LAYERED NETWORKS AS A TOOL FOR SOFTWARE DEVELOPMENT

Ira P. Goldstein and Daniel G. Bobrow

Xerox Palo Alto Research Center
Palo Alto, California 94304, U.S.A

Abstract: Most software systems use simple text files to represent the current state of an evolving design. We propose that layered networks provide a much better medium to represent the design of a software system and its documentation. PIE is an experimental personal information environment which provides users with descriptive structures for documents and programs. In PIE, alternative designs for programs and documents can be examined and manipulated within the systems context structured database. This data base also facilitates cooperative design among several people.

1. Introduction

The existence of a software problem is widely acknowledged: it is becoming progressively more costly to develop and maintain software systems. In this paper, we propose an application of AI technology, context sensitive databases, as a software development tool for decreasing these costs. We argue that context sensitive databases provide a representation for the evolution of software that is superior to traditional file systems. We have explored this proposition by means of an experimental Personal Information Environment (PIE) that has been used to manage the evolution of various software designs in its host environment.

PIE stores software designs in networks whose nodes represent the modules, procedures and other entites of the design, and whose links represent relationships among them. Links are asserted in *layers* and retrieved with respect to a sequence of layers termed a *context*. A layer is therefore a set of <node,link,node> assertions and represents a set of related changes to a design, while a context represents a particular version Goldstein Bobrow 80 provides a general introduction to the PIE system.

2. Previous Research in Artificial Intelligence

Various kinds of context sensitive databases have been explored in artificial intelligence research. They have primarily been used as a mechanism for representing alternative world views. (See, for example, Rulifson71; Hewitt71; SussmanMcDermott72; Hendrix 75; Cohen75). Generally the need to represent alternatives has arisen in planning programs. For example, a robot is analyzing alternative paths to reach some specified location. The terms *contexts* and *layers* are drawn directly from CONNIVER [SussmanMcDermott72].

Our application differs from previous AI research primarily in that previous applications focussed on the use of such databases by mechanical problem solvers. We are exploring the use of such databases in a mixed-initiative fashion with the user primarily responsible for their creation and maintenance.

3. The Inadequacies of Files

Most computing environments use files to express alternative designs. Users record significant alternatives in files of different names; the evolution of a given alternative is recorded in files of the same name with different version numbers. We argue that this use of files provides an inadequate structure for representing alternatives.

For example, consider how the development of the a software system consisting of several modules, where a module is a group of related procedures. Typically, the source code for each module for be stored in its own file-say, files A, B. . . . .If a programmer develops an alternative design that requires changes to several modules, how can he store this alternative? Typically, he would create files A', B', ... containing the new definitions plus any unaltered code for each module. The result is that the unaltered code is now stored redundantly. If subsequent development leads to modifications to this unaltered code, then these modifications must be made in both sets of files. The need for redundant editing becomes progressively worse as the number of alternatives grows.

How else might the programmer store his design if he wishes to avoid the need for redundant editing? One option is to place the common code into a separate file. Altering a given procedure common to more than one design would then take place in only one place. The cost of this storage strategy is that files would no longer serve to group related procedures of a design. Thus, to obtain design flexibility, the programmer would give up an equally important feature—modular representation.

Another option is to use conditional compilation statements in the original source code and avoid the need for multiple files and the associated redundant storage. However, the problem of examining the set of changes common to a redesign is now complicated by the distribution of these changes across many files.

A related problem is coordinating change. The programmer must maintain descriptions of how various alternative designs are distributed among files. Sometimes this is done by adopting conventions in naming files, as we have implicitly done in naming the files for our second design A', B,' But such conventions are an impoverished means to describe a configuration and fail as the space of alternatives grows in complexity. Explicit descriptions of such configurations that do not depend on naming conventions are preferable.

Finally, there is the problem that designs represented as configurations of files are not reflected in the operational software. The result is that it is cumbersome to examine the structure and performance of a design interactively within the system. Switching from one design to another requires reading and writing the appropriate files.

Thus, the traditional use of files is unsuitable for representing alternative designs for three reasons. (1) Files are inflexible. Their utility to store modular parts of a design must be sacrificed to avoid redundant storage of shared structure. (2) File names are impoverished as a vehicle to encode the intended coordination between different files as part of a common design. (3) The representation of alternatives is not integrated into the running software environment.

4. Source Code Control Systems

To remedy the first of these deficiencies, editors have been developed that store changes of source code to a base file. The most extensively used of these is the *Source Code Control System* [Rochkind75, Glasser78] developed as part of the Programmer's Workbench for the UNIX system (Ivie77). In SCCS, changes are stored in terms of lines deleted and inserted in a file called a *delta.*

SCCS has the virtue that shared structured is not stored redundantly. A delta contains only changes. The description of a particular version of a system is specified by a delta sequence. The explanation for the differences between this version and earlier ones is stored in the comments associated with the new delta files. However the changes are still stored with respect to lines, and have no connection with units of the program as thought of by the programmer.

5. Smalltalk

PIE employs layered networks to manage software development for any project undertaken in its host environment, Smalltalk [Ingalls76, Kay74]. To illustrate the system, we will briefly introduce Smalltalk, then describe a series of design exercises chosen to improve its implementation of the data structure for sets.

Smalltalk is an object-oriented language that extends the notion of class and instance found in Simula [Birtwistle73]. A class defines a group of procedures, termed *methods,* and a set of var.ables on which they operate. Each method is invoked by a message pattern. Every object is an instance of some class and stores particular assignments of values to the variables specified in the class.

Below Is a partial listing of class Set. Message patterns are shown in boldface and their methods appear below and indented The listing is incomplete: for example, the definition of the method for deleting elements from the set is not shown.

```
Class new title: 'Set' instance Variables:
'array n'

"Class Set employs an array with a position pointer
n to represent sets. The objects of the set are stored
in the array from position I to n.

Initialization Protocol
    init
        "This method Is conventionally executed when
        a new instance of class Set is created. It
        initializes the instance variables. The array
        variable is set to an array of size 8 and n is
        set to 0."
        [array <- Array init: 8.
         n <- 0.]

Public Protocol
    has: element
        "Testing whether an element belongs to the set
        is accomplished by iterating through the first n
        items in the array, checking for equality."
        [for: i from: I to: n dog
              [if : element = (array lookup; i) thens
        (return: true]],
         return: false]
    insert: element
        "A new element is added to the set if it is not
        already present."
        [if : (self has: element) then: [return: false]
        else : [self add: element]]

Private Protocol
    add: element
        "A new element is added by loading it into
        position n+1 of the array and incrementing the
        pointer. The array is copied into a larger array
        if its free positions are exhausted."
        [if : n=(array length) theng [array <- array
        growby: 10].
         array insert: (n<-n+1) with: element.]

Printing Protocol
    print
        "This method prints a set by printing the
        string 'a set'."
        ['a set' print.]
```

6. A Layered Redesign of Class Set

Our first design goal is to improve retrieval time by having the implementation convert from a sequential to a hashtable representation when the cardinality of the set exceeds some bound. The rationale for this redesign is that sequential access is less expensive in storage space and retrieval time when the set is small, but is not economic when the set is large.

We begin by generating a PIE description of the current implementation. PIE is able to generate a network describing any class in Smalltalk from the internal Smalltalk representation for the class. This network is stored as a collection of instances of class Node, a class we created to implement the behavior of a network database. The network generated from the initial implementation is stored in a new layer—say layer A. The layer is placed in a new context which we shall name the *hashing context.*

The next step in the design process is to define the changes to the present implementation. A layer is created and added to the context to store these changes. This is layer B in Figure 1.

Below is a PIE generated listing of the redesigned class with respect to the *hashing context.* Only parts of the public and private protocols are shown and comments have been removed. PIE has been instructed to highlight new assertions, derived from layer B, by printing them in boldface.

The listing shows a new type of variable in the class definition, limit is a *class variable.* The value of a class variable is available to all instances, limit is used to specify the size at which the internal representation switches from sequential to hashed.

```
Class new title: 'Set' instance Variables:
'array n'
   classVariables: 'Limit'
Public Protocol
   has: element
      [if : n<limit
       thens [fors i from: I to: n dog
                    if: element = (array lookup: i)
      thens [return: true]],
              return: false]
       else: [return: (self nashHas: element)].
   insert : element
      [if : (self has: element) then : [return: false].
       if : n=limit thens [self
      conuert from Sequential to Hash].
       if : n<limit thens [self odd: element]
          else : [self hash Add: element].]
Private Protocol
   add: element
      [if : n = (array length) thens [array <- array
      growby: 10].
       array Insert: (n<-n+1) with: element.]
   hash Add: element

   hash Has: element
```

The user can test his design by *installing* the hashing context. Installation causes the Smalltalk interpreter to employ the definitions asserted in the specified context. These definitions are not immediately installed to prevent premature modification of the underpinnings of the system before a design is complete.

If further debugging is needed following installation, the programmer can create a new layer to store the changes to his design, then reinstall the context with this layer dominating the old layers. By placing the edits of each debugging session in a separate layer, the programmer can undo a set of changes that have proved unsatisfactory by removing the layer from the context and reinstalling.

Design exercises rarely consist of a single iteration through the design/debugging loop In conducting this design exercise, a number of additional layers were added to the *hashing context.* For example, Layer C was added to correct the inconsistent treatment of the limit value, has: element treats limit as the lower bound of the hash representation while insert: element treats limit as the upper bound of the sequential representation. Layer C debugged the has: element procedure to use a < test for comparing the size of the set and the limit value.

Layers facilitate the comparison of alternatives For example, we considered different implementations of hashing in the redesign of class Set. Such analyses produced a network of layers as illustrated in Figure 2. The m and n layer sequences represent alternative designs.

Easy switching facilitated our obtaining comparative performance measures for the original linear design and our mixed linear and hashing design for implementing sets. The parameters which determine performance of a set implementation are the number of elements in a set. the ratio of membership tests to insertion and deletion, and the proportion of such tests which return false With two tests for every insertion, and 33% returns of false, the choice of 7 for limit allowed the mixed design to dominate; with five tests for every insertion, limit should be 4 to provide overall better performance for the mixed desigr

7. Redesigning the I/O Behavior of Sets

To illustrate the ability of layers to manage interacting designs, we continue our scenario by pursuing a second design goal. This goal is to improve the I/O behavior of sets. Specifically, the goal is for an instance of class Set to print showing its elements enclosed in braces, e.g. as {A,B}, if the size of the set is less than some bound. Presently all. instances of class set simply print as 'a set'. This redesign requires that we modify the printing method of class Set.

Before we make this change, we must decide where to store it. Since this is an independent modification of the code, our philosophy requires that we store these changes in a new layer, D To make it easier to test and adopt the printing changes independently of the hashing changes, we create a new context, to be called the *printing context,* for these changes.

This context begins from the initial design of class Set and, therefore, its first layer should be the same as the first layer of the hashing context. If we examine or install this context, we get an implementation of class Set that only has the improved printing behavior. Layer D is added to the printing context to store the changes involved in this redesign. Below is the new method for printing sets stored in layer D.

```
print
   [if : (self size)<4
   thens ['{' print.
          fors t from: 1 to: (self size) dog [(self
   element: I) print].
          '}' print.]
   elses ['a set' print.] ]
```

This redesign becomes more interesting if we decide to include a modification to the Smalltalk reader that allows the string printed to be reread as a set. To accomplish this, we must modify the reader to recognize braces. We could put the required changes in layer D. Changes recorded in a layer can span module boundaries. But since the reader changes are independent of the altered printing behavior, it is better practice to put this set of modifications in a separate layer, say layer E. We can therefore test the two parts of the design separately, i.e. we can first install layer D to examine the printing behavior, then install layer E to examine the reader.

Layer E could be placed in an entirely separate context, but since we presumably want to adopt both the changes to the reader and to the printing procedure, it makes sense to include this layer in the set printing context. However, since the alterations are modularly stored in a layer, we leave open the option of creating a separate context to store changes to the reader that includes layer E.

The printing context now contains layers that make a coordinated set of changes to more than one module of the system: in this case, both the reader and a particular abstract datatype. This is not an unusual situation—despite a modular design, some modifications inevitably cross module boundaries, since the modularity is based on a particular partitioning of the design space, and such partitionings are not unique.

8. Representing Contexts

Layers and contexts are described by PIE in the same network as software is described. Figure 3 shows part of the network representing the hashing and printing contexts. The rationale attribute links a layer or context to a textual description of its purpose and the focus attribute points to the major classes being modified by the design. The layers attribute of a context node points o a sequence of nodes representing the layers.

Describing layers and contexts in the network has two advantages. First, the user can search for a layer or context using the general network matching machinery provided by PIE. A search is initiated by specifying a description of some node in terms of constraints on the values of its attributes. Thus, a user can search for a node representing a context whose focus is class Set and whose rationale includes the substring *hashing*. The network description escapes the limitation of file systems in which the name of a file is burdened with the description of the file Second, the user can manipulate layers and contexts using the same network operations used to manipulate code—i.e. the addition, deletion or modification of the attributes of nodes.

0. Composite Contexts

After we have debugged our two redesigns of class Set, we will want to combine them. We can do this by creating a composite context built from the layers of the existing contexts. This is the *Set Redesign* context shown in Figure 4. We have not simply concatenated the layers of the hashing and printing contexts. This would produce the sequence: A, B, C, A, D, E. The second occurrence of layer A would inadvertently dominate the changes in layers B and C.

We may wish to impose the constraint that if new layers are added to the hashing or printing contexts, then they are automatically included in the composite context. This can be accomplished by a capability that PIE provides for defining procedures that are attached to nodes. These procedures are triggered by adding or deleting attributes of the node. By defining such a procedure and attaching it to the layers attribute of the hashing and printing contexts, we can have it synchronize these contexts with the composite context.

Concatenating new layers of the printing context to the hashing context is justified by the independence of the printing and hashing refinements. This is therefore a useful comment to include in the network. Figure 4 illustrates various design comments. Layers B and D are commented as independent refinements, layers D and E as dependent refinements and layer C as a repair for layer B. When layers are combined into new contexts, these comments are checked and the user is alerted to questionable combinations such as adding a layer without its subsequent repairs. This description also allows other programmers to examine the network of contexts and layers and understand their relationships.

Interactions between design decisions can lead to conflicting values in two layers. In SCCS, conflicts are noticed only to the extent that two modifications touch the same line of code. In PIE, the granularity of detected overlap is at the level of the method, if two layers have changed the same method, then there is a potential conflict. In addition, we have in PIE a mechanism for explicitly expressing dependencies among a number of methods. This allows us to find some potential interactions when there is no structural overlap.

To resolve conflicts between layers, a new layer could be added to the composite context that resolved any differences in design decisions. This layer would only apply to the composite design and not to the individual designs since it would not be included in their contexts.

Finally, we could have created a composite context. The *concatenated* context in Figure 4 is such a context. PIE treats the layer sequence of such a context as the concatenation of the layer sequences of its sub contexts. This combination strategy is appropriate when the constituent contexts are independent. But here, the common use of layer A makes it inappropriate

10. Communicating Contexts

We have discussed this design project so far from the standpoint of an individual programmer But many projects are *collaborative*. Layers facilitate cooperative design by supporting on line interaction that is analogous to two programmers scribbling on a common listing One programmer can transmit to another programmer a set of changes to the first programmers design by sending a layer PIE supplies comparison functions for identifying the changes between layers Below is a PIE generated listing of class Set with respect to the redesign context plus a layer transmitted by a collaborator. The collaborator's layer redefines the has: element procedure and includes an annotation stating the rationale for the change

```
Class new rule: 'Set' instance variables:
'array n'
   class Variables: 'limit'

Public Functions
   has: element
      [if : (self size <limit
       thens [return; (self seqllas: element)]
       else : [return: (self hasnHas; element)].
```

Annotation from Danny received 4/1/60, 3:15pm: I think that a more subroutinized definition will pay off in the tony run.

The advantages of this level of intimacy in communication would not ordinarily be sufficient to offset the disadvantages that might arise from an unwanted intrusion by the sender into the recipient's workspace A programmer might be justifiably relucant to load a file from a collaborator directly into his workspace Despite the appeal of being able to examine the new software with design tools in the software environment, the software may contain modifications that overlap and interfere with software developed by the programmer.

Layers avoid this problem. The sender's modifications are contained in a separate layer. Hence, they can bo loaded without destroying changes stored in separate layers. The user can install then, examine their performance, then undo them if desired simply by deleting the layer from his context.

11. Complex Designs

Massive redesigns can involve changes to hundreds of procedures distributed over dozens of modules. We believe that the machinery described here for layers and contexts provides capabilities that are suitable for such complex real world software problems based on our analysis of the deficiencies of present source control systems.

As with systems like SCCS and unlike systems that store the entire source code for a package in a single file, we store a set of changes modularly with very little redundant information. This allows forks in designs to be explored and facilitates the creation of specialized configurations from selected subsets of the layers.

Unlike SCCS, we represent software and design configurations in a network that has important structural advantages over textual descriptions. First, the network supports a representation in which there is a natural locus for all of the properties of a given object including both source and compiled code—such properties would typically have to be distributed across a textual listing or even across different files. A designer can readily examine all of the properties of some object by interrogating the network. Second, a network allows us to move two ways across a link. Thus we can move from the nodes describing variables to their classes or vice versa. A designer can therefore traverse the network to explore some the consequences of a redesign Third, the formalized descriptions of a network support search and matching operations to find desired objects of a design, including layers and contexts themselves Fourth, the network facilitates the formation of various kinds of composite designs. Hierarchical file directories provide some aid of this kind for file based source code systems, but are less powerful than the network architecture of PIE. Finally, the network strictly dominates text since a node can have a source code attribute that points at text describing the software.

12. Conclusions

PIE's ability to represent alternative designs comes at the price of a more complex representation. For this price to be affordable, the user interface must simplify the presentation and manipulation of the database In this paper however, we have not had the room to describe the display interface we use. An extended discussion can be found in GoldstemBobrow81. In that paper, we also discuss the use of additional description provided in the network to specify reasonable default behavior for the interface.

PIE is currently running with excellent response time on a Dorado, a high speed micro programmable personal computer that runs Smalltalk at approximately one million instructions/second [LampsonPier 80]. The response time is perceived in terms of the time to refresh the display interface following a new selection—a user perceives little or no delay. The criticial limitation of the present Smalltalk implementation is the size of its virtual memory. Smalltalk 76 [Ingalls78] supports an address space of only 48,000 objects. This includes all of the objects defining Smalltalk itself. As a result, there is insufficient space to build large PIE networks. No more than 1,000 nodes can be in the local address space at any one time Since there are approximately 3,000 methods in Smalltalk, it has not been possible to build a PIE network describing the entire Smalltalk system A new Smalltalk implementation soon to be available has a 31 bit virtual memory. Given this capacity, the possibility exists of transforming PIE from its present experimental status to a permanent part of Smalltalk's programming environment.

The techniques described in this paper have been tested in the context of the Smalltalk programming environment. However, they can be applied to any programming system that provides a layered network database and an interactive display interface. We believe that the payoff of such databases for software design and development will more than justify their cost in terms of retrieval time and storage.

19. Bibliography

[Birtwistle, 1973]
Birtwistle, G., Dahl, O. J , Myhrhaug, B., and Nygaard, C, *Simula Begin,* Auerbach, Philadelphia, 1973.

(Cohen, 1975]
Cohen, P., "Semantic Networks and the Generation of Context", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Tsibilisi: (1975), 134-142.

[Glasser, 1978]
Glasser, Alan L., "The Evolution of a Source Code Control System", in S. Jackson and J Lockett (eds), Proceedings of the Software Quality and Assurance Workshop, ACM, (1978) -pp. 122-125

[Goldstein & Bobrow, 1980]
Goldstein, IP. and Bobrow, D.G., "Descriptions for a Programming Environment", *Proceedings of the First Conference of the American Association for Artificial Intelligence.* Stanford: (1980), 187 189.

[Goldstein & Bobrow, 1981]
Goldstein, IP. and Bobrow, D.G., "Browsing in a Programming Environment", *Proceedings of the 14th Annual Hawaii International Conference on System Sciences,* Honolulu: (1981).

[Hendrix, 1975]
Hendrix, Gary G., "Expanding the Utility of Semantic Networks Through Partitioning", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence,* Tsibilisi: (1975), 115-121.

[Hewitt, 1971]
Hewitt, C, "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot". Ph.D. Thesis (June, 1971) (Reprinted in AI TR 258 MIT-AI Laboratory (April 1972.)

[Ingalls. 1978]
Ingalls, Daniel H., "The Smalltalk 76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* Tucson, Arizona (January, 1978) pp 9 16.

[Ivie, *77]*
Ivie, E.L. "The Programmers Workbench - A Machine for Software Development." *Communications of the ACM,* V. 20 No. 10 (October 1977) pp 746 53

[Kay, 1974]
Kay, A., SMALLTALK, "A Communication Medium for Children of All Ages". Palo Alto, California: Xerox Palo Alto Research Center, Learning Research Group (1974).

[Rochkind, 1975]
Rochkind, Marc J., "The Source Code Control System", *IEEE Transactions on Software Engineering* (December 1975) pp 364-370.

[Rulifson, 1971]
Rulifson, J., Waldinger, R., and Derksen, J., "A Language for Writing Problem Solving Programs", IFIP, 1971.

[Sussman & McDermott, 1972]
Sussman, G , & McDermott, D , "From PLANNER to CONNIVER • A Genetic Approach". *Fall Joint Computer Conference.* Montvale, N. J.: AFIPS Press (1972).

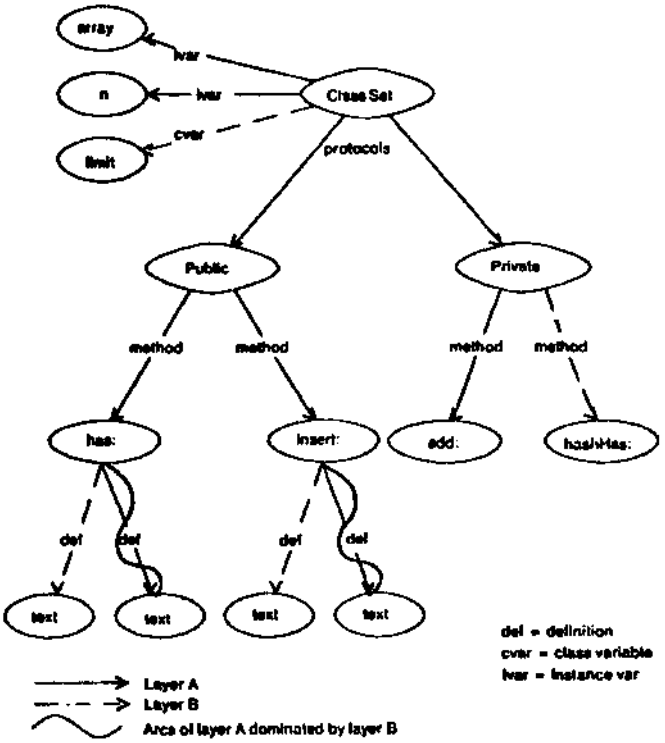

Figure 1. A partial view of a layered network representation for class Set. Layer A describes the original design; Layer B contains modifications. The Hashing Context contains both these layers with layer B dominating layer A.

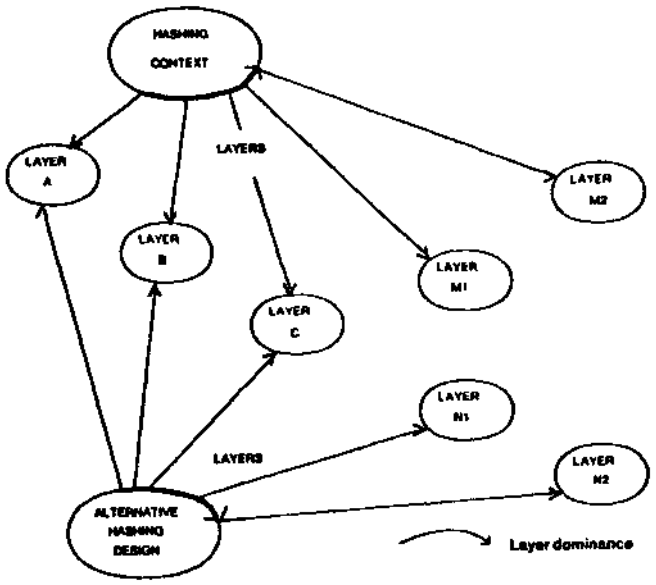

Figure 2. Alternative designs for hashing. The dominance arrow points from least to most dominant layer.

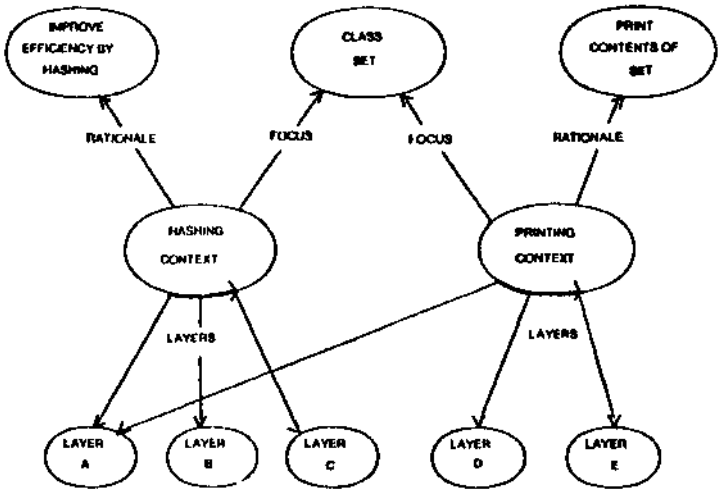

Figure 3. A partial view of the network description of layer and context nodes.

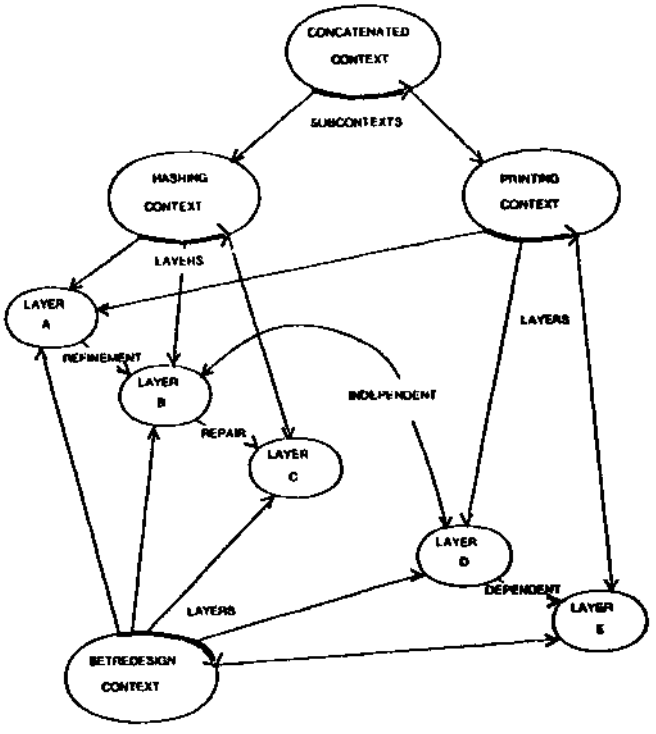

Figure 4. Two different kinds of composite contexts.