ANALOGICAL AND DEPENDENCY DIRECTED REASONING STRATEGIES FOR LARGE SYSTEMS EVOLUTION

Vasant Dhar
and
Matthias Jarke

August 1985

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

Working Paper Series

CRIS #100

GBA #85-66

A shorter version of this paper appears in the **Proceedings of the Sixth International Conference on Information Systems,** 1985.

Table of Contents

Abstract

The maintenance of large information systems involves continuous design modifications to designs in response to evolving business conditions or changing user requirements. Because of the complexity barrier associated with engineering such systems, changes can be *ad hoc* and prone to errors. Based on our observations of such a process in the oil industry, we believe 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 changing requirements, and to design parts of systems that might be "similar" to existing ones. In this paper, we describe a partially implemented 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 as well as the general domain-specific design rules on which such dependencies are based. This knowledge can then be applied to prototype refinement, systems maintenance, and the re-use of existing designs to construct "similar" design fragments.

1. INTRODUCTION

Research in systems analysis and design has resulted in several useful methods for the development of information systems. While these methods are effective in developing initial designs, they neither support the correction of design errors nor changes in previous design choices. As a result, changes in system design tend to be unprincipled, ad hoc, and error prone, failing to take cognizance of the *rationales* for previous design decisions. In this paper, we examine some of these shortcomings and present a knowledge based system architecture called REMAP that alleviates these problems. This architecture supports 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 REMAP architecture has resulted from our observations of a complex system design effort in a large oil company. This study has 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 general application-specific *rules*; however, these rules are seldom articulated explicitly by users or analysts. Second, when systems are developed in a piecemeal fashion following the prototyping idea (Jenkins, 1983), analysts apply *analogies* to transfer experience gained from one subsystem to "similar components" of another.

It seems clear 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, we argue 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 bases for dependencies among design decisions made by the analyst.

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.

In this paper 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 sufficient rich context for discussion, we use parts of the design that were actually developed in the system design in the oil company.

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. A formal model of our approach is presented in section 3, along with an overview of a partial implementation of the REMAP architecture. Section 4 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. A CLASSIFICATION OF 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. A 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, the system representation is restricted to the Structured Analysis level (DeMarco, 1978; Gane and Sarson, 1979). Note, however, that the problems described here, and our approach to solve them, are not restricted to this level but appear in any systems maintenance situation.

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 level 0 of the system. In this example, since Sales comprises the entire system, this can also be used as the 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. Level 2 (figures 4 and 5) are the bottom level decompositions of the bubbles 1 and 3. Each of the bubbles at this level have an associated mini-spec (not discussed here).

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.

DATA FLOW DIAGRAM CONVENTIONS

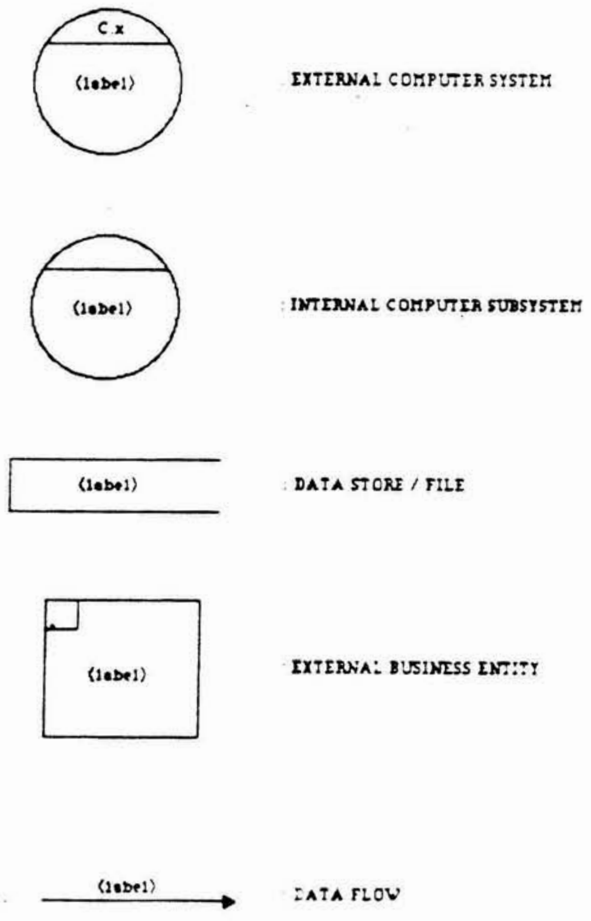


Figure 1

SALES ACCOUNTING SYSTEMS
CONTEXT DIAGRAM

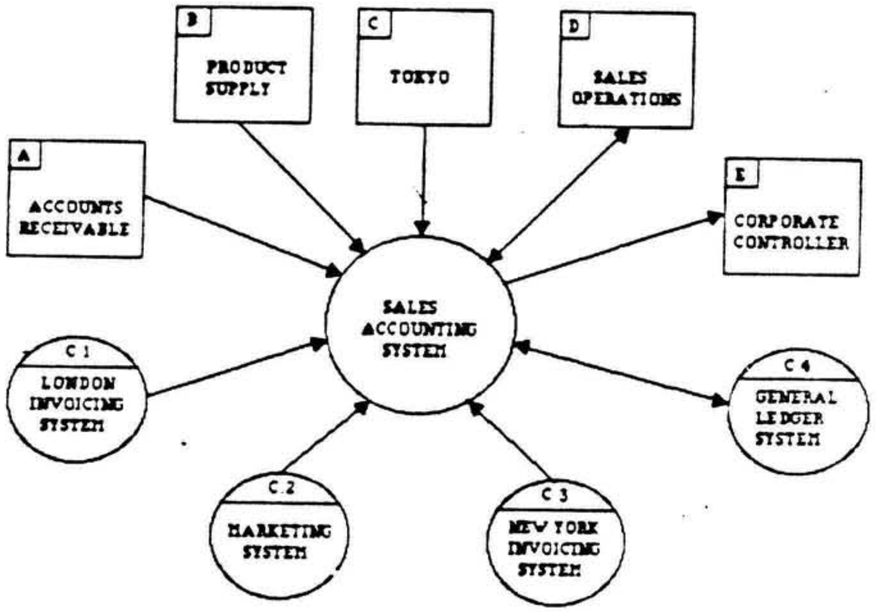


Figure 2

FUELS SALES (INITIAL)

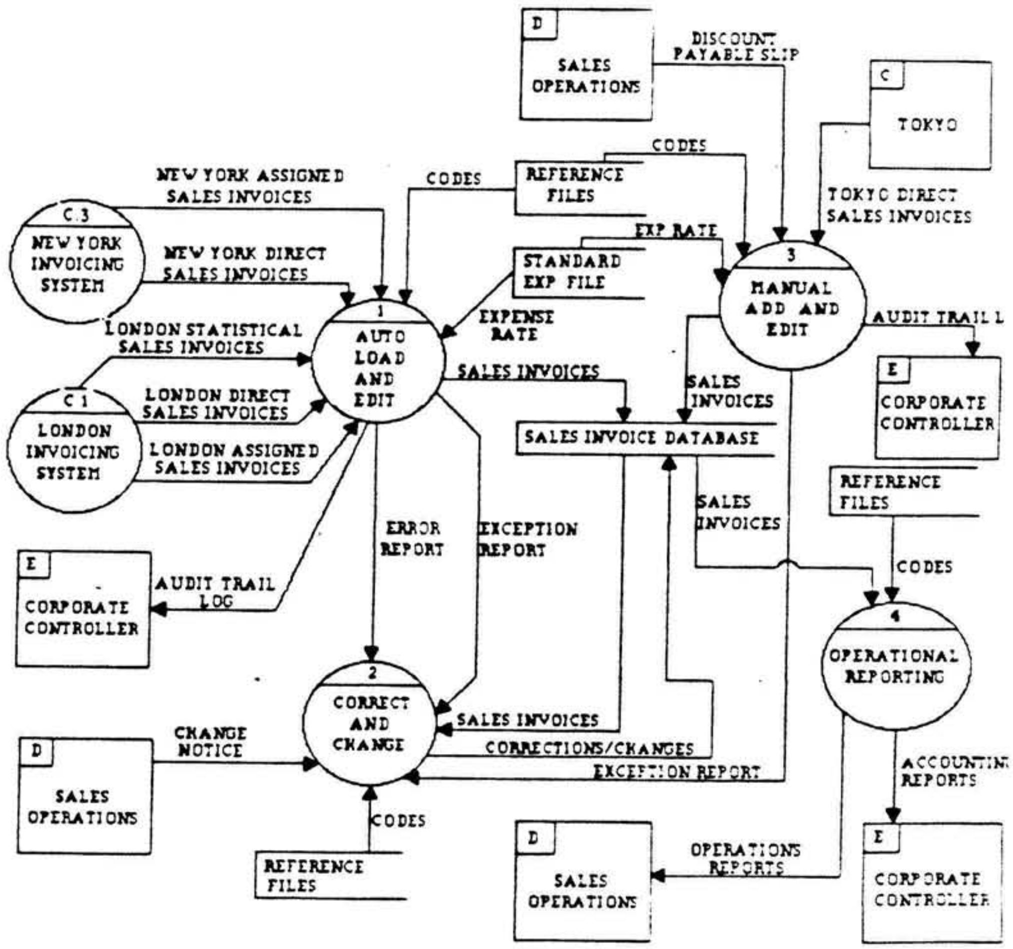


Figure 3

AUTO LOAD AND EDIT

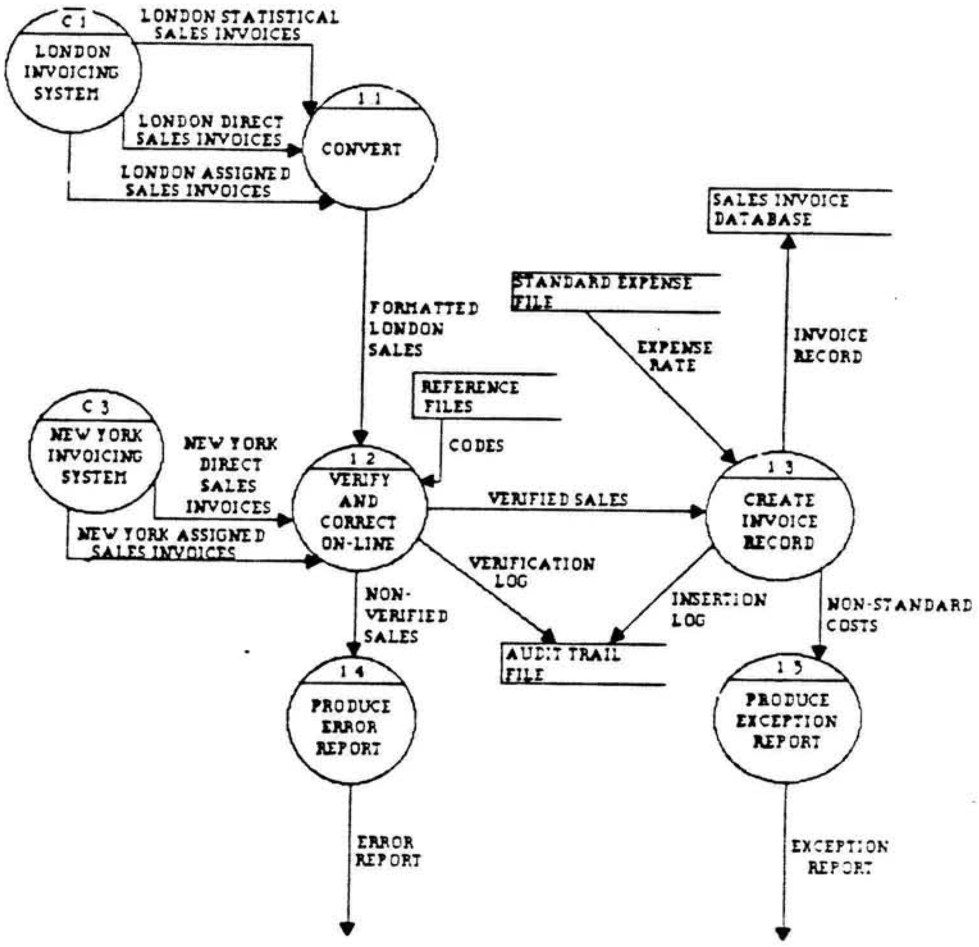


Figure 4

MANUAL ADD AND EDIT

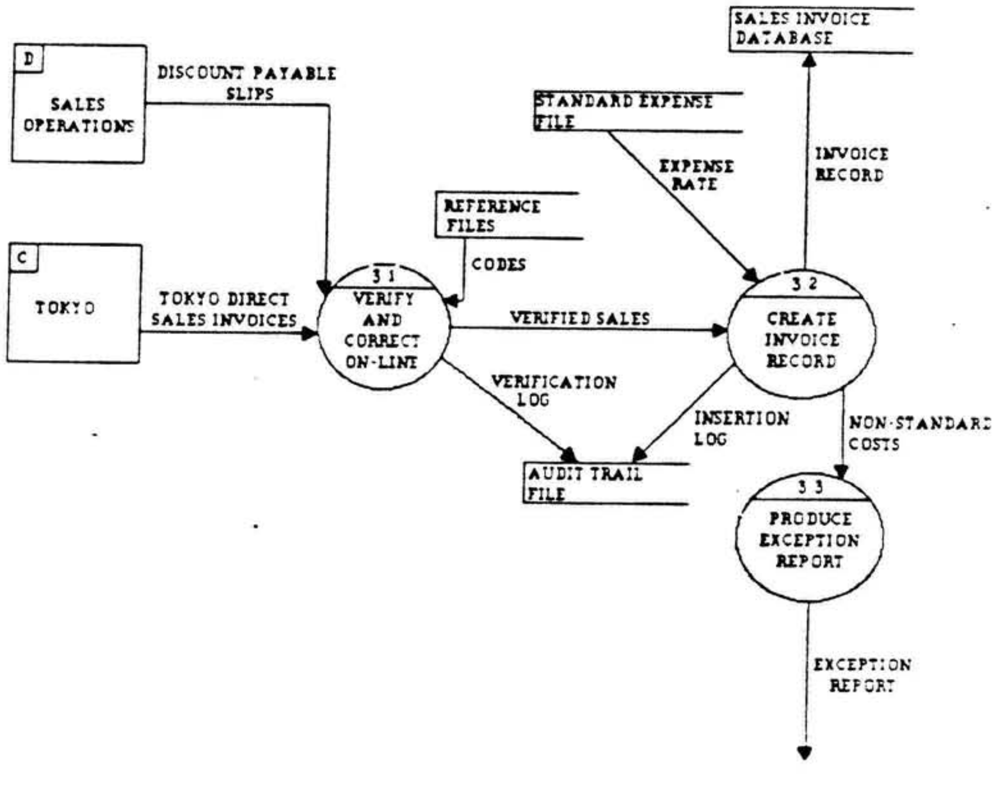


Figure 5

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 existence of the dataflows representing London invoices. On recognizing that London invoices are now not unformated, it should be able to detect the fact that conversion is unnecessary. Further, it show 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 rationales 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 design changes as well as major related modifications are needed in several sections of the code. In this case, 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.

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.

[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.

FUELS SALES (MODIFIED)

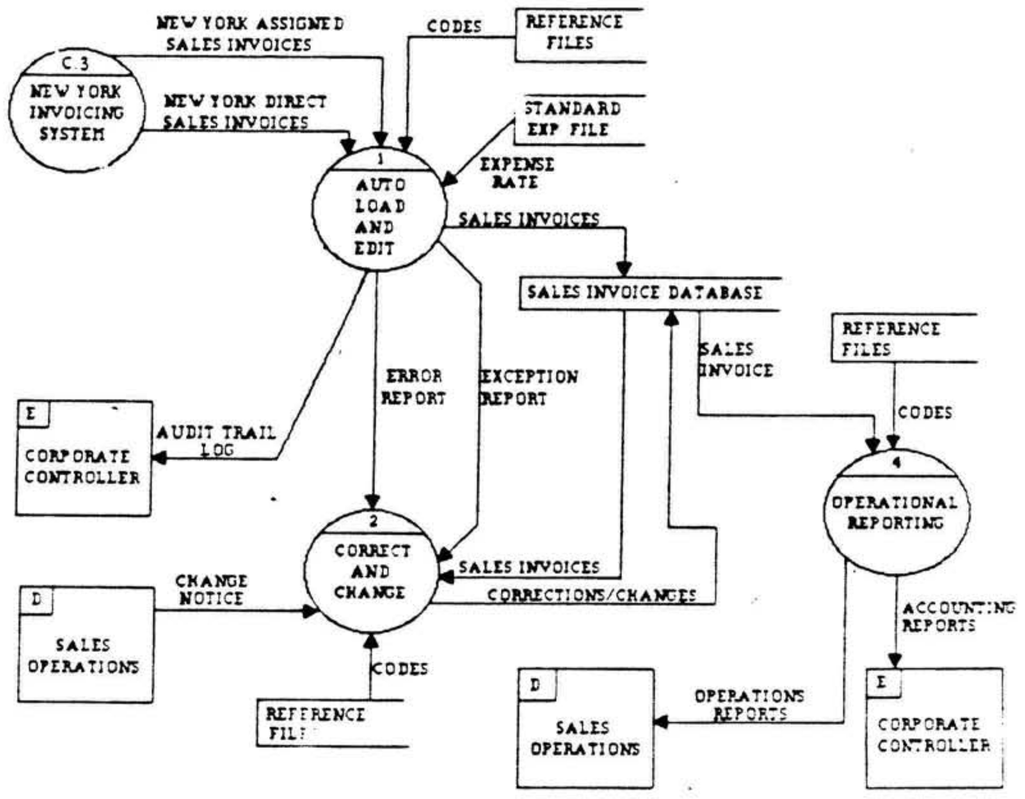


Figure 6

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. 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 (Simon, 1981), 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 rationale for a particular design follows from a subjective world-view of the designer.

If a program 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 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. REMAP: ARCHITECTURE AND IMPLEMENTATION

REMAP (REpresentation and MAintenance of Process knowledge) is a knowledge based system designed to address the needs identified in the previous section. It is apparent from the examples that application-specific knowledge plays a key role in reasoning about a design. This raises an important question, namely, how is this knowledge to be *acquired* by REMAP?

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 might be extracted and 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, and describe a partial implementation of some key features of the system.

3.1. Representing Designs Using Structured Objects

The REMAP model centers around *design objects*. The designer defines *instances* of such objects, whereas 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 since it is at the top of the hierarchy, 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 the designer, the system should be provided with a small initial knowledge base. In the Structured Analysis example used throughout this paper, this consists of the definition of object types corresponding to data flow diagram conventions. The five major components are defined below (cf. figure 7):

INITIAL OBJECT TYPE HIERARCHIES

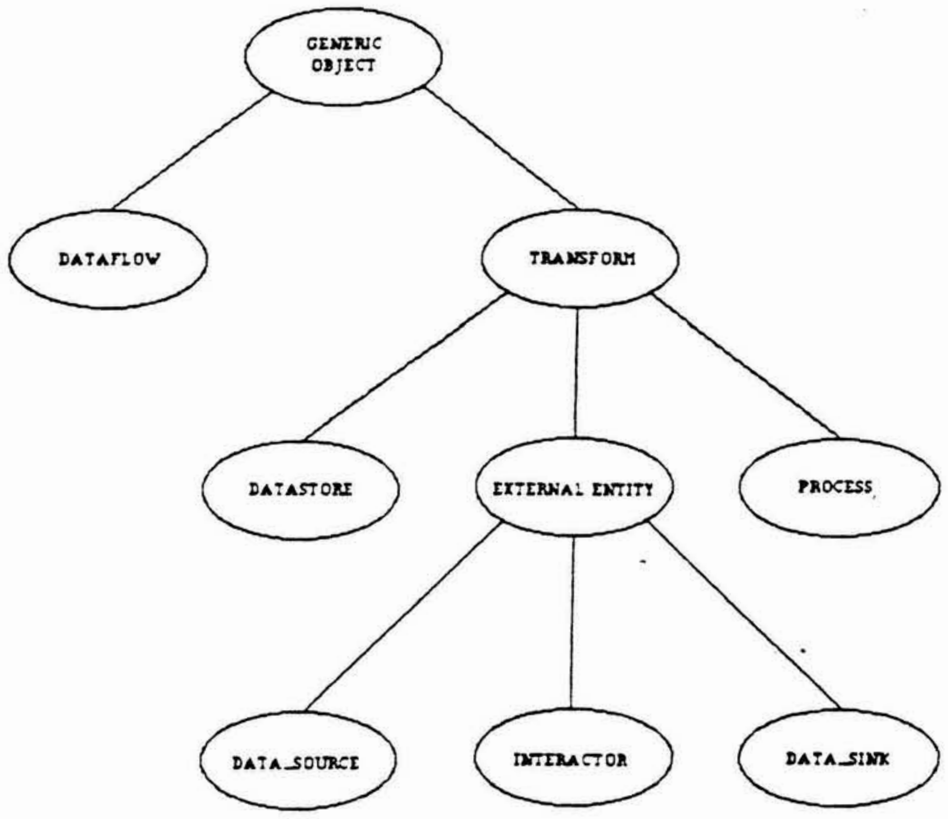


Figure 7

INITIAL GENERALIZATION HIERARCHY

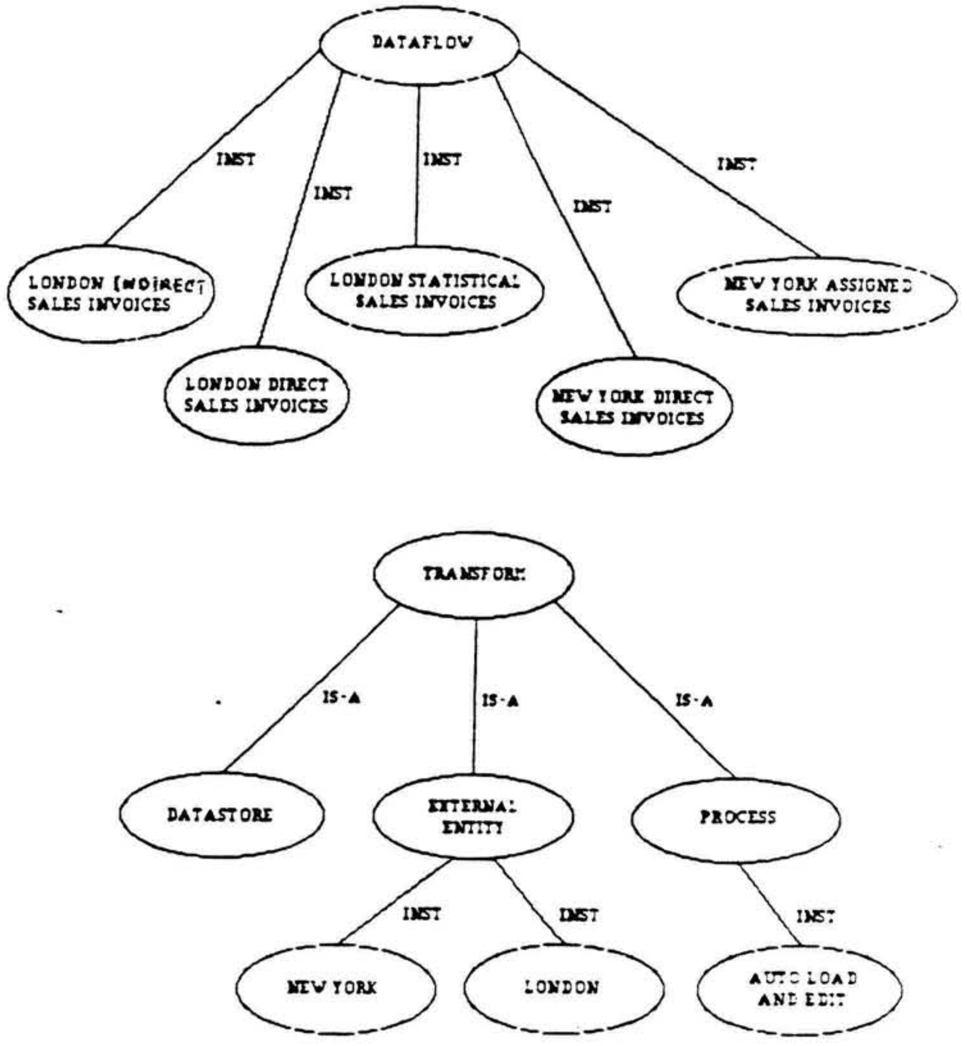

DATAFLOW
INST
INST
INST
INST
INST
LONDON INDIRECT SALES INVOICES
LONDON DIRECT SALES INVOICES
LONDON STATISTICAL SALES INVOICES
NEW YORK DIRECT SALES INVOICES
NEW YORK ASSIGNED SALES INVOICES
TRANSFORM
IS-A
IS-A
IS-A
DATASTORE
EXTERNAL ENTITY
PROCESS
INST
INST
INST
NEW YORK
LONDON
AUTO LOAD AND EDIT

Figure 8

```
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 broken down into 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.

This representation allows us to define data flow diagrams completely. 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, application-specific information is 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 (i.e., without using the "because-of" slot). To reason about such situations, additional knowledge structures are required, which we describe below.

3.2. Representing Rationales as Dependencies

Design decisions at the Structured Analysis level define bubble and dataflow objects. The *rationale* or *justification* of a decision consists, in turn, of other decisions. 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 is justified by the existence of New York and London invoices, which form its "set of support" (Doyle, 1978) or the cumulative reason for its existence. The convert operation is justified because London sales invoices are not formated correctly. Similar dependencies can be identified for other decisions.

The complete dependency network corresponding to a design may be viewed as incorporating the overall *purpose* of a set of design decisions. The general form of a dependency is:

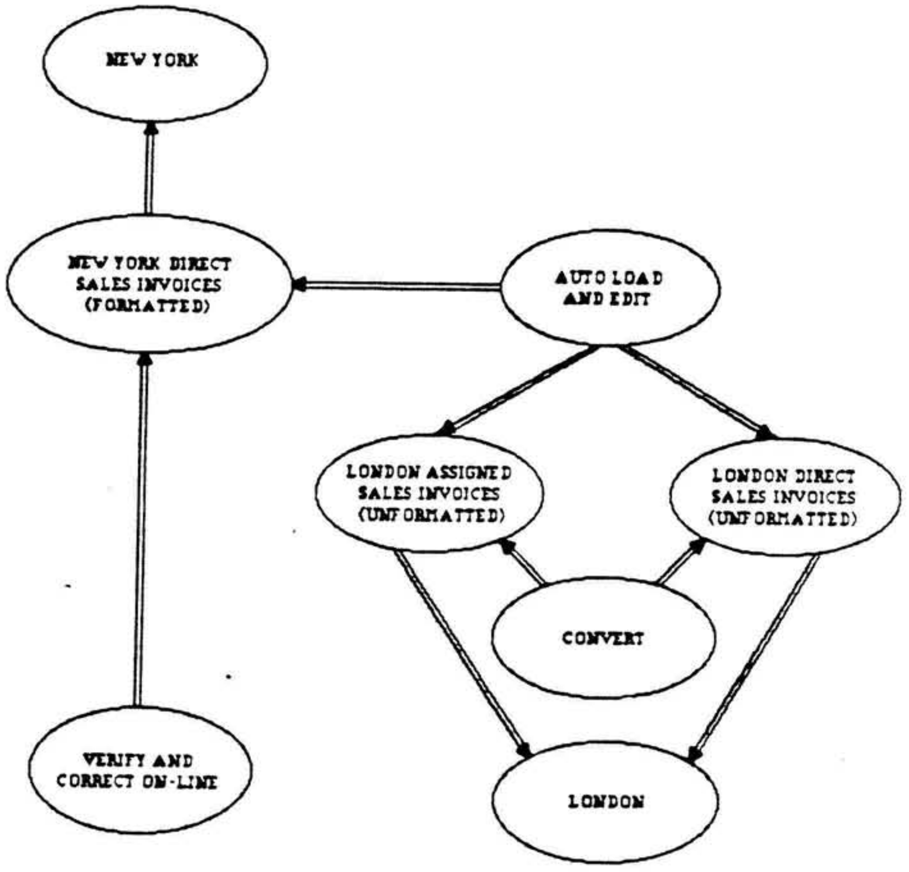
```
(<decision> <justification>)
```

where <decision> and <justification> are both object instances. In REMAP, each design object maintains a cumulative set of justifications in its *because-of* slot that constitutes its set of support.

In order to demonstrate the usefulness of this dependency network, let us 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, an existing dependency network such as the one on figure 9 can be used to assess certain ramifications of a change, a process commonly referred to as *belief maintenance* (Doyle, 1978). In the above example, conversion is *not* required for London

A DEPENDENCY NETWORK


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

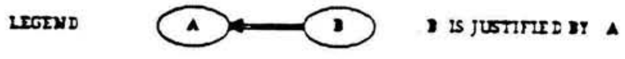


Figure 9

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, additional knowledge of a more general nature 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 (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, or office originating the invoice. 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. If the system is to be able to *learn* anything from existing designs, it must also have access to the general *rules on which the dependencies have been based* because the rules differentiate the important features of the relationship from the incidental. It must generalize the dependencies, which can be thought of as *examples*, into rules.

There are two approaches to example based generalization described in the literature. The controlling factors that dictate which of the two approaches to adopt are the *number of examples* that are available, and the extent of an *underlying theory* in the domain under consideration.

For domains with little or no theory, it is desirable to have a very large number of examples from which to generalize. In such situations, generalization involves detecting patterns in the data, which is essentially a process of *theory formation*. Examples of this approach can be found in Michalski (1983), Langely et. al (1983), and Smith (1980). Borgida and Williamson (1985) apply the same idea to the reorganization of database schemas based on the analysis of exceptions in a current schema.

In contrast, for generalization based on few or single examples, it is best if there exists a *strong theory* of the domain. The example can then be verified for correctness using the theory, and then generalized into a useful form (i.e. a rule) for *subsequent use* by maintaining only the essential features of the example -- where these features are derived as a side effect of applying the theory. Mitchell et.al (1985) and Mahadevan (1985) describe this approach in the context of digital circuit theory.

Unfortunately, neither of these approaches appears feasible for our problem since application areas are too diverse and idiosyncratic to provide a theory, and examples are few. This forces us to make a somewhat strong assumption about the system design process, namely, that when the designer specifies the purpose of a design fragment as part of an initial system design, the rationale is a reasonably accurate one. While the validity of this assumption is debatable in principle, it appears to be a reasonable one in practice (Yourdon, 1976) since the analysis and design process involves extensive exchange between users and analysts -- by the time design specifications are actually articulated in terms of a structured methodology (like SADT, or DFDs), there are well defined rationales for them. In light of this, REMAP requires the designer or user to generalize specific dependencies to design rules during the process of system analysis and design. This requires articulation of the justifications for choices, as well as of the general basis for the justifications. In effect, the "explanation" for the dependency forms the basis for constructing the rule.

A more crucial issue however, is what *form* these rules might take. On the one hand, the rule can be expressed in terms of objects and their slot values, for example:

```
{dataflow
        medium: computerized} ==> verify on line

{dataflow
```

```
        medium: paper} ==> perform conversion
```

If the medium slot has not been defined before, the type definition of dataflow can first be extended to include it. Nevertheless, there is a major problem with this scheme. Recall that so far, the generalization hierarchy for dataflows is extremely shallow including only one type, namely the dataflow (cf. figure 8). Adding additional slots for each rule will soon yield very complex object types. In looking at the different invoices -- which are instances of type dataflow -- it is apparent that *different* attributes are relevant in describing the various instances. For example, paper invoices might be distinguished by their *color*, an attribute that is irrelevant for describing computerized invoices. Thus, most slots in the extended dataflow type definition would remain unfilled for many objects.

This situation can be expected to occur in the early stages of the system analysis process, when the system is still unfamiliar with the application area. New design decisions could be added and instantiated as instances of an existing type although they differ qualitatively from other instances, and might therefore be better off described in terms of a different bundle of attributes.

When instances vary sufficiently, this indicates that the generalization hierarchy must be extended to include more specific subtypes. For example, extending the generalization hierarchy in figure 8 would involve creating two new types, namely paper-invoices and computerized-invoices and re-classifying the existing instances in light of this new classification. Further, computerized-invoices can then be broken down into magnetic-tape-invoices and on-line-invoices if appropriate.[2] The reconfigured generalization hierarchy would then appear as in figure 10, and in contrast to the rule representation above, the rule could

[2]This raises the following question: how might the program differentiate among situations where the generalization hierarchy should be extended versus those where little is to gained by extension? Although we have yet to address this question adequately, it appears that a reasonable heuristic for deciding when to extend the generalization might be based on the need for additional slots to differentiate newly defined object instances.

RECONFIGURED GENERALIZATION HEIRARCHY

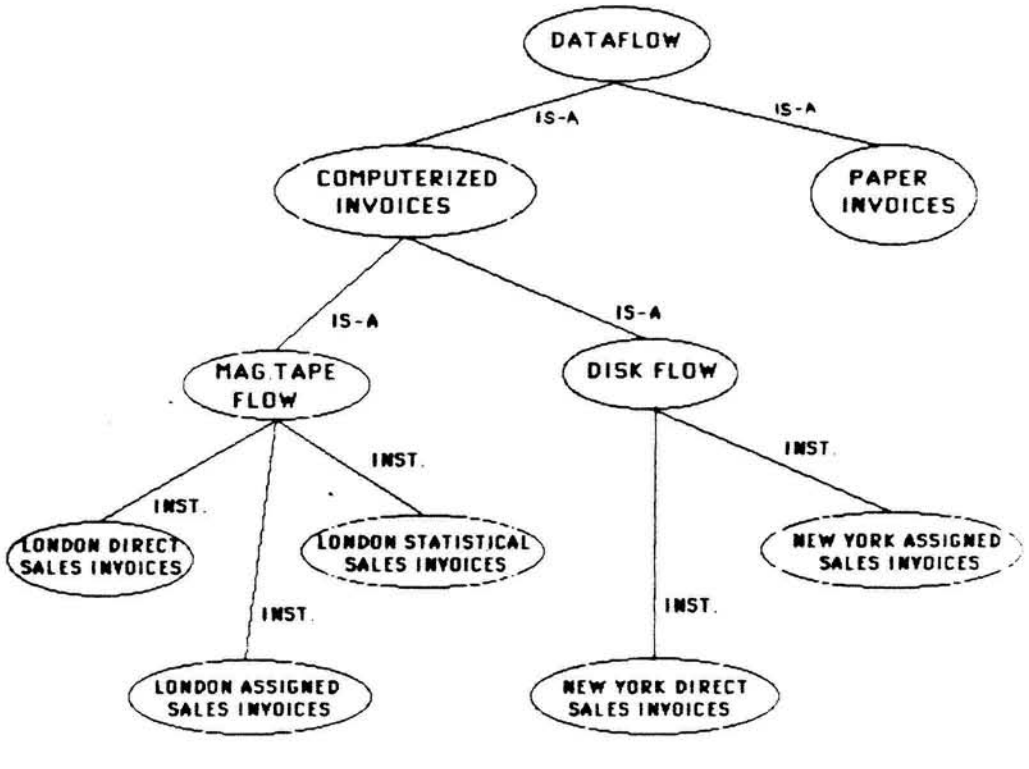

DATAFLOW
IS-A
IS-A
COMPUTERIZED INVOICES
PAPER INVOICES
IS-A
IS-A
MAG.TAPE FLOW
DISK FLOW
INST.
INST.
INST.
INST.
INST.
LONDON DIRECT SALES INVOICES
LONDON STATISTICAL SALES INVOICES
LONDON ASSIGNED SALES INVOICES
NEW YORK ASSIGNED SALES INVOICES
NEW YORK DIRECT SALES INVOICES

Figure 10

then be stated in terms of the newly defined object types.

To illustrate, such rules might appear as:

```
{computerized-invoices} ==> perform auto-load-and-edit

{paper-invoices} ==> perform manual-add-and-edit
```

It should be possible to use these rule structures in two ways. First, if an operation such as auto-load-and-edit is part of a design and has one or more computerized inputs coming into it, these should be added automatically to the operation's set of support. Second, if no such inputs are in the design, the rule can be used to compare "expected" reasons for the operation to the justifications provided by the user, or to suggest changes in designs that appear "inconsistent" with the knowledge in the rules.[3]

3.4. Analogical Reasoning Using Object Classification and Rules

In section 2, we 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 program must be able to utilize its knowledge concerning the purpose of "similar" design fragments. Specifically, it must determine what features of the new situation are the same as objects it already knows about, and then attempt to learn about the unique features of the new situation, represented in terms of one or several design objects.

As we have pointed out, knowledge about the various design objects is organized in the form of a generalization hierarchy, with rules referencing nodes in this hierarchy. 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 11, and computerized

[3]This assumes that the rule is "correct". An existing rule that turns out to be inaccurate, leads to a "contradiction" in which case the rule can be discarded by the belief maintenance machinery, or refined interactively.

but unformated invoices come in on magnetic tape from Venezuela, they would be classified as a "Magnetic-tape-invoices" dataflow. Once the most specific level is found, rules referencing objects at that level can be applied. For example, if Venezuela invoices had been formated, a rule calling for a "verify and correct on line" operation would be applied. Similarly, if a match on other attributes is found, more specific rules can be applied. In general, it is likely that rules applicable to some of the attributes of a new object will be found, while the system will not have rules dealing with others. For these, new rules must be extracted from the user.

If no rules are applicable to the newly defined object at the most specific level, the system can look for *more general* rules that might be applicable. Specifically, this involves moving up the generalization hierarchy until a rule is found that is applicable at a higher level of generality. In the example considered above, this would involve 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 an auto-load-and-edit operation be performed on them.

It should be noted that even though there may *not* be an object in the design that is similar to the new one, existing rules might still apply. For example, London invoices were previously unformated; this had required a convert operation which was subsequently eliminated when the form of these invoices was changed. However, the more general piece of knowledge that should have been extracted at that time in the form of a rule, now becomes applicable to Venezuela invoices.

Finally, we should distinguish between the analogical *reasoning* procedure described above, and the *learning* by analogy procedures of Winston (1979) 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. Drawing analogies between the examples leads to rule formation in the target domain. In contrast, analogical reasoning, as we have described it, involves determining what *applies* to a new situation *given* what is already known in the same domain. This requires breaking down the new situation into parts that the system already knows how to deal with. In our representation, this requires matching the new situation successively against more general knowledge about the application by moving up the generalization hierarchy. The procedure is described more formally in the following subsection.

3.5. REMAP Control Structure

In order to incorporate new knowledge and to reason about user critiques, the model requires an overall control structure that enables it to switch among design support and knowledge acquisition modes. Figure 11 provides a high-level transition network representation of the main modes.

The *add* mode is the usual starting point for a new system. The designer can add a set of proposed new design objects and their associated dependencies. The system can invoke the *analogical reasoning* mode to assist in this task to whatever extent possible. The *belief maintenance* mode is responsible for checking the consistency of proposed changes with respect to existing object types and rules. The *learning* mode interacts with the user in order to establish a generalization of dependencies that are not derivable from existing rules, possibly adding new rules and specifying new object types. The system then moves into the belief maintenance mode in order to check the compatibility and consequences of the newly acquired knowledge.

If there is an existing design to be improved, or reused for another system, the system will start in the *critique* mode. Here, the designer may want to change or add to certain parts of the design. Again, feasibility and possible learning opportunities induced by the change can be studied in the belief maintenance (for a formal description of belief maintenance

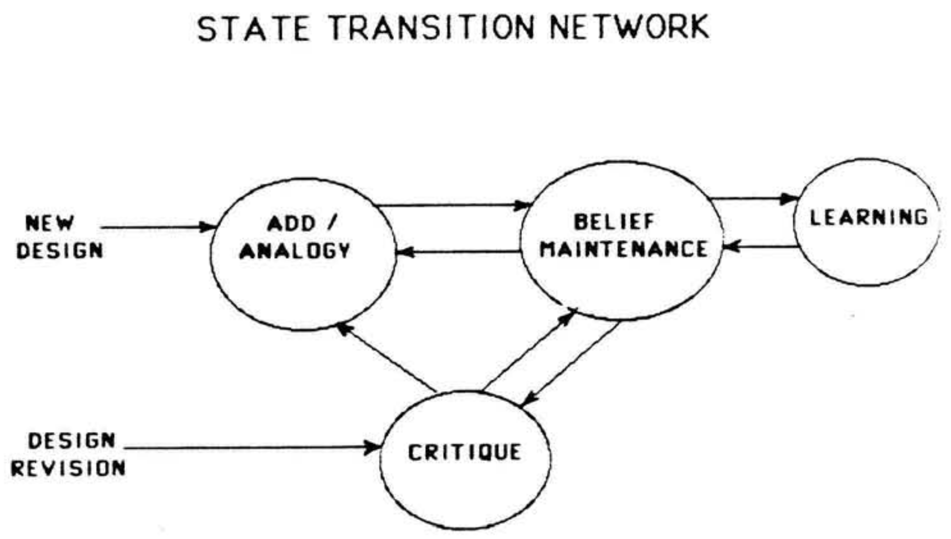

STATE TRANSITION NETWORK
NEW DESIGN
ADD / ANALOGY
BELIEF MAINTENANCE
LEARNING
DESIGN REVISION
CRITIQUE

Figure 11

algorithms, see Doyle (!979) and McAllester (1980)) and learning modes. The interaction of these components of the REMAP architecture is described below in "Structured English."

```
Add-mode:
1. DOWHILE user is entering object instances.
2. Accept object instances.
3. Invoke Analogical-Reasoning-mode
4. IF enabling conditions of a rule are
   satisfied by instances
     THEN 4a. Create dependencies generated by rule.
          4b. Invoke belief maintenance.
     ELSE 4c. Accept dependency.
          4d. Invoke Learn-mode

Learn-mode:
1. Extract essential features (slot values) of objects.
2. IF slot value is an object instance
      THEN 2a. Note its type
      ELSE 2b. IF needed slot does not exist
                  THEN Create-new-type-mode.
3. Propose generalization (rule) in terms of the identified
   or defined types.

Create-new-type-mode:
1. Record context (slot values) of object instance.
2. Define new data type corresponding to relevant slot of
   this instance. Establish an IS-A link to
   parent-of of the object instance.
3. Create a new instance of the new data type.
4. Assign slot values to the new instance
   corresponding to  the old instance.
5. Destroy the old object instance.

Critique-mode:
1. Accept user critique in the form of negation to existing
   decision, or addition to design.
2. IF negation
     THEN invoke belief maintenance
     ELSE invoke Add-mode.
```

Analogical-Reasoning-mode

```
1. Identify lowest level in generalization hierarchy into
   which the newly defined object can be categorized.
   Call this the focus.
2. While focus is not the root of the generalization
   hierarchy:
        2a. Find and instantiate rule applicable at focus.
        2b. IF applicable rule does not include all
                attributes of newly defined object
              THEN Invoke Learn-Mode.
```

3.6. Implementation Status

Some key aspects of the REMAP architecture have been incorporated in a small system intended to test their feasibility. The system contains an implementation of the object type hierarchy and an initial knowledge base about data flow diagrams. Knowledge is represented using FLAVORS (Moon and Weinreb, 1981), a LISP-based utility that supports object-oriented programming. An annotated example interaction is displayed in Exhibit 1. The example demonstrates some features of the various modes of figure 11, namely the "add" and "learn" modes in addition to setting up the dependencies. The user begins by informing the system about a new design fragment, namely, London invoices. Specifying the type of object being defined causes the system to query the user about the relevant attributes. The generalization hierarchy is then augmented with the new objects, and justifications including their general bases (the rules) are extracted. In summary, the implementation so far has the capability to accept data flow diagram object instances, to generalize dependencies to rules, and to expand the generalization hierarchy.

A feature currently being added is a connection between FLAVORS and McAllester's (1982) belief maintenance system, RUP, so we may utilize its belief maintenance machinery. In addition, a more habitable user interface is under development. In matching "similar" situations to ones known to REMAP (i.e. the analogical reasoning mode), we intend to use methods similar to Winston's analogical reasoning (Winston, 1979). Specifically, in

To begin with, the user may wish to enter a description of a fragment of the data flow diagram. In this example, we define the fragment in level 1 corresponding to the auto-load-and-edit. The "*" sign represents the system top level prompt. User input is underlined. Annotations to the dialogue are provided in parentheses.

```
* add
        type: external.entity
        name: London
        output: London-direct-sales invoices
                  London-assigned-sales-invoices
                  London-statistical-sales-invoices

        type: dataflow
        name: London-direct-sales-invoices
        from: London
        to: auto-load-and-edit
        medium: magtape

        o
        o
```

(Similarly, other dataflows are entered.)

```
        type: process
        name: auto-load-and-edit
        inputs: London-direct-sales invoices ...
        outputs: Sales-invoices Error-report Exception-report
        because-of: London-direct-sales-invoices
                      London-assigned-sales-invoices
                      London-statistical-sales-invoices
                      NY-direct-sales-invoices
                      NY-assigned-sales-invoices
```

(The "because-of" slot simply records the justifications for performing an auto-load-and-edit operation. At this point, on the basis of the information supplied by the user, the system augments the generalization hierarchies as indicated in figure 9. Now having noted the justifications, the system proceeds with attempting to construct rules based on generalizations of these dependencies.)

Identify the essential features of the object required for the dependency:

```
(dataflow name: London-direct-sales-invoices
          medium: magtape
          from: London
          to: autoload-and-edit)
 ==> (auto-load-and-edit)

: magtape London
```

(The user states that the fact that the input is on magnetic tape justifies the auto-load-and-edit operation (a more general reason, not expressable here, is that the input is *computerized* as opposed to manual). Also, the user states that the operation is required because there's an input from London. The system attempts to "generalize" this by considering whether this is true for inputs from external entities in general, as we see below.)

```
London is an instance of an external.entity.
Is the following rule true:
        ((magtape)(external.entity)) ==> auto-load-and-edit
: yes
```

(Having established this rule, the system will go through a similar process for other inputs that are not similar to the one just considered since the dependencies established by them are explainable by the rule just established. In this example, it would skip the remaining London invoices and proceed to establish the relationship between auto-load-and-edit and New York invoices.)

Exhibit 1

determining similarities, it will be necessary to determine the *important* aspects of a situation, and use them together with classification based knowledge in order to assess the correspondence of situations.

Although we believe that the current system demonstrates the feasibility of some important features of our model, substantial additional research will be required prior to a full-scale implementation. Leaving aside interface issues such as graphics input and output, there is a need to extend two aspects of the model itself.

On the one hand, the type hierarchy may grow very quickly; methodologies will be needed to keep it at a manageable size, and to help the user locate the correct type for a new instance. One way to reduce the number of types would be to allow multiple typing of instances and/or multiple inheritance in the type hierarchy. However, multiple inheritance introduces its own set of problems, such as inconsistency and search complexity.

On the other hand, the present model only handles justifications for existing design decisions. A more prescriptive approach could associate feasibility bounds for subsequent design decisions with an object. For example, it should be possible to state that the choice of paper invoices *exclude* the use of the auto-load-and-edit process, even if the alternative, manual-add-and-edit, has not yet been defined elsewhere. A further step is the *automatic choice* between several feasible alternatives (Dhar and Quayle, 1985), accomplished by the incorporation of a new object type *goal* into the model. A design object might be declared dependent on a goal object if it is the *optimal solution* with respect to that goal. As requirements change, design choices may have to be revised due to different trade-offs.

4. DISCUSSION

The REMAP concept 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. Other important features of such an environment such as language interfaces, editors, version controllers etc. (Konsynski, 1984) have yet to be interfaced to our system but are not currently the focus of this research.

The importance of REMAP's objectives is confirmed by two recent requirements studies on specification-based computing environments (Balzer, et.al, 1982) and on Artificial Intelligence tools for design support in general (as contrasted to information systems design) (Mostow, 1985). Balzer et.al. emphasize the need for supporting systems evolution at the design level as well as at 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 to the first goal by maintaining an evolving object type hierarchy (which for instance, would allow the definition of a new design language other than data flow diagrams), whereas the second is achieved by making each designer's justifications for design fragments explicit. Mostow (1985) also stresses the need for making design goals, design decisions and their rationales 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 (Narayanaswamy, et.al. 1985), and to other design objects at higher levels of abstraction. Integrated environments such as TRW's Software Productivity System (Boehm et.al, 1982) or TEDIUM (Blum, 1983) also allow the designer to relate design objects, programs, and test cases handicapped by the lack of a precise requirements specification language (Borgida et.al. 1984), and because the relationship between requirements and designs is not explained in terms of design decisions and their rationales.

Proponents of prototyping (Naumann and Jenkins, 1982) claim that systems changeability is automatically achieved or substantially supported through the prototyping process and cite case studies in support of this claim (Appleton, 1973). However, others have recognized that in complex systems, the prototyping idea must be applied at multiple levels of abstraction (Groner et. al, 1979). This in turn, requires substantial control of the process, taking into account the design rationales and rules learned from errors in previous prototypes (Dhar and Jarke, 1985). 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 (Jarke and Shalev, 1984; Minsky and Borgida, 1984; Morgenstern, 1983), this approach assumes that such constraints can be enumerated a priori. A more ambitious approach, embodied in the PLEXSYS project (Konsynski et.al, 1984) integrates constraint management into a full design support environment. PLEXSYS' dynamic metasystems (Kotteman and Konsynski, 1984) have represent 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 Davis' (1979) idea of using "meta models" to maintain and reason about object level knowledge contained in the MYCIN system (Shortliffe, 1976). 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 (Hewitt, 1985) 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, our approach has much in common with the Programmer's Apprentice (PA) project (Shrobe, 1979; Waters, 1982; Rich, 1984). 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 about evolving programs. However, there are two important differences. Our focus is on the more abstract parts of the design as opposed to 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 must build up this knowledge on the basis of application-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.

5. 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 is 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% and probably as much as 70% of software costs go into maintenance. Yet, problems of design evolution have not been adequately addressed by previous methodologies, whereas they constitute the focus of our approach. The work reported here is considered a first step towards 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; furthermore, the method also provides guidance in the propagation of proposed changes.

Finally, the analogy-based reasoning component of the method supports the reuse of code and designs in systems that are similar to existing ones. It also provides the designer of such systems with access to the rationales 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

Appleton, D.S., System 2000 Database Management System, GUIDE 37, Session IS-23, Cambridge, MA, November 1973.

Balzer, R., Dyer, D., Fehling., and Saunders., 1982. Specification-based computing environments, Proceedings 8th Very Large Data Base Conference, Mexico City, pp. 273-279.

Blum., B.I., 1983. A Workstation for Information Systems Development, Proceedings of the 7th IEEE COMPSAC Conference.

Boehm, B.W., Elwell, J.F., Pryster, A.B., Stuckle, E.D., and Williams, R.D., 1982. The TRW Software Productivity System, Proceedings of the 6th International Conference on Software Engineering.

Borgida, A., Greenspan, S., and Mylopolous, J., 1985. Using Knowledge Representation for Requirements Modeling, IEEE Computer (Special Issue on Requirements Modeling).

Borgida, A., and Williamson. K., 1985. Accomodating Exceptions in Databases and Refining the Schema by Learning from them., Proceedings of the 11th VLDB Conference, Stockholm, Sweden, August 1985.

CGI Systems Inc., 1984. Presenting PACBASE. Systems Development Software from CGI, Pearl River, NY.

Davis, Randall., 1979. Interactive Transfer of Expertise -- Acquisition of new inference rules, Artificial Intelligence, No.4.

De Marco, T., 1978. Structured Analysis and System Specification, Yourdon Press, New York.

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

Dhar, V., and Quayle, C., 1985. An Approach to Dependency Directed Backtracking Using Domain Specific Knowledge, in Proceedings of the 9th Joint International Conference on Artificial Intelligence (IJCAI), Los Angeles, CA.

Doyle, Jon., 1978. A Truth Maintenance System, AI Laboratory Memo 521, MIT

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

Greenspan, S., 1984. Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, Ph.D Thesis, Technical Report CRSG-155, University of Toronto.

Groner, C., Hopwood, M.D., Palley, N.A., and Sibley, W., 1979. Requirements Analysis in Clinical Research Information Processing -- a Case Study, IEEE Computer 12,9.

Hewitt, Carl., 1985. The Challenge of Open Systems, BYTE Magazine, April

Jarke, M., and Shalev, J., 1984. A Database Architecture for Supporting Business Transactions, Journal of Management Information Systems 1, 1, pp. 63-80.

Jenkins, Milton A., 1983. Prototyping: A Methodology for the Design and Development of Application Systems, Working Paper #227, Graduate School of Business, Indiana University, April 1983.

Konsynski, B.R., 1984. Advances in Information Systems Design, Journal of MIS, 1,3.

Konsynski, B., Kotteman, J., Nunamaker, J., and Stott, J., 1984. PLEXSYS-84: An Integrated Development Environment for Information Systems, Journal of Management Information Systems, volume 1, No. 3, Winter 1984-85.

Kotteman, J.E., and Konsynski B.R., 1984., Dynamic Metasystems for Information Systems Development, Proceedings of the 5th International Conference on Information Systems, Tucson, Az, pp. 187-204.

Mahadevan, S., 1985. Verification-based Learning: A Generalized Strategy for Infering Problem-Reduction Methods, in Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA.

Martin, J., 1982. Application Development Without Programmers, Prentice-Hall.

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

McCracken, D.D., 1980. A Maverick Approach to Systems Analysis and Design, Conference on Systems Analysis and Design: Foundation for the 1980s.

Michie., 1982. The State of the Art in Machine Learning, Introductory Readings in Expert Systems, D. Michie (ed., Gordon and Breach, UK.

Minsky, N., and Borgida, A., The Darwin Software-Evolution Environment, in Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development, Pittsburgh, PA, 1984.

Mitchell, T.M., Mahadevan, S., and Steinberg, L., 1985. LEAP: A Learning Apprentice for VLSI Design in Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA.

Moon, David. and Weinreb, Daniel., 1981. Lisp Machine Manual, MIT AI Lab.

Morgenstern, M., Active Databases as a Paradigm for Enhanced Computing Environments, Proceedings of the 9th VLDB Conference, Florence, Italy.

Mostow, J., 1985. Toward Better Models of the Design Process, AI Magazine, Spring 1985.

Narayanaswamy, K., Scacchi., and McLeod, D., 1985. Information Management Support for Evolving Software Systems, Computer Science Department, USC, Los Angeles, CA.

Naumann, J.D., and Jenkins, A.M., 1982. Prototyping: the New Paradigm for Systems Development, MIS Quarterly, 6,3.

Newell, Allen., and Rychener, Mike., 1978. An Instructible Production System, in F.Hayes-Roth and D.Waterman (eds.), Pattern Directed Inference Systems, Academic Press.

Orr, K., 1981. Structured Requirements Specification, Orr and Associates.

Protsko, L.B., Sorenson, P.G., and Tremblay, J.P., 1984. Automatic Generation of Data Flow Diagrams from a Requirements Specification Language, Proceedings 5th International Conference on Information Systems, Tucson, Az, pp. 157-171.

Reiner, D., Brodie, M., Brown, G., Fridel, M., Kramlich, D., Lehman, J., and Rosenthal, A., 1984. The Databse Design and Evaluation Workbench (DDEW) Project at CCA, Database Engineering, volume 7, no.4, December 1984.

Rich, Charles., 1984. A Formal Representation for Plans in the Programmers Apprentice, in Brodie,M.L., Mylopolous, J., and Schmidt, J.W. (eds.), On Conceptual Modeling, Springer, pp. 239-269.

Shrobe, Howard., 1979. Dependency directed reasoning for complex program understanding, Ph.d Dissertation, MIT.

Shortliffe, E.H., 1976. Computer-Based Medical Consultations: MYCIN. New York: American Elsevier.

Simon, H.A., 1981. The Sciences of the Artificial, 2nd ed., MIT Press, Cambridge, Mass.

Smith, S. F., 1980. A Learning Systemm Based on Genetic Adaptive Algorithms, Ph.D Dissertation, University of Pittsburgh.

Stallman, Richard. and Sussman, Gerald., 1977. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, Artificial Intelligence, volume 9, No.2, pp 135-196.

Waters, Richard., 1982. The programmer's apprentice : knowledge based program editing, IEEE Transactions on software engineering, no.1.

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

Winston, P.H., 1979. Learning and Reasoning By Analogy, CACM, vol. 23, No. 12, pp. 689-703.

Yourdon, E., and Constantine, L.L., 1978. Structured Design, Yourdon Press, New York.