ACQUIRING APPLICATION-SPECIFIC KNOWLEDGE DURING[1] DESIGN TO SUPPORT SYSTEMS MAINTENANCE

by

Vasant Dhar
P. Ranganathan

Information Systems Department
Graduate School of Business Administration
New York University
90 Trinity Place
New York, NY 10006

and

Matthias Jarke

University of Passau
West Germany

June 1986

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

Working Paper Series

CRIS #170
GBA #87-117

---

[1]We wish to thank Albert Croker for his assistance with the notation used to formalize the model.

Abstract

Most large systems development efforts proceed in a top-down fashion where initial specifications and requirements are incorporated into a high-level design, followed by programs based on this design. However, a major part of the software life-cycle effort is devoted to maintenance. While several existing methodologies aid in the initial phases of requirements and specification, they have proven to be of little value for maintenance. Changes in user requirements are often translated directly to the level of code, divorcing it from the high level design it was based on. After a few such changes, the programs may not correspond to any formal high-level design, making subsequent maintenance difficult. We argue that maintenance must be based on the knowledge used in synthesizing the high-level design. This requires a development environment where the knowledge about high-level designs is formally represented, and raises the question about how this knowledge will be acquired by the support environment in the first place. In this paper, we present a model that enables the support environment to acquire design knowledge through "learning by observation" of a designer engaged in specifying a high-level design. The knowledge that the learning system begins with is a generic object for expressing design decisions. Based on the input provided by the designer, and a limited interactive querying process, it constructs and continuously refines a taxonomic classification of application-specific knowledge and rules at an appropriate level of generality that capture the rationale of the design. This knowledge can be used subsequently for maintaining system designs and recognizing design situations similar to the ones it has knowledge about.

KEYWORDS: Knowledge-based Systems Maintenance, Software life cycle, knowledge acquisition and learning, object-oriented design.

Table of Contents

1. The Systems Maintenance Problem

Systems maintenance refers to changes that have to be made to computer programs after they have been delivered to a user. While there exist several well-established techniques that help in structuring the initial specification and high-level logical design, they prove to be of marginal value for maintenance. Fundamentally, this is because much of the semantics of the application domain are *implicit* in the primitives/structures that constitute the initial design supported by these techniques. This has two related consequences. First, the ramifications of specifications or requirements changes are not readily apparent at the design and hence the program level, and must be assessed completely by the designer. This requires the designer to alternate continually between the high level design and the low level programs, an activity which is cumbersome and prone to error. A consequence of this situation is that it encourages changes to be made directly at the level of code, thereby throwing the conceptual design and programs out of sync, rendering the design useless for purposes of maintenance. Over time, the relationship between the design and code can loosen considerably, placing a heavy burden on the designer to remember the associations between the application domain and the code; if this individual's involvement with the system ceases, making changes can become extremely difficult.

Our position is that while maintenance ultimately involves managing changes at the level of code, it can be greatly facilitated if we first attend to maintaining accurate higher level design specifications on which the programs are based. Coupling maintenance to design requires a development environment (henceforth environment) where application-specific knowledge about dependencies among various parts of the high-level design and the general *bases* for them can be accumulated in an appropriate form and used to reason about subsequent design changes. An important component of such an environment is a learning or knowledge acquisition mechanism that can extract the general bases for dependencies among design decisions expressed by the designer/analyst. This application-specific knowledge can be used subsequently in maintaining an increasingly complex software design, and in detecting similarities between new design situations and ones it has already encountered.

In this paper we present an implemented model for knowledge-acquisition/learning that is part of the larger environment we are developing for systems development and maintenance (Dhar & Jarke, 1985). A primitive object-oriented language is used for describing the high-level design in terms of situations (a set of attribute-value pairs) and design decisions/actions. A complete design is viewed as a set of "examples", each consisting of situation-action pairs. The learning process involves generating hypotheses about associations between design situations and actions, and an iterative refinement of these hypotheses based on successive examples. Further, in situations where

it is not possible to sufficiently constrain the plausible generalizations arising out of the examples, some of these generalizations can be eliminated via an intelligent querying process. Functionally, the queries can be viewed as generating examples that help discriminate among the hypothesized generalizations.

2. Representing Designs

Getting started on a design requires putting aside the details and the procedural aspects of the problem. A common technique for imposing structure on a problem is to break it down into connections among abstract "black boxes", and label these boxes and connections to designate features of the problem. Gradually, the function and structure of these components becomes specified. Over the last two decades, several methods have been developed for expressing designs. A limitation of most of these schemes, however, is that the semantics associated with their primitives and hence the application-specific labels attached to them is not precise. Further, the structure of programs can become language-dependent. These factors have led researchers to design high-level object-oriented specification languages which allow for data independence, and have a formal semantics associated with high-level design primitives so that they are machine interpretable (for example TAXIS (Borgida et.al. (1984)); Belkhouche & Urban (1986)).

Our approach toward developing a comprehensive design environment is in the spirit of these latter approaches, and involves the design of a set of ontological primitives for specifying designs. We are interested in extending the advantages of the object-oriented representation to very high level design specifications that have traditionally been expressed using other methodologies such as structured design. In this paper a limited subset of this language, namely, a set of structured object types is used to represent designs in terms of dataflows and transformations of dataflows. This scheme resembles structured design methods using dataflow diagrams (deMarco, 1979; Gane & Sarson, 1979; Yourdon & Constantine, 1979). However, we should stress that our model of learning is independent of any particular design method, and in this respect, the Structured Design Methodology is used for illustrative purposes only.

2.1. Design Primitives

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 transformations on data, 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. In order to keep the discussion simple, we limit ourselves to dataflows and transformations on them. Both dataflows and transformations are represented as structured objects.

In this representation, a design decision (also called an action) is a transformation that is required because of certain attributes of a dataflow. That is, the dataflow constitutes a situation, and the transformation is an action.[1] The learning task, as we shall shortly illustrate, is one of distinguishing between the important and the incidental attributes of the dataflow. Once this is done, a generalization of the situation can be constructed that incorporates in it the important features while ignoring the incidental.

2.2. An Example Set of Design Decisions

For purposes of illustration, we use the following example in this paper. We assume that an organization has a central computerized sales accounting system (the one being designed and/or maintained) that processes sales data from its two branches, one in New York and the other in London. These branches generate three types of sales invoices, namely, direct-sales-invoices, assigned-sales-invoices, and statistical-sales-invoices which indicate different types of sales. These invoices are *computerized*, that is, are accessible from a magnetic tape, or directly from disk. Further, New York invoices are *formated* according to some scheme whereas London invoices are *unformated*, because of which they must go through a *convert* operation before they can be processed. Since both offices generate computerized invoices, they can be automatically loaded into the system for processing; if the invoices had been non-computerized (i.e. paper invoices), manual editing and entry would first be required. A small fragment of a high-level design corresponding to the above description is presented in figure 1a, with a decomposition of part of figure 1a in figure 1b. The symbols have their usual meanings (see deMarco, 1979).

3. The Objective: Synthesizing the Generalization Hierarchy

Basically, our objective is to infer the general knowledge underlying a design where the design consists of a set of examples. Viewed differently, the design is an *instantiation* of a more abstract model relevant to the application domain.

Extracting plausible generalizations from examples is basically a learning task. It involves

---

[1] This is a convention. Other conventions for designating situations and actions (for example, see Orr (1981)) can also be adopted.

LEVEL 1 DATAFLOW DIAGRAM OF EXAMPLE

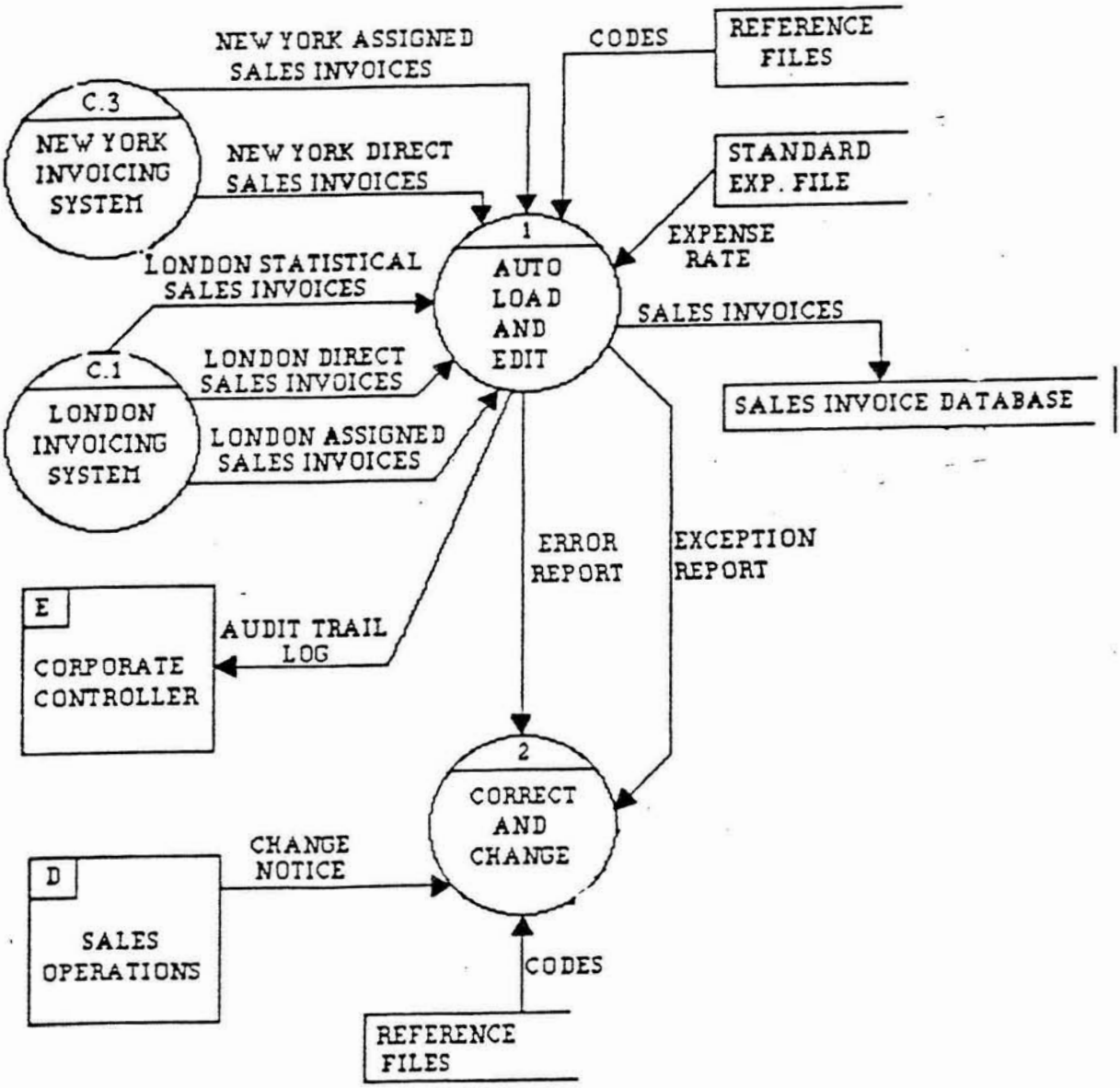

FIGURE 1a

LEVEL 2 DATAFLOW DIAGRAM OF EXAMPLE

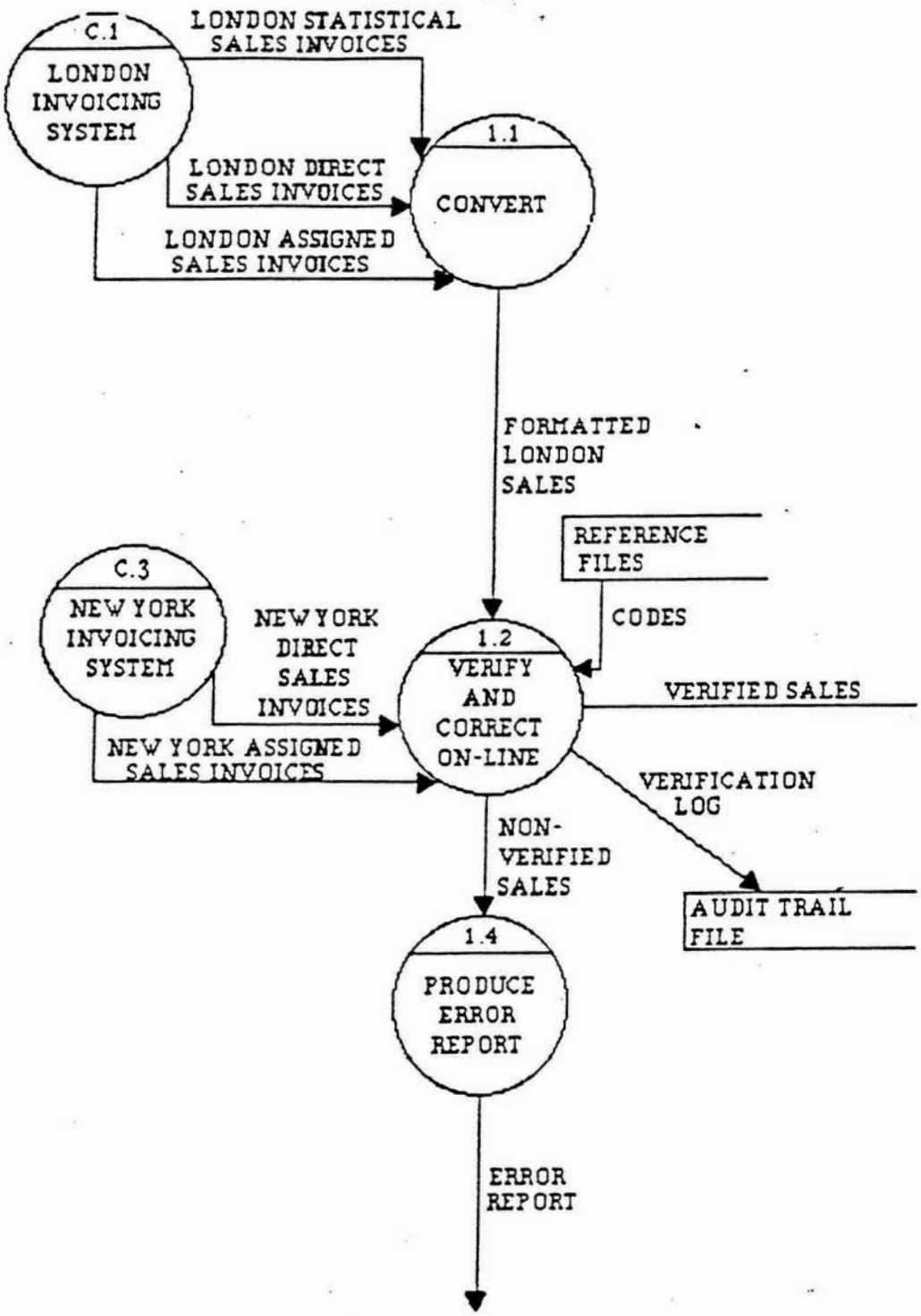

FIGURE 1b

generalizing situations into categories on which design decisions 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.

The problem of generating plausible generalizations is essentially a *search problem*. Most researchers in Psychology and Artificial Intelligence (AI) have in fact viewed Learning primarily as a heuristic search through a space of possible generalizations – also referred to as the hypothesis space (for example, see Simon (1977)). This approach has formed the basis for several AI systems such as those of, Waterman (1970), Sussman (1975), Lenat (1982) and Michalski (1980).

While search is an important ingredient of any learning mechanism, more recent work has focussed on representations for imposing structure on the hypothesis space to reduce the search. Broadly, this includes learning by analogy (Winston 1975), by being told (Davis 1979), learning based on candidate eliminations in the light of successive training instances (Mitchell 1977, 1983a), and learning by observing experts solve specific problems (Mitchell et al. 1985). In these approaches, emphasis is on incremental learning based on a small number of examples.

Our approach to forming general descriptions is based on 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 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.

4. An Example

We provide a formal notation for the data structures and the learning algorithm in the next section. To illustrate the example however, a brief description of these structures and the mechanics for operating on them is first necessary.

A situation is characterized in terms of an instance d_i of a generic object D that is used to express

the examples. It 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 $V_{i,j}$ 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 would represent 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.

To introduce the model, let us consider some design decisions from the sales accounting system mentioned earlier. To keep the example clear, we restrict the generic object D to four slots, called "from", "medium", "priority" and "frequency". These slots are relevant for defining dataflows in the design of a particular sales accounting system we have analyzed. The first example, designated E_1 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.

A representation of the possibilities, the hypothesis space of all possible rules based on the first example, is shown in Figure 2. 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 LEX [Mitchell, 1983a]. In those spaces, 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 objects. The difference, of course, is that these

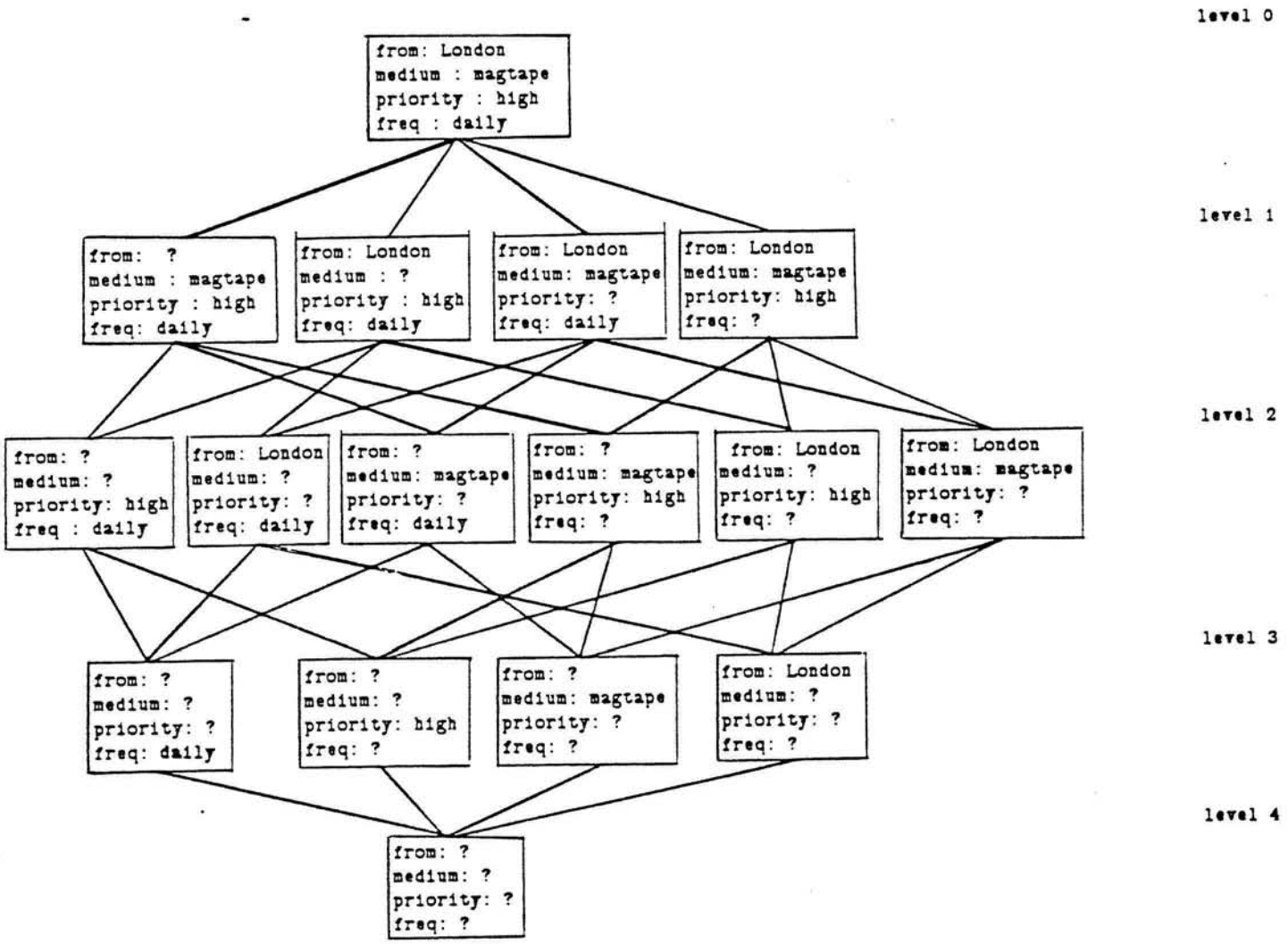

Figure 2. Hypothesis space for Auto-load-and-Edit (t_1) after E_1.

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 Figure 3. In summary, an initial hypothesis space generates a crude object taxonomy. As the space is refined, so is the taxonomy.

Let us consider what happens when another example, again representing a design decision, is presented.

```
E2 =
  { d2
      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. Ideally, 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, we designate the possibility of there being a superclass using a dot notation such as "magtape.disk". This designates a class that subsumes "magtape" and "disk". The system must query the user as to whether a suitable superclass exists which can characterize both "magtape" and "disk". If the user responds with "computerized", the system asks the user to enumerate other members belonging to the class labelled "computerized". 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 we return to in the next section.

The hypothesis space for t_1, shown in figure 2, is now refined to reflect these modifications. We

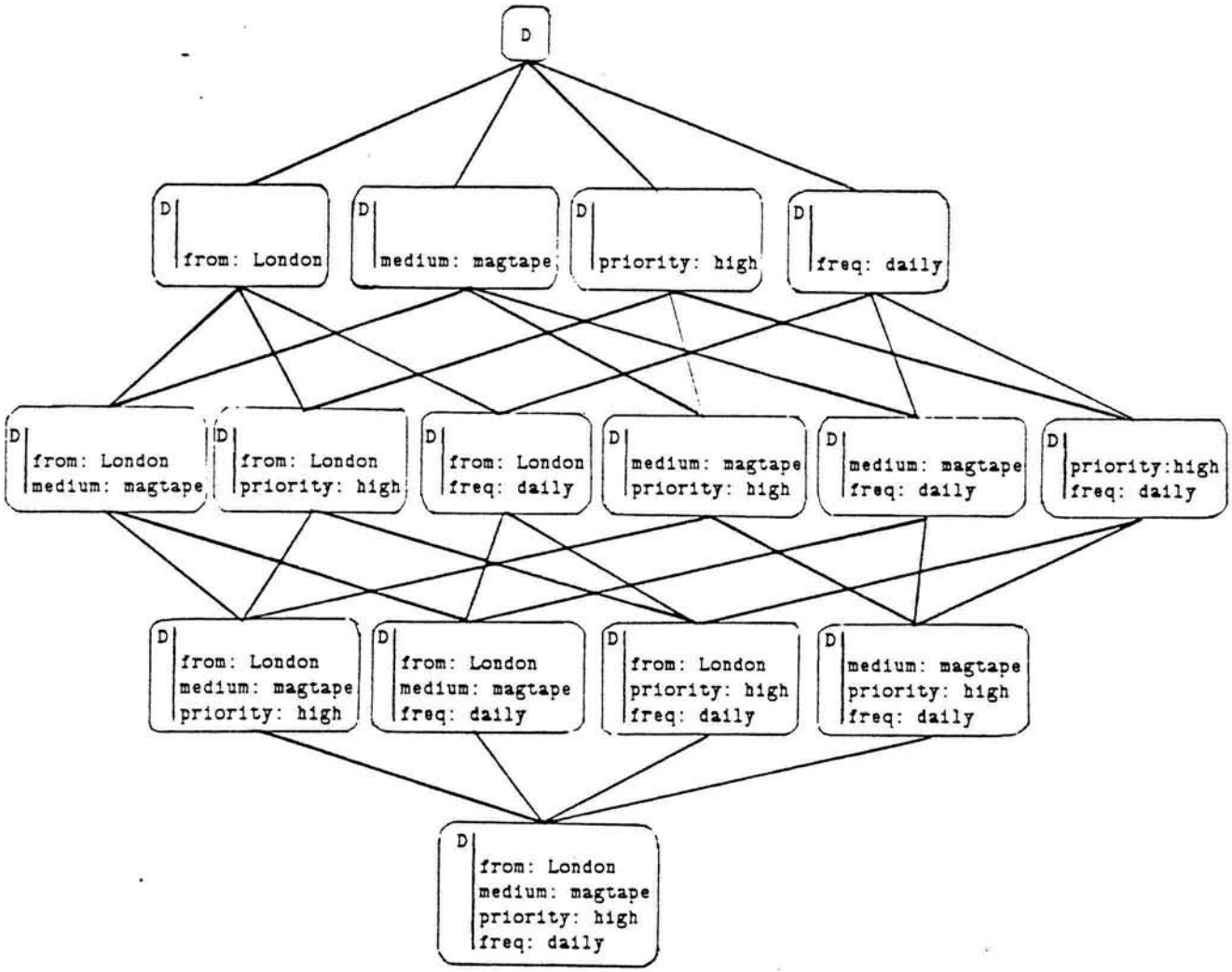

Figure 3. Generalization Hierarchy after E_1. Nodes in the hierarchy are specializations of D where slot and value pairs on the right of the vertical bar indicate restrictions on an object type. The lines joining the nodes are IS-A Links.

have replaced "magtape" by "magtape.disk" (instead of using the new label "computerized") in the relevant slots. This change reflects a modification of the object types in the hypothesis space as shown in figure 4. The dotted segment will be explained shortly. The generalization hierarchy is reorganized accordingly to incorporate the modified object type.

Let us now consider a third example:

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

This instance is a negative instance with respect to E_1 and E_2. 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 rationale for Manual-add-and-edit (t_2).

In 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 types were indicated in the dotted section of figure 3. In the refined hypothesis spaces of Auto-load-and-edit and Manual-add-and-edit (figure 4) these nodes are marked as eliminated.

The generalization hierarchy, reflecting the refined hypothesis spaces is also modified to that shown in Figure 6. It represents a union of the two hypothesis spaces.

As a final example, let us consider the following:

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

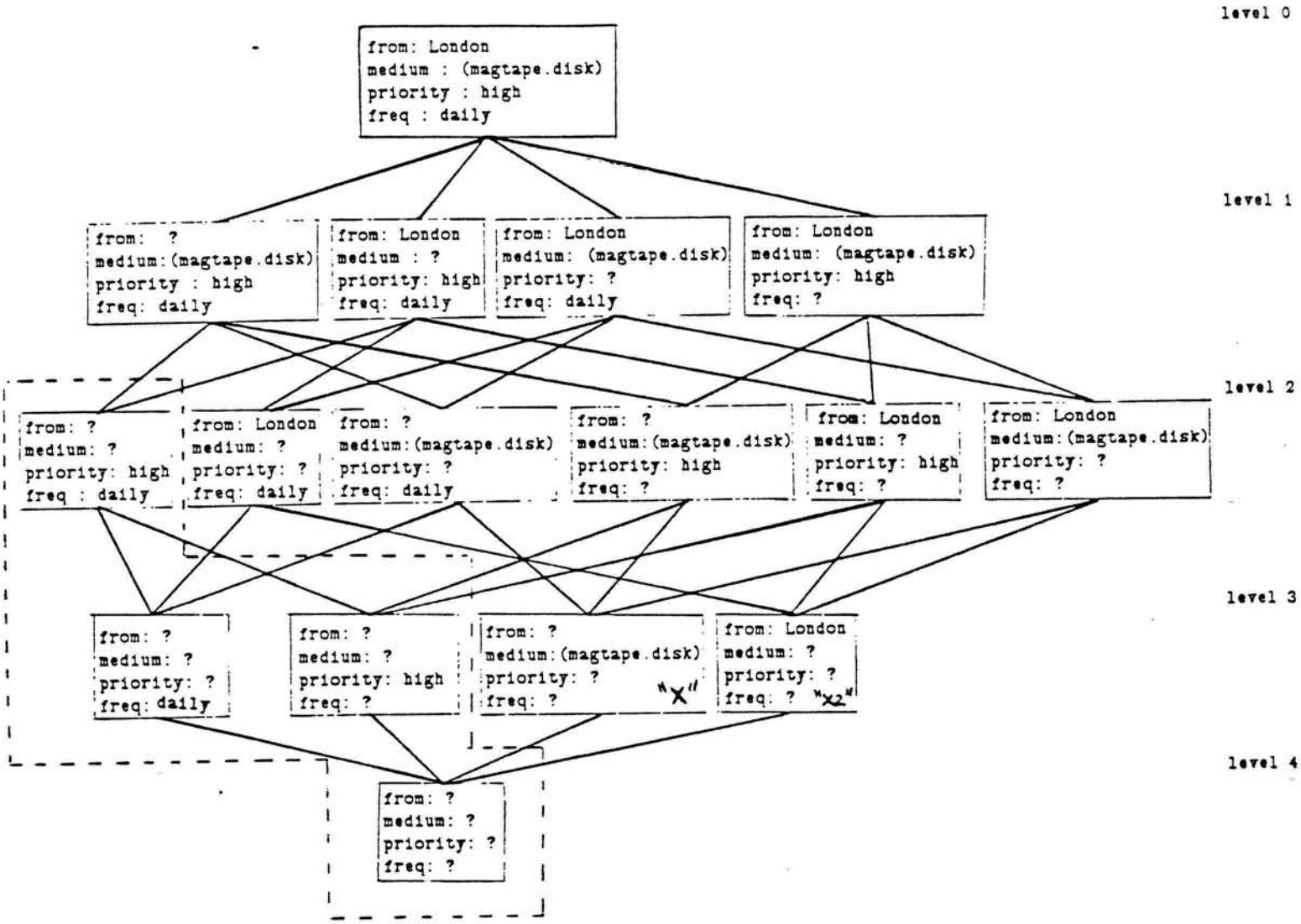

Figure 4. Hypothesis space for Auto-load-and-edit after E_2.

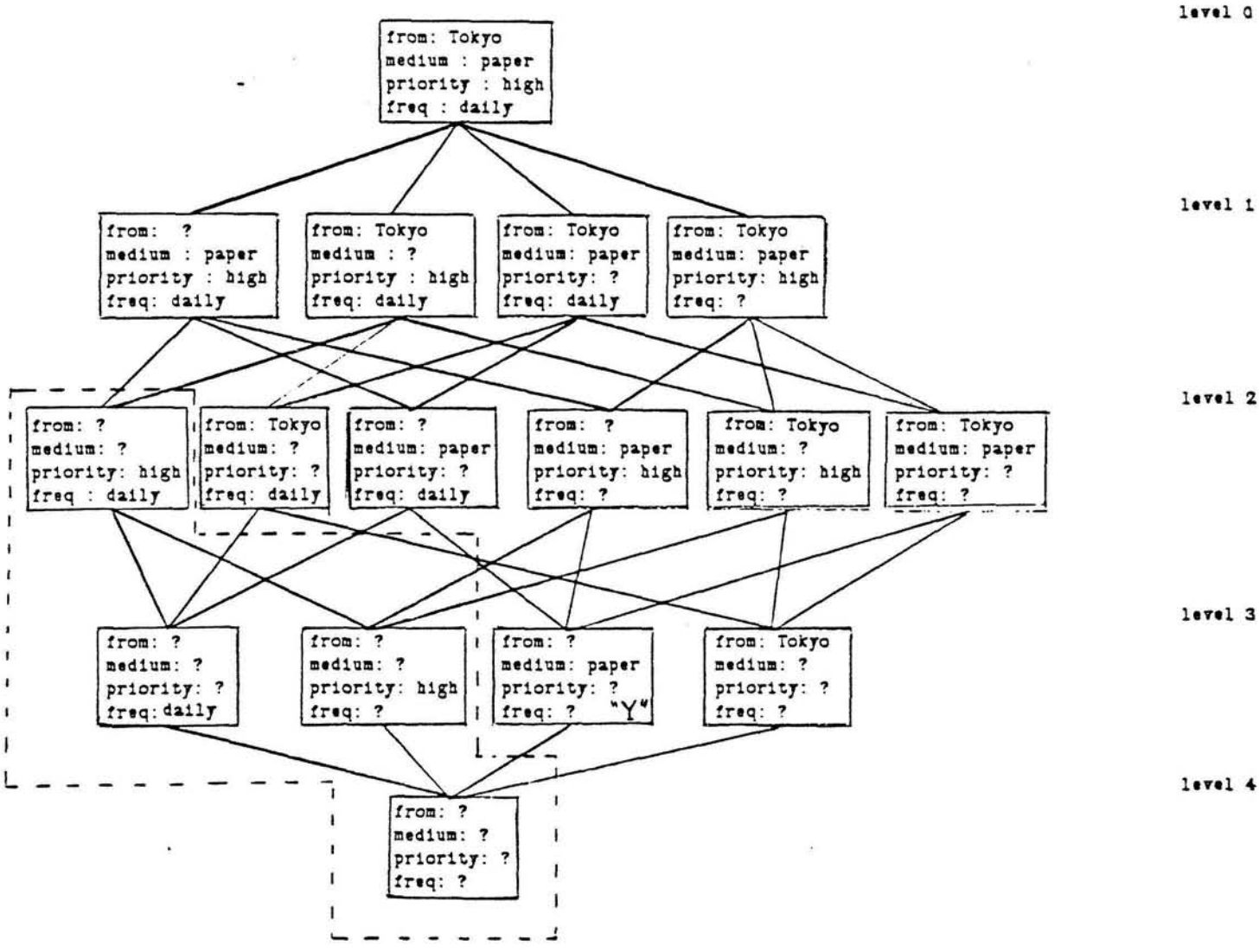

Figure 5. Hypothesis space for Manual-add-and-edit (t_2) after E_3. Comparision of this hypothesis space with that of t_1 leads to the removal of the dotted area from both hypothesis spaces.

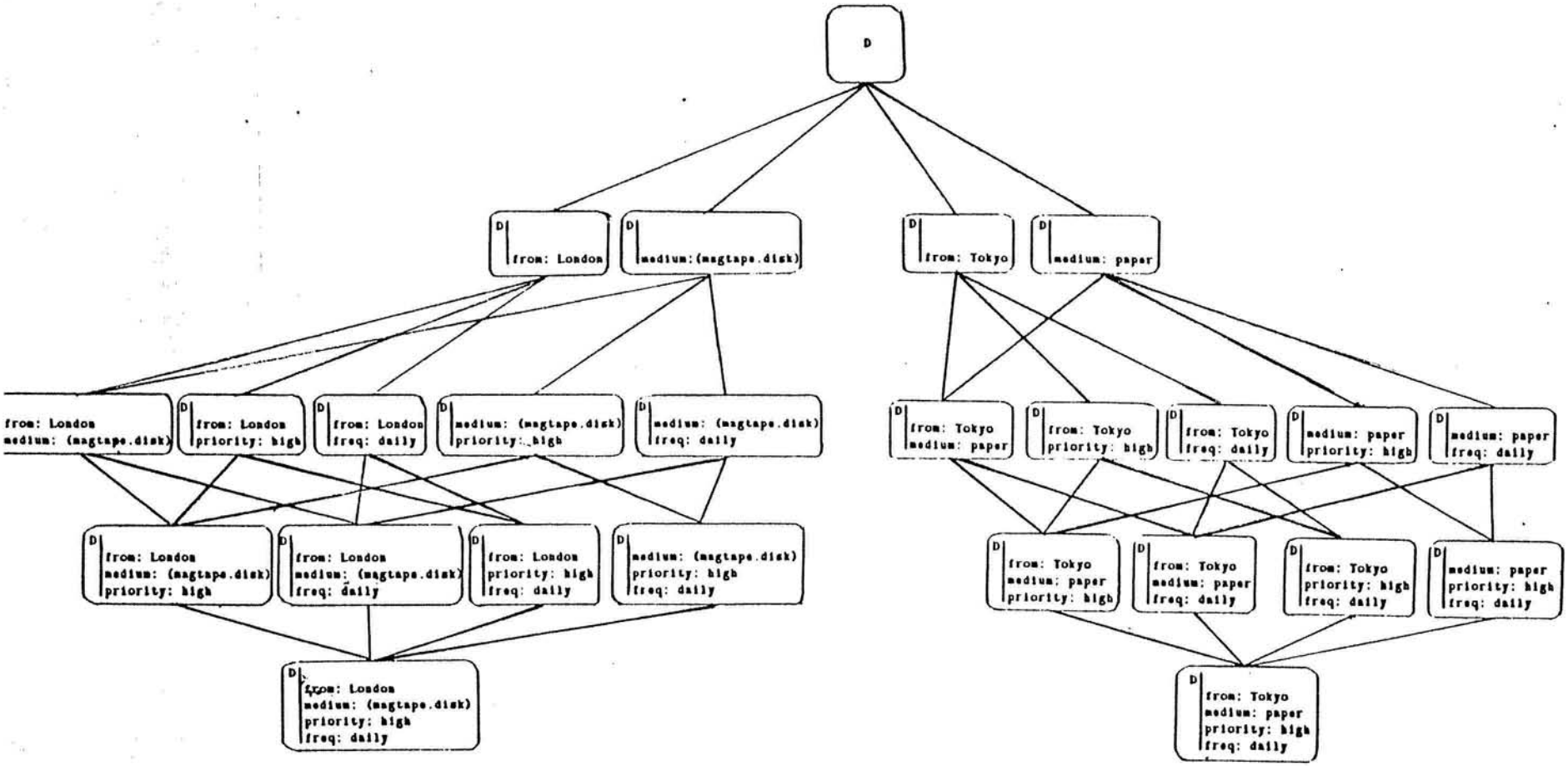

Figure 6. Generalization Hierarchy after E_3.

In comparing this example with 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 from the space, those 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 appropriate level of generality.

4.1. System-Generated Examples

Like other learning formalisms that generalize from examples, the effectiveness of our model is sensitive on 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 that vary only in terms of values of a few attributes 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 the examples alone. From a practical standpoint, however, if we are to use the results of the learning process for analogical reasoning, it is necessary to narrow down to a single hypothesis for each space. For this reason, it is necessary to have a mechanism that overcomes reliance on the examples alone. One way for the system to accomplish this 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 figure 4 where there are several competing hypotheses for Auto-Load-and Edit. Suppose the system 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 figure 5) 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 ?
```

If the users 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 mean s 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).

Given a hypothesis space (i.e corresponding to a design decision/action), from all the plausible hypotheses in that space, one of the possibilities is to begin with the most general or specific situation as the correct one (the one which expresses the rationale for the design decision). If we begin with the most general situation and the user responds negatively to the example, that node can be established as characterizing the most appropriate general class of situations for which the design decision is valid. In contrast, if one starts with a more specific hypothesis, a negative response would be of no value since more general situations might also be appropriate for that action. The system therefore begins with the most general node as the first example.

Since we are trying to generate a negative instance, the node in the example is actually picked from another hypothesis space – a node that "corresponds" to X. This corresponding node, marked "Y" in Figure 5, is one at the same level of generality in another hypothesis space; 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 breath-first. We employ a breath-first strategy. The rationale 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 is likely to differentiate among the transformations. If the example above had proved to be a positive instance, the system would have generated another query using the node "X2" in figure 4 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 breath-first strategy.

Figure 7 shows a generalization hierarchy where the nodes in figure 6 that are not relevant to the design decisions in the examples have been eliminated. For readability, we have relabeled some of the nodes. As we can see, the hierarchy represents the general situations that underlie that part of the design used in the examples.

5. The Model

We now describe the model underlying the learning process illustrated in the example.

5.1. Notation

Let D be the object type with slots $s_1, s_2, ..., s_p$, i.e., D is the p-tuple $<s_1, s_2, ...s_p>$.

Let $d_1, d_2, ...d_n$ be instances of D.

Let $V_{i,j}$ be the value of the slot s_j of instance d_i.

Thus, d_i is the ordered set of pairs $\{s_j{:}V_{i,j} \mid 1 \leq j \leq p\}$

Let $d_i ==> t_c$ indicate a decision "If d_i then t_c".

Let $C_i(k)$ be the set of all subsets of k pairs of d_i, i.e.,

$\forall S \in C_i(k),\ S \subseteq d_i \text{ and } |S|=k$

Let a specialization of D, denoted $D|_{s_j : V} = <s_1, s_2, s_{j-1}, V, s_{j+1}, ...s_p>$

Thus, $C_i(k)$ creates specializations of D at "level" k. Specializations at level k represent types where k slots are constrained to have a fixed value.

Let Φ be a function that maps specializations at level k into subsets of specializations at level k+1.

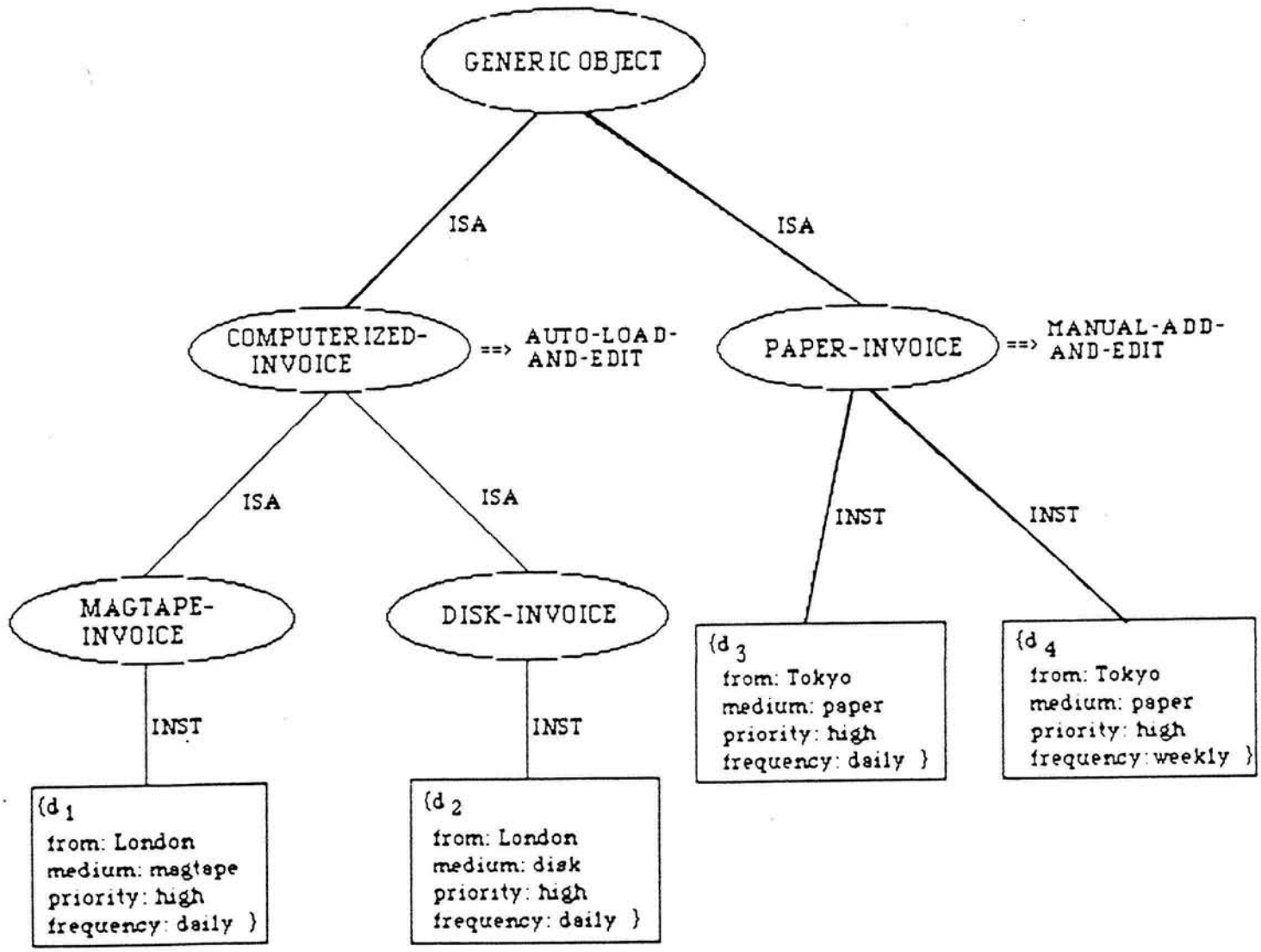

Figure 7. Final generalization hierarchy corresponding to the design examples.

Thus $\Phi : C_i(k) \rightarrow 2^{C_i(k+1)}$ s.t. $S \in \Phi(t)$ where $t \in C(k)$, $t \subseteq S$, and $C(k+1) = C(k)$ for $k \geq p$.

A hypothesis space, Ω_k, corresponding to "$d_i \Longrightarrow t_k$", is the lattice :

$\Omega_k = \{ C(i) \mid 0 \leq i \leq p\}$

which represents partially ordered set of specializations of D.

$\Omega_k(l)$ represents the set of nodes at level l of the hypothesis space Ω_k.

5.2. Algorithm

The following algorithm describes modifications to the set of hypothesis spaces $\{\Omega_1, \Omega_2, \ldots \Omega_m\}$ when an example in the form of "$d_h \Longrightarrow t_c$" is presented.

```
Let I = {t1, t2, ... tm}
Begin
    If tc ∉ I
     Then Begin
            I = I ∪ {tc}
            Ωc = Φ
            For K = 0 to p, Do:
                Ωc = Ωc ∪ Ch(k)
          End
     Else
          Begin
            If tb ∈ I s.t. tb = tc
               and tb is generated from "di ==> tb", do:
                 For k from 0 to p, do:
                     For j from 1 to p, do:
                         If sj:Vh,j ≠ sj:Vi,j
                             then Let Vi,j = Vi,j ∨ Vh,j

            For each t1 ∈ I s.t. t1 ≠ tc, do:
                For k from 0 to p, do:
                  For each El ∈ Ω1(k) and for each Ec ∈ Ωc(k),
                     If El = Ec,
                        then  Mark El and Ec as eliminated.
          End

   End
```

6. Discussion

In general, a system based on the learning by observation model described above is likely to be valuable in problem areas where experts routinely engage in design activity. As a taxonomy of objects in the application domain is gradually synthesized, the system can become useful for reasoning about analogous situations.

Our model of learning has been motivated by Mitchell's (1983) version spaces. The structure of our hypothesis spaces is similar to his version spaces which contain partial orderings of situations (left hand sides of rules) that are hypothesized to account for the actions (the right hand sides – decisions in our problem). The fundamental difference is that while Mitchell's version spaces are generated from a pre-existing generalization hierarchy of carefully selected object types and relationships relevant to a domain, our objective is to *synthesize* such a hierarchy from the hypothesis spaces generated by the examples.

A limitation of the model is that the object classifications it forms are limited by the adequacy of the generic object used to describe the examples. Ideally, the generic object must be supplied with all the properties needed to capture the important features of the examples. We believe this is a reasonable assumption for most domains. However, we are currently working on ways to enable a user to introduce new properties dynamically at any level of abstraction in the generalization hierarchy.[2]

A second limitation is because of the disjunctive problem. In programs that begin with a generalization language, the program makes inductive leaps that are biased by the content and structure of the generalization language (Mitchell, 1983b; Utgoff and Mitchell, 1982). As we illustrated in the example, the disjunctive problem arises in our model in another form: when faced with an "A or B" situation, the program is unable to *create* an appropriate superclass because it has no access to such a vocabulary. One way to create these is by embeding domain knowledge into the program – which runs counter to our goal. Alternatively, the expert could be requested to suggest a category – which is the method we have adopted.

Finally, since the model does not include a scheme for backtracking, a critical assumption underlying it is that there are no inconsistencies in the examples. We are currently investigating ways of incorporating the plausible reasoning machinery of Doyle (1979), McAllester (1982) and others into the model in order to deal with inconsistent examples.

---

[2] Currently, changing a type definition requires restating all previously expressed examples in terms of the modified type definition.

7. Conclusion

This research has been motivated by a real world problem where it is clear that knowledge-based support will be plausible only if a system manages to first acquire sufficient knowledge about the domain from a designer engaged in specifying the design. This is particularly important when domain-knowledge is embeded in decisions. If this decision making is observable by a computer, as we have described, it is possible to extract this knowledge by endowing the computer with the intelligence to learn through observation. Rather than place the burden on the designer to specify all the data types and operations on them, the system is able to infer the appropriate amount of domain knowledge for use in maintenence.

We have presented a model for learning through a process of observing design decisions. These decisions, viewed as examples, result in a space of partial orderings of plausible generalizations. This space is then refined using the constraint information in successive examples. Finally, an attractive feature of the model is that the results are independent of the order of the examples.

References

Belkhouche, B. & Urban, J.E., 1986. Direct Implementation of Abstract Data Types from Abstract Specifications, in *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, May 1986, pp. 649-661.

Borgida, A., Mylopoulas, J., & Wong, H., Generalization/Specialization as a basis for software specification, in *On Conceptual Modelling*, Springer Verlag, New York, 1984.

Davis, R., 1979. Interactive Transfer of Expertise – Acquisition of new inference rules, in *Artificial Intelligence*, Vol. 4.

DeMarco, T., 1979. *Strcutured Analysis and System Specification*, Prentice-Hall, 1979.

Dhar, V. and Jarke, M., 1985. Learning from Prototypes, in *Proceedings of the Sixth International Conference on Information Systems*, Indianapolis, Indiana.

Doyle, Jon., 1979. A Truth Maintenance System, *Artificial Intelligence*, vol 12, number 3, 1979, pp. 231-272.

Gane, C. & Sarson, T., 1979. *Structured Systems Analysis: Tools and Techniques*, Prentice-Hall, 1979.

Lenat, D.B., 1982. AM: Discovery in Mathematics as Heuristic Search, in R.Davis and D.B.Lenat (eds), *Knowledge-Based Systems in Artificial Intelligence*, pp 1-225.

McAllester, D., 1982. Reasoning Utility Package, AI Laboratory Memo 667.

Michalski, R.S., 1980. Knowledge Acquisition through Conceptual Clustering: a theoretical framework and an algorithm for partitioning data into conjunctive concepts, in *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 3. pp 219-244.

Mitchell, T., 1977. Version Spaces: A Candidate Elimination Approach to Rule Learning, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp 305-310.

Mitchell, T., 1983a. Learning and Problem Solving, in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pp 1139-1151.

Mitchell, T.M, 1983b. Generalization as Search, in *Artificial Intelligence*, Vol. 18, No. 2.

Mitchell, T.M., Mahadevan, S., and Steinberg, L.I., 1985. LEAP: A Learning Apprentice for VLSI Design, in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp 573-580.

Orr, K.T., 1981. *Structured Requirements Definition*, Ken Orr and Associates, Topeka, Kansas, 1981.

Simon, H.A., 1977. Artificial Intelligence Systems that Understand, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp 1059-1073.

Sussman, G.J., 1975. *A Computer Model of Skill Acquisition*, American Elsevier, New York.

Utgoff, P.E. and Mitchell, T.M., 1982. Acquisition of Appropriate Bias for Inductive Concept Learning, in *Proceedings of the 1982 National Conference on Artificial Intelligence.*

Waterman, D.A., 1970. Generalization Learning Techniques for Automating the Learning of Heuristics, in *Artificial Intelligence*, No. 1, pp 121-170.

Winston, P.H., 1975. Learning Structural Descriptions from Examples, in P.H. Winston (ed), *The Psychology of Computer Vision*, McGraw Hill, New York.

Yourdon, E. and Constantine, L.L., 1979. *Structured Design*, Prentice-Hall, New Jersey.