, since knowledge in the generic shell is organized in an inheritance network and since the knowledge elicitation and representation system have built-in inference capabilities, then the process of *instantiating* the generic shell and building a representation for an instance-level problem is greatly simplified, facilitated, and expedited.

3. BASIC CONCEPTS (PRIMITIVES) OF THE KNOWLEDGE REPRESENTATION MODEL

In this section we introduce the different types of objects in the knowledge representation language and the different levels of aggregation of these objects.

3.1 Types of Objects

We define the following types of objects:

- o Entities
- o Relationships
- o Transformations
- o Constraints

The concepts of entity and relationship have the same interpretation as in the Entity-Relationship (ER) model (Chen 1976). Thus, an *entity* is an object that can be distinctly identified; a *relationship* is an association among entities. Attached to entities and relationships are *attributes*, *transformations*, and *constraints*: they describe the *passive* (data) and the *active* (procedural) aspects of entities and relationships. (In the classic ER model, only the passive aspects of objects are represented.)

We define the following types of attributes:

a. *Deterministic data attributes*, to represent passive, descriptive information.

b. *Probabilistic data attributes*, to represent random events or quantities. Two value sets are defined for each discrete probabilistic data attribute: a "states-of-nature" set and a probability set. The information about a continuous probabilistic data attribute is expressed by a probability density function.

c. *Action/decision attributes* are abstract attributes representing *decision variables*.

d. *Transformation-based attributes* are attributes that are generated by transformations. An important case of this type are *objective/performance attributes*, to be described below.

A *transformation* T is a mapping of one set of attributes into another set of attributes. Symbolically,

$$T^{NAME}: \{ a_1^D, a_2^D, \ldots, a_n^D \} \rightarrow \{ a_1^R, a_2^R, \ldots, a_n^R \}$$

is a transformation whose name is NAME, $\{ a_1^D, a_2^D, \ldots, a_n^D \}$ is the set of *domain attributes* of T (i.e., the set of attributes that undergo transformation), and $\{ a_1^R, a_2^R, \ldots, a_n^R \}$ is the set of *range attributes* of T (i.e., the set of attributes that are formed by the transformation T operating on the domain attributes). Transformations represent functional, causal, definitional and other relationships in the problem domain. They represent primitive/atomic models, allow the creation and representation of abstract concepts, and the construction of complex decision models. Transformations represent active, procedural, and modeling-oriented conceptual knowledge. The following comments are also in order:

a. A transformation T is said to be in *implicit form* if the mapping from the domain space into the range space is not specified explicitly. The *explicit form* of T is specified in terms of a *model* (active procedure). The model underlying the transformation may be either *analytical* or *numerical*. Analytical models are usually mathematical/statistical (e.g., mathematical expressions, Regression Analysis). If the mapping (model) is complex and/or involves complex probabilistic data attributes in its domain, so that the mapping is not amenable for analytical formulation, then a numeric model (usually simulation) is employed.

b. The domain of T may include any type of attributes, as well as other transformations. If all the attributes in the domain are data attributes, then T is a *data transformation* (most statistical transformations are of this type). If some of the attributes in the domain are action attributes, then T is an *action transformation*.

c. A transformation T may have many attributes in its range. Also, the model for evaluating a transformation in explicit form is not necessarily unique.

A *constraint* is a relationship that expresses restrictions on the possible values of action/decision attributes. The restrictions may be specified in explicit form, in which case an *action-value set* is associated with each action attribute, and enumerates all the possible values that the action attribute may assume. Alternatively, the restrictions may be specified in implicit form, i.e., by a *constraint relationship*. The general form of a constraint relationship is:

< Transformation-Based-Attribute > . θ . < Constraint-Attribute >

where <Transformation-Based-Attribute> is generated by an action transformation T; <Constraint-Attribute> is an attribute that represents, for example, maximum availability/capacity of a resource (it may be either a data attribute or a transformation-based attribute); θ is a binary relation selected from the set

$\theta := \{EQ, NE, LT, LE, GT, GE\}$.

To each transformation T and to each constraint, we attach the following: (1) *Rules* for triggering the execution of one or more procedures for evaluating/computing the transformation or constraint, and (2) *Procedures* for computing the transformation or constraint.

3.2 Levels of Aggregation of Objects

We identify four levels of aggregation in the proposed language for knowledge representation to reflect the different levels of specificity and abstraction in the problem domain: (1) instances, (2) classes, (3) subschema, and (4) schema. More specifically:

a. An *instance* (of a class) is an object that can be distinctly identified.

b. *Classes* are groups of similar objects.

c. A *subschema* is an array of one or more related classes.

d. A *schema* is an ordered set of subschemas. A schema may have attached to it one or more *objective/performance attribute(s)*. This attribute measures the entire schema's performance. An *operator* such as OPTIMIZE (MAXIMIZE or MINIMIZE) or SATISFICE is associated with each objective/performance attribute. Each objective/performance attribute is computed by an *action* transformation.

Given the types of objects identified in the previous section, we obtain the following classification of objects:

Instances

a. Entities
b. Relationships
c. Attributes:
   1. Data Attributes:
      i. Deterministic
      ii. Probabilistic
   2. Action/Decision Attributes
   3. Transformation-Based Attributes
d. Transformations
e. Constraints

Classes

a. Entity Sets
b. Relationship Sets
c. Attribute Sets:
   1. Data Attribute Sets:
      i. Deterministic
      ii. Probabilistic
   2. Action/Decision Attribute Sets
   3. Transformation-Based Attribute Sets
d. Transformation Sets
e.Constraints Sets

4. THE EXTENDED ERA MODEL FOR CONCEPTUAL SCHEMA REPRESENTATION

We propose a graphical, semantic model for the conceptual schema representation of both "passive" and "active" knowledge, called the *extended (generalized) ERA model*. Recalling the organization of the PDKB (Figure 1), the extended ERA model is used to represent the conceptual schema of *generic problems* and the conceptual schema of *instance-level problems*.

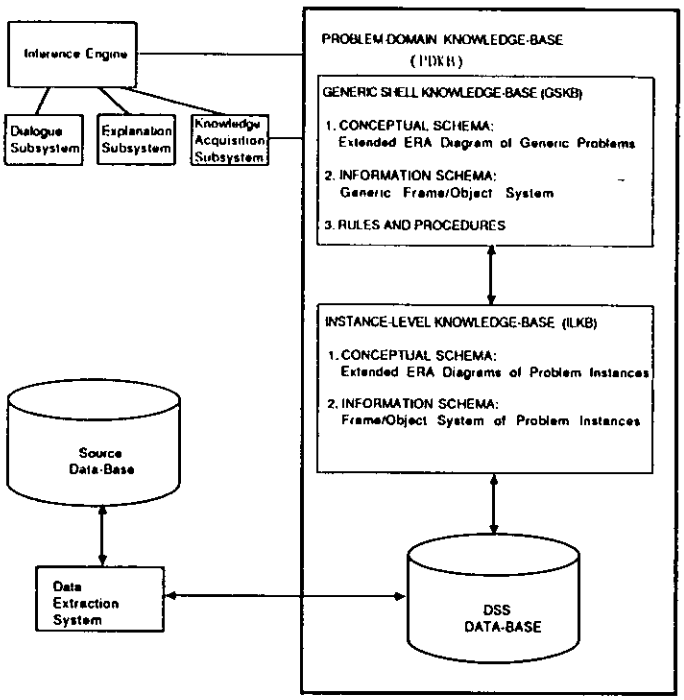

Figure 1. The Knowledge Elicitation and Representation System

The graphical representation of the conceptual schema is in the form of *extended ERA diagrams.* The following graphical symbols are used in extended ERA diagrams:

1. Entity sets are represented by rectangular boxes.

2. Relationship sets are represented by diamond-shaped boxes.

3. Attribute sets are represented by circular or oval boxes connected to entity/relationship sets by arcs.

4. Action and probabilistic attribute sets are designated as such.

5. Transformation sets are represented by the symbol T^{NAME} (where NAME is the name of the transformation) below the arc that connects the "Attribute.Set.Of" T^{NAME} with the transformation-based attribute set. The *implicit form* of T^{NAME} is shown in the legend accompanying the extended ERA diagram.

6. Constraint sets are graphically represented by an octagon-shaped box. Two dashed arcs are connected to each constraint set: one from the *<Transformation-Based-Attribute Set>* on the left-hand side (LHS) of the constraint relationship set, and one from the *<Constraint Attribute Set>* on the right-hand side (RHS). The constraint relationship is stated inside the octagon-shaped box, and the name of the constraint set is stated outside the box.

7. If an attribute set is generated/defined in a *different* conceptual subschema (i.e., a different extended ERA diagram), a dashed rectangular box represents the (external) subschema, with the name of the subschema stated in the box, and a dashed arc connecting it to the attribute set. The attribute set is usually an action attribute set, or a transformation-based attribute set.

4.1 Extended ERA Diagrams of Generic Problems

A generic shell of a problem domain is defined in terms of *generic classes* (i.e., generic entity sets, generic relationship sets, generic attribute sets, generic transformation sets, and generic constraint sets), *generic class dependencies*, *generic subschemas*, and *generic schemas*. Accordingly, generic objects only (with their attached generic

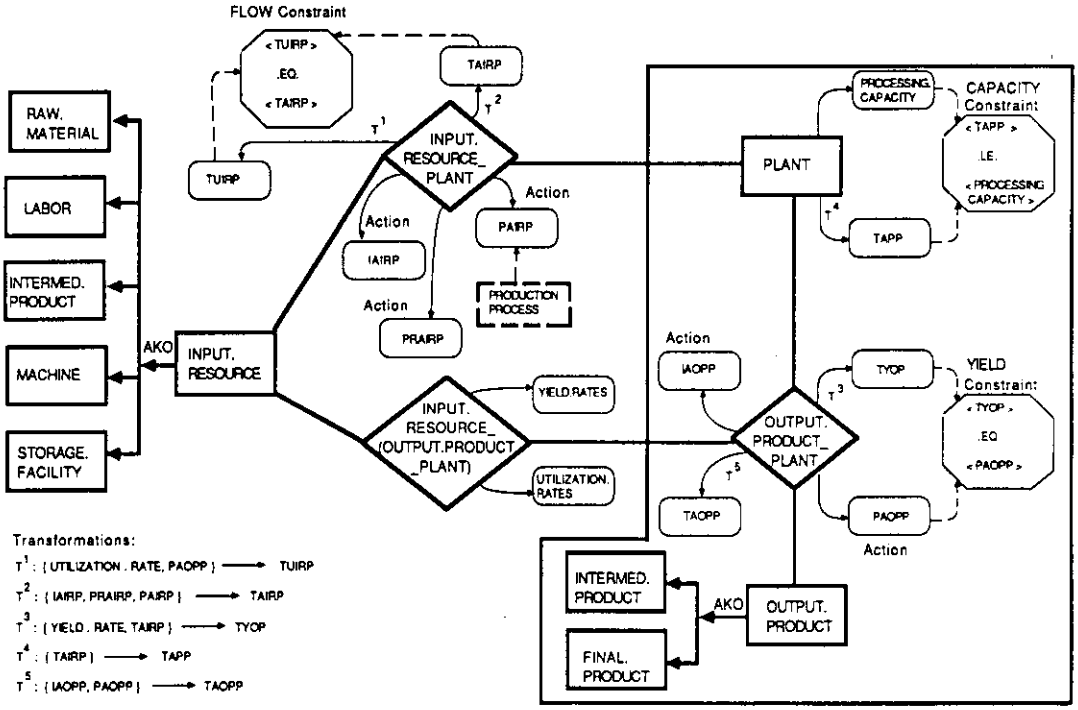

Figure 2. Extended ERA Diagram of a Generic Production Process

data and procedures) appear in the extended ERA diagrams of generic problems.

A generic product-mix problem is used for illustration purposes. There are two *generic subschemas* in a generic product-mix schema: a *generic production process* and a *generic distribution process.*

Figure 2 shows the extended ERA diagram of the conceptual-schema representation of the generic production process. A generic production process represents the conversion of *input-resources* to *output-products*, in *plants*, at some *planning-period.* Accordingly, three *generic entity sets* are defined in Figure 2: INPUT.RESOURCE (with the generic subclasses RAW.MATERIAL, LABOR, INTERMEDIATE.PRODUCT, MACHINE, STORAGE.FACILITY), OUTPUT.PRODUCT (with the generic subclasses INTERMEDIATE.PRODUCT, FINAL.PRODUCT), and PLANT. Three *generic relationship sets* appear: INPUT.RESOURCE-PLANT, OUTPUT.PRODUCT-PLANT, and INPUT.RESOURCE-(OUTPUT.PRODUCT - PLANT). (Notice that the last is a relationship set between an entity set and a relationship set.) The *generic action/decision* attribute sets are PAOPP (PRODUCTION.AMOUNT of OUTPUT.PRODUCT in PLANT), IAOPP (INVENTORY.AMOUNT of OUTPUT.PRODUCT in PLANT), PRAIRP (PURCHASE.AMOUNT of INPUT.RESOURCE in PLANT), PAIRP (PRODUCTION.AMOUNT of INPUT.RESOURCE in PLANT), and IAIRP (INVENTORY.AMOUNT of INPUT.RESOURCE in PLANT). (Notice that PAIRP is generated/defined by an *external* Production.Process, which means that some input resources in the *current* production process were output products in a *previous* production process. In other words, *multi-stage production systems* are allowed.) The following *generic transformation* sets are defined:

1. T^{TUIRP}, or TOTAL.UTILIZATION of INPUT.RESOURCE in PLANT. Its implicit form is T^{TUIRP}: {UTILIZATION.RATE, PAOPP} ->TUIRP.If I, P, and O are, respectively, the index sets of INPUT.RESOURCE,

PLANT, and OUTPUT.PRODUCT, then the explicit form of T^{TUIRP} is TUIRP[I,P]:= SUM[0](UTILIZATION.RATE[I,O,P] * PAOPP[O,P])where "SUM" is a summation symbol.

2. T^{TAIRP} is the transformation that computes the TOTAL.AVAILABILITY of INPUT.RESOURCE in PLANT T^{TAIRP}:{IAIRP, PRAIRP, PIRP} -> TAIRP, and in explicit form (T is the index for PLANNING.PERIOD, which we suppressed thus far): TAIRP[I,P,T]:=IAIRP[I,P,T-1] + PRAIRP[I,P,T] + PAIRP[I,P,T] - IAIRP[I,P,T].

3. T^{TYOP} is the transformation that generates TOTAL.YIELD of OUTPUT.PRODUCT in PLANT: T^{TYOP}: {YIELD.RATE,TAIRP} -TYOP, and in explicit form:TYOP:[O,P]:= SUM[I](YIELD.RATE[I,O,P] * TAIRP[I,P]).

   T^{TYOP} computes the output of the "blending" of some input resources in the process of producing an output product (as found, for example, in many chemical production processes). Notice that one of the attribute sets in the domain of T^{TYOP}, i.e., TAIRP, is a transformation-based attribute set. Thus, to evaluate T^{TYOP}, we first need to evaluate T^{TAIRP}.

4. T^{TAPP} is the transformation that computes TOTAL.AMOUNT (of all INPUT.RESOURCES) processed in PLANT: T^{TAPP}: {TAIRP} -> TAPP, and in explicit form: TAPP[P,T]: = SUM[I]ITAIRP[I,P,T]).

5. T^{TAOPP} is the transformation that computes the TOTAL.AVAILABILITY of OUTPUT.PRODUCT in PLANT: T^{TAOPP}: {IAOPP,PAOPP} -> TAOPP, and in explicit form:

TAOPP[O,P,T]: = IAOPP[O,P,T-1]+PAOPP[O,P,T]= IAOPP[O,P,T].

A generic production process includes three generic constraint sets:

1. *A Flow Constraint Set*:

   <TUIRP> .EQ. <TAIRP>.

2. *A Capacity Constraint Set*:

   <TAPP> .LE. <PROCESSING.CAPACITY>.

3. *A Yield Constraint Set*:

   <TYOP> .EQ. <TAOPP>.

Figure 3 shows the extended ERA diagram of a generic distribution process. It describes (in generic terms) the distribution of *products* from *sources* to *destinations* at some planning period. The generic entity sets, relationship sets, action attribute sets, transformation sets, constraint sets, etc., are described in Figure 3. Notice that the concept "distribution" is defined in a broad sense; for instance, it includes the sale of products to markets/customers.

4.2 Extended ERA Diagrams of Instance-Level Problems

An instance-level problem is a *specific* problem/application. As such, it is defined in terms of instance-level objects only, and so is the extended ERA diagrams representing it.

Using the product-mix domain for illustration purposes, we show in Figures 4 and 5 the extended ERA diagrams of two instance-level production processes. Figure 4 depicts the process in which Labor, Iron, Component A, and Component B (all of which are instances of Input-resources) are used to produce Final-Products 1, 2, and 3 in Plants I and II. Figure 5 depicts a typical "blending" process in the oil industry, in which Light-Crude and Heavy-Crude are used as inputs in Refineries A and B to produce Gasoline, Kerosene, and Industrial Fuel. It is important to notice that the instance-level production processes in Figures 4 and 5 represent two different instances (realizations) of the *same* generic production process shown in Figure 2.

5. LOOPS IMPLEMENTATION OF THE KNOWLEDGE ELICITATION AND REPRESENTATION SYSTEM

We propose a *frame system* for the representation of the information schema of both generic problems and instance-level problems. Using frames to represent the information schema of generic problem domains and to drive the knowledge elicitation and inference process by which instance-level information schemas are derived is quite natural, since frames are well suitable as data structures for representing prototyped/stereotyped concepts or situations (Minsky 1975, 1968). Frames also facilitate recall, inference, and interpretation.

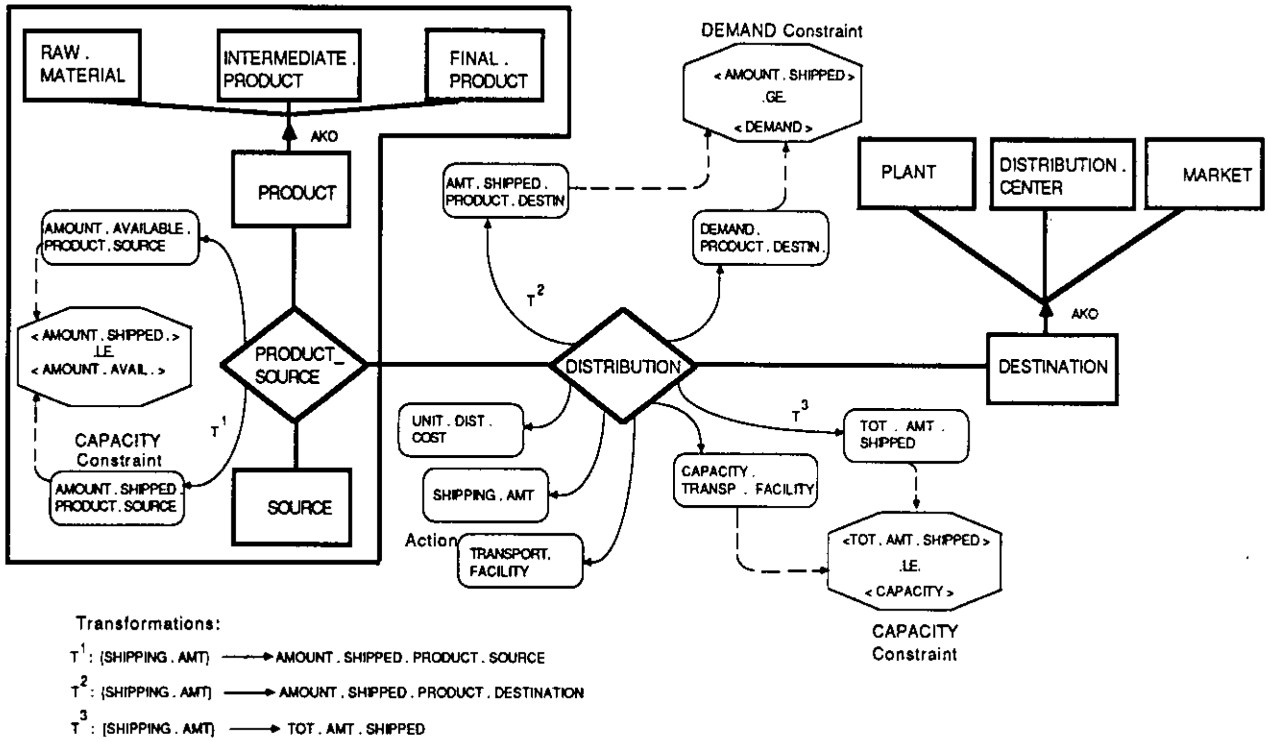

Figure 3. Extended ERA Diagram of a Generic Distribution Process

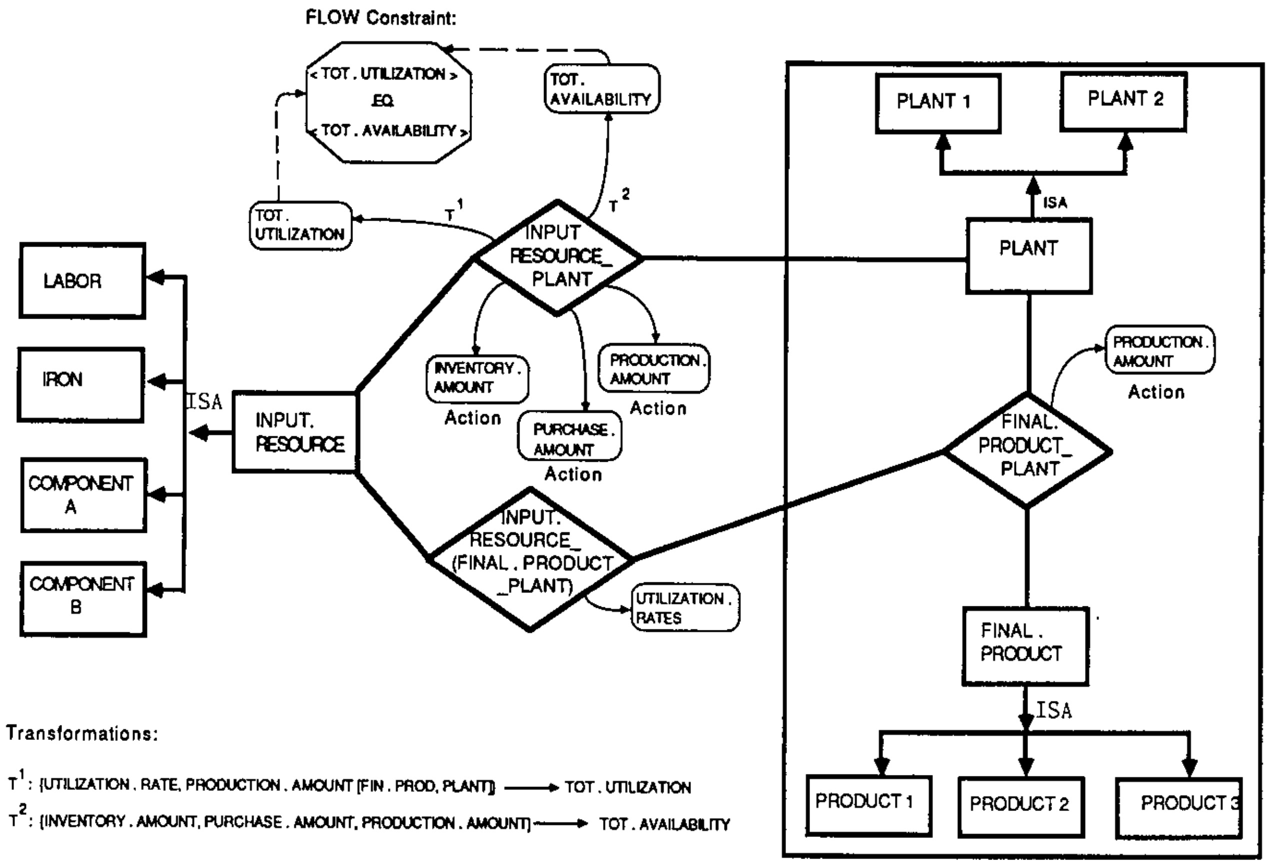

Figure 4. Extended ERA Diagram of a Production Process Instance

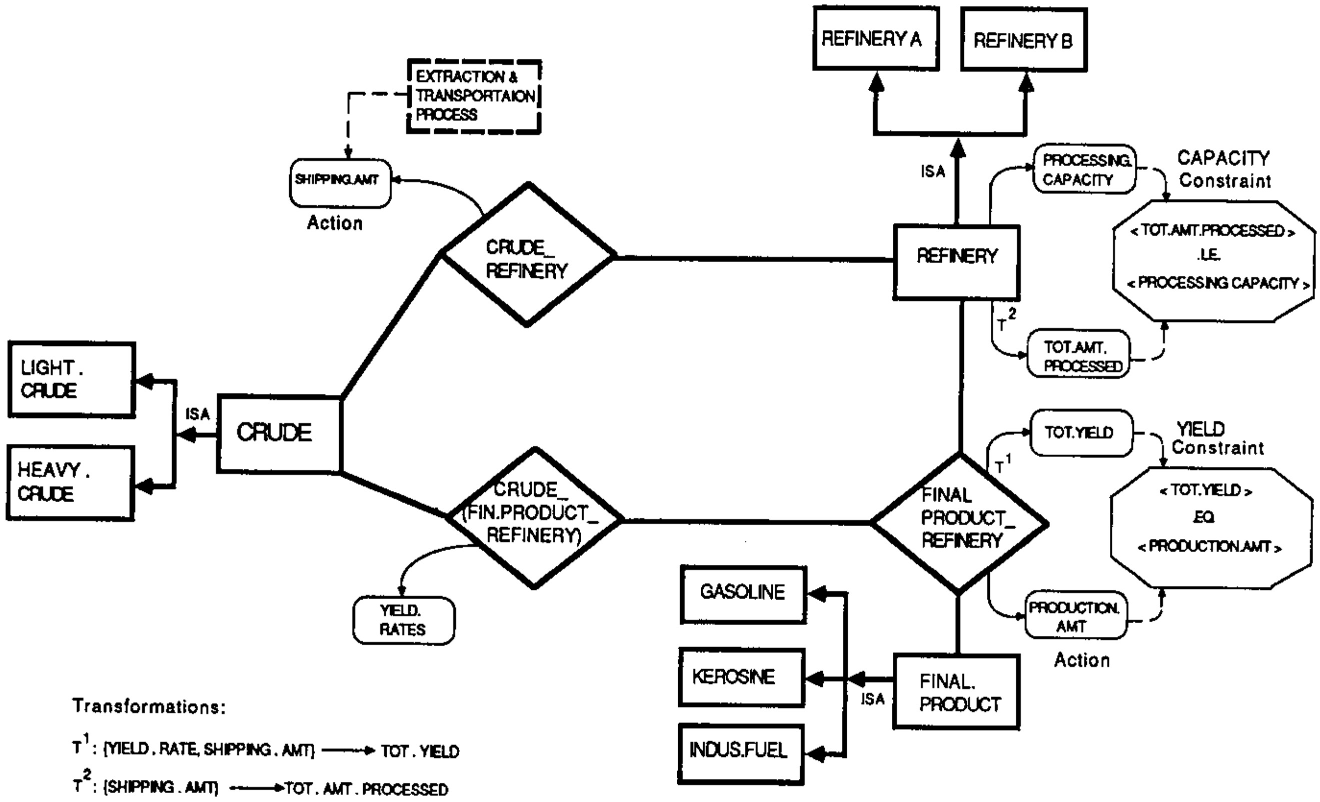

Figure 5. Extended ERA Diagram of Oil Refining Process Instance

5.1 Capability Requirements

A primary motivation for employing the concepts of generic conceptual schema and generic information schema is to simplify, facilitate, and support the process of elicitation and representation of knowledge about instance-level problems. To accomplish this objective, a programming environment for implementing the knowledge representation models and for supporting the knowledge elicitation process should possess the following capabilities:

1. The ability to represent objects such as entities and relationships along with the *passive* (data) and *active* (procedural) properties attached to these objects, such as data attributes, action attributes, transformations, and constraints.

2. The ability to represent different levels of specificity, abstraction, and aggregation. This requires the ability to represent *class-subclass* relationships, *membership* relationships, and *composite* ("part-of") relationships.

3. *Inheritance mechanisms*, so that a subclass in a class-subclass hierarchy automatically inherits certain properties of its superclass, and that instance-level objects inherit the attributes/properties of their classes.

4. *Inference mechanism* that, based on its knowledge about the generic problem-domain, will request the user to specify certain instance-level objects, and then proceed to automatically instantiate other generic objects/concepts (such as relationships, transformations, constraints) and, finally, create a frame system and extended ERA diagrams to represent the instance-level problem.

5. Access to procedural languages, subroutines, and software packages. In many cases, the evaluation of a transformation requires the execution of statistical models, mathematical programming, simulation, and other models. A convenient access to procedural languages/subroutines/packages is highly desirable in these cases.

6. The ability to manipulate and maintain attribute values of entities/relationships, including: (a) traditional database functions such as update, delete, append, etc., (b) referential constraint checking, to ensure that reference is never made to a nonexisting object, and (c) the automatic update of the value of a transformation-based attribute as a result of updating the value of a data attribute in the transformation's domain.

7. Explanations, help, and tutoring capabilities.

5.2 LOOPS Implementation of the Knowledge Representation System

The requirements concerning the support of knowledge representation and elicitation as listed above suggest that a multi-paradigm programming environment is needed in order to represent and manipulate the different types of knowledge in an IDSS environment and to support reasoning, inference, and inheritance requirements. For these reasons, we chose Xerox's LOOPS (Lisp-Object-Oriented-Programming-System) (Bobrow and Stefik 1981) as the implementation environment for the IDSS's PDKB and its knowledge elicitation system. LOOPS is built on top of the Interlisp-D programming environment and supports four programming paradigms: object-oriented, procedure-oriented, rule-oriented, and data-oriented. Under the *object-oriented* paradigm, programs are organized around objects that have aspects of both procedures and data. Procedures are invoked by sending "messages" between objects. Objects are organized in an inheritance network. The *procedure-oriented* paradigm allows users to build procedures in Interlisp and use the extensive environmental support of the Interlisp-D system. The *rule-oriented* paradigm permits users to create RuleSets, i.e., sets of condition-action production rules. The *data-oriented* paradigm allows users to create procedural attachments, which are procedures attached to variables or frame slots designated as "active." The attached procedures are automatically invoked and executed upon accessing the "active value" slots.

We will use the product-mix problem (whose extended ERA diagrams are shown in Figures 2 and 3) in order to illustrate the implementation of the knowledge elicitation and representation system on LOOPS. The implementation runs on a Xerox 1108 "Dandelion" Workstation.

Figure 6 shows a portion of the LOOPS code used to define the generic product-mix problem. Figure 7 shows the pseudo-code of some of the procedures that are attached to variables declared as "active" in Figure 6. (In the actual implementation, of course, these procedures are written in Interlisp.) The reader can verify that the code in Figure 6 represents the hierarchy and relationships of the generic product-mix problem as shown in Figures 2 and 3. The following comments highlight the features of the implementation.

a. The object-oriented paradigm is used to create objects (frames) in the problem domain and organize them in an inheritance network. The data-oriented paradigm is used to create active values that specify procedures to be invoked when the value of a variable (frame slot) marked as "active" is accessed. The procedure-oriented paradigm is used to build Interlisp procedures that compute transformations and constraints in the problem domain. The rule-oriented paradigm is used to trigger the execution of procedures for computing transformations and constraints. Rules are also used in the inference mechanism that interprets the user's input about instance-level objects in his application, instantiates classes in the generic schema, and builds a representation of the user's instance-level problem.

b. *Class hierarchy*. First, notice that a *class* in LOOPS is a description of one or more similar objects; an *instance* is an object described by a particular class; and a *metaclass* is a class whose instances are classes. In order to represent the hierarchy of objects in the generic product-mix problem (or in any other domain), we need the following kinds of relationships: composite relationships; subclass-class relationships; membership relationships. More specifically:

   1. *Composite relationships* describe "Part-Of" relationships between objects (e.g., "wing" is Part-Of "plane"). In LOOPS, this kind of relationship is created by the concept of *composite object*, which is described by

```
[DEFCLASS ProductMix
    (MetaClass Template doc
        (**  Composite object representing generic product-mix problems composed of
             generic production process and generic distribution process.))
    (Supers Object)
    (InstanceVariables
        (Name NIL doc)
        (Planning-Horizon NIL doc (* Time periods covered by current product-mix
             planning problem.))
             .
             .
             .
        ( . . .))
    (Methods
        (ObjectiveAttribute ProductMix.Objective doc (*  Computes the equation for
             the objective/performance attribute of the product-mix problem.))]

[DEFCLASS ProductionProcess
    (MetaClass Template PartOf  ($  ProductMix) doc (* Generic production process
        converts Input-Resources to Output-Products in Plants.))
    (Supers Object)
    (InstanceVariables)
        (Name NIL doc)
        (Description NIL doc)
             .
             .
             .
        ( . . . ))
    (Methods
        (ExtERADiagram ProductionProcess.Diagram doc (* Produces Extended ERA
             diagrams of a production process.))]

[DEFCLASS DistributionProcess
    (MetaClass Template partOF ($ ProductMix) doc (* Generic distriubtion process
        describes the distribution of products from sources to destinations.))
    (Supers Object)
    (InstanceVariables
             .
             .
             .
        ( . . . ))
    (Methods
        (ExtERADiagram DistributionProcess.Diagram doc (* Produces Extended ERA
             diagrams of a distribution process.))]
```

Figure 6. Class Definitions of Generic Product-Mix Problem in LOOPS

```
[DEFCLASS InputResource
    (MetaClass Template partOf ($ ProductionProcess) doc (* . . .))
    (Supers Object)
    (InstanceVariables
        (Name NIL doc)
        (Description NIL doc)
            .
            .
            .
        ( . . . ))

[DEFCLASS Plant
    (MetaClass Template partOf ($ ProductionProcess) doc (* ...))
    (Supers Object)
    (InstanceVariables
        (Name NIL doc)
        (Location NIL doc)
        (Manager NIL doc)
        (Processing-Capacity NIL doc)
        (TAPP #(NIL Calc-TAPP NIL) doc (* Compute the equation for the Total-Amount
            of all Input-Resources Processed in Plant.))
        (Capacity Constraint #(NIL Calc-CapacityConstraint NIL)
            doc (* Computes the Capacity-Constraint <TAPP>.EQ.<Processing-Capacity>
            for Plant.)))]

[DEFCLASS InputResource-Plant
    (MetaClass Class Edited: (* . . .))
    (Supers Object)
    (InstanceVariables
        (Plant-Id NIL)
        (InputResource-ID NIL)
        (PAIRP NIL doc (* Action/decision variable: Production-Amount of Input-
            Resource in Plant.))
        (PRAIRP NIL doc (* Action/decision variable: Purchase-Amout of Input-
            Resource in Plant.))
        (IAIRP NIL doc (* Action/decision variable: Inventory-Amount of Input-
            Resource in Plant.))
        (TUIRP # (NIL Calc-TUIRP NIL) doc (* Computes the equation for the Total-
            Utilization of Input-Resource in Plant.))
        (TAIRP # (NIL Calc-TAIRP NIL) doc (* Computes the equation for the Total-
            Availability of Input-Resource in Plant.)
        (FlowConstraint # (NIL Calc-FlowConstraint NIL) doc (* Computes the Flow-
            Constraint <TUIRP>.EQ.<TAIRP> for Plant.)))]

[DEFCLASS Machine
    (MetaClass Class Edited: (* . . .))
    (Supers InputResource)
    (ClassVariables)
    (Instance Variables
        (Machine-Speed NIL doc (* . . .))
        (Capacity NIL doc (* . . .))
        (Maintenance-Schedule NIL doc (* . . . ))]
```

Figure 6. Continued

creating a class whose metaclass is *Template*. In the representation of the product-mix problem in Figure 6, composite object templates are created for ProductMix, ProductionProcess, DistributionProcess, InputResource, OutputProduct, and Plant. The objects ProductionProcess and Distribution-Process are Part-Of the composite object ProductMix; in turn, the objects InputResource, OutputProduct, and Plant are Part-Of the composite object ProductionProcess. When the message *New* is sent to a composite class, then all of the parts starting with this class are instantiated.

2. *Subclass-class relationships* relate subclasses to their superclasses: they represent the AKO (A-Kind-Of) relationship. In LOOPS, the concept of *metaclass* describes the subclass-class relationship; the Supers list in a class definition shows the list of all the metaclasses of the class. In Figure 6, for example, Machine is a subclass of the metaclass template InputResource.

3. *Membership relationships* relate instances to a class: They represent the IS-A relationship. In LOOPS, instances of a class are created by sending the class the message *New*. For example, all the lowest level objects in Figures 4 and 5 (e.g., Iron, Component A, Plant 1, Product 2, Light Crude, Refinary A, Gasoline) are instances of classes defined in Figure 3 (in conceptual-schema form) and in Figure 6 (in LOOPS code form).

c. *Variables and methods.* Objects may have variables and methods. *Class variables* contain information shared by all instances of the class; *instance variables* contain information specific to an instance (see examples in Figure 6). Procedures are invoked by sending "messages" between objects; these procedures (or methods) are Interlisp functions. In Figure 6, ProductMix.Objective in the Methods declaration of the metaclass template ProductMix is the name of an Interlisp function that computes the equation for the objective/performance attribute (e.g., NetRevenue or Profits) for each instance of ProductMix. The function ProductMix.Objective is to be applied when the message *ObjectiveAttribute* is received. Similarly, the Interlisp function ProductionProcess.Diagram in the metaclass template ProductionProcess is to be applied when the message *ExtERADiagram* is received: this function creates Extended ERA diagrams for instances of ProductionProcess (such as the ones shown in Figures 4 and 5).

d. *Inheritance.* A subclass in the class hierarchy automatically inherits the properties of variables and the methods of its superclass, unless overridden in the subclass. The Supers list in the class definition specifies the superclasses from which properties and methods are inherited. In Figure 6, for example, the subclass Machine inherits properties from its superclass InputResource. Also, instance-level objects inherit values through default facets of the class.

e. *Attached procedures: active values.* If the value of a variable/slot is marked as "active," then the active value specifies the Interlisp procedures to be invoked when the value of the variable/slot is accessed (read or set). Active values are used to compute and generate the equations for *transformations* and *constraints* in the problem domain. Consider the instance variables TUIRP, TAIRP, and FlowConstraint in the definition of the relationship set Input-Resource-Plant in Figure 6 and their respective active values #(NIL Calc-TUIRP NIL), #(NIL Calc-TAIRP NIL), and #(NIL Calc-FlowConstraint NIL). Whenever the variable TUIRP, TAIRP, or FlowConstraint is accessed, the corresponding function (Calc-TUIRP, Calc-TAIRP, or Calc-FlowConstraint) is invoked. Each of these functions, whose *pseudo code* is shown in Figure 7, performs two tasks. First, it checks the conditions for triggering the execution of the procedure that computes the equation for the transformation or constraint for each instance of the relationship set InputResource-Plant. If these conditions are met, then the equation for the transformation or constraint is computed, using the generic procedure specified by the function. For illustration purposes, consider the instance (InputResource-Plant)[I=4, P=3, T=1] (where I, P, and T are, respectively, the indices for InputResource, Plant, and TimePeriod). Using the generic procedures in Figure 7, and assuming that the conditions for computing the transformations and constraint are met, then Calc-TUIRP may produce the equation TUIRP[4,3,1]: = 2 * PAOPP[1,3,1] + 5 * PAOPP [2,3,1], Calc-TAIRP may produce the equation

1. **Calc-TUIRP:**

   IF (UtilizationRate ISA Data-Attribute.Of (Input.Resource-(Output.Product-Plant)) Relationship) AND (PAOPP ISA Action-Attribute.Of (Output.Product-Plant) Relationship)

   THEN TUIRP[I,P,T]:=SUM[0]*(Utilization.Rate[I,O,P,T] * PAOPP[O,P,T]).

2. **Calc-TAIRP:**

   IF (IAIRP or PRAIRP or PAIRP ISA Action-Attribute.Of (Input.Resource-Plant) Relationship)

   THEN TAIRP[I,P,T]:=IAIRP[I,P,T-1] + PRAIRP[I,P,T] + PAIRP[I,P,T] - IAIRP[I,P,T].

3. **CALC-FlowConstraint:**

   IF (TUIRP AND TAIRP ISA Transformation-Based-Attribute.Of (Input.Resource-Plant) Relationship)

   THEN FlowConstraint[I,P,T]:=<TUIRP[I,P,T]>.EQ.<TAIRP[I,P,T]>.

Figure 7. Pseudo-Code of Procedures Attached to Active Values

TAIRP[4,3,1]: = IAIRP[4,3,0] + PRAIRP[4,3,1] + PAIRP[4,3,1] - IAIRP[4,3,1], in which case Calc-FlowConstraint produces the following flow-constraint:

<2*PAOPP[1,3,1]+5*PAOPP[2,3,1]>
.EQ.
<IAIRP[4,3,0]+PRAIRP[4,3,1]+PAIRP[4,3,1]-IAIRP[4,3,1]>

(The transformations involved in the product-mix problem and, consequently, the Interlisp functions to compute them are relatively simple. In other problem domains, however, the transformations may be quite complex and require, for example, a simulation model in order to compute them. Being built on top of Interlisp, LOOPS has an immediate access to Interlisp functions, as well as to other procedural languages (through interfaces), thus enabling it to compute complex transformations.)

5.3 The Knowledge Elicitation and Representation System (KERS)

The Knowledge Elicitation and Representation System (KERS) is a subsystem of the IDSS responsible for eliciting basic facts and other knowledge about instance-level problems/applications from the user, and creating information- and conceptual-schema representations of the user's instance-level problem/application. In order to simplify, facilitate, and expedite these tasks, KERS is designed as an expert system (see Figure 1):

a. Its *knowledge-base* is the *Problem-Domain Knowledge-Base* (PDKB), composed of Generic Shell Knowledge-Base (GSKB), Instance-Level Knowledge-Base (ILKB), and Data-Base. More specifically:

   1. The *Generic Shell Knowledge-Base* contains the following:

      1.1 A description of generic problems/applications in the enterprise's universe of discourse, in both a *conceptual-schema* representation (e.g., the Extended ERA diagrams for generic product-mix problems, Figures 2 and 3), and an *information-schema* representation (e.g., LOOPS code, Figure 6).

1.2 Rules and procedures for eliciting basic facts and knowledge about instance-level problems/applications from the user, and for *instantiating* the relevant generic shells and building conceptual- and information-schema representations of the instance-level problems/applications.

b. The *inference engine* contains the inference strategies and controls used during the instantiation process.

c. The *knowledge acquisition subsystem* is used to create conceptual- and information-schema representations of new generic problems/ applications, to update and modify existing representations, and to create new rules and procedures, and update and modify existing rules.

In addition, the KERS also contains an explanation subsystem and a dialogue subsystem.

KERS is also implemented in LOOPS, using the RuleSets in LOOPS's rule-oriented paradigm. Given the relevant generic shells in KERS's Generic Shell Knowledge-Base, the knowledge elicitation and representation process works as follows (we again use the product-mix problem for illustration purposes).

a. *Problem classification.* A user-system dialogue takes place at this step. The purpose is to classify the user's problem/application instance into a generic class. The inference process at this stage involves matching instance-level facts and knowledge (elicitated from the user) with knowledge about generic problems. Key words and terms, as well as the objectives of the analysis (as stated by the user), are also used in the identification and classification process. (In many cases, the user knows the classification of a given problem instance, in which case this step is bypassed.)

b. *Object instantiation.* In this step, the inference mechanism instantiates the generic objects/concepts and produces information- and conceptual-schema representations of the problem/application instance. The instantiation process works in a top-down fashion, following the class hierarchy of the generic problem/domain representation. In terms of the conceptual-schema representation, the *subschemas* are first identified, then the generic *entity sets* of each subschema are instantiated, then the generic *relationship sets*, followed by the instantiation of the generic *transformation* and *constraint sets*. This strategy is implemented in terms of the information-schema representation (described in Figure 6) as follows:

1. The user-system dialogue identifies the *composite relationships* in the domain, i.e., the composite objects and the objects that are declared as Part-Of composite objects. Recalling the composite relationships defined in Figure 6, assume that the user indicates that the first process in his product-mix application is a production process. The system then informs the user that the (generic) objects InputResource, Output Product, and Plant are Part-Of the metaclass template ProductionProcess.

2. The user-system dialogue identifies the relevant subclasses in each *subclass-class relationship*, using the Supers list in the class definitions of Figure 6. Assuming that the ProductionProcess instance is the one shown in Figure 4, it is determined that RawMaterial, Labor, and IntermediateProduct are the relevant subclasses of the superclass InputResource, and that FinalProduct is the relevant subclass of OutputProduct.

3. The classes in the *membership relationships* are instantiated. To this end, the user is asked to instantiate *entity sets* only. In the example of Figure 4, the user specifies that Iron is an instance of the RawMaterial class; Labor is an instance of the Labor class; Components A and B are instances of the IntermediateProduct class; Products 1, 2, and 3 are instances of the FinalProduct class; and Plant 1 and 2 are instances of the Plant class. Then, the system automatically instantiates the *relationship sets*, based on its knowledge about the entity sets instances and on the definition of generic relationship sets (e.g., the class InputResource-Plant in Figure 6).

4. The *values of variables* of objects are instantiated. To this end, the system presents the user with the *instance variables* of each object defined in the generic information schema, and the user is asked to supply values for these variables. For example, the user will specify the values of the instance variable UtilizationRate for each instance of the object InputResource-(FinalProduct-Plant) in Figure 4. Then, the inheritance mechanisms are applied according to the class hierarchy: each subclass inherits the values of *class variables* of its superclasses as specified in the Supers list; each instance inherits values through default facets of its class.

5. The Interlisp procedures specified in the *active values* are automatically invoked and executed at this stage: they compute the equations for the transformations and constraints of each instance of a class whose definition contains active values. In the example of Figure 4, a Total. Utilization equation (transformation T^1), a Total.Availability equation (T^2 in Figure 4), and a Flow.Constraint equation are produced for each instance of the relationship set InputProduct-Plant. These equations are produced, respectively, by the generic procedures Calc-TUIRP, Calc-TAIRP, and Calc-FlowConstraint.

c. *Generation of instance-level Extended ERA diagrams.* Given the information about the instances of all the objects, KERS produces the Extended ERA diagrams of the subschema/schema. In the example of the production subschema, this task is performed by sending the message *ExtERADiagram* to the object Production Process (see Figure 6), which activates the Interlisp function ProductionProcessDiagram that generates the diagram shown in Figure 4.

The following comments are in order:

1. *Explanation and tutoring.* At each step, the system provides the user with explanation, help, and tutoring support. This includes a detailed explanation of the structure of the generic shell of the domain (e.g., generic product-mix problems) with examples, prototypes, etc. It also includes explanations of each object and the variables, procedures, and methods attached to it, through the declaration "doc" ("documentation") in the class definitions (see Figure 6). Finally, the reasoning process in KERS is made *transparent* through LOOPS's *audit trail* mechanism, which gives an account and explanation for its reasoning process.

2. *Verification and validation.* During the object instantiation process, the system will verify with the user its "understanding" of the semantics of the instantiation. As an example, consider the instantiation of the relationship sets (e.g., InputResource-Plant, OutputProduct-Plant in Figure 2). The system presents the user with each *possible* instance of, say, OutputProduct-Plant; the user then indicates if the instance exists or not. Furthermore, the instance-level Extended ERA diagrams are also used for verification/validation: the point here is that verification/validation is best done using a *graphical conceptual representation* of the problem/application rather than a detailed data-structure representation. Finally, LOOPS provides delete, append and other functions for update/revision purposes.

3. The design of KERS simplifies, facilitates, and expedites the process of knowledge elicitation and representation. First, the use of the concept of generic problem domains provides a convenient conceptual template for knowledge elicitation and representation. Second, the system requires the user to provide only the necessary minimal information about objects in his application; other objects and constructs are then generated automatically by the system.

REFERENCES

Bobrow, D. G., and Stefik, N. *The LOOPS Manual*, Technical Report KB-VLSI-81-13, Knowledge Systems Area, Xerox, Palo Alto Research Center, 1981.

Chen, P. P. "The Entity-Relationship Model: Toward a Unified View of Data." *ACM Transactions on Database Systems*, Vol. 1, March 1976, pp. 9-36.

Minsky, M. "Framework for Representing Knowledge." In P. H. Winston (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.

Minsky, M. (Ed.) *Semantic Information Processing*. MIT Press, Cambridge, 1968.