Dependency Directed Reasoning and Learning in Systems Maintenance Support

VASANT DHAR AND MATTHIAS JARKE

Abstract—The maintenance of large information systems involves continuous modifications in response to evolving business conditions or changing user requirements. Based on evidence from a case study, we show that the systems maintenance activity would benefit greatly if the *process knowledge* reflecting the *teleology* of a design could be captured and used in order to reason about the consequences of changing conditions or requirements. We describe a formalism called REMAP (REpresentation and MAintenance of Process knowledge) that accumulates design process knowledge to manage systems evolution. To accomplish this, REMAP acquires and maintains dependencies among the design decisions made during a prototyping process, and is able to learn general domain-specific design rules on which such dependencies are based. This knowledge cannot only be applied to prototype refinement and systems maintenance, but can also support the reuse of existing design or software fragments to construct similar ones using analogical reasoning techniques.

Index Terms—REMAP, systems maintenance support.

I. Introduction

METHODS for the analysis and design of information systems are often effective in developing initial designs but rarely support the correction of design errors or changes in previous design choices due to changing requirements. As a result, changes in system design tend to be unprincipled, ad hoc, and error prone, failing to take cognizance of the *justifications* for previous design decisions. In this paper, we examine some of these shortcomings and present a knowledge based system architecture called REMAP that strives to alleviate these problems. REMAP supports an iterative design and maintenance process by preserving the knowledge involved in the initial and evolving design, and making use of this knowledge in analogous design situations.

The research that led to the REMAP architecture was stimulated by our study of a complex system development effort (several related systems with hundred-thousands of lines-of-code each). This study revealed several types of *process knowledge* that are instrumental in developing and maintaining such systems. First, the design process consists of a sequence of interdependent design decisions. The *dependencies* among decisions are typically based on application-specific justifications. In the case study, such justifications were frequently laid down on paper in design documents. While general domain-dependent *rules* typically underlie these justifications, these rules are seldom articulated explicitly by users or analysts. Second, when systems are developed in a piecemeal fashion following the prototyping idea, analysts apply *analogies* to transfer experience gained from one subsystem to "similar components" of another.

It is the purpose of this paper to demonstrate—by analyzing the evidence from our case study, by developing the REMAP architecture and by presenting the most crucial parts of its implementation—that the development and maintenance process would benefit if this knowledge about dependencies and the general bases for them could be accumulated in an appropriate form, and used to reason about subsequent design changes. Specifically, this paper argues that a knowledge based support tool for this must have the following architectural components:

1) a classification of application specific "concepts" into a taxonomy of design objects, and mechanisms for elaborating this structure as more knowledge is acquired by the system.

2) a representation for design dependencies and mechanisms for tracing repercussions of changes in design.

3) a learning mechanism for extracting general rules from dependencies, associated with a mechanism to check new design objects or dependencies for consistency with the rules.

4) an analogy based mechanism for detecting similarities among parts of similar subsystems. This mechansim should make use of the classification in the generalization hierarchy to draw analogies between systems parts.

We describe each of these components in terms of the specific feature of process knowledge that they deal with and how this knowledge is represented. In order to establish a sufficiently rich context for discussion, the examples are parts of the design that were actually developed in an oil company. For readability, these examples are only represented graphically as data flow diagrams at a high level of abstraction. However, as described in Section III of the paper, the internal knowledge representation of REMAP is object-oriented and can accommodate a wide range of practically useful languages for requirements analysis, system design, and programming.

The remainder of this paper is organized as follows.

Manuscript received August 30, 1985; revised August 29, 1986.

V. Dhar is with the Department of Information Systems, Graduate School of Business Administration, New York University, 624 Tisch Hall, 40 West 4th Street, New York, NY 10003.

M. Jarke is with the Department of Computer Science, University of Passau, Passau, West Germany.

IEEE Log Number 8718388.

Section II begins with detailed real-world examples that are used to show the need to maintain process knowledge and to identify different kinds of such knowledge. The REMAP architecture is presented in Section III. Section IV describes in detail the learning component as a central part of the architecture. Section V provides a discussion relating the model to previous work in systems analysis and artificial intelligence. We conclude with a summary of possible applications which may benefit from the REMAP approach.

II. Classification of Design Process Knowledge

In this section, examples from a case study in the oil industry are used to illustrate different forms of process knowledge. Four classes are identified: specific knowledge about design dependencies (at the level of *instances*), general knowledge about design rules, knowledge about the essentiality of conditions for certain design decisions, and knowledge about analogical properties between design situations.

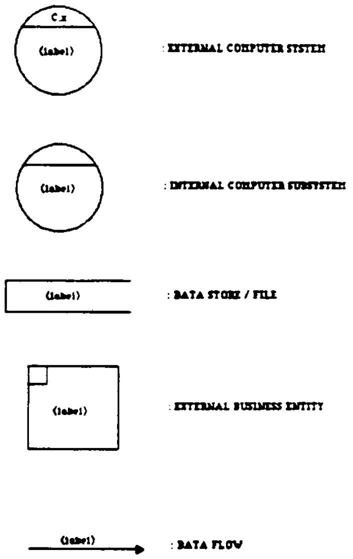
A. The Case Study

The problem studied in the oil company involves the desin and subsequent maintenance of a series of sales accounting systems for different products of the company, here referred to as OC. OC sells oil and natural gas-based products with different characteristics to its subsidiaries and to outside customers in different parts of the world. Sales Accounting at OC's Corporate Headquarters requires generating various integrated reports for purposes of audit and control. Input to Sales Accounting is based on invoices generated from transactions in a number of offices in the U.S. and abroad.

For the sake of readability, we describe systems using the structured analysis representation [9], [14]. However, the problems described in this section and our approach toward solving them are not confined to this representation.

In structured analysis, systems designs are described in terms of data flow diagrams at various levels of abstraction. A data flow diagram is a network where the nodes represent processes, external entities, or data stores (files), and directed arcs represent the data flows from one node to another. Process nodes are frequently called "bubbles"; each bubble can be decomposed into a lower-level data flow diagram. Bubbles at the bottom level have associated minispecs on which the program designs are based. Data flow and data store information is managed in data dictionaries. Fig. 1 shows the notational conventions used in this paper.

Part of the structured top-down design of OC's Sales subsystem is illustrated in Figs. 2–5. Fig. 2 shows a context diagram which depicts the relationship of the system to external entities. Figs. 3, 4, and 5 are data flow diagrams for levels 1 and 2 of the sales system. Further decomposition and implementation, possibly using different languages, would finally lead to a working system; however, the level of detail given in Figs. 2–5 is sufficient to describe the problems of systems maintenance and our solution to them.


C.x
(label)
: EXTERNAL COMPUTER SYSTEM
(label)
: INTERNAL COMPUTER SUBSYSTEM
(label)
: DATA STORE / FILE
(label)
: EXTERNAL BUSINESS ENTITY
(label)
: DATA FLOW
Fig. 1. Data flow diagram conventions.

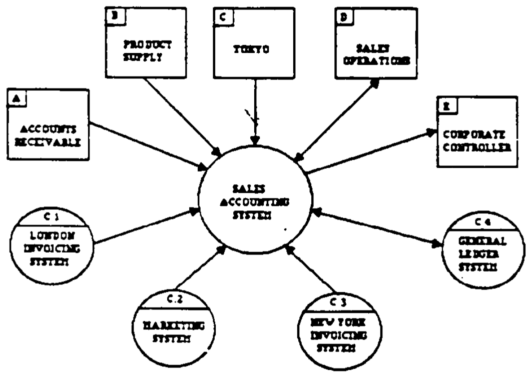


Fig. 2. Sales accounting systems context diagram.

We now illustrate the problem of design adaptation using three scenarios. Each requires a different extent of modification to the original design, and illustrates the need for a different aspect of process knowledge. All of the examples involve external requirements changes but similar problems also occur during the refinement cycle.

B. The Role of General and Specific Knowledge

London Sends Formatted Invoices: In the original design, the difference between the New York and London invoices was that the former were accessable *formatted* whereas the latter were received *unformatted*, on magnetic tape. Hence, a minor "convert" operation was required to bring the inputs into a format required by the "verify and correct on line" operation (bubble 1.1)

As a simple change, suppose that the London office be-

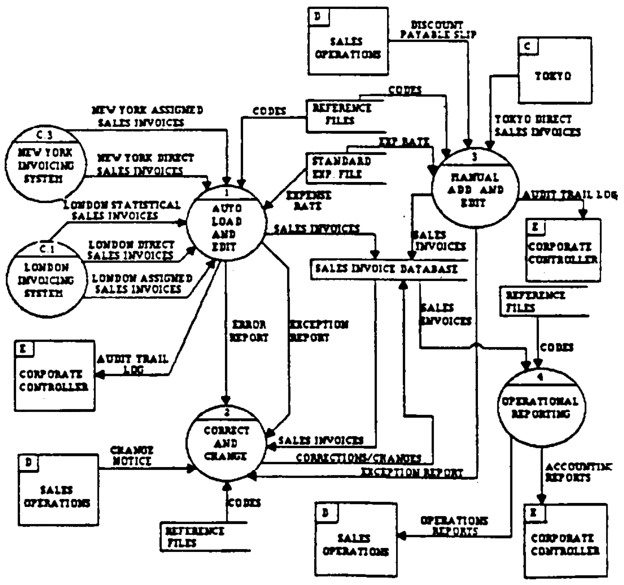


Fig. 3. Fuel sales (initial).

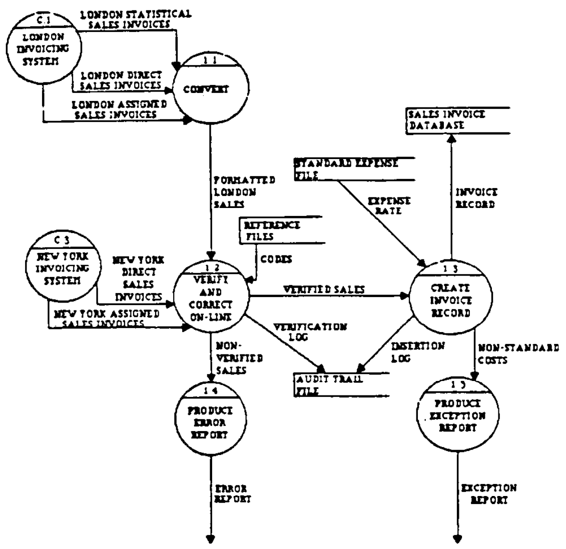

C 1 LONDON INVOICING SYSTEM
LONDON STATISTICAL SALES INVOICES
LONDON DIRECT SALES INVOICES
1 1 CONVERT
LONDON ASSIGNED SALES INVOICES
SALES INVOICE DATABASE
FORMATTED LONDON SALES
STANDARD EXPENSE FILE
EXPENSE RATE
INVOICE RECORD
REFERENCE FILES
CODES
C 3 NEW YORK INVOICING SYSTEM
NEW YORK DIRECT SALES INVOICES
1 2 VERIFY AND CORRECT ON-LINE
NEW YORK ASSIGNED SALES INVOICES
VERIFIED SALES
1 3 CREATE INVOICE RECORD
VERIFICATION LOG
NON-VERIFIED SALES
INSERTION LOG
NON-STANDARD COSTS
AUDIT TRAIL FILE
1 4 PRODUCE ERROR REPORT
1 5 PRODUCE EXCEPTION REPORT
ERROR REPORT
EXCEPTION REPORT

Fig. 4. Auto-load-and-edit.

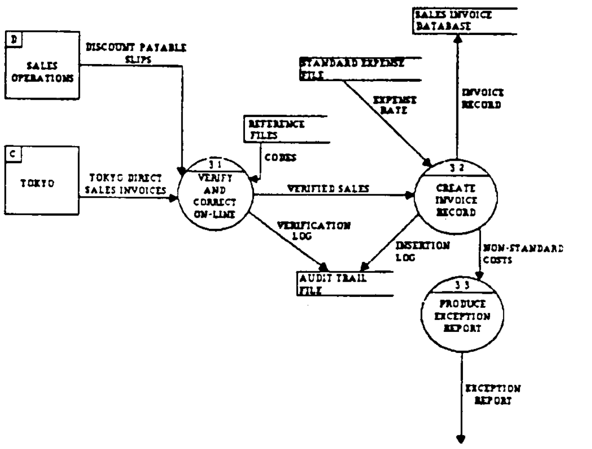


Fig. 5. Manual-add-and-edit.

gins to send correctly formatted invoices on magnetic tape to central headquarters. What kinds of design modifications are required?

It is clear that the change is not at a high enough level to affect the more abstract part of the design in Fig. 3. However, at the next lower level (Fig. 4), the "convert" bubble is not required anymore since the London invoices should now proceed directly for verification.

In order to be able to assimilate this minor change, a designer must know that in the existing design, the convert bubble is dependent on the nature (i.e., unformatted) of the dataflows representing London invoices. On recognizing that London invoices are no longer unformatted, it should be able to detect the fact that conversion is unnecessary. Further, he should also know that *in general*, formatted invoices proceed directly for on-line verification. Based on this, he should direct London invoices to the "verify and correct on line" operation.

In summary, we have used two types of knowledge in understanding the existing design and the effects of changes to it: *general knowledge* about domain-specific constraints (i.e., unformated invoices require conversion), and *specific knowledge* about the purpose of existing design objects in the form of justifications for existing design choices (i.e., the existence of the convert bubble in Fig. 4 depends on the existence of unformatted invoices).

C. The Role of Essentiality

London and Tokyo Will Not Sell Fuels Anymore: This represents a more radical type of change than the first. Intuitively, it seems clear that major design modifications are needed at several levels of analysis, design, and implementation. For example, lack of invoices from Tokyo obviates the need for a manual add and edit operation at level 1 (a *manual* input operation was required because these were *paper* invoices). However, the *auto* load and edit is still required because New York invoices must still be processed.

This example illustrates the idea of *essentiality* in design; the Tokyo invoices dataflow was an *essential* input for manual add and edit. In a more general sense, the *purpose* of a manual add and edit operation was to process paper invoices. The other inputs to it (the discount payable slips, codes and expenses) were *auxiliary*, and in fact *dependent* on Tokyo invoices.[1] In effect, bubble 1 stays (although some of its lower level components corresponding to London operations are removed), while bubble 3 must be deleted. The revised level 1 dataflow design is shown in Fig. 6.

It should also be noted that although the manual add and edit operation is no longer necessary, some of the lower level operations associated with it are still required in order to process New York invoices. At the programming level, this means that the code corresponding to those operations is not deleted since it is shared with the auto load and edit process.

D. The Role of Analogy

The Venezuela Office Will Sell Fuels: This corresponds to a high level change this is likely to induce widespread changes into the existing design. First, some additions must be made at level 1. The types of changes, however, depend on the nature of the sales invoices from Venezuela. If the invoices are computerized, an input into bubble 1 is required whereas paper invoices would call for introducing a manual add and edit operation. Similarly, at the next lower level, the operations required would depend on other, more detailed features of the invoices (i.e., are they formatted, unformatted, etc.).

This example illustrates the use of *analogy* in reasoning about a new situation. Design additions at the various levels depend on how "similar" the Venezuela invoices are to existing ones, and the design ramifications of these similarities and differences. This type of reasoning requires a system to carry out an elaborate match between design parts the system currently knows about, and a new design in order to draw out their analogous features. Specifically, it requires some notion of what are the *important* dimensions in the analogy being sought. In this example, relevant attributes in drawing the analogy are the *medium* of the invoices, that is, whether they are computerized or manual, and whether they are *formatted*. Once the important features are realized, the design ramifications become clear.

[1]This illustrates the "nonuniform" nature of dataflow diagram entities, that is, relationships among "unconnected" entities, and the design consequences that can emerge due to changes in them.

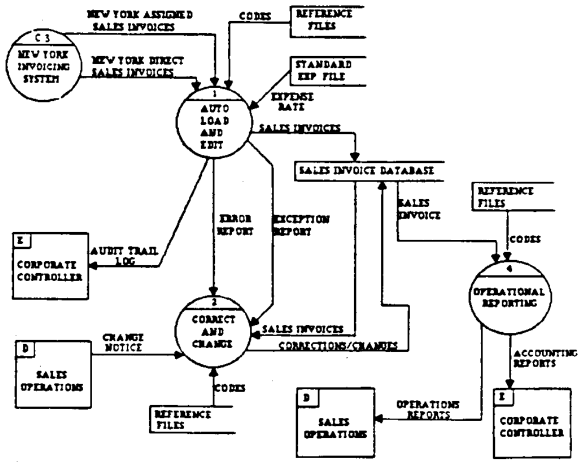


Fig. 6. Fuels sales (modified).

E. Summary: The Need for Teleological Knowledge

In walking through the examples, we have attached fairly rich interpretations to the various design components that are *implicit* in the design, i.e., not necessarily represented or even representable in structures such as data flow diagrams or any other purely outcome-oriented knowledge representations. These interpretations derive from the *purpose* of the application which cannot be determined from looking at the resulting design alone. That is, the design is an artifact [35] whose teleological structure is imposed by the *designer's* conception of the problem. This conception may change repeatedly during the evolutionary design process. In other words, there is no *a priori* "theory" relating problems to designs; rather, the justification for a particular design follows from a subjective world-view of the designer.

If a support system is to be able to reason about the types of changes illustrated in the examples, it must have the knowledge that reflects the teleology of the design. Because such highly contextual knowledge about a potential application area is impossible to design into a system *a priori*, the knowledge must be *acquired* by the supporting system *during* system design. To do this, the program must be equipped with mechanisms that enable it to learn about design decisions in an application area that it knows nothing about at the start of the design. It must then apply this growing body of acquired knowledge to reason about subsequent modifications to an existing design, or to construct new designs based on new but similar requirements. In the following section, we describe an architecture called REMAP that is geared toward the extraction and management of the process knowlege involved in systems development and maintenance.

III. The REMAP Architecture

It is apparent from the examples that application-specific knowledge and experience plays a key role in reasoning about a design. This raises an important question, namely, how can a *system* acquire such knowledge?

In most projects involving the construction of a knowledge based system, the system builder constructs the model of expertise by first specifying a representation, and then accreting the knowledge base in accordance with the precepts underlying the chosen representation. Unfortunately, large scale application developments take place in a wide variety of domains that may have little in common. This uniqueness of each application situation discourages construction of a knowledge base that might be valid for a reasonable range of applications.

If a knowledge based system is to be able to support the process of systems analysis and design, it must have an initial representational framework, and mechanisms to augment this framework with domain specific knowledge that captures the purpose of design decisions and relationships among them. As more is learned, it should be possible to use this process knowledge to reason about design changes, and draw analogies in extending a design to deal with new situations.

In the following subsections, we develop a knowledge representation for this process knowledge, and present a model of how it is used by the REMAP system architecture. Each of the components of this architecture illustrates the use of a certain type of process knowledge. We conclude the section by illustrating how these components interact through a global control structure. A detailed example of the most important subsystem within the architecture—the learning component—is presented in Section IV.

A. Representing Design Outcomes Using Structured Objects

The REMAP model centers around *design objects*. The designer defines *instances* of such objects, and the REMAP system maintains a *generalization hierarchy* of object *types*. The structure of an object type definition in the hierarchy is as follows:

```
OBJECT TYPE
   type_name : <string>
   child_of : <set of object types>
   parent_of : <set of object types>
   components: <set of slots>
   operators : <set of procedures/methods>
```

The "child-of" and "parent-of" components position an object type in the generalization hierarchy. "Componets" slots describe typical aspects of an object instance of the given type. As an example, consider the initial top-level definition of a generic object type.

```
OBJECT TYPE
   type_name : generic_object
   child_of : ( )
   parent_of : unknown
   components: (identifier : <string>
      type      : <string>
      because_of : < set of objects>)
   operators : (define, remove)
```

This means that any object will have an identifier, a type, and a "because-of" slot. The generic object type has no parent, and its children are yet to be specified. The "because-of" slot defines the *raison d'etre* of an object instance and will be further discussed in the next subsection.

A "generic" object provides very little structural information about its semantics. It is therefore useful to *specify subtypes* for which additional slots are defined in order to capture the meaning of object instances of such a subtype. This can be represented using a generalization hierarchy of object types as shown in Fig. 7. Some instances of dataflows and transforms used in the three scenarios of Section II are shown in Fig. 8.

In principle, the system could begin with the generic object type and then learn all subtypes from scratch. Since such a procedure would be rather cumbersome for the designer, the system should be provided with an initial set of object types useful for a broad range of domains, for instance, those associated with the analysis, design, and implementation languages in use. For example, if the designer were to work with data flow diagrams, the initial knowledge base of object types might contain the following definitions (cf. Fig. 7):

```
OBJECT TYPE
   type_name : dataflow
   child_of : generic_object
   parent_of : unknown
   components: (part_of : dataflow;
      medium : <string>;
      from, to : process)
   operators : (redirect, nostart, noend)

OBJECT TYPE
   type_name : transform
   child_of : generic object
   parent_of : (process, external, datastore)
   components: (inputs, outputs : <set of dataflows>)
   operators : ( )

OBJECT TYPE
   type_name : process
   child_of : transform
   parent_of : unknown
   components: (part_of : process)
   operators : (expand, noinput, nooutput)

OBJECT TYPE
   type_name : datastore
   child_of : transform
   parent_of : unknown
   components: (data_structure :
     <set of data elements>)
   operators : (define_structure, noinput, nooutput)
```

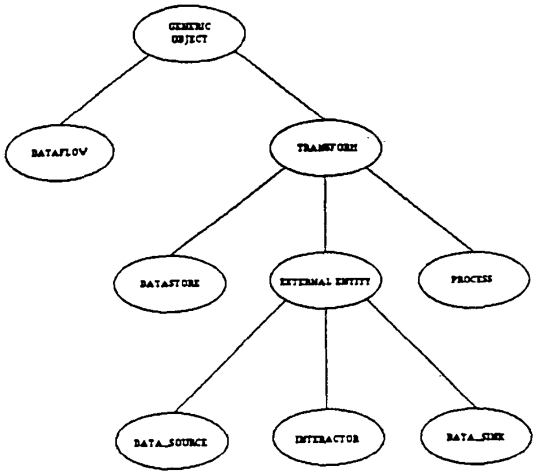


Fig. 7. Initial object type hierarchies.

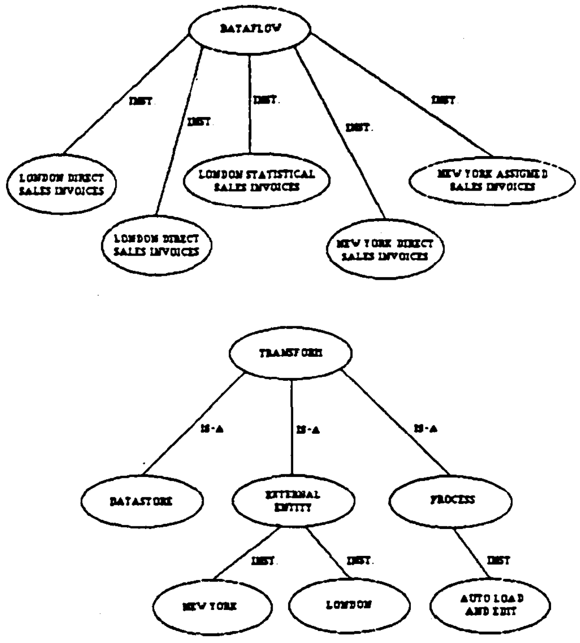

DATAFLOW
INST.
INST.
INST.
INST.
INST.
LONDON DIRECT SALES INVOICES
LONDON STATISTICAL SALES INVOICES
NEW YORK ASSIGNED SALES INVOICES
LONDON DIRECT SALES INVOICES
NEW YORK DIRECT SALES INVOICES
TRANSFORM
IS-A
IS-A
IS-A
DATASTORE
EXTERNAL ENTITY
PROCESS
INST.
INST.
INST
NEW YORK
LONDON
AUTO LOAD AND EDIT

Fig. 8. Initial generalization hierarchy.

```
OBJECT TYPE
   type_name : external_entity
   child_of : transform
   parent_of : unknown
   components: ( )
   operators : ( )
```

External entities could be further refined to data source, data sink, and interactor. The slot value "unknown" refers to the fact that the slot values should be, but have not yet been, defined.

As an example of *instance definitions*, consider the following description of the "London" external entity and one of the sales invoice dataflows generated by it (cf. Fig. 8).

```
{identifier : London
type      : external_entity
because_of : ( )
inputs   : ( )
outputs   : (London-direct-sales-invoices,
             London-assigned-sales-invoices,
             London-statistical-sales-invoices)

{identifier : London-direct-sales-invoices
type      : dataflow
because_of : (London)
part_of   : ( )
medium      : magnetic tape
from      : London
to     : auto-load-and-edit}
```

Similarly, instances corresponding to other object types can be defined. Note, that the instance definitions have all the slots defined in their immediate type, as well as inheriting those of their supertypes.

Besides the definition of design objects, it is also possible to perform "syntactic" consistency checks using information in the hierarchy. As a simple example, if a bubble has no inputs, it must be removed or new inputs must be defined. However, certain types of application-specific information are not maintained in this representation. For instance, if London invoices become "formatted," ramifications of this change cannot be assessed using the knowledge in the hierarchy alone. To reason about such situations, additional data structures are required, which we describe in the following subsections.

B. Representing Design Processing Using Dependencies

REMAP views a design process as a set of interrelated *design decisions*. Design decisions are represented in terms of *justified actions*. An action consists of adding, deleting or changing a design object; its justification consists of previous actions. A design decision is represented in REMAP as a two-part data structure called *dependency*:

$$(<\text{justification}> \;==>\; <\text{action}>)$$

where <justification> and <action> are references to object instances.

To illustrate, consider Fig. 9 which shows a network of dependencies among a few of the dataflows and bubbles considered so far. Specifically, the auto-load-and-edit object is justified by the existence of New York and London invoices (both objects), which form its "set of support" [12].

In order to demonstrate the usefulness of this dependency network, reconsider the first scenario where the London invoices become formatted. In this case, the convert operation is no longer required since its *essential* support elements have been eliminated. Similarly, in the second scenario where the London office does not sell fuels anymore, no more invoices are generated from London. Again, no conversion operation is required. However, the

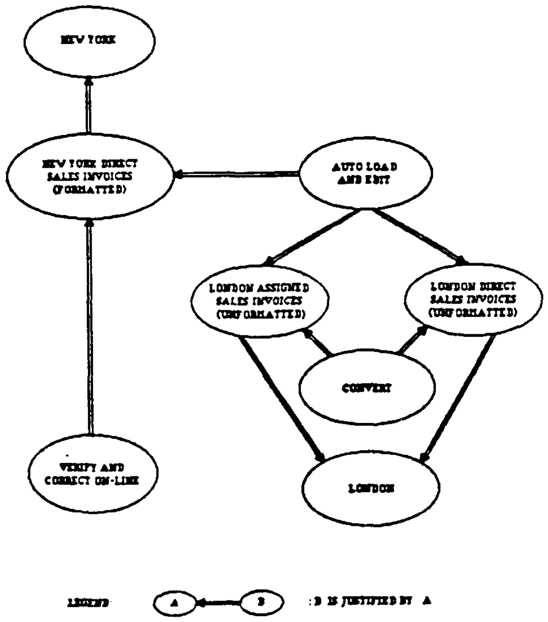

NEW YORK
NEW YORK DIRECT SALES INVOICES (FORMATTED)
AUTO LOAD AND EDIT
LONDON ASSIGNED SALES INVOICES (UNFORMATTED)
LONDON DIRECT SALES INVOICES (UNFORMATTED)
CONVERT
VERIFY AND CORRECT ON-LINE
LONDON
LEGEND
A
B
: B IS JUSTIFIED BY A

Fig. 9. A dependency network.

auto load and edit operation is still required because New York invoices are still to be processed.

In general, a dependency network can be used to assess certain ramifications of a deletion or change in previous design decisions. Such processes are commonly referred to as *belief maintenance* [12]. In the above example, conversion is *not* required for London invoices. However, the dependency network does not indicate how these invoices *should* be treated because this knowledge is not expressed in the network. In order to assess the complete repercussions of the change, more general (object type level) knowledge is required. For example, to realize that formatted London invoices should be treated like New York invoices (and should proceed directly for verification), it is necessary to know that *in general* formatted invoices are verified directly. This knowledge can then be used to reason about all object instances corresponding to formatted invoices.

C. Learning as Rule Formation

Dependency information as indicated in Fig. 9 is represented in terms of *object instances*. For example, the auto-load-and-edit object (bubble 1) is justified by the two kinds of dataflow objects originating from London. An object type corresponding to this invoice dataflow might have slots such as data, amount, frequency, and source. However, not all slots are relevant to the justification. For example, the auto-load-and-edit is performed because the invoices are computerized, regardless of their other features. A general rule that subsumes this dependency would therefore state that computerized invoices require auto-load-and-edit. It is the purpose of REMAP's learning component to acquire such rules.

In forming a rule, however, the system must first learn the relevant category of object types (i.e., computerized invoices) that will constitute the left-hand side of the rule. If we consider ''dataflow'' as being a generic object with the structure described earlier, what the system must do is to form a specialization of it, where the specialization involves restricting the value of one or more slots of the generic object. For example, a computerized invoice can be considered a specialization of the dataflow object with the medium slot being restricted to values that belong to the set ''computerized entities'' like disk or magnetic tape.

Basically, the learning procedure views each dependency (stated in terms of object instances) as a *training instance* consisting of a situation object and an action object. Each training instance has an associated *hypothesis space* which consists of possible generalizations of the situation object. A training instance is termed *positive* with respect to its action object, and *negative* with respect to all others. As more and more examples (i.e., dependencies) are provided in the course of a systems development process, irrelevant elements of the various hypothesis spaces are eliminated and the system converges on generalizations (i.e., type definitions and rules) that are consistent with the examples. If a hypothesis space shrinks to the point where no generalizations can be found, this indicates inconsistencies in the design or in the design rule base and must be corrected by the user. In order to accelerate convergence of the hypothesis space, REMAP can provide system-generated examples for categorization as positive or negative training instances by the user. The learning procedure is described in detail in Section IV.

To summarize, the learning objective is twofold: to form appropriate specializations of the predefined object types relevant to the application domain, and to establish relationships in the form of rules between these specialized object types. This results in a growing generalization hierarchy such as that of Fig. 10, and in rules that are applicable at various levels of abstraction.

D. Analogical Reasoning Using Object Classification and Rules

The effort of learning a flexible object type hierarchy and general design rules associated with it pays off in two ways. First, types and rules can be used to check the correctness of new design object instances added to a design. The second advantage is less obvious but potentially more important. When requirements changes demand the construction of new design objects in addition to the removal of existing ones, *analogical reasoning* methods can be employed to explore the possibility of reusing fragments of existing designs, based on the general knowledge acquired by REMAP's learning component.

For example, Section II-D introduced a scenario where a new operation was added, namely, sales of fuels from Venezuela. In order to assimilate such a change into an existing design, a system must be able to utilize its knowl-

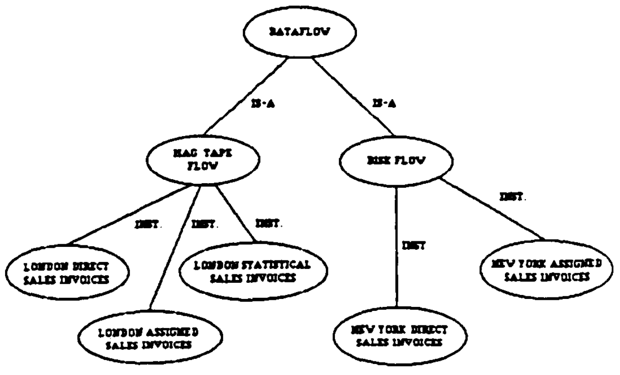

DATAFLOW
IS-A
IS-A
MAG TAPE FLOW
DISK FLOW
INST.
INST.
INST.
INST
INST.
LONDON DIRECT SALES INVOICES
LONDON STATISTICAL SALES INVOICES
LONDON ASSIGNED SALES INVOICES
NEW YORK DIRECT SALES INVOICES
NEW YORK ASSIGNED SALES INVOICES

Fig. 10. Reconfigured generalization hierarchy.

edge concerning the purpose of "similar" design fragments. Specifically, it must determine what attributes of the new situation are the same as objects it already knows about, and then treat the new object accordingly.

In order to categorize a new object, it is necessary to first determine, if possible, the most specific level of abstraction in the generalization hierarchy that is applicable to it. For example, if REMAP's current knowledge about dataflows is that shown in Fig. 10, and computerized but unformated invoices come in on magnetic tape from Venezuela, they are classified as an instance of the magnetic-tape-invoices type. Rules referencing this type can be applied to it in order to create new object instances automatically.

If no rules are applicable to the newly defined object as the most specific level, *more general* rules might be applicable. This involves moving up the generalization hierarchy as long as applicable rules are found. In the example, this involves gathering rules applicable to magnetic-tape invoices, then computerized invoices, and finally dataflows in general. For Venezuela invoices, we can see that one of the rules mentioned in the previous section will apply at the level of computerized invoices, suggesting that the existing auto-load-and-edit operation (or a new instance of it) be performed on them.

It should be noted that even though there may *not* be an object in the current design that is similar to the new one, existing rules learned during previous design processes might still apply. For example, London invoices had been originally unformatted; this had required a convert operation which was subsequently eliminated when the form of these invoices was changed. However, since a rule on formatted versus unformatted invoices was retained which now becomes applicable to Venezuela invoices, the old convert operation could be reinstalled, or a similar one implemented if the formatting differs at a lower level of abstraction than shown in our examples.

E. REMAP Control Structure

In order to incorporate new knowledge and to reason about user critiques, REMAP requires an overall control structure that enables it to switch among design support

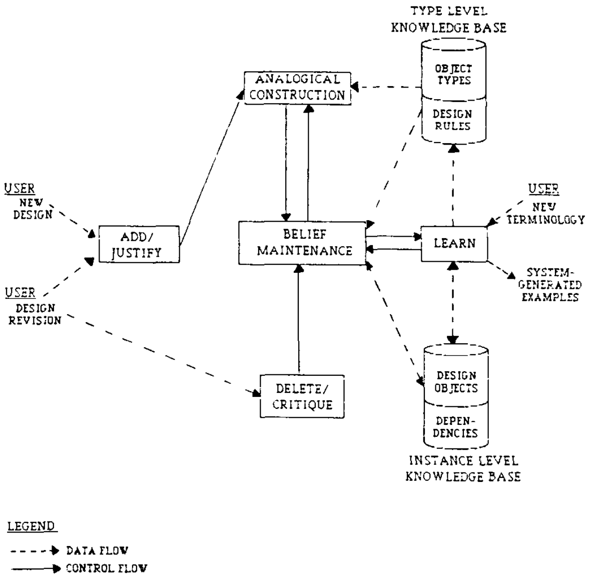


Fig. 11. Summary of REMAP architecture.

and knowledge acquisition modes. Fig. 11 provides an architectural summary of the system. The architecture consists of five modes of operation and two knowledge bases. One knowledge base describes the design objects and dependencies at the instance level, whereas the other one is a meta-knowledge base which contains the object type hierarchy and the general design rules. We shall first describe the functionality of the architecture for two typical scenarios and then present a semiformal summary of the interaction of the modes in a structured-English notation.

Consider first a scenario where the user wants to add a new design object. The *add* mode accepts a design object and its associated justification (i.e., a dependency plus possibly a detailed description of the design object). The *analogical reasoning* mode assists first in identifying the type of objects. It then tries to apply design rules to generate additional objects dependent on the one entered by the user. If the system has accumulated knowledge about the application domain, rule application might continue down to the implementation level. For each action, the *belief maintenance* mode is responsible for entering objects and dependencies to the instance-level knowledge base. If existing rules are not applicable to the new objects, the *learning* mode assumes control and attempts to form a generalization (rule) from the dependency (this is described in detail in Section IV). The learning model also comes into play if a contradiction is encountered, in which case it initiates interaction with the user in order to correct the object instances, or to establish new rules and, if necessary, specify new object types. The system then returns to the belief maintenance mode in order to do the required changes at the instance level and to trace the consequences of the newly acquired knowledge, returning control to the analogical-construction-mode.

If parts of an existing design are to be removed, the system will start in the *critique* mode. In this case, the belief maintenance mode is responsible for tracing which dependent objects can also be removed from the design, by following the chains of dependencies in the instance-level knowledge base. Updates to a given design object can be considered as deletions followed by additions of the new version.

We now give a high-level summary of the algorithms underlying each mode. We should point out, however, that the learning mode description will be more understandable after reading Section IV, which is a walk-through of the algorithm using a detailed example.

```
Add-mode:
1. Accept object instance i and its justification object
   j.
2. Call Analogical-construction-mode (i, j).

Analogical-construction-mode (inst, just):
1. Position inst and just in type hierarchies, finding
   types ti and tj.
2. Call Belief-maintenance-mode (''add'', inst, just,
   ti, tj).
3. FOR each rule r of form ''ti => x'' or ''tp =>
   x''
     where ti is a subtype of tp DO
     IF an object instance corresponding to x does not
     exist
            THEN create object x.
     Call Analogical-construction-mode (x, inst).

Delete/critique-mode:
1. Accept object o to be removed.
2. Call Belief-maintenance-mode (''del'', o, nil, nil,
   nil).

Belief-maintenance-mode (op, inst, just, ti, tj):
1. IF op= ''del''
     THEN IF just = { }
            THEN Remove inst from each set of support.
               \* Note that the description of inst is not
               removed *\
            FOR EACH object obj with empty set of
            support DO
            Call Belief-maintenance-mode (''del'', obj,
            nil, nil, nil).
     ELSE \*op = ''add'' *\
        Add just => ist to the dependency base.
        Add the description of inst to the design object
        base.
        Call Learn-mode (just, inst, ti, tj).

Learn-mode (i, j, ti, tj):
1. FOR EACH rule tj => x where x incompatible with
   i,[2]
        Request correction by user.
2. IF there exists a dependency k ==> i \* positive
   training instance *\
     THEN IF new slots
            THEN Establish new terminology with
            user.
   FOR EACH x ==> y
        IF ti = type of x     \* negative training in-
        stance *\
          and i incompatible with y
          THEN Reduce hypothesis spaces for i and
          y.
3. Provide system-generated examples for further type
   refinement.
```

IV. Synthesizing the Generalization Hierarchy

Inferring plausible object types and rules from design decisions (dependencies) can be considered a learning task.[3] It involves generalizing situations (the left-hand side of the instance level dependency) into subtypes on which design decisions (the right-hand side) might be based. For example, if sales invoices coming from London are computerized (a situation) and are processed directly by computer (a decision), a plausible generalization is that computerized invoices in general can be processed by computer. It therefore makes sense to create a category called ''computerized invoices'' and a general rule stating that computerized invoices are to be processed directly. These two types of knowledge can then be used to recognize new instances of such invoices, and how they are to be processed. The problem of course, is to distinguish among the important and the incidental attributes of the situation.

Our approach to forming general descriptions is based on the construction of a structured hypothesis space (a lattice data structure) for each decision. This space contains possible generalizations of situations for each decision. These generalizations are gradually eliminated or refined with successive examples. For a design expressing

[2]A design object is called incompatible with another one if both constitute alternative actions for the same situation. Without loss of generality, actions that are not equal can always be considered incompatible if the right level of abstraction is chosen.

[3]We would like to acknowledge the significant input of Padmanbhan Ranganathan in developing the learning strategies presented in this section. These strategies are described in more detail in [11].

many situation-action pairs, the ultimate goal is to synthesize a taxonomy of appropriate situation descriptions, each corresponding to a decision expressed in the design. Specifically, the aim is to synthesize a generalization hierarchy of concepts relevant to the application domain that contains general situation descriptions on which the design decisions are based.

Formally, a situation is characterized in terms of an instance d_i of one of the object types in the existing hierarchy described as in Section III-A. This object type, hence called D, has slots $s_1, s_2, s_3, \cdots, s_p$. An instance d_i consists of the set of *pairs* of properties $\{ s_j : V_{i,j} \}$ where V_{ij} is the value of the jth slot. An operator that is applicable to this situation is represented as t_k. In the application domain, $d_i ==> t_k$ represents a design decision to perform t_k in the situations described as d_i. If this first example is followed by the example "$d_j ==> t_k$", this example represents a positive training instance for t_k whereas the example $d_j ==> t_l$ represents a negative training instance for t_k. The learning goal is to converge on those properties of examples that are, by themselves or in combination, relevant to the design decisions, and to acquire the necessary terminology interactively.

A. Designer Generated Examples

To introduce the learning model, consider some design decisions made by a systems designer/analyst from the sales accounting system. To keep the example clear, we restrict the description of object type D (a special kind of data flow) to four of the slots, namely, "from," "medium," "priority," and "frequency." The first example, designated E_1, corresponding to a small design fragment from Fig. 3, is:

```
E1 =
  {d1
    from: London
    medium: magtape        ==> Auto-load-and-edit
    priority: high
    frequency: daily}
```

where Auto-load-and-edit is an action performed on a dataflow characterized by the left-hand side. The set {from:London, medium:magtape, priority:high, frequency:daily} represents the situation d_1. The operator t_1 that is applicable to d_1 is Auto-load-and-edit. Based on this example alone, the following possibilities arise:

1) All pairs of d_1 are relevant in deciding on t_1.

2) Only some combination of the pairs are relevant to t_1.

3) All pairs of d_1 are merely incidental, that is, t_1 is performed on *all* instances of D regardless of their properties.[4]

A representation of the possibilities, the hypothesis space of all possible rules based on the first example, is shown in Fig. 12. A question mark indicates that there is no restriction on the slot value. The figure represents a hypothesis space for t_1, extending from the most specific hypothesis, at level 0, down to the most general one at level 4.

[4]In this section, we ignore the case that a *new* slot might be necessary to distinguish object subtypes. This case would simply be handled by user intervention.

It is worth contrasting such a hypothesis space with those that are constructed *using* an *a priori* taxonomy of object types such as is done in the learning system LEX [24] where nodes represent situations characterized in terms of the types in the *existing* taxonomy. We interpret our hypothesis space in the same way, as consisting of object types. The difference is that these types are *implicit* in our hypothesis space and need to be characterized explicitly. Specifically, the nodes contain specializations of D, that is, subtypes with restrictions on values of certain slots. In our example, nodes at level 1 are those where values of any three slots have restricted values and the fourth slot can take any value. Similarly, level 4 consists of the most general object type, where values of all 4 slots are unrestricted. In effect, each of the nodes in the hypothesis space is a specialization of D, corresponding to a particular object type. The generalization hierarchy corresponding to this hypothesis space is shown in Fig. 13. In summary, an initial hypothesis space generates a crude object taxonomy. As the space is refined, so is the taxonomy.

Now another example, again representing a design decision, is presented.

```
E2 =
  {d2
    from: London
    medium: disk     ==> Auto-load-and-edit
    priority: high   freq:daily }
```

Comparison to E_1 shows that only the value of the "medium" slot is different. The second example calls for the same right hand side and is therefore a positive training instance with respect to E_1. The fact that both left-hand sides, which represent slightly different situations, have the same right-hand side leads to the following possibilities:

1) The values of the "medium" slot are irrelevant in determining which operator is to be applied, since changing them made no difference to the action to be performed.

2) Alternatively, the values may in fact be essential, if they belong to some generic category which requires performing t_1. For example, "magtape" and "disk" could both belong to a "superclass" called "computerized" which could be what requires t_1. This situation requires creating a new term, in this case *computerized*, that will characterize the new superclass. However since the system has no domain knowledge for generating this type of vocabulary, the system must query the user. If the user responds with "computerized," the system asks the user to enumerate or characterize other members belonging to

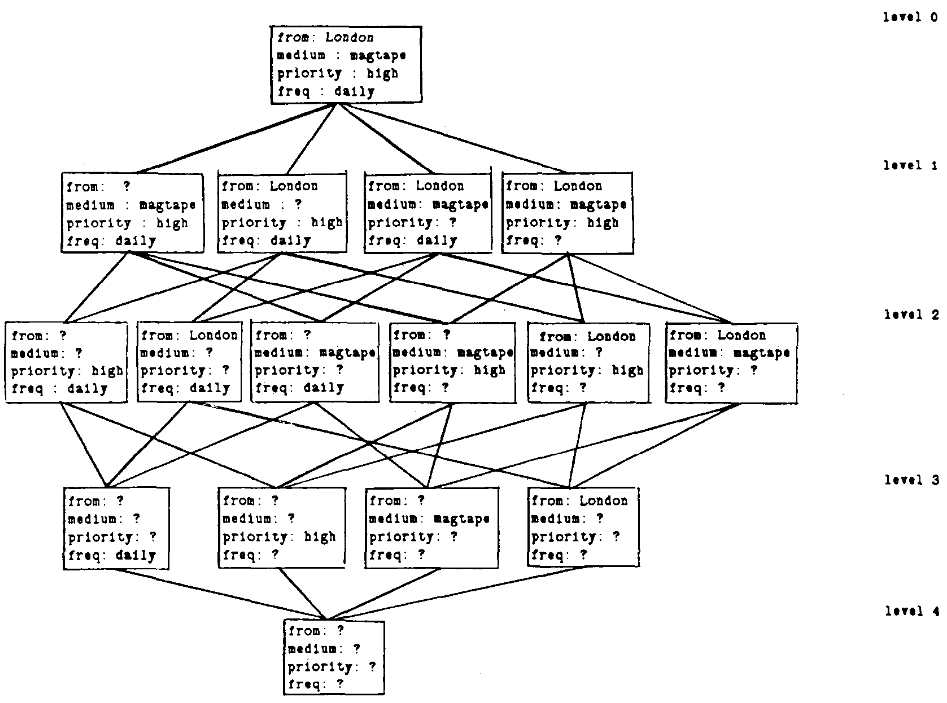


Fig. 12. Hypothesis space for auto-load-and-edit (t_1) after E_1.

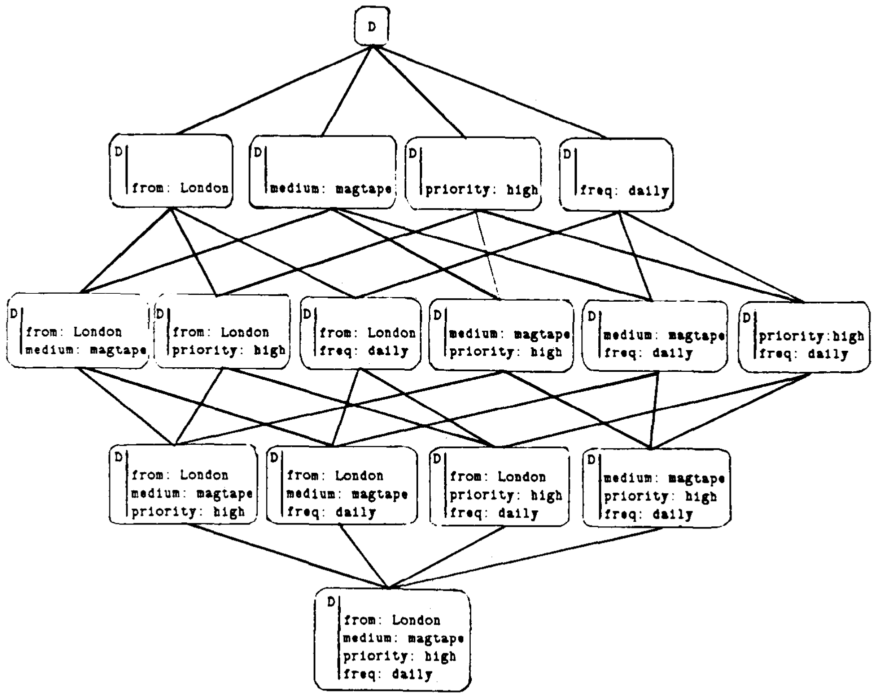

D
D from: London
D medium: magtape
D priority: high
D freq: daily
D from: London medium: magtape
D from: London priority: high
D from: London freq: daily
D medium: magtape priority: high
D medium: magtape freq: daily
D priority:high freq: daily
D from: London medium: magtape priority: high
D from: London medium: magtape freq: daily
D from: London priority: high freq: daily
D medium: magtape priority: high freq: daily
D from: London medium: magtape priority: high freq: daily

Fig. 13. Generalization hierarchy after E_1. Nodes in the hierarchy are specializations of D where slot and value pairs on the right of the vertical bar indicate restrictions on an object type. The lines joining the nodes are IS-A Links.

this class. This information can be used to recognize other instances of the new class.

Both these possibilities are represented in the hypothesis space. In the second case, certain nodes in the hypothesis space are generated to accommodate the information in the positive training instance. This is the well known *disjunctive problem* which occurs in generalization from examples.

The hypothesis space for t_1, shown in Fig. 12, is now refined to reflect these modifications. We have replaced "magtape" by "computerized" in the relevant slots. This change reflects a modification of the object types in the hypothesis space as shown in Fig. 14. The generalization hierarchy is reorganized accordingly to incorporate the modified object type.

Now consider a third example:

```
E3
  {d3
     from: Tokyo
     medium: paper   ==> manual-add-and-edit
     priority: high
     freq: daily }
```

This instance is a negative example with respect to the decision auto-load-and-edit. Comparison of this new training instance with E_1 and E_2 reveals the following:

1) The values of slots "priority"and "freq" are the same in all three instances. This implies that the "priority" and "freq" pairs do not, by themselves or in combination, discriminate in deciding which operator should be applied.

2) The values of the slots "from" and "freq" could, in conjunction with values of other slots, provide the justification for Manual-add-and-edit (t_2).

In the light of the evidence from the third example, it is apparent that object types corresponding to

```
D|                 D|             D|
 |priority: high    |freq : daily  |priority: high
                                   |freq: daily
```

do not discriminate among the examples, and can therefore be eliminated from the two hypothesis spaces so far. The nodes corresponding to these tyes are indicated in the dotted section of Fig. 14. In the refined hypothesis spaces of auto-load-and-edit and manual-add-and-edit (Fig. 15) these nodes are marked as eliminated.

The generalization hierarchy, reflecting the refined hypothesis spaces is also modified to that shown in Fig. 16. It represents a union of the two hypothesis spaces.

As a final example, consider the following:

```
E4=
  {d4
    from: Tokyo
    medium: paper   ==> Manual-add-and-edit
    priority: high
    freq: weekly }
```

In comparing this example to E_3 we find that only the value of the "freq" slot is different. As in the second example, this results in the possibility that the two values "daily" and "weekly" belong to some superclass. Accordingly, the hypothesis-space for manual-add-and-edit is augmented to reflect this possibility, and the corresponding changes are induced in the generalization hierarchy. Finally, this is a negative instance with respect to the hypothesis space for t_1. In this case, it has no effect on the hypothesis space of t_1.

To summarize, the concept formation problem described above has the following features. An example, reflecting a design decision, leads to the construction of a lattice structure called a hypothesis space which is interpreted as a partial order of plausible concepts that account for the decision. Subsequent examples refine the hypothesis space. Specifically, positive instances suggest higher order concepts which result in an expansion of the taxonomy of objects. Negative instances are used to eliminate concepts previously hypothesized to differentiate between design decisions. In this way the taxonomy of objects is refined, with the expectation that the irrelevant concepts will be eliminated as plausible differentiators, enabling the system to converge on rules at the approximate level of generality.

B. System-Generated Examples

Like other learning formalisms that generalize from examples, the effectiveness of our model is sensitive to the nature of the examples. If provided with "good" examples, the model converges quickly on the right hypothesis for a decision; for our problem, the best discriminatory power results from examples where situations varying only in a few attribute values require different decisions (the negative instances). However, in general, the strategy above cannot guarantee that the system will converge on the most appropriate hypothesis in each hypothesis space based on design observations alone.

One way for the system to overcome total reliance on the designer's examples is to *generate* additional examples that will help it discriminate among competing hypotheses in each space. Since the real discriminating power is provided by negative instances, it makes sense to try and generate descriptions that will prove to be negative instances in the various hypothesis spaces. To illustrate, consider Fig. 14 where there are several competing hypotheses for Auto-Load-and Edit. Suppose the sytem wants to establish the node marked "X" as the correct hypothesis for Auto-Load-and-Edit (reasons for why X are explained shortly). To generate a negative example, the system picks the "corresponding node" (marked "Y" in Fig. 15) from another hypothesis space. The system thus generates the example, posed as a query to the user:

```
For { dataflow
    from: ?
    medium: paper
    priority: ?
    freq: ?  }
Will you do Auto-Load-and-Edit?
```

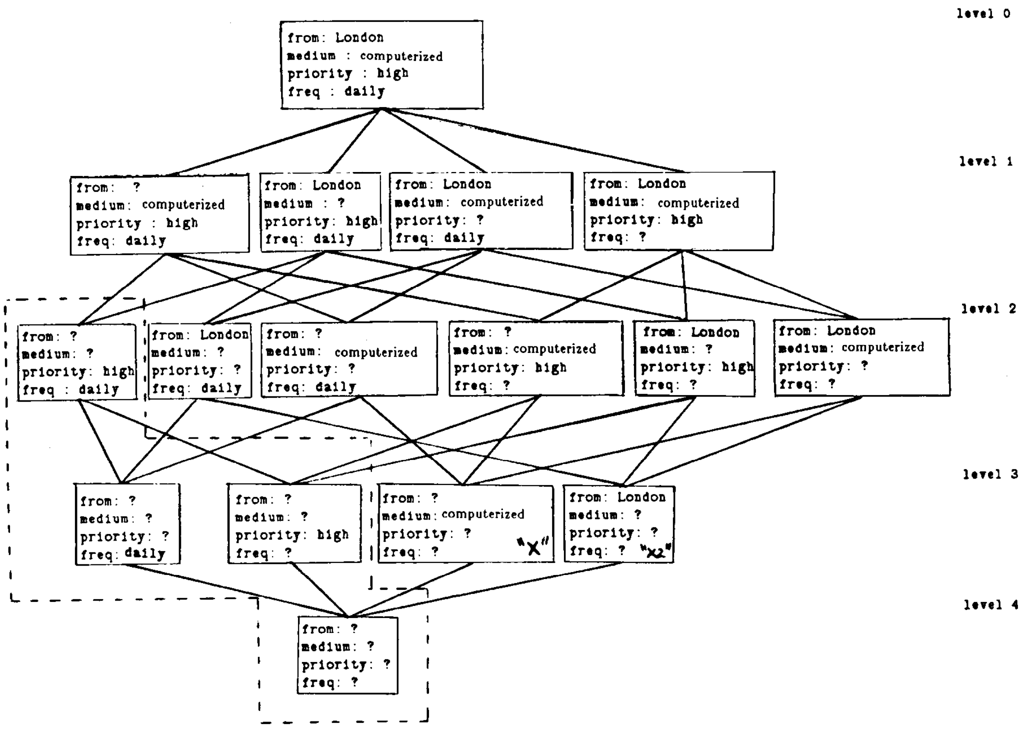

level 0
from: London medium : computerized priority : high freq : daily
level 1
from: ? medium: computerized priority : high freq: daily
from: London medium : ? priority: high freq: daily
from: London medium: computerized priority: ? freq: daily
from: London medium: computerized priority: high freq: ?
level 2
from: ? medium: ? priority: high freq : daily
from: London medium: ? priority: ? freq: daily
from: ? medium: computerized priority: ? freq: daily
from: ? medium: computerized priority: high freq: ?
from: London medium: ? priority: high freq: ?
from: London medium: computerized priority: ? freq: ?
level 3
from: ? medium: ? priority: ? freq: daily
from: ? medium: ? priority: high freq: ?
from: ? medium: computerized priority: ? freq: ? "X"
from: London medium: ? priority: ? freq: ? "X2"
level 4
from: ? medium: ? priority: ? freq: ?

Fig. 14. Hypothesis space for auto-load-and-edit after E_2.

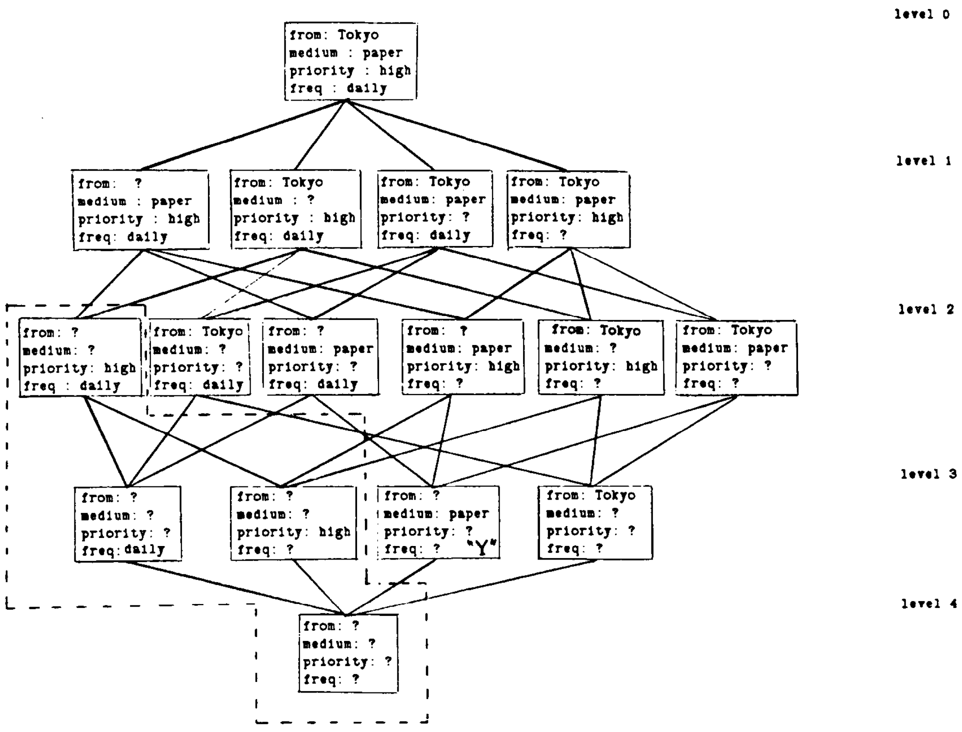

level 0
from: Tokyo medium : paper priority : high freq : daily
level 1
from: ? medium : paper priority : high freq: daily
from: Tokyo medium : ? priority : high freq: daily
from: Tokyo medium: paper priority: ? freq: daily
from: Tokyo medium: paper priority: high freq: ?
level 2
from: ? medium: ? priority: high freq : daily
from: Tokyo medium: ? priority: ? freq: daily
from: ? medium: paper priority: ? freq: daily
from: ? medium: paper priority: high freq: ?
from: Tokyo medium: ? priority: high freq: ?
from: Tokyo medium: paper priority: ? freq: ?
level 3
from: ? medium: ? priority: ? freq: daily
from: ? medium: ? priority: high freq: ?
from: ? medium: paper priority: ? freq: ? "Y"
from: Tokyo medium: ? priority: ? freq: ?
level 4
from: ? medium: ? priority: ? freq: ?

Fig. 15. Hypothesis space for manual-add-and-edit (t_2) after E_3. Comparison of this hypothesis space with that of t_1 leads to the removal of the dotted area from both hypothesis spaces.

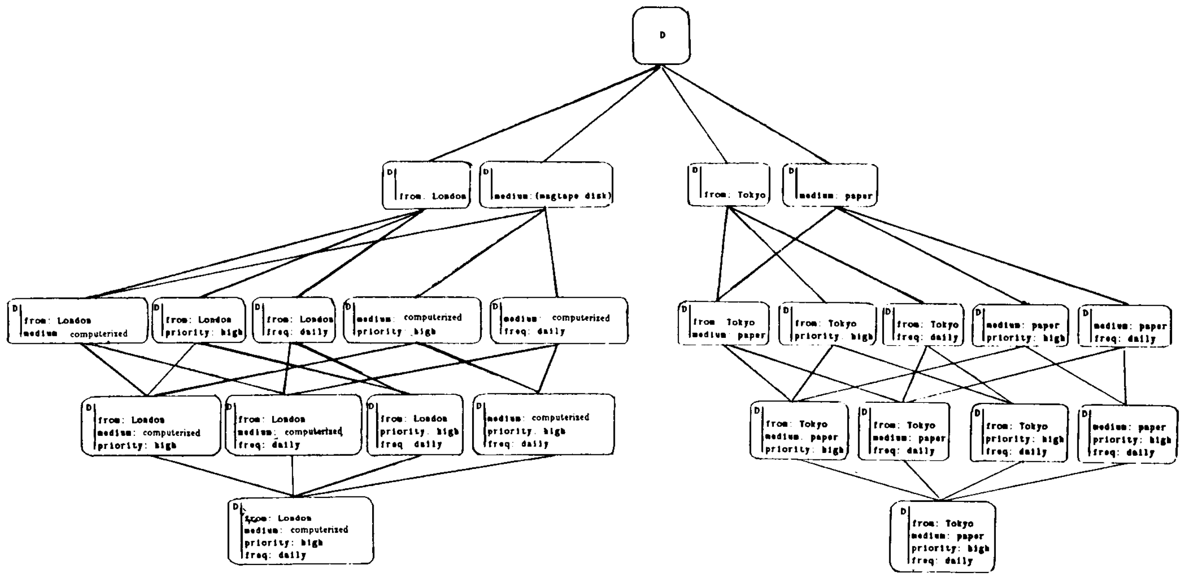

D
D from: London
D medium: (magtape disk)
D from: Tokyo
D medium: paper
D from: London medium: computerized
D from: London priority: high
D from: London freq: daily
D medium: computerized priority: high
D medium: computerized freq: daily
D from: Tokyo medium: paper
D from: Tokyo priority: high
D from: Tokyo freq: daily
D medium: paper priority: high
D medium: paper freq: daily
D from: London medium: computerized priority: high
D from: London medium: computerized freq: daily
D from: London priority: high freq: daily
D medium: computerized priority: high freq: daily
D from: Tokyo medium: paper priority: high
D from: Tokyo medium: paper freq: daily
D from: Tokyo priority: high freq: daily
D medium: paper priority: high freq: daily
D from: London medium: computerized priority: high freq: daily
D from: Tokyo medium: paper priority: high freq: daily

Fig. 16. Generalization hierarchy after E_3.

If the user's response is negative, it is clear that the node marked as "X" represents the most general correct hypothesis for Auto-Load-and-Edit. In this example, it means that the value of the "medium" slot is the sole discriminator in deciding on Auto-Load-an-Edit instead of Manual-Add-and-Edit. On the other hand, if the user responds in the affirmative, further querying is needed.

The above scenario raises two questions: 1) how does the system generate the example? and 2) what happens if the example turns out to be a positive training instance (i.e., the user's response is affirmative)?

To generate examples, one could begin with either the most general or the most specific element of a given hypothesis space. If we begin with the most general situation and the user responds negatively to the example, the node can be established as characterizing the most appropriate general class of situations for which the design decision is valid. Since we are trying to generate a negative instance, the node in the example is actually picked from another hypothesis space (Fig. 15)—a node that "corresponds" to X (Fig. 14). This corresponding node, marked "Y" in Fig. 15 (level 3), is at the same level of generality as X; only the value(s) of the discriminating slot(s) are different.

In addition to a method for choosing an initial hypothesis, the system must also have a search strategy for exploring the remaining nodes if its initial examples prove to be positive training instances. There are several ways to organize the search, the extremes being depth-first and breadth-first. We employ a breadth-first strategy. The justification for this is that in a design organized in terms of incremental transform of data, differences in one or only a small number of attribute-value pairs are likely to discriminate among the transformations. If the example above had proved to be a positive instance, the system would have generated another query using the "X2" in Fig. 14 as the situation in the example query, before proceeding to a more specific level.

To summarize the querying mechanism, the system attempts to establish a node at the most general level in one hypothesis space as the correct (characterization of the) situation. To accomplish this, the system generates an example, using as the situation a corresponding node in another hypothesis space, and attempts to establish via a query, whether the example is a positive or negative training instance with respect to the decision of that space. Further examples are generated using a breadth-first strategy.

Fig. 17 shows a generalization hierarchy where those nodes in Fig. 16 that are not relevant to the design decisions in the examples have been eliminated. As we can see, the hierarchy represents the general situations that underly that part of the design used in the examples. It is identical to Fig. 10.

V. Discussion

Some key aspects of the REMAP architecture—the object-oriented knowledge representation, the belief maintenance component, and the learning component—have been implemented in a Lisp environment. Design objects are represented using FLAVORS [26], a Lisp-based utility that supports object-oriented programming. In addition, dependencies are represented using the Reasoning Utility Package [21]. A designer interface and further refinements to the learning algorithms are under development. In a related project [17], the integration of these concepts with advanced knowledge representation languages for software development (RML [6] and TAXIS [29]), and with the DBPL database programming language [13] is studied to form a development and maintenance environment for database-intensive information systems software.

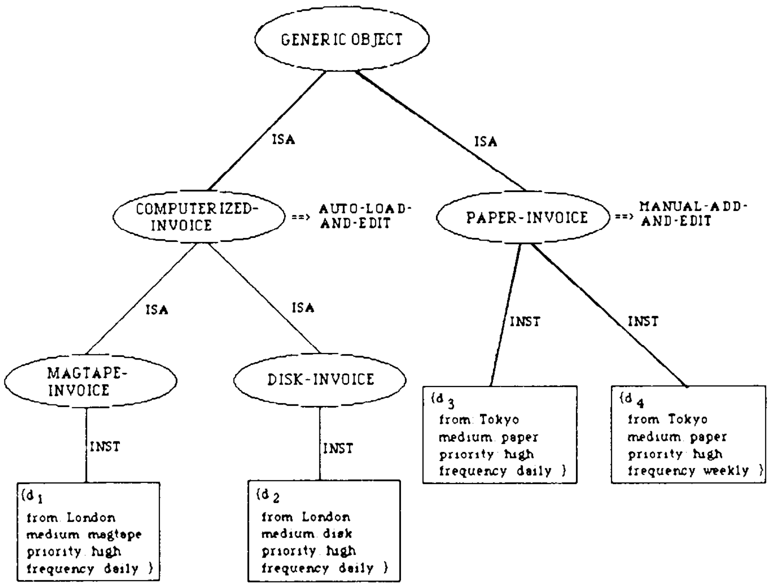

GENERIC OBJECT
ISA
ISA
COMPUTERIZED-INVOICE
==> AUTO-LOAD-AND-EDIT
PAPER-INVOICE
==> MANUAL-ADD-AND-EDIT
ISA
ISA
INST
INST
MAGTAPE-INVOICE
DISK-INVOICE
{d 3 from Tokyo medium paper priority high frequency daily }
{d 4 from Tokyo medium paper priority high frequency weekly }
INST
INST
{d 1 from London medium magtape priority high frequency daily }
{d 2 from London medium disk priority high frequency daily }

Fig. 17. Final generalization hierarchy corresponding to the design examples.

The REMAP formalism can be viewed as a knowledge-based tool for the representation and maintenance of design process knowledge, to be employed as part of an integrated software development and maintenance environment. The importance of REMAP's objectives is confirmed by two recent requirements studies on specification-based computing environments [2] and on artificial intelligence tool for design support in general (as contrasted to information systems design) [28]. Reference [2] emphasizes the need for supporting systems evolution at a high level design level as well as the software level. In particular, they suggest that design tools should be changeable, and that inter-user interaction should be supported. We believe that REMAP contributes primarily to the first goal by synthesizing an evolving object type hierarchy (which for instance, would allow the definition of a new design language other than data flow diagrams) relevant to the application domain. The second goal is partially achieved by allowing for each designer's justifications for design fragments to be made explicit. Reference [28] also stresses the need for making design goals, design decisions, and their justifications explicit.

In contrast to these recognized demands, existing databases or knowledge bases for software development tend to focus on the management of design objects rather than on the process knowledge captured by REMAP. Design databases evolved from the data dictionary concept which provides system-wide management of data structures as an aid in keeping notation in the systems designs and programs "consistent." It was soon realized that the data dictionary idea also applied to the management of process/module libraries [30], and to other design objects at higher levels of abstraction. Integrated environments such as TRW's Software Productivity System [4] or TEDIUM [3] also allow the designer to relate design objects, programs, and test cases or requirements specifications. However, these systems are somewhat handicapped by the lack of a precise requirements specification language [5], and because the relationship between requirements and designs is not explained in terms of design decisions and their justifications.

Proponents of prototyping [31] claim that systems changeability is automatically achieved or substantially supported through the prototyping process and cite case studies in support of this claim [1]. However, others have recognized that in complex systems, the prototyping idea must be applied at multiple levels of abstraction [15]. This in turn, requires substantial control of the process, taking into account the design justifications and rules learned from errors in previous prototypes [10]. While some researchers claim that such control can be provided by domain or other technique specific standards, policies, and constraints to be enforced in the development and maintenance environment [18], [23], [27], this approach assumes that such constraints can be enumerated *a priori*. A more ambitious approach, embodied in the PLEXSYS project [19] integrates constraint management into a full design support environment. PLEXSYS' dynamic metasystems [20] have represented application-specific knowledge in terms of an "axiomatic" model that can propagate certain types of changes to the object level where design decisions are represented. This approach is similar in spirit to that of TEIRESIAS [8], which uses a "meta model" to maintain and reason about object level knowledge contained in the MYCIN system [34]. Several other knowledge base management components of AI systems have been structured along similar lines.

While this approach has proven successful in situations where the scope of applications known to the meta-model can be defined in advance, it has fundamental limitations if the application domain is not known *a priori*. Under such circumstances, the high level model, even if definable, may become general to the point of missing the subtleties involved in an application area. What is needed instead, is a mechanism by which the high level model itself can be synthesized on the basis of experience in the application area. Consequently, REMAP follows an "open systems" approach [16] that begins by representing knowledge about relationships among instances in a domain in terms of dependencies, and generalizes some of these into a growing corpus of rules. In this way, the process knowledge involved in building an application can be used for incremental modification of designs, and where possible, to acquire knowledge in terms of application specific rules.

Methodologically, the instance level operations of our approach have much in common with those of the Programmer's Apprentice (PA) project [32], [36], [33]. The PA is an intelligent system that is designed to assist expert programmers with the maintenance of large programs. Like REMAP, the PA uses a dependency network of choices in order to represent and reason abut evolving programs. However, there are two important differences. Our focus is on the more abstract parts of the design as well as on the level of coding. More importantly, because of the diversity of applications, we are unable to assume a fixed library of "cliches" or programming constructs, but had to develop a learning method to build up this knowledge on the basis of specific designs. However, once our system has constructed and organized a library of cliches, they could be used to reason about "analogous" situations in a similar manner as the PA.

In concluding this section, we should distinguish between the analogical *reasoning* procedure described here for applying experience to new design decisions, and the *learning* by analogy procedures of [37] and others. In analogical learning, there is typically a domain where a known theory already exists in the form of rules or some other convenient representation; examples from this domain are then matched with examples from a domain in which the learning is to occur. In contrast, our learning scheme involves a novel combination of "learning by observation" and "learning by being told" which supports the acquisition of new terminology along with the recognition of conceptual structures and rules. In this way, our approach differs both from pure learning-by-example methods where no existing theory is assumed [22], [7], and from theory-based learning [25] where a good understanding of either the domain itself, or at least of an analogous domain is needed.

VI. Conclusions

The approach proposed in this paper suggests a novel way of thinking about systems evolution which emphasizes the designer's assumptions and justifications, rather than generally valid "meta-theories" of design. This reorientation can be of particular importance in the presence of multiple designers since many apparent "logical contradictions" may arise as a result of different *perspectives*, each based on a different set of assumptions.

From a practical viewpoint, the emphasis on design changes is of particular importance since it is estimated that at least 50 percent and probably as much as 70 percent of software costs go into maintenance. The work reported here is considered a first step toward a process-oriented design environment which is expected to have important applications in at least three areas. First, the prototyping method of systems development is enhanced by a learning component that prevents the repetition of design errors and supports a better formal understanding of the system's domain. Second, the undesirable practice of just updating program documentation in the maintenance phase of the software life cycle is replaced by a methodology for maintaining consistent designs; finally, the formalism provides a way of assessing the ramifications of real or proposed changes.

Finally, the analogy-based reasoning component of the method can support the reuse of designs in systems that are similar to existing ones. It also provides the designer of such systems with access to the justifications for the original design, thus permitting the encapsulation of required design differences and the identification of suitable alternatives. This controlled "cloning" capability is particularly valuable in organizations that have to construct a large number of functionally similar systems for different divisions. If process knowledge is not maintained automatically, such organizations have to rely on the experience and loyalty of a few key individuals.

References

[1] D. S. Appleton, *System 2000 Database Management System*, Guide 37, Cambridge, MA, in Nov. 1973.

[2] R. Balzer, D. Dyer, Fehling, and Saunders, "Specification based computing environments," in *Proc. 8th Int. VLDB Conf.*, 1982, pp. 273-279.

[3] B. I. Blum, "A workstation for information systems development," in *Proc. 7th IEEE COMPSAC Conf.*, 1983, pp. 116-120.

[4] B. W. Boehm, J. F. Elwell, A. B. Pryster, E. D. Stuckle, and R. D. Williams, "The TRW software productivity system," in *Proc. 6th Int. Conf. Software Engineering*, 1982, pp. 148-156.

[5] A. Borgida, S. Greenspan, and J. Mylopolous, "Using knowledge representation for requirements modeling," *Computer*, vol. 18, pp. 82-91, 1985.

[6] A. Borgida, J. Mylopoulas, and H. Wong, "Generalization/specialization as a basis for software specification," in *On Conceptual Modelling*. New York: Springer-Verlag, 1984.

[7] A. Borgida and K. Williamson, "Accomodating exceptions in databases and refining the schema by learning from them," in *Proc. 11th VLDB Conf.*, 1985, pp. 72-81.

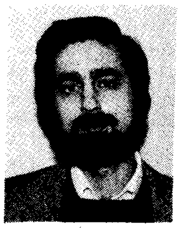
[8] R. Davis, "Interactive transfer of expertise," *Artificial Intell.*, vol. 12, pp. 121-158, 1979.

[9] T. deMarco, *Structured Analysis and System Specification*. New York: Yourdon, 1978.

[10] V. Dhar and M. Jarke, "Learning from prototypes," in *Proc. 6th Int. Conf. Inform. Syst.*, Dec. 1985, pp. 114-133.

[11] V. Dhar, P. Ranganathan, and M. Jarke, "Taxonomic concept formation and refinement from examples," Dep. of Inform. Syst., New York Univ., Working Paper 170, Dec. 1987.

[12] J. Doyle, "A truth maintenance system," Massachusetts Inst. Technol., Cambridge, AI Lab. Memo. 521, 1978.
[13] H. Eckhardt, "Draft report on the database programming language DBPL," Univ. Frankfurt, 1985.
[14] C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
[15] C. Groner, M. D. Hopwood, N. D. Palley, and W. Sibley, "Requirements analysis in clinical research information processing—A case study," *Computer*, vol. 12, pp. 100-108, 1979.
[16] C. Hewitt, "Implications of open systems," in *Managers, Micros, and Mainframes: Integrating Systems for End Users*, M. Jarke, Ed. New York: Wiley, 1986.
[17] M. Jarke, J. Mylopoulos, J. W. Schmidt, and Y. Vassiliou, "KBMS for software development," in *Proc. Xania Workshop Large-Scale Knowledge Base Management and Reasoning Systems*. New York: Springer-Verlag, 1987.
[18] M. Jarke and J. Shalev, "A database architecture for supporting business transactions," *J. Management Inform.*, vol. 1, pp. 63-80, 1984.
[19] B. Konsynski, J. Kotteman, J. Nunamaker, and J. Stott, "PLEXSYS-84: An integrated development environment for systems development," *J. Management Inform. Syst.*, vol. 1, 1984.
[20] J. E. Kotteman and B. R. Konsynski, "Dynamic metasystems for information systems development," in *Proc. 5th Int. Conf. Inform. Syst.*, Tucson, AZ, 1984, pp. 187-204.
[21] D. McAllester, "Reasoning utility package," Massachusetts Inst. Technol., Cambridge, AI Lab. Memo 667, 1982.
[22] R. Michalski, *Mahine Learning*. Palo Alto, CA: Tioga, 1983.
[23] N. Minsky and A. Borgida, "The Darwin software evolution environment," in *Proc. SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development*, Pittsburgh, PA, 1984.
[24] T. Mitchell, "Learning and problem solving," in *Proc. Eighth Int. Joint Conf. Artificial Intelligence*, 1983, pp. 1139-1151.
[25] T. Mitchell, S. Mahadevan, and L. Steinberg, "LEAP: A learning apprentice for VLSI design," in *Proc. 9th Int. Joint Conf. Artificial Intell.*, Los. Angeles, CA, 1985, pp. 573-580.
[26] D. Moon and D. Weinreb, *Lisp Machine Manual*, Massachusetts Inst. Technol., Cabridge, AL Lab., 1981.
[27] M. Morgenstern, "Active databases as a paradigm for enhancing computing environments," in *Proc. 9th VLDB Conf.*, Florence, Italy, 1983, pp. 34-42.
[28] J. Mostow, "Towards better models of the design process," *AI Mag.*, vol. 6, pp. 44-66, Spring 1985.
[29] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong, "A language facility for designing database intensive applications," *ACM Trans. Database Systems*, vol. 5, 1980.
[30] K. Narayanaswamy, W. Scacchi, and D. McLeod, "Management support for evolving software systems," Dep. Comput. Sci., Univ. Southern California, Los Angeles, 1985.
[31] J. D. Naumann, and A. M. Jenkins, "Prototyping: The new paradigm for systems development," *MIS Quart.*, vol. 10, pp. 24-40, 1982.
[32] H. Shrobe, "Dependency directed reasoning for complex program understanding," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, 1979.
[33] C. Rich, "A formal representation for plans in the programmers apprentice," in *On Conceptual Modeling*, M. L. Brodie, J. Mylopolous, and J. W. Schmidt, Eds. New York: Springer-Verlag, 1984.
[34] E. H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*. New York: American Elsevier, 1976.
[35] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA: M.I.T. Press, 1981.
[36] R. Waters, "The programmer's apprentice: Knowledge based program editing," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 1-20, 1982.
[37] P. H. Winston, "Learning and reasoning by analogy," *Commun. ACM*, vol. 23, pp. 689-703, 1979.



Vasant Dhar received the Bachelor of Technology (B.Tech.) degree in chemical engineering from the Indian Institute of Technology, and the Ph.D. degree in information systems with a major in artificial intelligence from the University of Pittsburgh, Pittsburgh, PA.

He is an Assistant Professor of Information Systems at the Graduate School of Business Administration, New York University, New York, NY. He has been a full-time member of the faculty at New York University since 1983. His primary research interests are in the empirical and theoretical aspects of artificial intelligence. Much of his research involves empirical investigation of problem-solving processes in domains involving design, planning, and decision-making, and the design of representational formalisms needed to build intelligent systems in these domains. He has written a number of articles on knowledge representation, heuristic search, and methodological issues in the development of knowledge-based systems.

Matthias Jarke received the Diploma degrees in business administration and computer science in 1977 and 1979, respectively, and the Ph.D. degree in economical sciences in 1980, all from Hamburg University, Hamburg, West Germany.

He is now a Professor of Computer Science at the University of Passau, West Germany. Formerly, he held faculty positions at New York University and Johann Wolfgang Goethe University, West Germany. His research interests include the design, evaluation, and optimization of high-level database interfaces to end users, decision support systems, and expert systems, both from a systems programming and from a user perspective. He is currently leading ESPRIT project DAIDA which investigates knowledge base management systems for database software development and maintenance. He is the author of a number of articles and book chapters on computer science and business subjects, and of four books in these areas.

Dr. Jarke is a member of the Association for Computing Machinery, the American Association for Artificial Intelligence, and the IEEE Computer Society.