DEPENDENCY DIRECTED REASONING AND LEARNING IN SYSTEMS MAINTENANCE SUPPORT

Vasant Dhar
New York University

and

Matthias Jarke
University of Frankfurt

March 1987

Center for Research on Information Systems
Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

STERN #IS-87-20

To appear in the **IEEE Transactions on Software Engineering** (1987 volume).

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 can not only be applied to prototype refinement and systems maintenance, but can also support the re-use of existing design or software fragments to construct similar ones using analogical reasoning techniques.

1. 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 underly 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 mechanism should make use of the classifications 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 3 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. Section 2 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 3. Section 4 describes in detail the learning component as a central part of the architecture. Section 5 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.

2. 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.

2.1. The Case Study

The problem studied in the oil company involves the design 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 US and abroad.

For the sake of readability, we describe systems using the Structured Analysis representation [9], [14]. However, that 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 mini-specs on which the program designs are based. Data flow and data store information is managed in data dictionaries. Figure 1 shows the notational conventions used in this paper.

Part of the structured top-down design of OC's Sales subsystem is illustrated in

figures 2 through 5. Figure 2 shows a context diagram which depicts the relationship of the system to external entities. Figures 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 figures 2 to 5 is sufficient to describe the problems of systems maintenance and our solution to them.

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.

2.2. The Role of General and Specific Knowledge

"London Sends Formated Invoices". In the original design, the difference between the New York and London invoices was that the former were accessable *formated* whereas the latter were received *unformated*, 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 begins to send correctly formated 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 parts of the design in figure 3. However, at the next lower level (figure 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, the system must know that in the existing design, the convert bubble is dependent on the nature (i.e. unformated)

DATA FLOW DIAGRAM CONVENTIONS

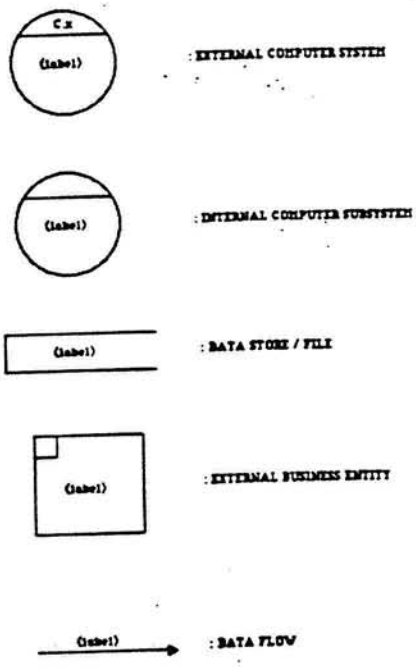

Figure 1

SALES ACCOUNTING SYSTEMS CONTEXT DIAGRAM

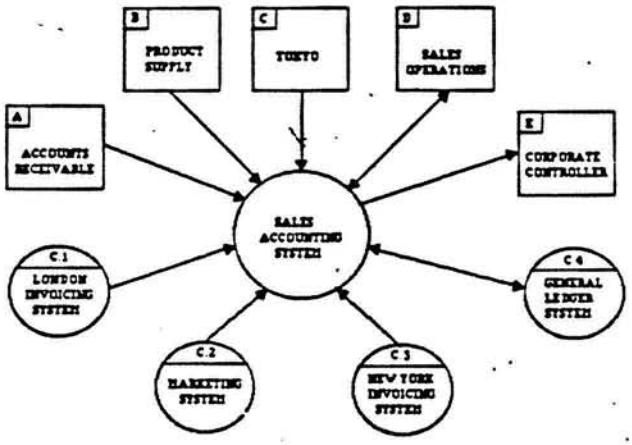

Figure 2

FUELS SALES (INITIAL)

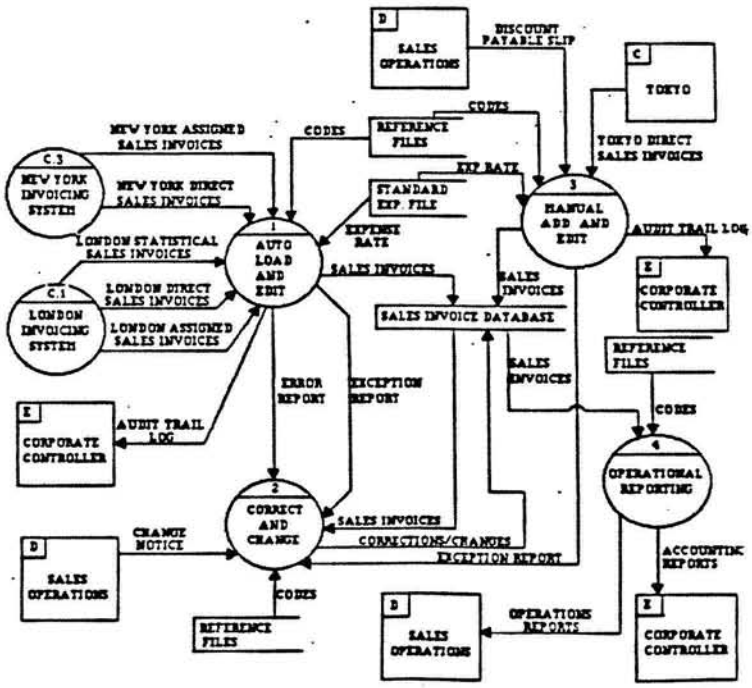

Figure 3

of the dataflows representing London invoices. On recognizing that London invoices are no longer unformated, it should be able to detect the fact that conversion is unnecessary. Further, it should also know that *in general*, formated invoices proceed directly for on-line verification. Based on this, it 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 figure 4 depends on the existence of unformated invoices).

2.3. 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 figure 6.

---

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

AUTO LOAD AND EDIT

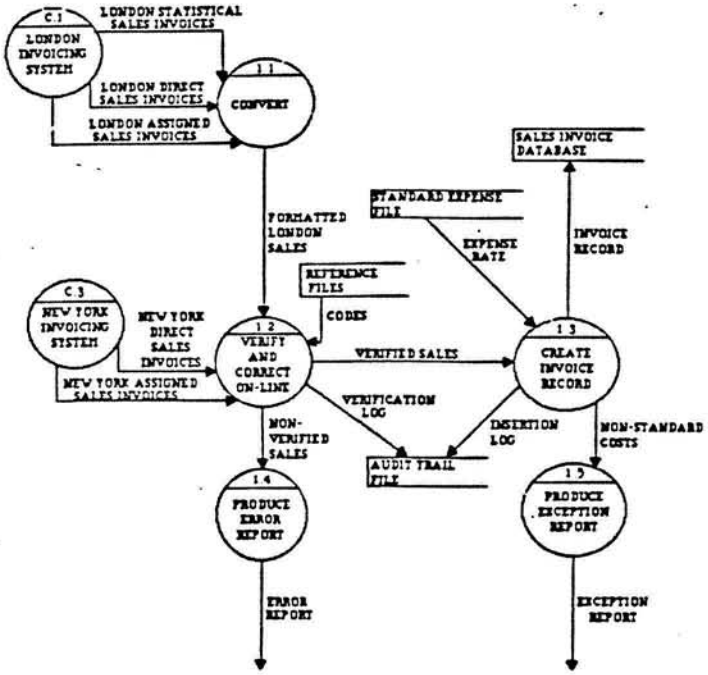

Figure 4

MANUAL ADD AND EDIT

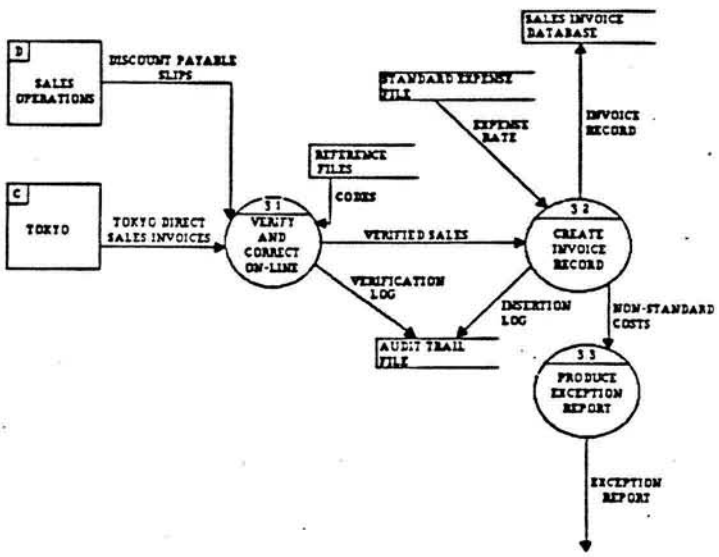

Figure 5

FUELS SALES (MODIFIED)

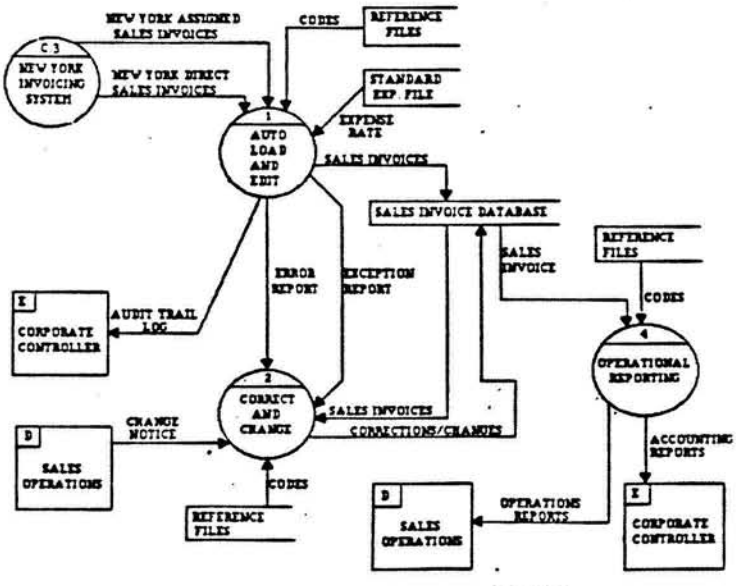

Figure 6

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.

2.4. The Role of Analogy

"The Venezuela Office Will Sell Fuels". This corresponds to a high level change that 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 formated, unformated, 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 *formated.* Once the important features are realized, the design ramifications become clear.

2.5. 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 representation. These interpretations derive from the *purpose* of the application which cannot be determined from looking at the resulting design alone. Since the design is an artifact [35], its 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 about the types of changes illustrated in the examples, it must have a formal representation for 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 knowledge involved in systems development and maintenance.

3. 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 4.

3.1. 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. "Components" 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 figure 7. Some instances of dataflows and transforms used in the three scenarios of section 2 are shown in figure 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

INITIAL OBJECT TYPE HIERARCHIES

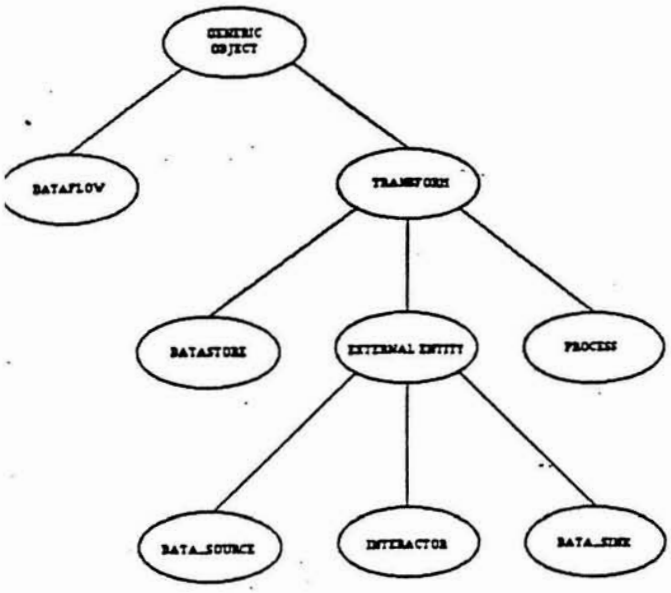

Figure 7

INITIAL GENERALIZATION HIERARCHY

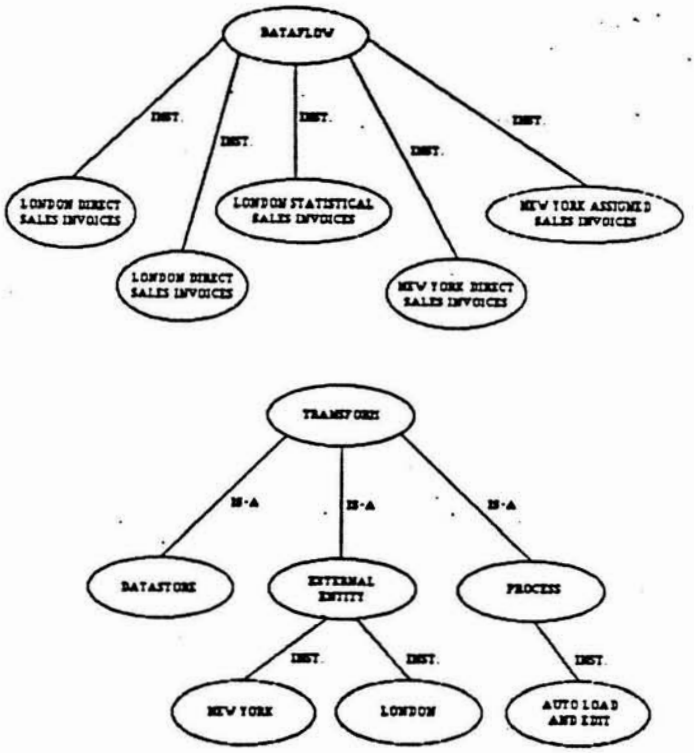

Figure 8

A DEPENDENCY NETWORK

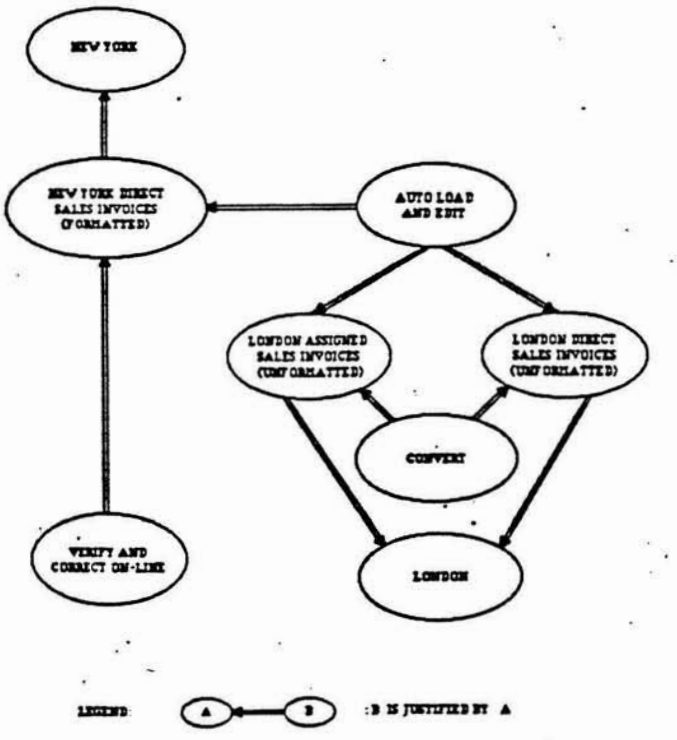

Figure 9

RECONFIGURED GENERALIZATION HIERARCHY

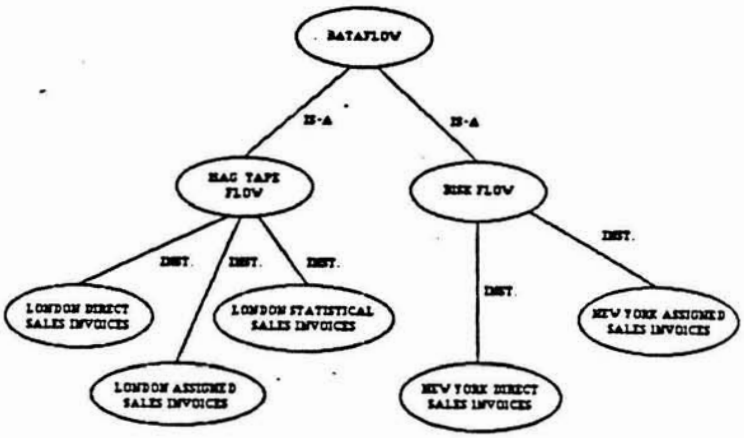

Figure 10

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. figure 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)

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. figure 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 "formated", 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.

3.2. Representing Design Processes 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*:

(<justification> ==> <action>)

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

To illustrate, consider figure 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 formated. 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 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 formated London invoices should be treated like New York invoices (and should proceed directly for verification), it is necessary to know that *in general* formated invoices

are verified directly. This knowledge can then be used to reason about all object instances corresponding to formated invoices.

3.3. Learning as Rule Formation

Dependency information as indicated in figure 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 4.

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 figure 10, and in rules that are applicable at various levels of abstraction.

3.4. 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 re-using fragments of existing designs, based on the general knowledge acquired by REMAP's learning component.

For example, section 2.4 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 knowledge 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 figure 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 at 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 unformated; this had required a convert operation which was subsequently eliminated when the form of these invoices was changed. However, since a rule on formated vs. unformated invoices was retained which now becomes applicable to Venezuela invoices, the old convert operation could be reinstalled, or a similar one implemented if the formating differs at a lower level of abstraction than shown in our examples.

3.5. 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 and knowledge acquisition modes. Figure 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 semi-formal 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 4). 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 4, 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* => *inst* to the dependency base.
   Add the description of *inst* to the design object base.
   Call *Learn-mode (just, inst, ti, tj).*

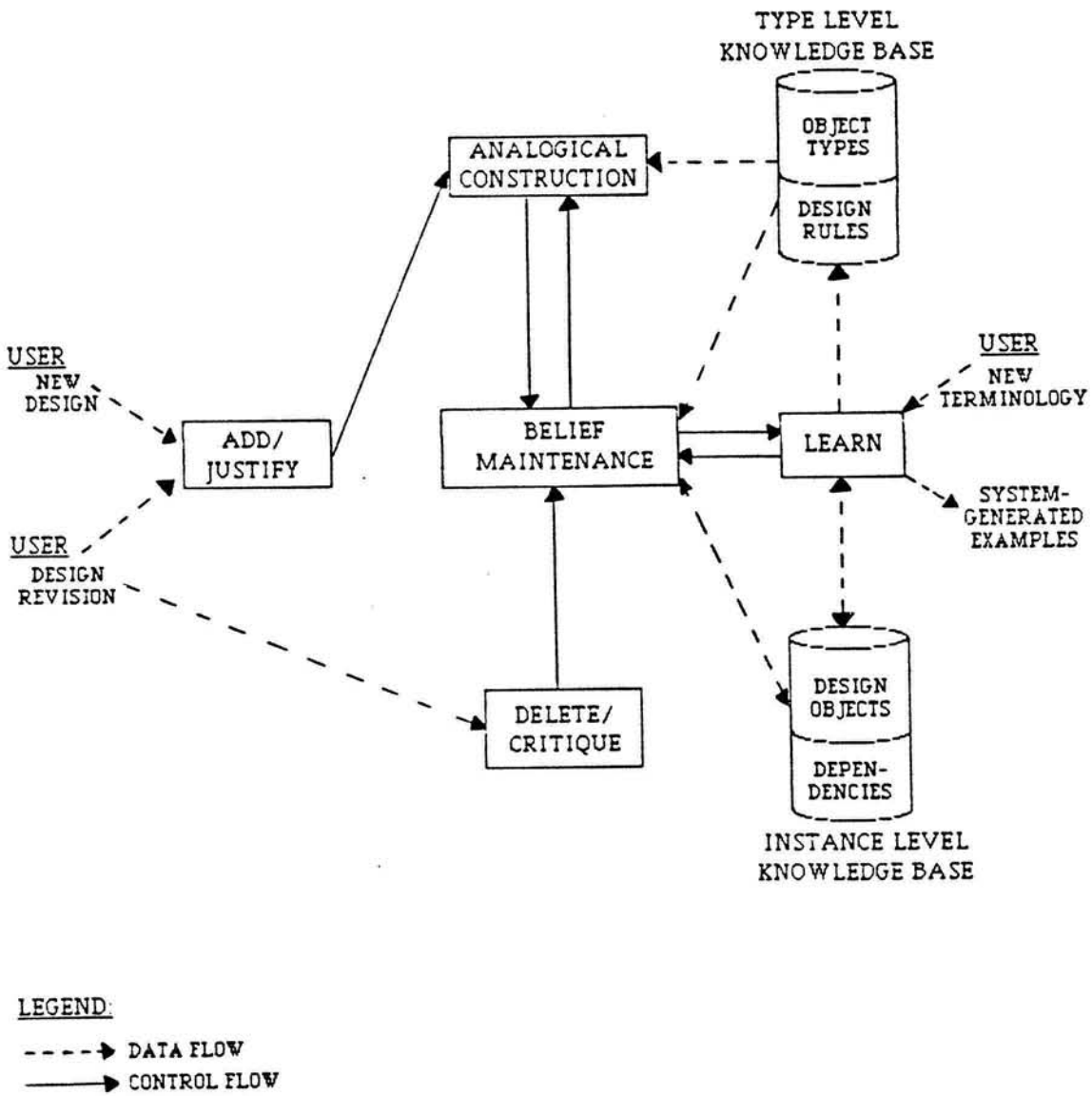

Figure 11. Summary of REMAP Architecture.

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

4. Synthesizing The Generalization Hierarchy

Infering 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 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

---

[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].

contains possible generalizations of situations for each decision. These generalizations are gradually eliminated or refined with successive examples. For a design expressing 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 3.1. This object type, hence called D, has slots $s_1, s_2, s_3, ..., s_p$. 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 j^{th} 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.

4.1. 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 figure 3, is:

E_1 =
{d_1
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 Figure 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.

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

---

[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.

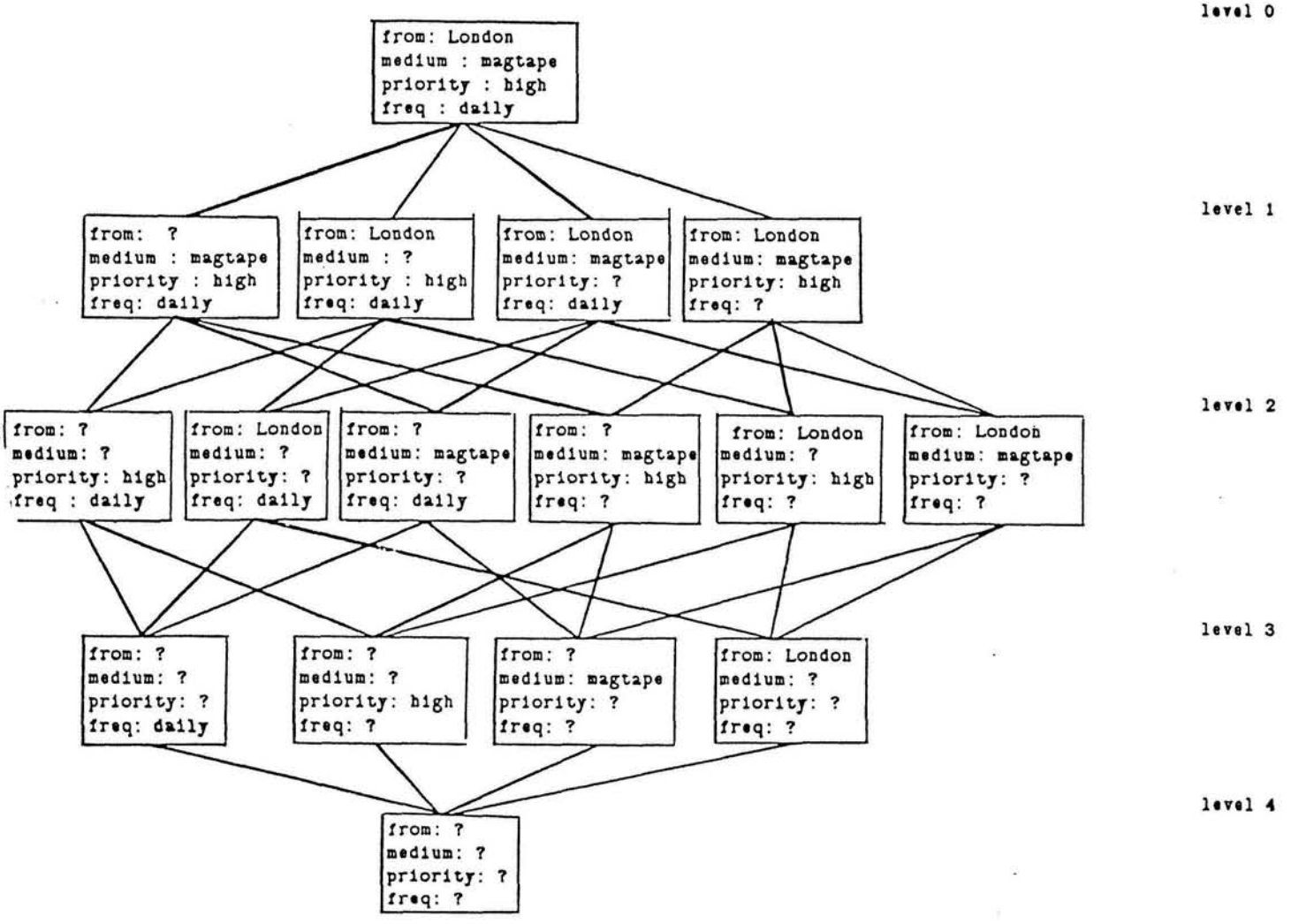

Figure 12. Hypothesis space for Auto-load-and-Edit (t_1) after E_1.

hypothesis space is shown in Figure 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.

$E_2 =$
{ d_2
from: London
medium: disk ==> Auto-load-and-edit
priority: high freq: daily }

Comparison with 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 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 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 accomodate 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 figure 12, is now refined to reflect these modifications. We have replaced "magtape" by "computerized" in the relevant

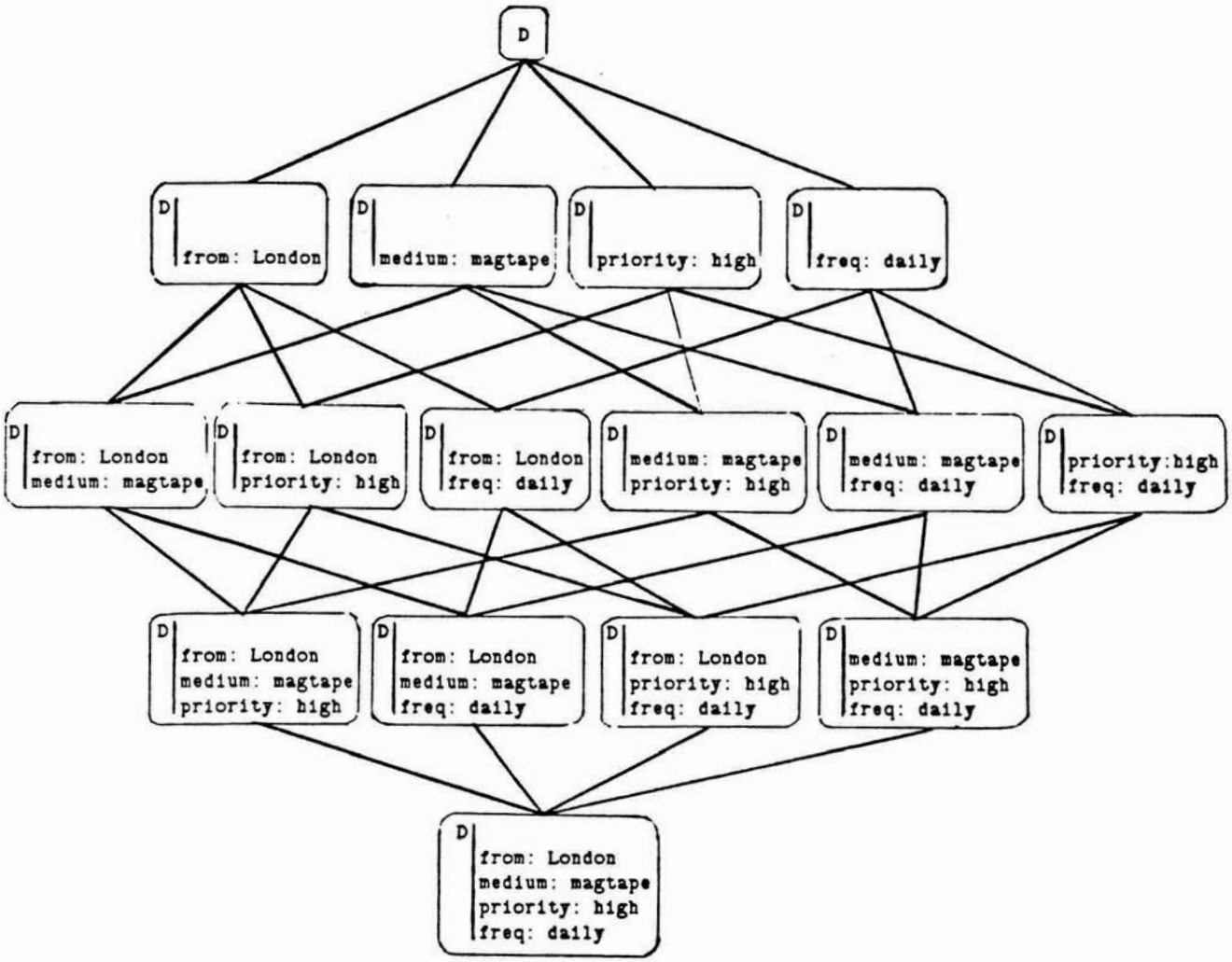

Figure 3. Generalization Hierarchy after E_1. Nodes in the hierarchy are specializations of D where slot and value pairs on the right of the vertical bar indicate restrictions on an object type. The lines joining the nodes are IS-A Links.