Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1994), *8*, 125–142. Printed in the USA.

Knowledge compilation using constraint inheritance

ROSEMARY CHABOT (née HORNER)
Multi-Vendor Customer Services Applied Research Group,
Digital Equipment Corporation, 334 South Street, Shrewsbury, MA 01545

DAVID C. BROWN
Computer Science Department, Worcester Polytechnic Institute,
100 Institute Road, Worcester, MA 01609

(Received December 1, 1992; Accepted September 15, 1993)

Abstract

Design knowledge is continually refined and expanded through experience. This research is concerned with design knowledge expressed as constraints. A simple learning mechanism simulates an expert designer's ability to incrementally adjust her knowledge when presented with slightly new problems. In response to unsatisfied expectations during the design process the system will examine its general knowledge about the design artifact, discover some relevant constraining knowledge, and convert that knowledge into a design constraint for future use. This process, referred to as constraint inheritance, should automatically improve the problem-solving performance.

Keywords: Learning; Design; Knowledge Compilation; Constraints

1. LEARNING WHILE DESIGNING

There should be no argument about the fact that designers learn while designing. It is also clear that investigating design systems that learn is an interesting and challenging area for research (Brown, 1991; Reich, 1991; Maher, 1992). However, what is not clear is how we should conduct this research, and whether it is really needed.

Currently, we know how to implement routine or near-routine design tasks quite well, especially if they are parametric design problems [a long list of references is possible here; for a sample, see Tong & Sriram (1992a)]. We do not yet know how to deal with nonroutine or more conceptual design tasks very well, although there is some work in this area (Gero & Maher, 1993; Tong & Sriram, 1992b).

For convenience, we will refer to the former class of problems as RP problems (for routine parametric) while the latter will be referred to as NRC (nonroutine conceptual) (Brown, 1992). Both classes represent extremes of a wide variety of types of problems.

It has been suggested that one can only learn what one can represent. We know how to represent the knowledge and processes that are used in RP design, as is clear from the wide variety of systems of that type. Consequently, this makes a suitable target for learning. However, it is not clear that anyone would claim that such knowledge was known and agreed upon for NRC design. This raises a doubt about whether this is a suitable area in which to investigate learning.

Why do we need intelligent design systems? One answer is that such a system could take care of the mundane design activity, leaving the harder, more challenging problems for humans. Mundane design activities are likely to be in the RP class.

Another reason is to provide support for the designer as she progresses through the design. A large amount of this support is to provide appropriate standard information in some form or another perhaps disguised as criticisms or suggestions. This is probably routine activity.

Other reasons are to do with consistency, and coverage. Issues of the consistent correctness of designs are more likely to refer to RP problems than NRC problems. However, the issue of coverage (i.e., ensuring that the designer adequately considers all the alternatives) is likely to refer to both RP and NRC problems, although the impact of a poor choice due to incomplete investigation is greater for NRC problems.

If the above arguments are correct and complete, we can deduce that there is a strong "need" for systems that handle RP problems. From the previous argument, we

Reprint requests to: David C. Brown, Computer Science Department, Worcester Polytechnic Institute, 100 Institute Road, Worcester, MA 01609.

can also conclude that such systems are a viable area for current research into learning in design systems. This may not be so true for NRC problems, despite the obvious need to do more research into NRC design.

1.1. Our research

Not surprisingly, our work has been concerned with learning within RP IntCAD (Intelligent Computer Aided Design) systems (Spillane & Brown, 1992). We have investigated how design knowledge in RP systems might change with experience. Thus, a system should improve its performance in response to the most recent set of design problems. So far, we have only been concerned with constraints and have studied how they can be added (reported here), deleted (Meehan & Brown, 1990) and moved (Sloan, 1988) within the knowledge.

More recent work is investigating how evidence from knowledge such as function, structure, and design cases, can be integrated so that an IntCAD system can learn appropriate decompositions for design problems (Liu & Brown, 1992).

While some learning in design work seems to be tyrannized by the existing classes of machine learning (rote, explanation-based learning, etc.)—mechanisms looking for problems—we are trying to look at each ingredient of an RP IntCAD system and ask how it could be learned. It is not clear that the full power of standard machine learning methods is always required.

1.2. Overview of the constraint inheritance mechanism

This paper is about *constraint inheritance*, which is a form of incremental refinement of knowledge. It is also a method that simulates an expert designer's ability to adjust her own compiled knowledge of design constraints on-the-fly when presented with slightly new problems in a routine design domain (Horner, 1989).

Constraint inheritance relies on the idea of *expectations*, which are predictions about the values of design attributes. We conjecture that expectations are formed by induction from past designing activity or observations of past designs. That mechanism is not addressed by this work.

Constraint inheritance first detects an *expectation violation* during the design process, compiles a constraint from relevant general knowledge about design objects, and moves the new constraint from the general knowledge to the routine design knowledge (i.e., it *inherits* it from the general knowledge). This general design object knowledge may represent the structure, behavior, and function of the object being designed. Because this general knowledge is stored without regard to its intended use, it can be very costly to reason with. Constraint inheritance can reduce the overhead associated with reasoning with this general knowledge by transforming it into structures that are highly tuned for routine design problem solving and are immediately available for direct use—essential features of compiled knowledge.

2. RESEARCH MOTIVATIONS

This work is part of a research program that is investigating the different types of knowledge and mechanisms that cause design problem solving to become routine. There are many different types of knowledge and problem-solving mechanisms that are used cooperatively when a designer solves a routine problem. A major motivation of this piece of the research is to determine possible origins of design constraints. If they can be pinpointed, then it should be possible to understand the content and effect of this knowledge, as well as understanding the process of learning new knowledge of this type.

We refer to *knowledge compilation* as the process by which design knowledge becomes routine. Knowledge compilation is a human mental activity that involves incremental learning from experience (Brown, 1989a; Goel, 1991). Knowledge compilation transforms existing knowledge into new forms, with the intent being to improve problem-solving efficiency. In this work the transformation method is constraint inheritance.

This research also addresses the issue of how an expert designer's knowledge is continually refined and expanded through experience. We have built and demonstrated a simple learning mechanism in a routine design system that simulates an expert designer's ability to incrementally adjust her knowledge about design constraints when presented with slightly new problems. The power of incremental learning is that as all design problems vary slightly with time, the design knowledge should be automatically enhanced.

A further goal of this research is to produce mechanisms that are reasonable hypothesis about human behavior. It is apparent that a reminding and a memory reorganization process runs parallel with problem-solving processes (Schank, 1982). This research integrates dynamic modification of design knowledge, and a form of reminding, with problem-solving processes.

A final research goal is to decrease the brittleness of expert systems. Many of today's expert systems have rigid boundaries. They need to know in advance the types of situations that will occur. These expert systems, when faced with a situation that is not anticipated, may apply inappropriate knowledge, or fail. Such expert systems have been labeled as *brittle*. These expert systems must be able to recognize their own boundaries and be able to expand these boundaries thus increasing their competence.

Expert systems that are emulating human behavior must be able to emulate learning processes, transform their existing knowledge, and degrade gracefully when missing or incorrect knowledge is encountered (Davis,

1989). Constraint inheritance is a learning mechanism that can decrease the brittleness of a design expert system by automatically improving its problem-solving performance, as well as its competence. The expected result is that as new design constraints are inherited, less time will be spent on incorrect design decisions that eventually lead to design failures and large amounts of backtracking.

3. ROUTINE DESIGN

We consider a design *routine* when the design knowledge, problem-solving strategies, and problem-solving alternatives (plans) are known prior to the beginning of the design process (Brown, 1992). This is not the case for other kinds of design, with labels such as "creative" or "innovative." Brown and Chandrasekaran (1989) state that problem decompositions, attributes, and actions to handle failures also need to be known in advance for routine design. The most common form is referred to as routine parametric design, where values are produced for a predetermined set of design parameters.

Design is called routine because the design has been performed repeatedly before, but each time with slight variations in the design requirements, resulting in known design structures and known control patterns. Because of this past experience, the problem-solving knowledge is transformed into an efficient form for the solution of similar problems. Although this form is highly efficient, routine design may still involve complex knowledge-based problem-solving activity, involving selection and failure handling.

4. DESIGN SPECIALISTS AND PLANS LANGUAGE

The *Design Specialists and Plans Language* (DSPL) is a domain-independent expert system building language for expressing routine design knowledge (Brown, 1985). It can also be thought of as a modeling language since it represents the design process. It is a non-rule-based hierarchical knowledge representation language and is capable of expressing many different types of routine design knowledge. The actual design is carried out when the DSPL code is executed.

DSPL uses a generic form of knowledge-based reasoning called Plan Selection and Refinement. A major portion of design is "situation recognition," and DSPL captures the knowledge and heuristics that are necessary for the recognition of familiar situations.

DSPL allows declaration of type, structure, and associations between a wide variety of cooperating entities, referred to as *agents*. We use the term "agent" because they cooperate, they communicate by passing messages, and because each one has a particular role to play during design. These agents can be *specialists*, *plans*, *sponsors*, *selectors*, *tasks*, *steps*, *constraints*, *failure handlers*, and *redesigners*. Captured within these agents is the "how to" design knowledge. DSPL agents are organized into a hierarchy that reflects the expert designer's conceptual view of the design object. This view is most probably a functional decomposition of the design object. As form tends to follow function, this leads to a strong relationship between the form of the design object and the form of DSPL. The important point here is that the DSPL language can mimic the expert designer's problem-solving abilities.

Of the design agents mentioned above, the constraint is the most important for this research, as we are concerned with the automatic construction of new DSPL constraints. Constraints play a key role in determining the correctness of design, and in triggering backtracking.

This research is concerned with enhancing the performance of DSPL by having DSPL knowledge adjust itself as a result of experience, using a simple learning mechanism. The gradual extension of DSPL to incorporate machine learning will allow for more powerful and flexible design expert systems to be built.

4.1. The ball point pen (BPEN) domain

DSPL was developed after studying routine design problem solving in the domain of air-cylinder design. A ball point pen (BPEN) was chosen to be an alternative, simple test domain for this research. A major reason for selecting the BPEN domain was that the topology of BPEN was very clear. Figure 1 depicts the structure of the BPEN. BPEN is not overly complex, yet it was not obviously simple either. The possible components, their interconnections, and their relationships were all known in advance, thus making BPEN a good domain for routine design. In addition, the dependencies between the BPEN component attributes provide many design constraints.

5. SURFACE AND DEEP LEVELS OF KNOWLEDGE

Several researchers have made the distinction between surface and deep models of expertise (Klein & Finin, 1987; Brown & Sloan, 1987; Chandrasekaran & Mittal, 1983). However, there are varying degrees of knowledge "depth," and such terms are relative.

Surface knowledge is mainly based on heuristics and associations. It is built up from experience. This is the level where one can find expert knowledge and problem-solving strategies that have been tightly organized and reduced. DSPL is a form of surface knowledge. An important feature of surface knowledge is that solution methods tend to be selected and executed rather than constructed. Once a situation is recognized, the associated action is performed. Surface knowledge tends to be brittle.

In contrast, *deep* knowledge is a more abstract and powerful level of knowledge. For routine design, the cor-

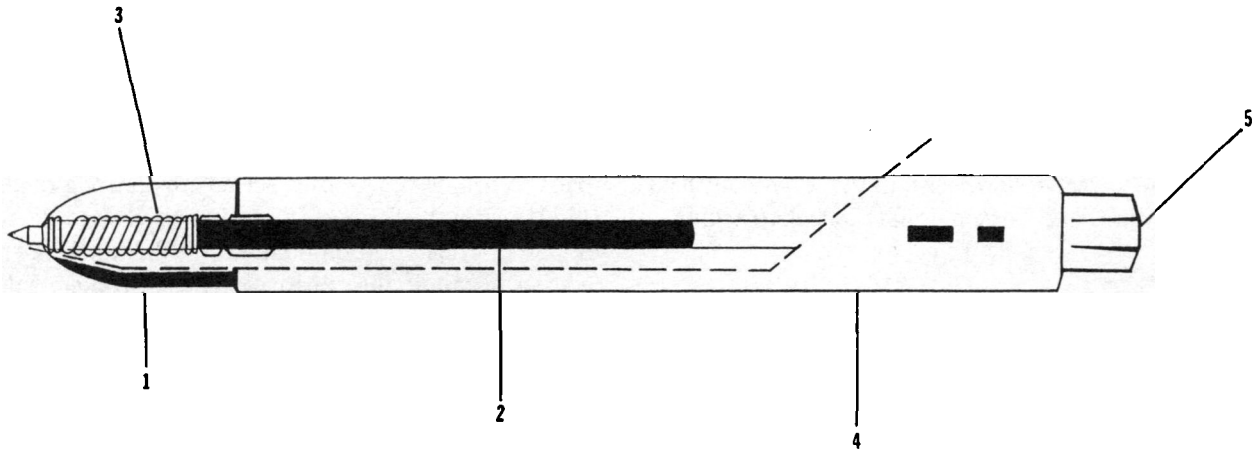

Fig. 1. BPEN structure diagram.

responding deep knowledge could be about the shape, structure, behavior, or function of the design object. Deep knowledge may be task independent. It may come from first principles, axioms, laws, or other information found in textbooks or reference manuals. One must inspect deep knowledge and reason with it. Because of this, deep level knowledge can handle problems that are not explicitly anticipated; problems that are unexpected or deviant. Deep knowledge implicitly underlies, and can be used to explain, patterns of surface knowledge. Therefore, this deep knowledge plays a very important role in the incremental restructuring of existing surface knowledge, or the creation of new surface knowledge.

For example, in a knowledge base about a simple, routine ball-point design, the constraint REFILL COMPONENT OUTER DIAMETER < SPRING COMPONENT INNER DIAMETER ensures that the refill component fits inside the retraction spring component. This is an example of surface knowledge. In its deeper form, it is equivalent to an instance of a SURROUNDS relation that refers to the general descriptions of the REFILL component and the SPRING component.

An expert system that can represent both surface and deep levels of knowledge, and automatically generate surface knowledge from deep knowledge, should be able to learn on its own. An expert system that can represent both deep and surface levels of knowledge allows the possibility of accessing deeper knowledge when surface knowledge is inadequate. Expert systems of this type will be able to solve new design problems more efficiently.

6. THE COMPILATION PROCESS

Learning is the normal situation, rather than the exception. Experts are continually reorganizing knowledge as part of their problem-solving activity. We believe human expert designers perform knowledge compilation while they design (Ericsson & Smith, 1991). Not all researchers agree on the definition of knowledge compilation (Fisher et al., 1992). We have based our work on the definition given in Section 2.

Compilation transforms existing knowledge into new forms in order to improve problem-solving efficiency. Problem-solving efficiency is increased since compilation produces short-cuts in reasoning, that is, deep reasoning is replaced with shallow (surface) reasoning.

Knowledge compilation involves a shift in both the knowledge representation used and the problem-solving required. The shift in knowledge representation is to a form appropriate for immediate use. The shift in problem-solving causes different forms of reasoning to occur. For example, deep reasoning involves processes such as functional reasoning, three-dimensional visualization, and reasoning from first principles. This type of reasoning is costly. After compilation, the reasoning process is much simpler, with highly efficient, script-like structures being used for problem solving.

As more complex expert systems evolve, it will be necessary to have access to domain knowledge that is stored at a deep level. Knowledge compilation can reduce the overhead associated with reasoning with this deep knowledge by transforming it into structures that are highly specialized, tuned, and most effective for a given type of problem-solving task (Chandrasekaran & Mittal, 1986; Keller et al., 1989).

Knowledge compilation also refers to the process of adding, from experience, new knowledge that improves the performance of existing knowledge (Pazzani, 1986). Design systems need to be able to react to requirements that may change slightly over time. Upon failures, these reactive systems should make adjustments that allow the system to continue to be useful and efficient (Brown, 1989a).

There are different views of whether the system should produce the same results after compilation or not (i.e., does it still produce the same designs? Are those designs

correct?). In the work reported here, for a given set of requirements, the system can produce a different design after compilation. Improvement is not just in speed. In work that is more in the machine learning "mainstream," for example (Laird, 1992), pre- and postcompilation equivalence is taken as normal.

Compilation can be viewed as having two logical phases: a formation phase and an adjustment phase. The formation phase is concerned with the initial assembly and structuring of deep knowledge by experience into surface form. The adjustment phase is associated with the modification and relocation of surface knowledge for greater efficiency (Brown & Sloan, 1987). Constraint inheritance both adds to and modifies surface knowledge, thus having characteristics from both the formation and adjustment phases.

7. THE CONSTRAINT INHERITANCE PROBLEM

In this research we are concerned with constraints that may be missing from routine design knowledge. In such a situation, a designer may make an incorrect design decision but not discover her mistake until later. We hypothesize that this discovery will often occur immediately after deciding a related design attribute's value. The designer mentally "steps back" and notices that the design decision is not consistent with her general knowledge of the design object (Brown, 1989b). She will use her own general object knowledge to allow her to "figure out" a solution to the current design problem. As she is reasoning with this general knowledge, she may locate a piece of design knowledge that is important to her problem solving.

Once the constraining general knowledge has been identified, the designer can include this knowledge as a design constraint into her surface knowledge of how to design that object. The constraint is inherited from the general object knowledge to a specific location within the routine design knowledge about that object. Once this constraint has been noted as being necessary for this situation, it can be used for all future design activity (see Fig. 2). This improves subsequent problem solving by avoiding similar failures.

The term "inheritance" has been used to suggest the transfer of information (constraints) from a general level to a specific level. The problem is how to implement

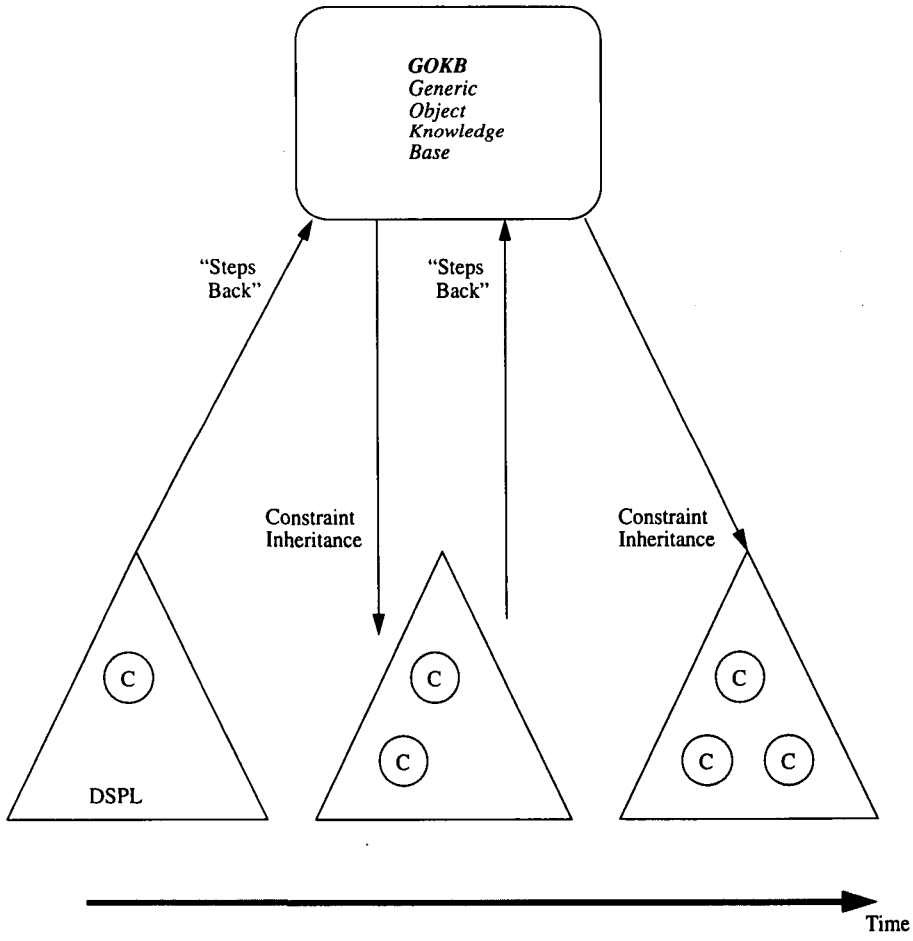

Fig. 2. The constraint inheritance problem.

this general description in the context of DSPL problem solving.

8. CONSTRAINT INHERITANCE DESIGN

Constraint inheritance uses several components and related, but distinct, design knowledge bases. Each plays a unique and important role in the entire design. We will first define the characteristics of each component and then discuss how they combine to form the constraint inheritance mechanism.

8.1. Design knowledge bases

To design an object, one must know all the components and the relationships between them. Hence part decomposition and spatial relationships are essential for object representation. Currently, in "vanilla" DSPL, there is no explicit knowledge that indicates how and where components fit together. Yet this is exactly the deeper type of knowledge that experts depend on for guidance during less routine problem solving.

In addition, in order to perform constraint inheritance, once must inherit from a deeper knowledge level to the DSPL (surface knowledge) level. Consequently, a general knowledge base about the design object is required. We will refer to this as the Generic Object Knowledge Base, or GOKB.

Originally, DSPL utilized only two main collections of knowledge, the DSPL knowledge base and the Design Data Base (DDB). The DSPL knowledge base is the surface knowledge, that is, the DSPL code that represents "how to" do the design. The DDB contains the current state of the design as the design progresses. The DDB will eventually contain the final design, that is, the design attributes and their final values. Note that the GOKB is not the same as either of these collections. The GOKB contains declarative, general knowledge about design objects.

This research included the task of designing and building a small GOKB. This knowledge is in the form of a hierarchy of object schema. This representation contains varying levels of detail. The DDB can be considered an instance of a portion of the GOKB.

The GOKB represent, or has pointers to, the structure, shape, behavior, function, features, and material hierarchies of the design object. Not all of these aspects were implemented in the work reported here. The GOKB is intended to model what the designer knows about the components and materials that will be used during the design. It is stored without regard to its intended usage.

8.2. Design constraints

Design constraints play a crucial role in any design process. This is because constraint knowledge is the primary method of detecting design failures. Constraints provide a way of declaratively representing the knowledge an expert has that confirms that a design decision should lead to a correct design. Constraints may refer to design attributes of the design object. Constraints may also refer to the process of assembling the design object or to the design process (Brown & Chandrasekaran, 1989). At a more abstract level, constraints may test for plausibility, feasibility, legality, or consistency of a design decision (Sloan, 1988).

Some constraints are qualitative. Such constraints are typically found in deep knowledge since this is what qualitative reasoning generally uses. Examples of structural qualitative constraints found in the GOKB are "attached-to," "serially-connected-to," "above," and "surrounded by." These relationships are not dealing directly with actual dimensions, compared to relationships such as "greater-than" or "less-than." General functional relationships of the design object or physical law relationships can also be found in the GOKB.

In contrast, surface knowledge (e.g., DSPL) typically contains quantitative constraints. A quantitative (numeric) constraint generally involves some type of calculation. These calculations can vary in complexity. DSPL constraints are generally quantitative constraints. The constraints in DSPL are represented as agents and are activated (i.e., asked for their response) by other agents.

Our hypothesis is that in routine design, general, qualitative knowledge about an object's structure and function have become transformed into a fixed series of calculations and constraints. Some additional evidence for this idea is provided by Liu's research on generating decompositions from functional and structural knowledge (Liu & Brown, 1992).

8.2.1. Varieties of design constraint

There are a large number of constraint types. They can be classified by their content, purpose, or location – where by "location" we mean where the constraints are placed in the "how to" knowledge. For examples of different categories of constraints see Sriram and Maher (1986), Brown and Breau (1986), and Stauffer and Slaughterbeck-Hyde (1989).

Hard versus *soft* is one category. Hard constraints are either satisfied or violated. There are no degrees of satisfaction. If a hard constraint is violated, then the design decision becomes unacceptable. These types of constraints are rigidly enforced; they cannot be relaxed or ignored.

Soft constraints may produce results such as "poorly satisfied," "satisfied," or "well satisfied," perhaps by testing against a range of values. Soft constraints may be relaxable [see, for example, Meehan & Brown (1990); Descotte & Latombe (1981)]. Error tolerances may be associated with soft constraints. These tolerances provide bounds in which the tested value of a constraint must fall.

If error tolerances are involved, a constraint's importance may grow as a tolerance limit is approached.

8.2.2. DSPL design constraints

In DSPL, constraints are distinguished from other kinds of design knowledge, that is, a constraint is a distinct DSPL entity. DSPL constraints can determine the suitability of incoming requirements or design attribute values, and can determine the applicability of some design knowledge.

Constraints can be explicitly associated with agents in the design knowledge hierarchy. Thus they can decide the ultimate success of any DSPL agent. This association is predefined and is not something that occurs at runtime. Constraints are tested, not posted or propagated. In DSPL, constraints are satisfied during a form of search, not by using a consistency algorithm.

The type of DSPL constraint with which this research is concerned tests a single relationship between two or more design attributes to see if it is within some limit. The design attributes and other knowledge needed to perform the test are immediately available, can be calculated, or can be retrieved from the design data base (DDB). The constraints are preconditions to the decision to store a design attribute value in the DDB. This decision is located at the end of every Step agent.

DSPL constraints are considered to be hard constraints. When DSPL constraints are violated, a failure message is returned to the calling design agent. Decisions are made at that point as to what to do about the failure. Constraints can provide suggestions about what to do in failure situations.

8.3. Design expectations

A *design expectation* is a designer's prediction about a design attribute's value. Since expectations are produced by experiences they inevitably reflect the design knowledge. However, they are likely to be approximations of the actual values. For example, for BPEN an expectation might be "expect the length of the body component to be four times the length of the head component." Expectations do not involve complicated calculations, unlike the decision making within a DSPL step. This is because expectations are simply predictions based on past design observations. In our system, expectations may consist of a default value, a range of possible values, or a simple relation between design attribute values.

The function of expectations is not to constrain but rather to alert. Great confidence should not be placed on the expectation value. Since expectation values are not precise, we use positive and negative tolerances attached to expectations to provide a potential range within which the value should fall.

In this research, simple *expectation rules* that relate design attributes are used as the source for expectations. We are using the term "rule," in the context of expectations, to mean a predicted relationship. An expectation rule may be triggered when a value for a *key design attribute* has been decided. There will exist another attribute that is in some way dependent upon the key design attribute. The expectation reflects the dependency that exists between both design attributes. The *expectation value*, which the expectation rule produces, is for the *dependent design attribute*. In the above example, "body length" is the dependent attribute, while "head length" is the *key* attribute. That is, deciding the head length will produce an expectation about the body length.

Note that expectations and constraints are two very distinct types of knowledge. They serve different purposes. Expectations produce a value, or a range of values. In contrast, a constraint tests a value.

We have observed that a designer's past experiences produce significant background knowledge about a domain. Expectations are an important source of knowledge because they prime the design system with some of that background knowledge. This knowledge can help control the design process.

Another motivation for the use of expectations is that this is routine design research. As routine design requires the design knowledge to be known in advance, there should be a wealth of expectations within the design knowledge. Therefore, routine design provides an excellent environment in which to use and study expectations.

It is worth noting at this point that the use of expectations is quite common in AI systems. For example, Chandrasekaran and Punch (1987) use "diagnostic expectations" to carry out expectation-based data validation. Other uses have been during parsing and for vision. Well-founded expectations reduce the amount of computation required during situation recognition.

8.3.1. Design expectation components

In summary, there are four design expectation components used in constraint inheritance. The first is the *key design attribute* on whose value other design attributes will depend. Expectations are *triggered* or evaluated upon decision of a key attribute's value. The second is the *dependent design attribute*, which is the *target* of the expectation.

The third and fourth components of an expectation are the *expectation rule* and the *expectation value*. The expectation rule is associated with the key design attribute. This rule is a calculation that is performed when the key design attribute has been decided. It calculates the expectation value, which is then associated with the dependent attribute.

For example, consider again the expectation that predicts that the length of the body component is four times the length of the head component. The expectation rule is "head length $\times$ 4." When the key design attribute, head length, is decided, the expectation rule will be evaluated.

The expectation value is the result of the evaluation of the expectation rule. For this example, consider the head component length to be 1.5 inches. The expectation rule yields an expectation value of 6 inches. This value will then be associated with the body length attribute. The expectation value will be used for comparison at a later point in the design when the value of the dependent design attribute (body length) is being decided.

8.3.2. Design expectation types

There are three basic types of expectations that a human expert designer might use. The first type is a *Context Independent* (CI) expectation, where the expectation is not dependent upon the current state of the design. This type of expectation does not require a key design attribute. Hence, they do not have an expectation rule. These expectations are predictions of default values (Minsky, 1975). In routine design, certain values may be expected in advance, regardless of design specifications. For example, a CI expectation could be that the material of the BPEN body component should always be "plastic."

The second type of design expectation depends upon the initial design requirements. We refer to these as *Context Dependent on Requirements* (CDR) expectations. Again, these expectations do not make use of the current state of the design. The key design attribute is always an attribute of the initial requirements.

The final type is referred to as *Context Dependent upon the Design* (CDD). These expectations depend on the current state of the design. This type of expectation has both a key and dependent design attribute (see Fig. 3). Our sample BPEN expectation, described above, is an example of this expectation type.

8.3.3. Design expectation evaluation time

Different design expectation types are used at different times. All of these expectation evaluations occur prior to finalizing the design decision for the dependent design attribute. Once the evaluation occurs the expectation value is stored in the DDB, associated with the dependent design attribute. This allows for easy access to the expected value when the design decision is being made for the dependent design attribute.

In the case of CI expectations, evaluation occurs prior to the beginning of the design. When DSPL is initialized it loads the design expectations. The CI expectation eval-

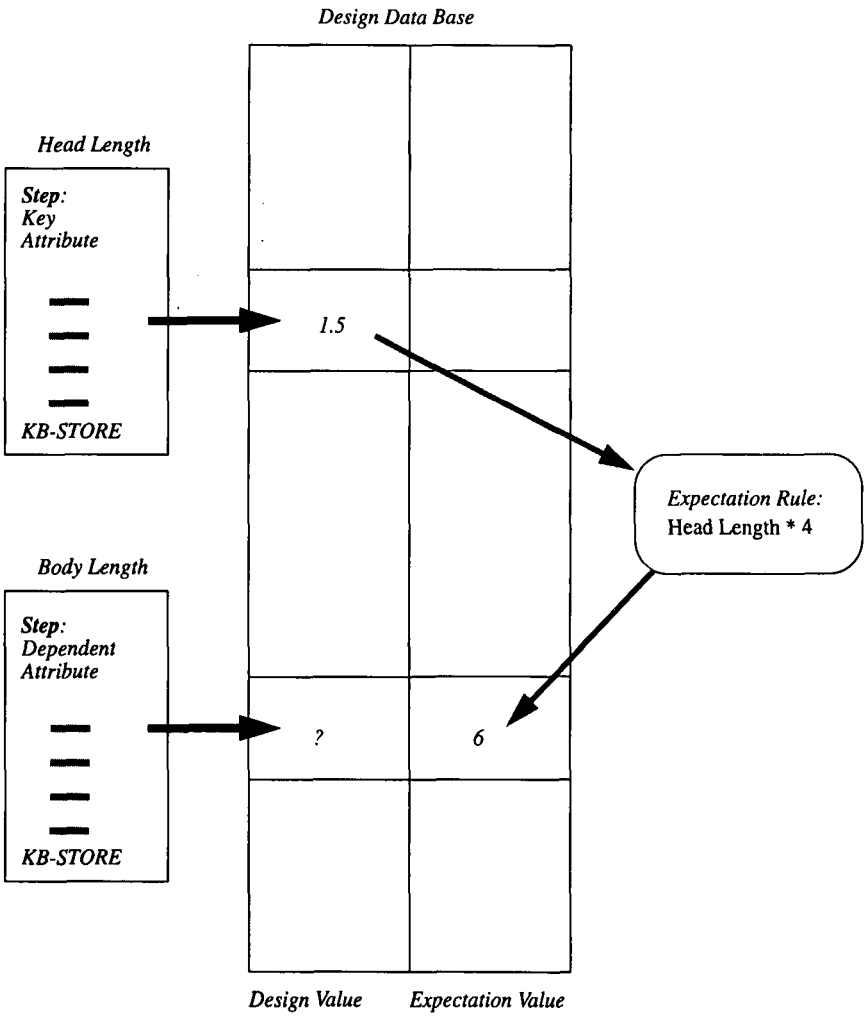

Fig. 3. Context dependent on design expectations.

uation is immediately made and the expectation values are stored with their associated dependent design attributes.

The evaluation of CDR expectations also occurs prior to the beginning of the design. These evaluations are performed immediately following the loading of the initial requirements. Once the requirements have been loaded, the CDR expectations, dependent upon requirement attributes, access them and perform any calculations that might be required by the expectation rule. As before, the expectation values are stored with their associated dependent design attributes.

The CDD expectation evaluation occurs during the actual design. It is performed at the time when the key design attribute is decided. Once again, the expectation values are stored in the DDB with their associated dependent design attributes.

An alternative time for expectation evaluation could be at the time when the dependent attribute is decided. This might be referred to as "late evaluation," as opposed to evaluating the expectation when the key design attribute is decided, which would be "early evaluation." The problem with this scheme is that the expectation no longer plays the role of a prediction, but acts more as a constraint. We prefer the scheme of early evaluation because of its predictive role and since one can imagine a human expert designer using a similar problem-solving mechanism.

8.4. Expectation violations

It has been said that "The greatest opportunity for learning takes place when an expectation that you have is violated by experience" (Edelson, 1992). An *expectation violation* occurs when an inconsistency is noticed between a design attribute's value and its expectation value. This research uses expectation violations as a signal to start reasoning with the deeper GOKB knowledge. Here we are concerned with expectation-failure-driven learning. Violations alert the system that there maybe an inadequacy with the surface knowledge. Inherited constraints should act to reduce those inadequacies.

When an inconsistency is detected, an appeal is then made to the GOKB. If relevant constraining knowledge is found, it can cause a constraint to be compiled out of this general knowledge into surface knowledge. The function of this new constraint will be to test the value of the design attribute for which there was an expectation.

Thus, expectation failures lead to a reevaluation of the current design knowledge. The result is the realization that knowledge may be missing, which may lead to the addition of a constraint. This leads to earlier detection of faulty decisions in subsequent design attempts. This new constraint will circumvent the need to refer back to the deeper knowledge from which this constraint originated. This is efficient because the more costly deeper knowledge need only be accessed after an expectation violation (see Fig. 4).

8.5. Triggering mechanism

Constraint inheritance should occur during the design phase of a session with DSPL. The inheritance mechanism invoked should interrupt DSPL, perform constraint inheritance, and then control should be returned to DSPL again. The constraint inheritance triggering mechanism that this research chose to model is triggered when expectations about a dependent design attribute value are in conflict with the dependent attribute's newly decided value.

An alternative triggering mechanism might have been that before updating any design attribute value to the DDB, one might browse the GOKB, allowing the value to be validated against the general object knowledge. If the value passes inspection, then the update to the DDB could occur, otherwise constraint inheritance would be triggered. Unfortunately, this is not the type of behavior that one could imagine an expert designer doing.

8.6. Design features

Design features are very important to the design process and should also be represented in the GOKB. Design features are considered to be any geometric form or entity used in reasoning during design activities. These features have originated in the reasoning processes associated with

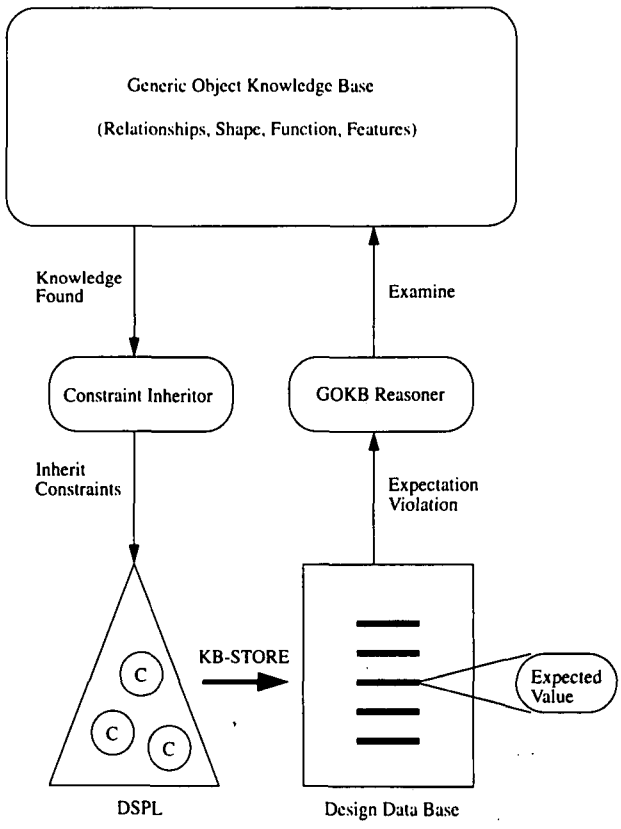

Fig. 4. *Constraint inheritance solution.*

these activities (Dixon et al., 1987). Features are extremely important to design and to manufacturing, because features provide more information than the basic primitives that are found in solid modeling systems.

Features often relate directly to the functions of the object. By representing information at the feature level, these higher level abstractions can be treated as entities that can be reasoned about. Features also enhance an object's description.

This research uses features to provide an alternative view of the object's structure. Features are used to focus reasoning in the GOKB when an expectation is violated. Expectations are tied to features. A feature contains a list of the components that are involved with designing that feature. For example, the slot in the body (see Fig. 1) is a feature of that component. The BPEN components that are involved with this feature are the body component, refill component, and pushing component. The scope of reasoning is controlled by only allowing those components in the list and their interrelationships to be analyzed. Thus, features provide an indexing mechanism from the expectation violations into the GOKB.

9. CONSTRAINT INHERITANCE SOLUTION

An expectation violation is the noticing mechanism that signals the possibility of missing knowledge in the DSPL design knowledge. What follows is an algorithm that summarizes our Constraint Inheritance Solution. Details are presented in the following sections.

1. Decision made for a Design Attribute.
2. Is there an expectation for this attribute in DDB?
   Yes: Was expectation violated?
      Yes: GOKB Reasoning Strategy takes control.
         Was relevant knowledge found in GOKB?
         Yes: Transform GOKB knowledge to a DSPL constraint.
            Inherit constraint into DSPL code.
            Execute constraint.
            Was new constraint violated?
            Yes: Break out of algorithm and return to DSPL failure handling or redesign.
            No: continue to 3.
         No: continue to 3.
      No: continue to 3.
   No: continue to 3.
3. Store Design Attribute Value in DDB.
4. Is there an expectation rule for this attribute?
   Yes: evaluate rule and update DDB.
   No: continue to 5.
5. Continue with design, returning to 1 when another Design Attribute is decided.

9.1. Constraint inheritance system architecture

Inheritance takes place in three phases: *Design Initialization*, *Task Initialization*, and *KB-STORE Processing* (see Fig. 5). Each phase interrupts the DSPL interpreter, and is triggered by a unique event. Design initialization is triggered only when the design knowledge is loaded. Task Initialization is triggered at the start of every DSPL task. KB-STORE Processing is triggered every time a design decision is made for a design attribute value. The following sections will describe these phases in more detail.

9.2. Design Initialization

This phase executes prior to the beginning of the design and performs the function of priming the DSPL system with the knowledge needed (i.e., the DSPL agents and the three different types of expectations).

The key design attribute of each Context Dependent on Design (CDD) expectation is traced to the feature to which it belongs. The system will link the expectation knowledge to the feature object. Each feature will maintain a list of pointers to all the expectations that will be evaluated during its design. This provides a means of packaging all expectations together with their associated feature. Since a DSPL Task is responsible for designing one small logical, structural or functional section of the design component (corresponding to a feature), it is appropriate to package CDD expectations in the same coherent manner. Thus, when a new DSPL Task is entered (i.e., to design a new feature), the system will focus on only the appropriate expectations.

Once all the expectations have been loaded and all the relevant structures have been initialized, the system is ready to begin the design. At this point, the CI and CDR expectations have been evaluated and have primed the design system. CDD expectations must, of course, be evaluated during the course of the design.

9.3. Task Initialization

DSPL Task Initialization is the second phase. It identifies the design feature associated with a task. It occurs on entry to every DSPL Task. Once the feature has been determined, the associated CDD expectations will be retrieved. The grouping of all the expectations about the attributes of a given feature with a representation of that feature has the effect of modularizing the expectation knowledge. Otherwise, expectations would be independent bodies of knowledge that just "float around" in the DSPL system.

9.4. KB-STORE Processing

KB-STORE is recognized by the DSPL interpreter as a command to update the DDB with an attribute's value.

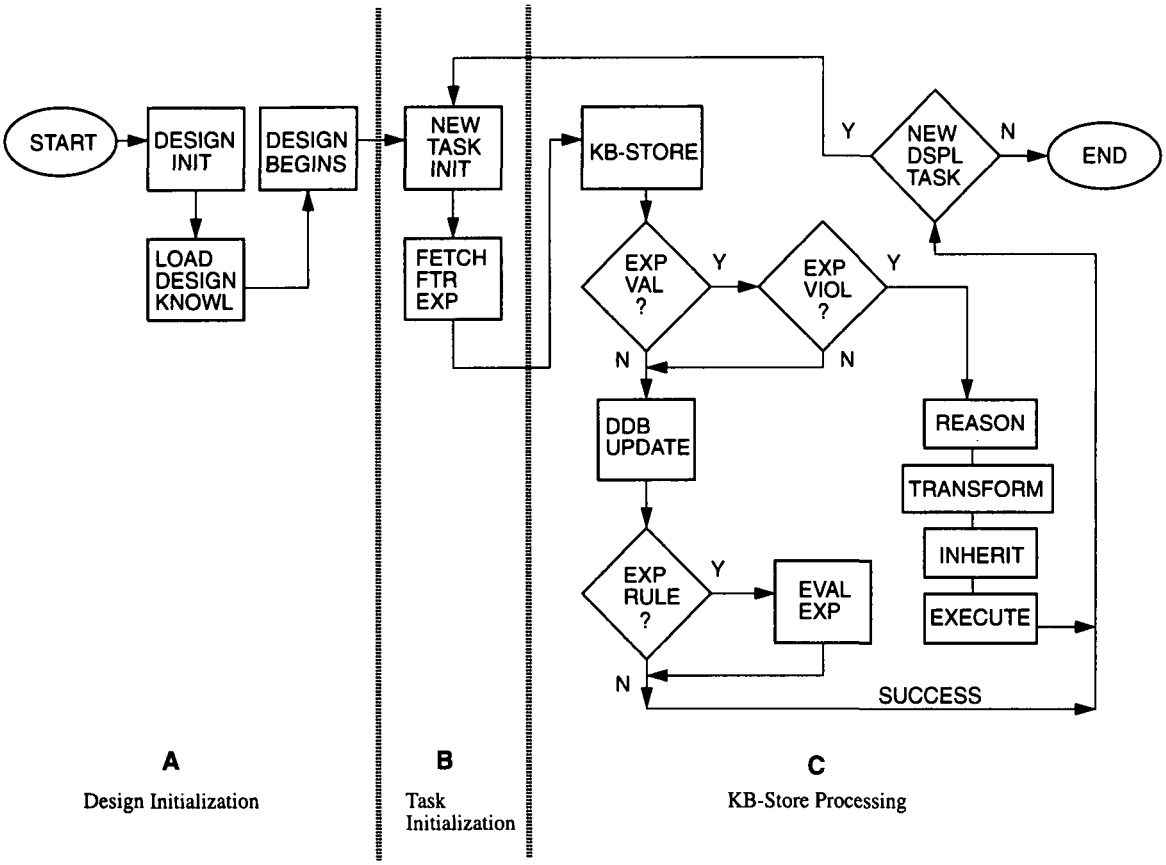

Fig. 5. Constraint inheritance architecture.

It is at this point, immediately prior to the KB-STORE command's execution, that constraint inheritance will be triggered upon an expectation violation. The KB-STORE processing phase is comprised of the following functions: *Expectation Value Query*, *GOKB Reasoning Strategy*, *DDB update*, and *Expectation Rule Query* (see the following subsections).

The burden of the constraint inheritance mechanism on the DSPL system is minor. The two queries, Expectation Value Query and Expectation Rule Query, check to see if their respective preconditions are met. The precondition for Expectation Value Query is that an expectation value is present. The precondition for Expectation Rule Query is met when an expectation rule is present. In the case where both preconditions are not met, then the KB-STORE function will execute as normal.

9.4.1. Expectation Value Query

This function identifies if there is an expectation value for the newly decided design attribute. If an expectation does not exist, control is directly passed to the DDB update function for normal processing.

If an expectation value is discovered in the DDB, then it is examined. There are two types of expectation value: numeric and symbolic. Numeric expectation values have positive and negative *tolerances* associated with them. By using tolerances, the range of expected values may be defined. For example, if an expectation value for the width of the body component is 0.5″, it may have positive and negative tolerances of 0.2″, which translates into an expected value range from 0.3″ to 0.7″.

We use tolerances, since expectations are intended to be approximate values. Both the positive and negative tolerances have to be considered when testing for a violation. If the value falls within the tolerance range, then constraint inheritance will not be triggered, and control will be passed directly to the DDB Update function.

Tolerances are not assigned for symbolic expectation values. In this case, the expected value must match the design attribute's value. If the values are equal, constraint inheritance will be bypassed.

When expectation value (of either type) is found to be inconsistent with the design attribute's value, then a DSPL expectation violation structure is created. This structure holds information about the violation, including the key and dependent attributes and their values. It also records the relationship between the attributes (i.e., the expectation rule), the name of the DSPL Step in which the expectation violation was discovered, as well as a list of related components that may be involved when design-

ing the current feature. It is at this point that the GOKB Reasoning Strategy is invoked.

9.4.2. GOKB Reasoning Strategy

The GOKB Reasoning Strategy is comprised of the following components: the *Reasoner*, the *Transformer*, the *Inheritor*, and the *Executor*. These components are detailed below. In summary, a subset of the GOKB knowledge is analyzed. If relevant constraining knowledge is found, it will be transformed into a DSPL constraint. This new constraint is put into the DSPL Step where the expectation violation occurred and then the new constraint is executed.

9.4.3. The Reasoner

The Reasoner first analyzes the *role descriptors* (RDs) of the dependent (target) attribute, since the expectation was for this attribute. RDs describe the potential values for a design attribute. They may enumerate the possible values (e.g., the value must be a member of {steel, iron, tin}) or specify bounds for possible values (e.g., the value must be within the range from 5 through 15).

If the relevant knowledge discovered was an enumerated role descriptor, the Reasoner will pass this knowledge to the Transformer with the instructions to create a constraint that performs a membership test.

If the relevant knowledge discovered was a bounded role descriptor, the Reasoner will instruct the Transformer to create two new constraints. One of these constraints will test the lower bound, while the other constraint will test the upper bound. An alternative would have been to construct one constraint with two tests, but the current DSPL constraint format forbids that.

If the RDs do not contain relevant knowledge, then the relationships of the key and dependent attributes are analyzed to see if a relationship exists between them. If one does, then it is examined to see whether it is similar to the relationship that the expectation had been testing. If such a relationship is discovered, then a constraint can be built from this mutually agreed upon knowledge.

Figure 6 illustrates an example of a *structural description* (SD) called SPRING-SURROUNDS-REFILL. This SD points to a relation called SURROUNDS. SDs represent relationships between RDs. The SURROUNDS relation points to the "surrounded" part which is the REFILL RD. The "surrounding" part is the SPRING RD. Thus, this relationship is in common between these two design attributes.

If no relations are found to be in common between the key and dependent attributes, then the related components that came from the current feature will be analyzed. It is possible that these related parts may provide a link, through transitive structural descriptions, between the key and dependent attributes.

Figure 7 illustrates some of the GOKB relations for the following example. Assume the expectation "BodySize should be greater than RefillSize" was violated. There are no SDs between REFILL and the BODY in this example. So the current feature being designed is accessed, that is, BODYSIZE-FTR. This feature's knowledge states that the REFILL, BODY, and SPRING are in-

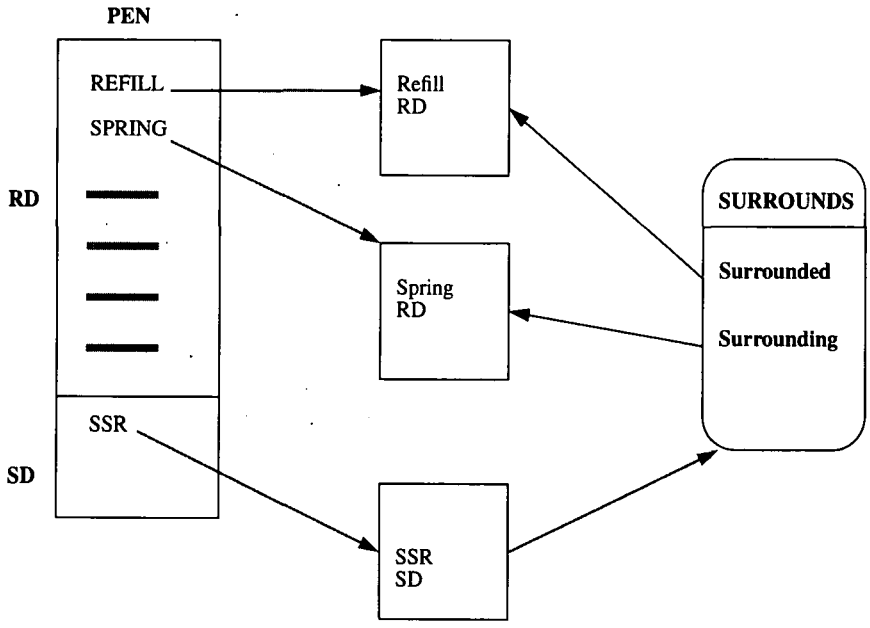

Fig. 6. Common structural descriptions.

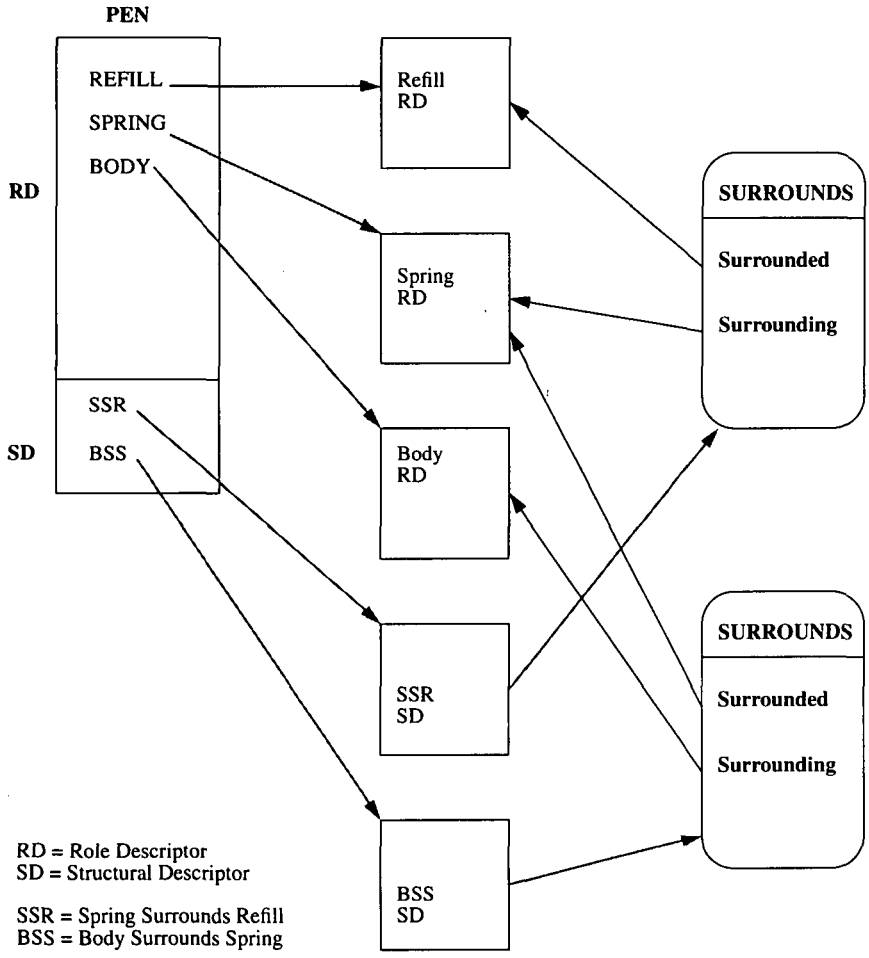

Fig. 7. Related structural descriptions.

volved in the design of BODYSIZE. A connection is established between the REFILL and the BODY by the SDs SPRING-SURROUNDS-REFILL and BODY-SURROUNDS-SPRING. The knowledge contained within the SDs will be integrated with other GOKB knowledge to form the DSPL constraint.

The new constraint would test the following relationship: BodySizeID > RefillOD. We conclude this since Body surrounds Spring and Spring surrounds Refill. In addition, we associate a quantitative test of ">" with the symbolic relation SURROUNDS. We also use the deeper knowledge, which states that outer diameters are larger than inner diameters and that these diameters are involved in the SURROUNDS relation.

If no relationships are found, then a negative report would be made to the DSPL system, constraint inheritance would terminate and the DSPL design process would continue. This is done for two main reasons. First, it is possible that the expectation is erroneous. Remember, we do not put great confidence in expectations. Expectation violations only *initiate* the GOKB reasoning strategy. Constraints are only inherited from the GOKB when a relevant explanation has been found by the reasoner. Second, extensive reasoning in the GOKB can be time-consuming and costly, leading to inefficient problem solving.

9.4.4. The Transformer

The Transformer component supervises the transformation of the relevant GOKB knowledge into a DSPL constraint structure. This component must also verify that similar constraints do not already exist.

In this prototype system the creation of a DSPL constraint is kept simple. The constraint will contain only bare essentials, such as the constraint body (the test to be performed), the name of DSPL Step that calls this constraint, and a failure message that indicates, among other things, that the constraint was inherited. This provides a link back to the originating deep knowledge. Failure suggestions have not been included, as they are the results of experience. Kwauk's approach to automatic suggestion generation in DSPL could be used to add these suggestions (Kwauk & Brown, 1988). The GOKB may also be a source for these DSPL suggestions.

9.4.5. The Inheritor

The Inheritor component modifies the DSPL step in which the expectation violation occurred, by adding a command that will execute the new constraint. The location of this command will always be immediately prior to the KB-STORE command. This has the desired effect of verifying the design decision immediately before the DDB update.

9.4.6. The Executor

The Executor component has the task of executing the newly inherited constraint. Since control is still within the constraint inheritance system at this point, the DSPL interpreter will need to be told the result of the constraint execution, just as if the constraint had always been there. Generally, the result will be a failure, as we expect the expectation violation to predict the result of the constraint. Control is returned to the DSPL interpreter along with the constraint result.

If the result of an inherited constraint's test is "failure," failure handling will be initiated. As the failure of an inherited constraint is not a routine failure, it is unexpected, hence failure correcting suggestions and redesign knowledge will not be available. If the design eventually does succeed, it will probably be because other key design attributes were redesigned, causing modification of the dependent design attribute's value. A successful design also means that the newly inherited constraint will have been tested and passed, ensuring that the new design attribute's value is valid.

9.4.7. DDB update

This function simply updates the DDB with the design attribute's value. It can be sure that no expectations or constraints have been violated due to this design decision.

9.4.8. Expectation Rule Query

The last component of the KB-STORE processing is the Expectation Rule Query. It is responsible for checking if any expectations are triggered by the new value. This is accomplished by accessing the current task's expectation list. If there are no expectations for the newly decided attribute, then KB-STORE processing is successfully completed and the design process continues.

If an expectation is found that needs to be triggered, the expectation rule is used, and the expected value is attached to the DDB location of the dependent design attribute. The value of a design attribute in the DDB is a pointer to a structure that has two fields, for the design attribute's value and the expected value.

This function can insert a pointer to the expected value, that is, to an expectation structure. The benefit of this method is that all the information concerning an expectation is localized into one structure. Once the expectation structure and the expected value of the design attribute in the DDB have been updated, then KB-STORE processing terminates and control is returned to the design process.

10. CONSTRAINT INHERITANCE SYSTEM PERFORMANCE

Constraint inheritance increases efficiency in two ways. First, the missing or inadequate surface knowledge needs only to be discovered once. Hence, reasoning in the GOKB occurs only once. Second, it is probable that prior to constraint inheritance an incorrect decision can remain undetected for a while. Once constraint inheritance occurs, a test will be made to try to prevent this decision. Thus, constraint inheritance can prevent a design decision error from propagating through the design process. This has the effect of reducing the amount of backtracking required by the design system, thus eliminating unnecessary effort.

Constraint inheritance may influence the execution path through the DSPL. It can prohibit an unacceptable design from being produced and produce a correct design instead. An inherited constraint that fails when tested may force the design process to follow a different and perhaps more efficient path.

When performing empirical evaluation, details such as execution time, the number of data base fetches and stores, or the amount of memory required could be considered. For DSPL, one should consider the amount of backtracking, failure handling, or redesigning needed, or the number of design agents needed to produce a valid design. The expectation is that the time spent backtracking, failure handling, and redesigning will be reduced after constraint inheritance, as problems will be detected earlier. Consequently, on average, the total number of agents used in the trace should be reduced.

A goal of the constraint inheritance system is to improve the quality of future problem-solving inferences. Constraint inheritance improves the problem solving by providing immediate access to new, usable problem-solving knowledge. However, the design run time may not always be reduced, since the GOKB may have to be explored. Reasoning in deep knowledge is time-consuming and costly. Knowledge compilation produces the benefits of using deep knowledge without the penalty of repeated deep reasoning.

Constraint Inheritance works in a domain-independent manner. As long as the design knowledge is encoded in DSPL, and the GOKB uses the same representations, then the methods described will be able to work. Its performance will vary depending on how many expectations are encoded, how many get violated during a design run, and how complete the design knowledge is to start with. Performance also varies depending on the requirements given, and how much inheritance has already been done.

11. THE PROTOTYPE IMPLEMENTATION

Both DSPL and the constraint inheritance prototype are implemented in Common LISP. The knowledge within the GOKB is represented in PCL-ONE, a prototype system that provides the basic knowledge representation functionality of KL-ONE (Brachman & Schmolze, 1989). PCL-ONE utilizes a structured semantic network. The PCL-ONE system and the DDB were implemented in Portable Common Loops (PCL), which provides object-oriented programming. PCL is a predecessor of the CLOS, the Common LISP Object System (Keene, 1989). CLOS was not available at implementation time.

Figure 8 depicts the entire system implementation. The bottom layer is Common LISP. The next layer up consists of two distinct interpreters: the DSPL interpreter and the PCL interpreter. The design knowledge for the BPEN domain is supported by the DSPL interpreter. The PCL interpreter supports both PCL-ONE and the DDB for BPEN. Likewise, PCL-ONE supports the BPEN GOKB.

12. EVALUATION OF COMBINED COMPILATION

Constraint inheritance (CI) is concerned with adding new constraints to DSPL. Constraint absorption (CA) is concerned with removing ineffective constraints from DSPL (Meehan & Brown, 1990). Constraint absorption is also a form of adjusting knowledge as a result of past experiences. An evaluation of design system performance, with both of these compilation systems running, was produced recently (Spillane & Brown, 1992). The Combined Compilation Method System (CCMS) was developed to study the compilation effects, both alone and together, of these two mechanisms. The BPEN domain was used. The CCMS results indicate that, in general, and in this domain, knowledge compilation *does* enhance design expert system performance.

For CI the execution time for some runs was longer after CI than it was before. This is due, in part, to a different and longer design path being used. The longer paths are caused by the increased number of constraints in the knowledge base.

At least one of the designs that would have been produced before the constraint was inherited was unacceptable. The design produced after CI was correct because the inherited constraint found the error during the design process. This forced the system to backup and use another plan, thus correcting the design.

Several design runs failed both before and after CI. However, after CI the failure was found earlier in the design process because of the inherited constraints. This resulted in reduced execution time for these runs.

The averaged results show that the execution time was shorter after CI and CA than it was before [see Spillane & Brown (1992) for more details]. This indicates that the knowledge base was more efficient after it was compiled by either or both compilation methods. In the evaluation tests, Inheritance resulted in a 9.64% speed up in the execution time.

13. FUTURE WORK

Expectations and their implementation have raised many issues. The first issue is to determine how expectations can be produced from previous experience using induction and generalization.

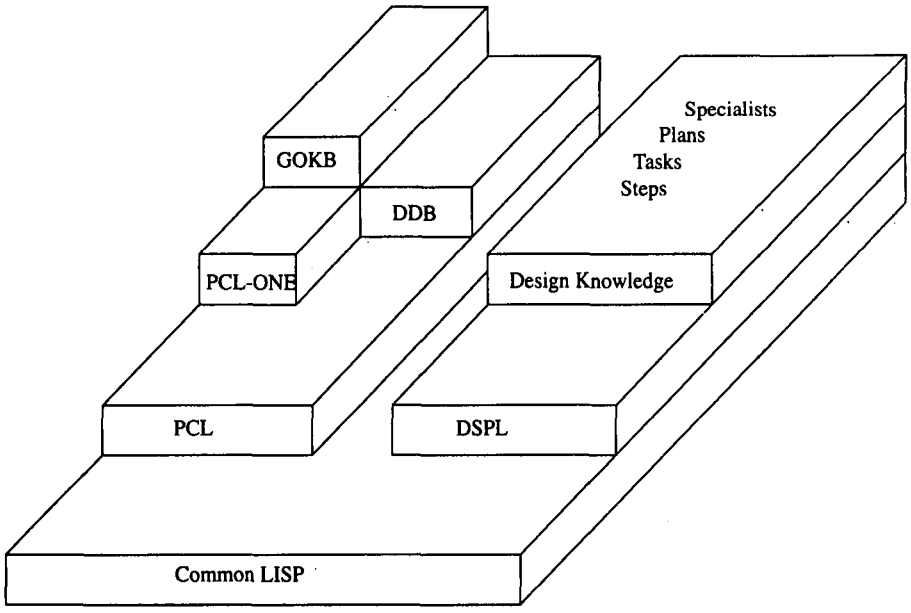

Fig. 8. System implementation.

Another set of issues is concerned with discovering conflicts concerning the expectations themselves. Assume that a decision is made, an expectation rule evaluated, and it is discovered that the resulting expected value is in conflict with the initial design specifications. This conflict may be caused by the fact that the expectation is erroneous, or that it needs to be context dependent. What should happen in this case? Should the design fail or just continue on after making a note of this discovery? The discovery in this case is now much more than an expectation violation.

The situation where multiple expectations exist for the same target (dependent) design attribute is also of interest. What happens if there are multiple expected values and these values are in conflict? Does one expected value have priority over another? If so, why, and does it depend on the expectation's triggering (key) attribute? Does a certainty factor for expectations need to be introduced? Is an expectation resolver required to investigate and resolve the multiple conflicting expected values?

If an expectation violation occurs and constraint inheritance fails, could the expectation be converted into some sort of weak constraint? This would probably require use of some form of certainty factor for expectations, as a weak expectation would make a terrible constraint.

Another issue is whether constraint inheritance should be triggered by just a single expectation violation. One of the attributes of an expectation is the violation count. Although this attribute has not been used in the prototype, its purpose is to trigger constraint inheritance only after the number of violations has passed some threshold. It could indicate whether the expectations were compelling enough to require constraint inheritance or whether violations were just "flukes" and should be recorded and ignored.

A Possible Solutions Knowledge Base (PSKB) would be a very efficient way of making explicit some knowledge that is currently only implicit within DSPL. The type of knowledge that could be represented within the PSKB would be knowledge of what kinds of designs this routine knowledge could lead to. Currently, this is implicit in DSPL, and inadequately represented in the DDB. The PSKB would be associated with both the DDB and the GOKB. The DDB and all its design attribute values would be an instance of part of the PSKB. On the other hand, the PSKB would contain better specified descriptions of the design object and its components then the GOKB. Case-Based Reasoning could also be used in conjunction with a PSKB. One could even imagine that the PSKB could be yet another source of design expectations.

14. ROLE OF MACHINE LEARNING

In general, learning is associated with some degree of improvement of a given task, due to past experience. Learning changes knowledge to allow improved performance. Constraint inheritance is a form of learning since there is a transformation from a less efficient to a more efficient representation of knowledge.

The main role of machine learning in this system has been that of automatic knowledge base refinement. The system developed has demonstrated that errors or inadequacies can be detected in reasoning that uses compiled knowledge. An explanation for this error or inadequacy is then sought in a deeper level of knowledge. If an explanation can be constructed from relevant deep knowledge, then a transformation occurs. This operationalization causes a task to be carried out better than before.

More specifically, constraint inheritance can be viewed as a form of failure-driven learning. Two important facets of learning are associated with failure-driven learning. These are "when to learn" and "what to learn" (Pazzani, 1986). Constraint inheritance knows that the time of learning begins upon an expectation violation, and that it is surface knowledge that is missing or inadequate.

Note that constraint inheritance as a form of learning is related to EBL (Ellman, 1989), is in the spirit of knowledge-base refinement (Ginsberg et al., 1985) and is related to incremental learning (Cohen & Feigenbaum, 1982; Carbonell, 1986, pp. 384-390).

15. CONCLUSIONS

This research has provided a better understanding of mechanisms in routine design problem solving. This should contribute to a better understanding of design in general. Experience plays a crucial role in the adjustment of routine design knowledge. Constraint inheritance is an incremental learning system that improves its problem-solving knowledge through problem-solving experiences.

It has been shown that the DSPL interpreter can use constraint inheritance to improve the performance of a DSPL knowledge base. By gradually extending DSPL to incorporate machine learning capabilities, such as constraint inheritance, more flexible and powerful design expert systems can be built.

The prototype system developed has demonstrated that some errors or inadequacies can be detected in compiled knowledge. An explanation can be sought for this problem by searching in a deeper form of knowledge. If an explanation is found in the relevant deep knowledge, then a transformation can occur, resulting in a new constraint. This can result in improved performance.

Expert systems have emerged into real world applications including government, medicine, engineering, finance, and manufacturing. If these knowledge-based expert systems are going to be of continued use in the future, they must possess the ability to monitor and adjust themselves in a changing environment. With knowledge compilation, the rigid boundaries of expert systems can be changed, and systems can become less brittle.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant DDM-8719960. We would also like to acknowledge the assistance of Bill Sloan, Maryann Spillane, Jingwen Liu, Ed Meehan, Ed Large, and the other members of the WPI AI in Design research group.

REFERENCES

Brachman, R.J., & Schmolze, J.G. (1989). An overview of the KL-ONE knowledge representation system. In *Readings in Artificial Intelligence and Databases* (Mylopoulos, J. and Brodie, M., Eds.), pp. 207-222. Morgan Kaufmann, San Mateo, CA.

Brown, D.C. (1985). Capturing mechanical design knowledge. *Proc. ASME Int. Computers in Engineering Conf.*, 121-129.

Brown, D.C. (1989a). Compilation: The hidden dimension of design systems. *Proc. 3rd IFIP WG 5.2 Workshop on Intelligent CAD*, Osaka, Japan, *Intelligent CAD, III* (Yoshikawa, H., Arbab, F., and Tomiyama, T., Eds.), 1991, pp. 99-108. North-Holland, Amsterdam.

Brown, D.C. (1989b). Adjusting constraints in routine design knowledge. Preprints of the *NSF Engineering Design Research Conference*, UMASS, Amherst, MA.

Brown, D.C. (1991). Emergent themes in intelligent CAD. In *Intelligent CAD, III* (Yoshikawa, H. and Arbab, F., Eds.), pp. 17-19. North-Holland, Amsterdam.

Brown, D.C. (1994). Routineness revisited. In *Mechanical Design: Theory and Methodology* (Waldron, M. and Waldron, K., Eds.). Springer-Verlag, Berlin, in press.

Brown, D.C., & Breau, R. (1986). Types of constraints in routine design problem-solving. In *Applications of Artificial Intelligence in Engineering Problems* (Sriram, D. and Adey, R., Eds.), pp. 383-390. Springer-Verlag, Berlin.

Brown, D.C., & Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies.* Morgan Kaufmann, San Mateo, CA.

Brown, D.C., & Sloan, W.N. (1987). Compilation of design knowledge for routine design expert systems: An initial view. *ASME Computers Eng. Conf.*, 131-136.

Carbonell, J.G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In *Machine Learning: An Artificial Intelligence Approach*, (Michalski, R.S., Carbonell, J.G., and Mitchell, T.M., Eds.), Vol. II, pp. 371-391. Morgan Kaufmann, San Mateo, CA.

Chandrasekaran, B., & Mittal, S. (1983). Deep versus compiled knowledge approaches to diagnostic problem-solving. *Int. J. Man Machine Studies 19*, 425-436.

Chandrasekaran, B., & Punch, W.F. (1987). Data validation, a step beyond traditional sensor validation. *Proc. 6th Natl. Conf. AI*, AAAI-87, 778-782.

Cohen, P., & Feigenbaum, E. (Eds.) (1982). Learning from examples. In *The Handbook of Artificial Intelligence*, Vol. 3, p. 360. William Kaufmann, Los Altos, CA.

Davis, R. (1989). Expert systems: How far can they go? *AI Magazine 10(2)*, 65-76.

Dixon, J.R., Cunningham, J.J., & Simmons, M.K. (1987). Research in designing with features. *Proc. 1st IFIP WG 5.2 Workshop on Intelligent CAD, Intelligent CAD, I* (Yoshikawa, H. and Gossard, D., Eds.), pp. 137-148. North-Holland, Amsterdam.

Descotte, Y., & Latombe, J.C. (1981). GARI: A problem-solver that plans how to machine parts. *Proc. 7th Int. Joint Conf. AI*, 766-772.

Edelson, D. (1992). When should a cheetah remind you of a bat? Reminding in case-based techniques. *Proc. 10th Natl. Conf. Artificial Intelligence*, 667-672.

Ellman, T. (1989). Explanation-based learning: A survey of programs and perspectives. *Computing Surveys*, ACM *21(2)*, 163-221.

Ericsson, K.A., & Smith, J. (Eds.) (1991). *Towards a General Theory of Expertise.* Cambridge University Press, Cambridge.

Fisher, D., Subramanian, D., & Tadepalli, P. (1992). An overview of current research on knowledge compilation and speedup learning. *Proc. ML92 Workshop Knowledge Compilation and Speedup Learning.*

Gero, J.S., & Maher, M.L. (Eds.) (1993). *Modelling Creativity and Knowledge-Based Creative Design.* Lawrence Erlbaum, Hillsdale, NJ.

Ginsberg, A., Weiss, S., & Politakis, P. (1985). SEEK2: A generalized approach to automatic knowledge base refinement. *Proc. 9th Int. Joint Conf. on AI*, Los Angeles, Vol. 1, pp. 367-374.

Goel, A.K. (Ed.) (1991). *IEEE Expert*, Special Issue on Knowledge Compilation, *6(2).*

Horner, R. (1989). *Knowledge Compilation Using Constraint Inheritance.* M.S. Thesis, Computer Science Dept., WPI.

Keene, S.E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison-Wesley, Reading, MA.

Keller, R.M., Baudin, C., Iwasaki, Y., Nayak, P., & Tanaka, K. (1989). Compiling special-purpose rules from general-purpose device models. Report No. KSL 89-49, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, CA.

Klein, D., & Finin, T. (1987). What's in a deep model? A characterization of knowledge depth in intelligent safety systems. *Proc. 10th Int. Joint Conf. AI*, 559-562.

Kwauk, R., & Brown, D.C. (1988). Generating and applying failure recovery suggestions in hierarchical design systems. In *Artificial Intelligence in Engineering: Diagnosis and Learning* (Gero, J.S., Ed.), pp. 29-50. Elsevier, Amsterdam.

Laird, P. (1992). Dynamic optimization. *Proc. 9th Int. Workshop on Machine Learning*, ML92, 263-272.

Liu, J., & Brown, D.C. (1992). The generation of decomposition knowledge for near routine design problems. *Proc. Int. Conf. Appl. AI Eng.*

Maher, M.L. (Ed.) (1992). *Preprints of the Machine Learning in Design Workshop '92*, AID'92, Second Int. Conf. on AI in Design, Pittsburgh, PA.

Meehan, E., & Brown, D.C. (1990). Constraint absorption and relaxation using a design history. *Proc. ASME Design Theory Method. Conf.*, 193-202.

Minsky, M. (1975). A framework for representing knowledge. *The Psychology of Computer Vision* (Winston, P.H., Ed.), pp. 221-277. McGraw-Hill, New York.

Pazzani, M. (1986). Refining the knowledge base of a diagnostic expert system: An application of failure-driven learning. *Proc. 5th Natl. Conf. AI*, AAAI-86, 1029-1035.

Reich, Y. (1991). Design knowledge acquisition: Task analysis and a partial implementation. *Knowledge Acquisition 3(3)*, 237-254.

Schank, R.C. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, Cambridge.

Sloan, W. (1988). *Constraint Migration: A Failure-Based Method of Knowledge Adjustment in Routine Design Expert System.* M.S. Thesis, Computer Science Dept., WPI.

Spillane, M.B., & Brown, D.C. (1992). Evaluating design knowledge compilation mechanisms. In *Intelligent Computer Aided Design* (Brown, D.C., Waldron, M., and Yoshikawa, H., Eds.), pp. 351-373. Elsevier Science Publications B.V. (North-Holland), Amsterdam.

Sriram, D., & Maher, M.L. (1986). The representation and use of constraints in structural design. In *Applications of Artificial Intelligence in Engineering Problems* (Sriram, D. and Adey, R., Eds.), Vol. 1, pp. 355-368. Springer-Verlag, Berlin.

Tong, C., & Sriram, D. (1992a). *Artificial Intelligence in Engineering Design*, Vol. 1. Academic Press, San Diego.

Tong, C., & Sriram, D. (1992b). *Artificial Intelligence in Engineering Design*, Vol. 2. Academic Press, San Diego.

Stauffer, L., & Slaughterbeck-Hyde, R. (1989). The nature of constraints and their effect on quality and satisficing. *Proc. ASME Design Theory Method. Conf.*, 1-7.

Rosemary Chabot is a Senior Software Engineer at Digital Equipment Corporation. She is currently working in the Multi-Vendor Customer Services Applied Research Group. She holds a M.S. in Computer Science from Worcester Polytechnic Institute and a B.S. in Math/Computer Science from the University of Hartford. She is a mem-

ber of AAAI. Her research interests include model-based reasoning, distributed AI, and machine learning.

David C. Brown is a Professor of Computer Science at Worcester Polytechnic Institute, and has B.Sc., M.Sc., M.S., and Ph.D. degrees in Computer Science. He is a member of the ACM, IEEE Computer Society, and the AAAI. He is on the Editorial Boards of two applied AI journals. He is an Advisory Committee member for several AI in Engineering and Design Workshops and Conferences. Current research interests include computational models of engineering design, and the applications of AI to engineering and manufacturing. He is the author, with B. Chandrasekaran, of the book *Design Problem Solving: Knowledge Structures and Control Strategies*, and a co-editor of the book *Intelligent Computer Aided Design*.